Chapter 1

APPLICATION OF GRAPH TRANSFORMATION TO VISUAL LANGUAGES

R. BARDOHL, G. TAENTZER
Department of Computer Science, Technical University Berlin
D-10587 Berlin, Germany
E-mails: {rosi,gabi}@cs.tu-berlin.de

M. MINAS
Department of Computer Science, University of Erlangen-Nürnberg
D-91058 Erlangen, Germany
E-mail: Mark.Minas@informatik.uni-erlangen.de

A. SCHÜRR
Department of Computer Science, University Bw München
D-85577 Neubiberg, Germany
E-mail: schuerr@informatik.unibw-muenchen.de

Graphs may be used to define visual sentences consisting of a set of visual symbols such as graphic primitives, icons or pictures, and a set of relations in between. Graph grammars are thus a natural means for defining a visual language in terms of its graph language. Using graph grammars for visual language definition, types of visual languages are tightly coupled with types of graph languages. Expressive power of a visual language type has to be considered carefully against efficiency of tools, e.g., when parsing visual languages. Context-free as well as different kinds of context-sensitive languages are considered in this chapter. Based on these concepts, two VL environment generating tools and languages—DiaGen and GenGEd—are presented that comprise generators for visual editors and parsers.
Graph transformation languages themselves form a kind of very high-level visual programming languages, where the underlying data model is a graph. These languages use graph transformation rules for manipulating graph instances on an appropriate level of abstraction. General purpose graph transformation languages PROGRES and Agg as well as special purpose languages DiaGen and GenGEd are discussed and compared with each other.

1.1 Introduction

Nowadays visual languages and visual environments emerge in an increasing frequency. This development is driven by the hope to provide considerable productivity improvement when using visual tools and languages. Visual languages are developed for very different tasks. Languages for visual modeling are accompanied by visual programming languages, visual database query languages and so forth. To get an overview of the recent developments the reader is referred to [1,2,3].

While visual languages are used for many different tasks it is still not clear how to define the term “visual language”. At least, visual languages have to possess a multi-dimensional representation of language elements. Similarly to textual languages, visual languages can be classified wrt. to certain paradigms such as constraint-based, data flow or rule-based. Languages based on graphs and graph transformation, the main topic of this book, are considered to be rule-based. Some of them are used as general purpose programming languages, others as special purpose languages for generating visual programming tools.

Although visual languages are used wide-spread, there is still a standard visual formalism as the Backus-Naur-Forms missing to define their syntax and semantics. All the currently used grammars for syntax definition of visual languages, such as picture layout grammars [4,5], or constraint multiset grammars [6], use textual instead of visual notations, i.e., multi-dimensional representations have to be coded into one-dimensional strings. Using graph grammars instead we have a much more natural and itself visual formalism for this purpose. Visual sentences consisting of a set of visual symbols such as graphic primitives, icons or pictures, and a set of relations such as “above” or “inside”, are defined by a graph. Graph rules define the language’s grammar. Its graph language determines then the whole visual language. There are several different graph transformation formalisms (compare [7]). A subset of them being applied to visual language definition is discussed in this chapter.

The theory of visual languages, e.g., a language hierarchy [8] and parsing algorithms [5,9,10,11] are very much oriented at the pendants for textual languages. Using graph grammars for visual language definition, the type of visual languages is usually not restricted to be context-free, but more general types of languages are often used. Theoretical results concerning independency of graph rule application can profitably be applied to efficiently parsing context-free as well as less restricted languages.

This chapter is organized as follows: Section 1.2 gives an overview of visual languages and environments. It concentrates on visual programming languages, especially rule-based languages. The tasks of defining the syntax of visual lan-

guages by means of graph grammars and of generating syntax-directed editors as well as parsers for certain classes of visual languages are the main subjects of Section 1.3, 1.4, and 1.5, respectively. Whereas Section 1.3 introduces into the syntax definition of context-free visual languages as well as less restricted ones by different kinds of graph grammars, Section 1.4 and 1.5 concentrate on tool generation based on these graph grammar definitions. Section 1.4 presents several generators for visual language editors which support syntax-directed and free-hand editing. In Section 1.5 several parsing algorithms are discussed.
Graph transformation based visual languages and corresponding tools are considered in Section 1.6 which completes the overview of visual languages given in Section 1.2. Mainly, general purposes languages PROGRES and AGG as well as VL environment generating languages DIAGEN and GENGED are discussed and compared with each other.

1.2 Visual Languages and Environments

This section introduces the reader to the world of visual languages with a main emphasis on visual *programming* languages. Its first Subsection 1.2.1 explains why it is so difficult to find a precise distinction between visual and textual programming languages and environments. The following Subsection 1.2.2 and Subsection 1.2.3 are nevertheless an attempt to come up with working definitions for visual programming languages and environments. They contain a brief survey about the history of visual programming and review major classification attempts. Subsection 1.2.4 presents afterwards that class of visual languages in more detail, which contains all graph transformation languages as special cases. This is the class of rule-based visual programming languages with its most important subclass of icon-rewriting languages. Finally, Subsection 1.2.5 summarizes the state of the art of visual programming.

1.2.1 Characterization of Visual Languages

Any book or survey paper about visual programming languages starts with an attempt to distinguish between visual and textual programming languages and their environments. In the very first moment this seems to be no problem at all, based on the assumption that visual languages have a graphical representation and that textual languages use a stream of characters as their representation. But characterizing a visual programming language as a language with a graphical representation defers the problem to the definition of the term *graphical representation*. Therefore, Brad Myers came up with another definition of the term *visual programming (VP)* in [12]:

> *"Visual Programming (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one dimensional stream."*

At a first glance, this definition based on the number of meaningful dimensions of a program (representation) seems to solve all our problems. But soon it will be obvious that the new definition raises further questions: Are languages like Cobol 73 [13], where the first 11 columns of a line are reserved for specific purposes, really visual programming languages? Are Nassi-Shneiderman diagrams [14], which simply offer another representation of a conventional program's flow of control, a visual programming language? And what about all those commercial "visual" programming environments—like VisualBasic from Microsoft Corp. [15]—which are user interface programming extensions of textual programming languages? In the following subsections we will answer these questions and come up with a slightly different working definition of the term "visual programming language".

1.2.2 Visual Programming Languages

The history of visual (programming) languages started about 30 years ago, when first prototypes of computers with graphics displays and gesture-based input devices (light pen) became available [16]. Usually mentioned ancestors in survey papers like [17,18,19] are:

- AMBIT/G [20] and PLAN2D [21], which are early examples of graph rewriting languages and which were developed to simplify pointer programming tasks.
- Pygmalion [22,23], which is probably the first icon-based programming-by-example language.
- GRAIN [24], a so-called Graphics-oriented Relational Algebraic INterpreter, which has a text-oriented user interface, but supports storage of pictures in and retrieval of pictures from a relational data base system.
- Grail [25] and PIGS [26] which are both imperative programming languages with a graphical representation of the flow of control and a textual representation for basic statements; Grail is based on flowcharts and machine language statements, whereas PIGS combines Pascal with Nassi-Shneiderman diagrams.

Hundreds of visual (programming) languages followed these trend-setting ancestors. Almost all of them are not general purpose programming languages, but were developed for various kinds of system modeling purposes. The artificial intelligence community invented for instance graphical notations like Sowa's Conceptual Graphs [27] for classification and reasoning purposes. And nowadays popular structured or object-oriented software engineering methods offer their users an abundance of different diagram types. Some well-known examples are data flow diagrams and class diagrams in OMT [28] as well as Message Sequence Charts and Statecharts in UML [29].
Reviewing the list of presented examples, we have difficulties to find one common principle which distinguishes visual and non-visual (programming) languages. This is due to the fact that some of them, like GRAIN, manipulate visual objects, but have a textual representation. Others, like Entity Relationship diagrams, have a visual representation, but are not real programming languages. And again others, like PIGS, are definitely real programming languages, but offer just an additional graphical representation for a traditional textual programming language like Pascal.
This observation caused Shu to introduce the following classification for visual languages [19]:

1. *Visual information processing languages* have a text representation, but are designed for the manipulation of inherently visual objects. One example of this category is the aforementioned image data base system (language) GRAIN [24].

2. *Visual interaction languages* deal with objects which have an imposed graphical representation. Examples of this kind are user interface programming languages like VisualBasic [15].

3. *Visual programming languages* are finally those languages which offer graphical constructs for programming purposes. One example of this category is Pygmalion [23].

Having some problems with this attempt to classify and characterize visual (programming) languages, we return to a variant of Myers' definition of a visual programming language (VPL) [12]:

> *"A visual programming language is a programming language with at least some constructs which have an either inherently or superimposed multi-dimensional representation".*

This definition includes all languages mentioned above, except GRAIN with its purely text-oriented representation.

Relying on such a broad definition of the term visual programming language, we are now ready to deal with the classification system for VPLs developed by Burnett and Baker in [30]. It classifies VPLs mainly with respect to their underlying programming paradigm and the type of visual representation. Other aspects, which are taken into account, are available language features, language implementation issues, language purpose, and theoretical background.
Usually distinguished (2-dimensional) visual representation types are

- *diagrammatic* representations with 2-dimensional shapes as object visualizations and polylines as (binary) relationship visualizations,
- *iconic* representations, where spatial relationships like "above" or "adjacent" between icons play the role of explicit connections between objects of diagrammatic representations,
- and *form-based* representations as they are used in spreadsheet applications and spreadsheet-based general purpose programming languages.

The classification of VPLs with respect to their underlying *programming paradigm* in [30] lists categories like concurrent language, constraint-based language, data flow language, form-based language, and rule-based language.
Visual graph transformation languages such as AGG (cf. Chapter ?? and Subsection 1.6.2), PAGG (cf. Chapter ??), or PROGRES (cf. Chapter ?? and Subsection 1.6.1) belong to the category of rule-based VPLs or, more precisely, to the subcategory of diagrammatic rule-based VPLs. As a consequence, we will continue our survey with a brief discussion of rule-based VPLs in Subsection 1.2.4. Section 1.6 contains a more detailed comparison of visual graph transformation languages.
For further details concerning other categories of visual languages the reader is referred to the IEEE series of Visual Language conference proceedings [31] as well as to paper collections like [32,33] or to the following two special issues of IEEE Computer [34,35].

1.2.3 Visual Programming Environments

The history of *visual programming environments (VPEs)* is tightly bound to the history of visual programming languages covered by the previous subsection. Repeating their development from a slightly different point of view would be rather boring. Therefore, we have adopted the approach of Allen Ambler and Margaret Burnett in their VPE survey paper [18] to broaden our view and include all ancient software tools with visual/graphical user interfaces. Such a decision is justified by the fact that many important concepts of VPEs were first available in tools which are either not programming environments or centered around textual programming languages.

Such a history should probably start with the system SketchPad [16], which was built at the beginning of the 60ies on the experimental U.S. Army, Navy, and Air Force computer TX2. Being a constraint-based, extendible graphics editor with a "What you see is what you get" user interface, it was decades ahead of its time. Push buttons, switches, and menus were already used for command activation purposes. A light pen together with drag, drop, and snap to grid functionality allowed its users to construct graphics interactively.

NLS/Augment [36] is another mind-setting VPE ancestor. It is the first attempt to implement Vannevar Bush's vision of a "Mind Machine" [37,38], a hybrid of nowadays used hypermedia editors and the world wide web. Its father, Doug C. Engelbart, invented the (three button) mouse. NLS/Augment was probably the first implementation of a *hypermedia tool* with support for multi-user hypertext editing and video conferencing.

Moving the focus from editing systems to programming environments, we have to mention two environments as further milestones for the development of modern user interfaces. The Smalltalk environment [39] was the first one which used *overlapping windows*, whereas the Cedar environment [40] introduced the less successful concept of *tiling windows*. These windows do not overlap, but rearrange themselves semiautomatically such that the available space is always completely covered.

The program visualization system BALSA and the generic programming environments Pecan and Garden added then the important idea of *multiple views* for the same logical data structure to the concept of window-based user interfaces. BALSA [41] gives its users extensive support for the construction of rather different types of graphical read-only views for monitoring the execution of programs. Pecan [42], on the other hand, was probably the first programming environment which supported visualization and editing of programs via multiple textual and graphical views. Its textual views were generated from programming language syntax definitions, whereas its graphical views were still hardwired and handcoded. Soon afterwards its successor Garden [43] became famous for its ability to automate parts of the development process of editable graphical program views. Furthermore, both Pecan and Garden as well as a many purely text-oriented programming environments, like the Cornell Program Synthesizer (CPS) [44], established the necessary technology for incremental analysis and compilation of manipulated programs. These were the necessary prerequisites for building highly responsive VPEs, where edited programs are continuously reevaluated.

It is quite obvious that all above mentioned systems had an impact on the development of VPEs, as they are used nowadays. They made important contributions to the appearance of their user interfaces and their responsive style of programming based on the tight integration of editing, analysis, and execution functionalities. Nevertheless, most of these systems are not VPEs with respect to the following definition:

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

SketchPad, NLS/Augment, and BALSA are not programming environments, whereas Pecan and CPS do not support visual programming languages. The only exception is the generic programming environment Garden, which was used, for instance, to build Petri net tools.
For further attempts to characterize VPEs and to clarify their relationships to software visualization systems like BALSA the reader is referred to the paper [45]. It partitions the field of software visualization into algorithm and program visualization. It defines VPEs as a subclass of program visualization tools with appropriate means for static code or data visualization.

1.2.4 Rule-Based Visual Programming Languages

The idea of programming with rules (productions) was strongly influenced by the long mathematical tradition to use axioms and inference rules for reasoning purposes. It is also closely related to the research area of formal language theory, where productions are used to describe, generate, and recognize languages and their sentences.
In order to be able to distinguish properly between (1) language modeling with rules (productions) and (2) programming with rules, we will use the following terminology throughout this chapter:

1. A *grammar* uses its productions to derive all sentences of a language from a given axiom. It distinguishes between intermediate derivation results in the form of nonterminal sentences and terminal sentences which cannot be rewritten. The "natural" semantics of a grammar is the set of its generated terminal sentences.

2. A *rewriting or transformation system*, on the other hand, takes a given set of facts as input and uses its rules to answer questions about these facts or to transform them into a desired state. It has no axiom and makes no distinction between nonterminal and terminal elements. The "natural" semantics of a rewriting system is a binary relation over a given domain of sentential forms.

In this subsection we are only interested in the story of rule-based programming, i.e., point (2) above. The related aspect (1) of defining grammars for visual languages will be postponed to Section 1.3. The development of rule-based programming language began about 25 years ago with the construction of expert systems like the medical diagnosis system MYCIN [46]. EMYCIN, the development language of MYCIN, and OPS5 [47] are two early examples of textual rule-based programming languages. Early examples of visual graph rewriting languages are AMBIT/G [20] and PLAN2D [21].
Postponing the presentation of graph rewriting (transformation) languages to Section 1.6, our history of *visual rule-based programming languages (VRPLs)* starts some years later on with systems like BITPICT [48], ChemTrains [49], KidSim [50], Vampire [51], and PictorialJanus [52]. All of them, except PictorialJanus, belong to the category of *icon rewriting languages*. Their underlying knowledge base is not a set of facts with a superimposed graphical representation, but a two-dimensional picture, usually called workspace or grid, with or without an underlying logical interpretation.
All these languages adhere to the simple *recognize-select-execute* model of rule-based languages:

1. *Recognize*: all or some occurrences (matches) of left-hand sides of rules (LHS) are computed.

2. *Select*: a predefined strategy is used to select one rule and one occurrence of this rule if recognize returns more than one applicable rule or more than one occurrence.

3. *Execute*: the selected rule replaces the selected occurrence of its left-hand side (LHS) by a copy of its right-hand side (RHS).

The whole execution process stops as soon as no more rules are applicable.
Despite their common execution model, VRPLs differ from each other with respect to many aspects like the structure of their workspace, the details of the pattern matching process, and so on:

- In BITPICT [48] workspaces as well as LHS and RHS of rules are simple pixel grids; a rule matches an area of the workspace which is identical to its LHS after geometric reflection and rotation operations. The system does not support modeling of higher-level entities.

- KidSim [50] uses different types of icons instead of a single type of pixel as grid entries. Its matching process does not involve geometric reflection and rotation operations, but supports abstraction by means of subclassing. A more general icon in a rule's LHS matches any more specific icon on the workspace. Icons may possess properties (like size) and rules may query and manipulate these properties.

- Vampire [51] is another step further on from the simple data model of bit matrixes to logical graph structures. It adds the concept of abstract spatial relationships—like "above"—and logical connections between icons to the pure icon rewriting concept of KidSim. It is, therefore, no longer necessary to write n different rules for processing all pairs of icons which are 1 to n grid units above each other.
- ChemTrains [49], finally, is a kind of missing link between pure icon rewriting languages and graph rewriting languages, which are no longer sensitive to geometric relations of regarded objects. It manipulates graphical objects like boxes and lines and regards only two types of spatial relationships between them: connects and contains. It is therefore no longer well-suited for manipulating object representations.

At the current time visual rule-based languages are mainly used to prototype interactive simulation programs, computer games, and visual programming environments with iconic user interfaces. Iconic VRPLs like KidSim are said to be well-suited as a first programming language for school children [50]. Simple KidSim rewrite rules are defined by first creating the expected initial situation and then editing this situation until it becomes the desired result. The initial situation corresponds to the left-hand side, the desired situation to the right-hand side of a new rewrite rule.

This approach gradually breaks down when we have to deal with non-toy problems. In this case means of abstraction are urgently needed such that the LHS of a rule does not only match exact copies on the workspace, but a whole family of similar situations. This was the main motivation for adding concepts like class hierarchies or textually defined and constraint icon attributes to the VRPL Vampire [51].

Other well-known shortcomings of pure rule-based languages are completely independent from the fact whether they have a visual or a textual representation. It is often said that production systems are easier to maintain and to modify than conventional programs [53]. In one case we define a new rewrite rule in order to add a new feature. It is then up to the system's recognize-select-execute cycle to call the new rewrite rule whenever possible. In the other case adding a procedure declaration to a program has no effects until the program's flow of control has changed, too. But the lack of any user-defined control structures often turns out to be a disadvantage, too. Without having any imperative means to control the order of rule applications we have to resort to tricks like adding extra elements to LHS and RHS of rules. The resulting production system has an implicitly defined flow of control and its rules may no longer be added or removed independently from each other. Regulated grammars or *programmed rewriting systems*—such as the graph transformation languages PAGG (cf. Chapter ??) and PROGRES (cf. Chapter ?? and Subsection 1.6.1)—address this problem by combining a rule-based style of programming with additional means to control the selection of applicable rules.

Last but not least, all presented rule-based languages neglect the necessity to specify the set of all meaningful workspace configurations separately from the set of rewrite rules that manipulate these configurations. A Vampire user may e.g. create any arrangement of connected icons on the workspace as the system's initial situation. But often a set of rewrite rules makes some assumptions about the (non)existence of certain icon configurations on the initial workspace. Therefore, a Vampire user will never know in advance whether a nontrivial set of rewrite rules is able to process a given input or not. This situation could be improved if icon hierarchies played a more important role. They can be extended to *class diagrams* with additional *integrity constraints*, as they are used in object-oriented modeling language like UML [29] as well as in the graph transformation language PROGRES of Chapter ??.

1.2.5 Summary of Visual Languages and Environments

On the previous pages we sketched the history of visual (programming) languages and environments. It became clear that visual languages are nowadays used by computer scientists for rather different purposes like knowledge modeling, data base design, or object-oriented (structured) analysis and design of software systems. Outside the computer science community, the usage of visual notations for developing architectural designs, circuit designs etc. has a long ongoing tradition. Still used text-oriented languages for modeling chemical processes or programming electronic components are currently replaced by visual languages, too.

It is safe to say that VPLs based on the concept of flow charts, the data flow programming paradigm, or the finite state automaton model are much more popular than those VPLs which support a rule-based style of programming. As a consequence there is a certain lack of VPLs which may be used to program abstract data manipulations on a very high level. This point of view is confirmed by the existence of hundreds of visual data base design languages, the existence of dozens of visual data base query languages, but the existence of only a handful of visual data base manipulation languages [54].

Despite the wide-spread usage of visual modeling, if not visual programming languages, 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—like picture layout grammars [5], or constraint multiset grammars [6]—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.
Throughout the rest of this chapter we will demonstrate that the graph transformation paradigm is one promising approach to overcome the deficiencies mentioned above: graph grammars are used in Section 1.3 as a powerful visual syntax definition formalism, graph transformation based (meta) tools are used in Section 1.4 to generate VL programming environments (with a main emphasis on VL editors), and they are used in Section 1.5 to generate VL parsers. Last but not least Section 1.6 shows that graph transformation based languages 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.

1.3 Defining the Syntax of Visual Languages

This section explains how graph grammars are used for visual language modeling purposes. Its first subsection explains the distinction between a visual language's concrete and abstract syntax, whereas the following subsections present different classes of (hyper)graph grammars of increasing expressiveness.

1.3.1 Concrete and Abstract Syntax of Visual Languages

Sentences of visual languages may often be regarded as assemblies of pictorial objects with spatial relationships like 'above' or 'contains' between them, i.e., their representations are a kind of directed graphs. Such a *spatial relationships graph* (SRG), which describes the structure of a visual sentence seen as a picture, is often complemented by a more abstract graph, the so-called *abstract syntax graph* (ASG). An ASG provides the information about the syntax and semantics of a visual sentence in a more succinct form. It is geared towards the interpretation or compilation of the regarded sentence.

The distinction between ASGs and SRGs—as introduced in [55]—has been inspired by the Model-View-Controller concept of Smalltalk and the traditional distinction between abstract and concrete syntax for textual languages. It is a prerequisite for modeling those visual languages, which offer different representations of the same concept simultaneously (e.g., a pure diagrammatic representation, a textual representation, ...). As both SRGs and ASGs are graphs, *graph grammars* are a natural means for defining the concrete and the abstract syntax of visual languages. An SRG graph grammar defines the language of all legal (spatial) combinations of regarded pictorial objects, whereas an ASG graph grammar defines the underlying (more abstract) language of all interpretable object configurations.

Other VL syntax definition approaches use relations or attributed multisets as their underlying data models. The main (dis)advantage of graph grammars compared with *relational grammars* [56] is that they enforce a strict distinction between objects (nodes) and relations between objects (arcs, 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* [5] or *constraint multiset grammars* on the other hand [8] are more important. The latter two approaches enforce a language modeler 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. For further details concerning differences of various VL syntax definition approaches and their accompanying parsing algorithms the reader is referred to Section 1.5.

The following Figure 1.1 shows the relationships between SRGs and ASGs and their purposes in more detail. SRGs and ASGs of a VL sentence must be derivable from the axioms of their corresponding graph grammars and may be manipulated using a syntax-directed editor. As already mentioned, once constructed ASGs are used for interpretation and compilation purposes, whereas SRGs are used to layout and to display (render) visual language representations. Please note that SRGs may also be produced as the output of a scanning and pattern recognition process as described in [57]. Furthermore, it is sometimes desirable to manipulate SRGs using a low-level graphics editor. A graph grammar parser is then responsible for recreating the new graph's derivation history.

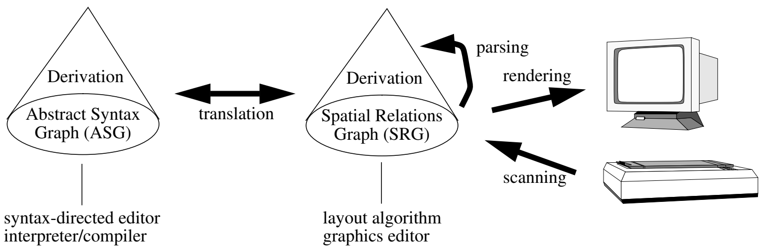


Figure 1.1: Three levels of visual sentence representation.

Please note that the derivation histories of SRGs and ASGs have to be modeled as separate data structures. A usual syntax-directed editor for ASGs or SRGs manipulates thereby the explicitly represented derivation history of a graph together with the graph itself. We will see later on in Section 1.4 that it is often possible to replace the graph grammar definition of a visual language by a graph transformation system, which defines all legal syntax-directed editor operations for a language of graphs without maintaining any derivation histories. It is then no longer possible to manipulate SRGs using a low-level graphics editor and to use a parser for regenerating the corresponding derivation history as well as the related ASG as shown in Figure 1.2.

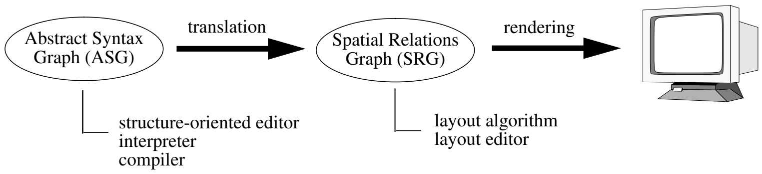


Figure 1.2: A simplified visual sentence representation approach.

Until now, we did not explain how ASGs and SRGs are related to each other, i.e., how an update of one graph is translated into an update of the related graph. In the most general case the relationships between ASGs and SRGs are modeled as coupled graph grammars as introduced in [58,59]. Assuming that any ASG graph grammar production is related to a single SRG graph grammar production and vice versa, it is a straightforward task to keep the ASG and SRG derivation histories and thereby the underlying graphs in a consistent state. In the absence of explicitly represented derivation histories a more primitive procedure has to be used to translate directly any ASG or SRG update into an update of the corresponding graph (as sketched in Figure 1.2.)

The following subsections explain in more detail how graph grammars may be used to define the syntax of visual sentences in order to distinguish correct visual sentences from incorrect ones and to define their structure. Graph grammars are briefly introduced in the following Subsection 1.3.2. Subsections 1.3.3 and 1.3.4 then present two different approaches how to represent visual sentences by graphs. Subsection 1.3.3 uses hypergraphs, whereas Subsection 1.3.4 uses graph structures. For each of these approaches, different types of graph grammars used for describing VL syntax are discussed, too. These subsections as well as the following Section 1.5 about graph grammar parsers disregard the distinction between ASGs and SRGs to a certain extent. The discussion related to Figure 1.1 and Figure 1.2 will be resumed later on in Section 1.4. There we will see that all presented graph-based VL environment (editor) generators realize only subsets of the conceptual VL representation model of Figure 1.1.

1.3.2 Graph Grammars

A visual language with a set of valid visual sentences that are represented by some kind of graphs (hypergraphs in Subsection 1.3.3 and graph structures in Subsection 1.3.4) corresponds with the language of the visual sentences' representations. Now we will briefly introduce general graph and hypergraph grammars. In the following subsections, we will then define several classes of grammars—*context-free* (CF) hypergraph grammars, *context-free* hypergraph grammars *with embeddings* (CF_e), and unrestricted graph grammars—which are apt to define such languages of graph representations. For these grammars, we will present parsing algorithms in Section 1.5. Such a parser is necessary for distinguishing correct visual sentences from incorrect ones and for constructing the logical structure of correct diagrams.

A graph grammar is similar to a string grammar in the sense that the grammar consists of finite sets of labels for nodes and edges, an axiom, i.e., an initial graph, and a finite set of productions. Each node, respectively edge, can be equipped with an element of the label set. These elements can be used as types or as terminal or nonterminal symbols. Each production consists of a left-hand-side graph (LHS) and a right-hand-side graph (RHS). It describes how an occurrence of the LHS in a graph can be replaced by the RHS; the resulting graph is said to be derived from the previous one by a rewrite step. The language of a graph grammar is the set of all graphs which can be derived in a finite sequence of rewrite steps from the initial graph.

A hypergraph grammar is a graph grammar with hypergraphs instead of graphs and hyperedges instead of edges. The axiom of a hypergraph grammar is a graph consisting of a single hyperedge that visits each of the axioms nodes and

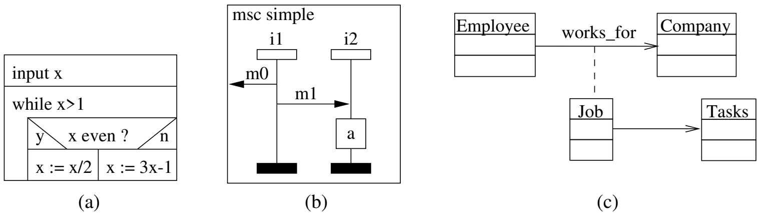


Figure 1.3: Instance of a Nassi-Shneiderman diagram (a), of a Message Sequence Chart (b), and of a UML class diagram (c).

that has a nonterminal label ("*axiom label*").

Similar to string grammars certain types of graph grammars with different generative power are distinguished. Each type is defined by imposing certain restrictions on their productions and permitted derivation sequences. In Subsections 1.3.3 and 1.3.4 we will discuss three grammar types. The reason for restricting grammars is that the existence of parsers for a grammar and their efficiency highly depend on these restrictions.

To motivate and illustrate the need for different graph grammar types we will use Nassi-Shneiderman diagrams (NSDs), Message Sequence Charts (MSCs), and a specific kind of UML class diagrams as running examples. NSDs are a VL for representing structured programs [14] by attaching boxes to obtain rectangular diagrams. MSC is a language for the description of interaction between entities [60]. A diagram in MSC describes which messages are interchanged between process instances, and what internal actions they perform. UML [29] offers a comprehensive VL for system modelling. It contains, e.g., class diagrams for describing object-oriented classes and their relationships. In our example we restrict class diagrams to classes represented by boxes and associations represented by arrows between classes. Associations may be augmented by association classes; this is visualized by a box representing the association class connected to the association with a dashed line. Figure 1.3 shows sample diagrams for NSD, MSC, and UML class diagrams.

Also similar to string grammars, graph grammars can make use of attributes. As described in the following, we extend nodes and edges by attributes which may be used to represent actual positions and sizes of pictorial objects. E.g., for the NSD example, we will use two node attributes x and y for the coordinates of the box vertex represented by this node. Of course, other information like color and texture can also be represented by attributes. Among other things, the next subsection and Section 1.4 briefly discuss how attributed graph grammars may be used to define the syntax of VL sentences. Attributed graph grammars may process attributes quite similar as attributed string grammars [61]: each node and each (hyper)edge with the same label has to carry attributes of the same types. Each production describes relationships between attributes which are used by the production. Whenever a production is going to be applied, i.e., when the LHS is matched with a host subgraph, matching attributes are identified. The relationships between the production's attributes either restrict application of the production (if the attributes of the matched subgraph do not satisfy required relations) or they contain information how to compute attribute values which were yet unknown.

1.3.3 Hypergraph Representation of Visual Sentences

This subsection explains how hypergraphs and hypergraph grammars can be used to model visual sentences. Primarily, we will discuss graphs which are closely related to the visual appearance of a VL sentence, i.e., the sentence's SRG. Of course, the same graph concepts can be used for the ASG. However, the structure of ASGs is less determined by the sentence's visual appearance, but more by its logical structure and how the sentence is further processed within the VL environment. Since we focus on VLs as such but less on their use within VL environments, we emphasize SRGs.

We will first describe how hypergraphs may be used as a representation of visual sentences. We will then discuss how the syntax of certain VLs can be expressed by context-free hypergraph grammars. A more expressive grammar approach that allows to describe a less restricted syntax is discussed afterwards. Even more general kinds of hypergraph grammars may be used in this context, but we restrict our discussion to these kinds of syntax since they allow for more or less efficient parsing which is described in Section 1.5. Subsection 1.3.4 will continue with a different approach for representing visual sentences. Since parsing is not an issue of this approach, a more general kind of syntax is described there.

Hypergraph Model There are two ways to use graphs for modeling visual sentences: Pictorial objects are either represented by nodes and relationships between them by edges, or pictorial objects are represented by edges and nodes are used for connections between related objects. A variant of this first approach is discussed in detail in Subsection 1.3.4. This subsection explains the second approach in detail. It is an appropriate means for representing visual sentences in a uniform way. In particular, free-hand editing as used by DiaGen (cf. Subsection 1.4.3) benefits from this approach. However, it requires hypergraphs instead of graphs as in the other approach.

A hypergraph is a generalization of a directed graph with a finite set of nodes and a finite set of hyperedges. Each hyperedge is labeled and has a certain number of tentacles. The hyperedge label determines the number of tentacles. Each tentacle is connected to a node. Directed edges are hyperedges with a source and a target tentacle. This definition of hyperedge-labeled hypergraphs over a certain label set corresponds to 0-hypergraphs defined in [62].

A visual sentence consists of a set of pictorial objects that are related in a way specific to the VL. Such objects are indivisible visual structures that are created, deleted, and related to other objects using editors in a VL environment. Hypergraphs offer a natural representation of visual sentences: each pictorial object is represented by a single hyperedge labeled by the object's type. Its tentacles model the object's attachment areas where it can be attached (i.e., related) to other objects. Nodes are used as connecting entities: pictorial objects related by their attachment areas are modeled by the corresponding tentacles being connected to common nodes.[a]

In our NSD example (Figure 1.3a), pictorial object types and thus the labeling alphabet consist of *text*, *cond*, *while*, and *until* for simple statement boxes, condition boxes, while and until polygons. The attachment areas are the boxes' vertices. The hypergraph for the NSD of Figure 1.3a is shown in Figure 1.4a. Hyperedges are depicted similar to diagram components in order to visualize the correspondence between tentacles and attachment areas.

For MSC (Figure 1.3b), we have a labeling alphabet *msc*, *start*, *end*, *line*, *message*, *action*, and *text* representing surrounding boxes, start and end boxes, vertical lifelines, message arrows, action boxes, and labeling text. A surrounding box has the borderline, an area for lifelines, and a subarea for labeling text as attachment areas. A message's attachment areas are head and tail of the arrow as well as an area for the labeling text. The other diagram components have similar attachment areas. Special consideration is needed for lifelines: A natural representation would be a hyperedge representing the whole lifeline. This hyperedge would need as many tentacles as there are actions, incoming, and outgoing messages plus two additional ones for beginning and end of the line. However, dealing with hyperedges with variable number of tentacles makes problems with syntax definition. Therefore, we represent lifeline segments between each two events or actions instead of the whole lifeline. Figure 1.4b shows the hypergraph representing the MSC shown in Figure 1.3b.

Context-Free Syntax The syntax of a VL distinguishes between correct and incorrect sentences of the VL. When visual sentences are represented

[a] There are VLs using general spatial relationships between their components which have to be represented by hyperedges, too. Such VLs are not considered here.

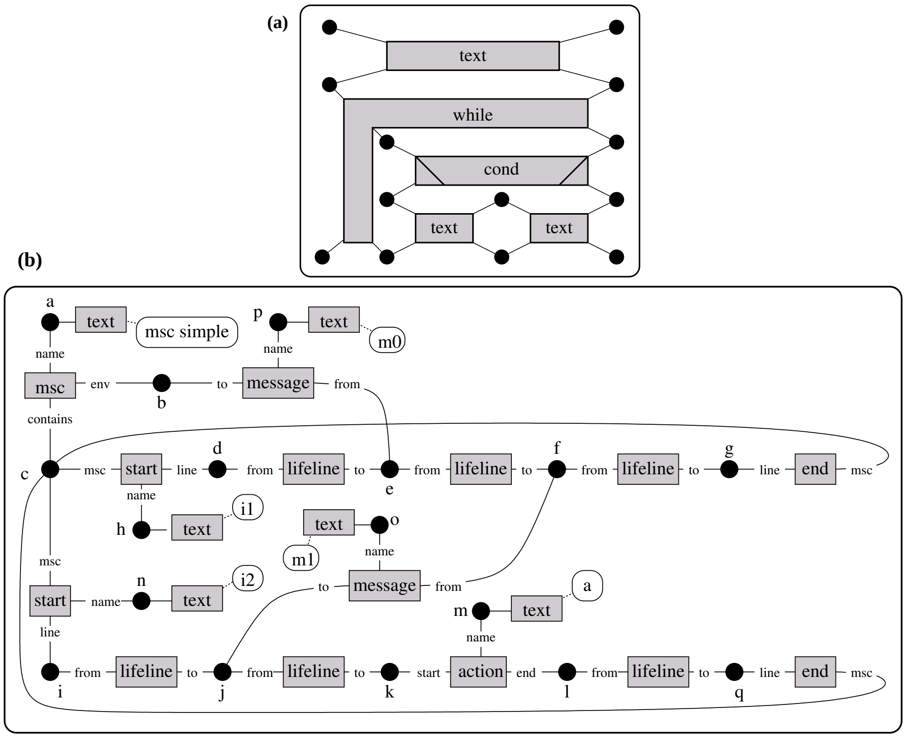


Figure 1.4: Hypergraph representation of the NSD (a) and the MSC (b) in Figure 1.3. Nodes are depicted as black dots, hyperedges as boxes, and tentacles as lines connecting hyperedges with nodes. Tentacles of (b) are labeled where necessary. Node labels in (b) are used as references in Figure 1.7. Ovals represent text attributes of the associated hyperedges.

by hypergraphs, hypergraph grammars can be used to define the language of hypergraphs representing correct visual sentences. The simplest hypergraph grammar type is the class of *context-free hypergraph grammars* which are quite similar to context-free string grammars and which are discussed in detail in [62]. A context-free hypergraph grammar has context-free productions only. The LHS of each production consists of a single hyperedge labeled by a nonterminal symbol. The RHS consists of an arbitrary hypergraph over terminal and nonterminal symbols. Each node of the LHS has a corresponding node of the RHS. Informally spoken, a rewrite step using a context-free production consists of removing the LHS hyperedge from the host hypergraph and pasting in the RHS instead. The correspondence between LHS and RHS nodes determines

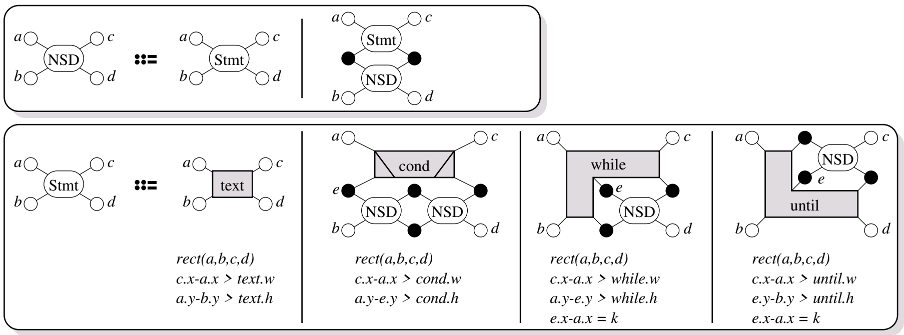


Figure 1.5: Context-free hypergraph productions for NSDs with constraints on node and hyperedge attributes. $rect(a, b, c, d)$ is an abbreviation for some equations requiring that a, b, c, and d represent vertices of a rectangle. k is some constant value.

how the RHS has to fit into the remainder of the host hypergraph after removing the LHS. Because of this behaviour, context-free hypergraph grammars are also called hyperedge replacement grammars [62].

The hypergraph representations for NSDs can be specified using a context-free hypergraph grammar. Its terminal symbols are *text*, *cond*, *while*, and *until*, the types of NSD components. Nonterminal symbols are *NSD* and *Stmt*. *NSD* is also the grammar's axiom label. The context-free hypergraph productions are shown in Figure 1.5, that also contains constraints on node and hyperedge attributes which are explained later. Terminal and nonterminal hyperedges are depicted as gray resp. oval white boxes. Productions are represented as $L ::= R$ where productions with the same LHS separate the different right-hand sides by $|$. Corresponding LHS and RHS nodes are equally labeled.

The derivation sequence creating a hypergraph from the axiom indicates that the hypergraph is a member of the grammar's language. In general, this derivation sequence is not unique; many permutations of a derivation sequence are equivalent derivation sequences. A derivation sequence is hence not an adequate representation of the hypergraph's syntactic structure. However, a derivation sequence starting from a hypergraph with a single hyperedge in a context-free hypergraph grammar defines a *derivation tree* in almost the same way as a derivation sequence in a context-free string grammar does. Informally spoken, the derivation tree has hyperedges as its nodes. The root is the starting hyperedge, and each node has those hyperedges as children which replace the node's hyperedge in the derivation sequence. A formal definition of derivation trees for context-free hypergraph grammars and a detailed discussion of their properties and relationships to derivation sequences can be found in [62,63]. The derivation tree of a terminal hypergraph starting at the axiom represents the set of all equivalent derivation sequences and thus the syntactic structure of the hypergraph. Therefore, this tree is the preferred representation of the derivation history as shown in Figure 1.1.

If nodes and hyperedges have attributes, context-free hypergraph grammars can be extended for attribute evaluation purposes similar to attributed context-free string grammars [61]: each production is extended by expressions that assign values to some node or hyperedge attributes of its LHS or RHS depending on other node or hyperedge attributes of the LHS or RHS. Expressions get evaluated when a derivation tree of a hypergraph is completed: Some attribute values are externally set, the others are calculated using the expressions. The problem of value assignment thus consists of evaluating all the expressions in a sequence such that each required value has been either set externally or computed earlier. Inconsistent, circular, and undetermined evaluation sequences may occur if there are no restrictions on expressions attached to productions. For briefness we do not explain in detail how these problems are solved. We rather briefly explain a more general approach: each production is extended by *constraints* on any node or hyperedge attributes of its LHS and RHS. Constraints are more general than value assignments of regular attributed grammars since constraints do not impose a certain evaluation sequence. Figure 1.5 shows the productions for the NSD example extended by constraints which are linear inequalities here. Attributes are referenced as $n.a$ where n is a node or hyperedge label and a one of n's attributes. Each node carries attributes x and y for the coordinates of the represented box vertex. Furthermore, *text*, *cond*, *while*, and *until* have attributes w and h representing width and height of the contained text. Attribute values are again computed when a derivation tree is completed: The hyperedge attributes are externally defined by represented typed text. However x and y values have to be determined such that all constraints implicitly defined by the derivation tree are satisfied. For our NSD example each assignment of values to attributes represents a valid NSD layout. Computing such valid values assignments is in general NP-complete, however for certain types of constraints efficient evaluation algorithms are known, e.g., for constraints based on linear inequalities as in our NSD example [64].

Context-Free Syntax with Embeddings Context-free hypergraph grammars are very limited in their expressiveness. Each context-free production is only permitted to refer to a single hyperedge which is replaced when the production is applied. It is not possible to refer to more than one hyperedge which is frequently necessary, e.g., for our MSC example: Appropriate productions have to create message hyperedges between any two nodes that are visited by lifeline hyperedges. Any production describing this situation has to refer to these lifeline hyperedges. Hence, context-free grammars cannot be used for describing MSC. Instead of context-free hypergraph grammars, we now consider context-free ones with embeddings, a slightly extended but much more expressive grammar class, which can describe MSC hypergraph representations.

A *context-free hypergraph grammar with embeddings* ($\mathrm{CF}_e\mathrm{HG}$) is a context-free hypergraph grammar with additional *embedding productions*: the RHS of an embedding production is the same as its LHS with an additional hyperedge. The application of an embedding production thus simply adds a hyperedge to a given context. However, in order to make efficient parsing possible (cf. Section 1.5), there is a restriction on permitted derivation sequences: embedding productions must not be applied to a context that contains hyperedges which have been created by any previous rewrite step using an embedding production.

The hypergraph representations for MSCs can be specified using a $\mathrm{CF}_e\mathrm{HG}$. Its terminal symbols are the types of MSC diagram components. Nonterminal symbols are S, M, and *Seq*. S is the axiom label. Figure 1.6 shows the productions. The last two productions are embedding productions which embed a message into an appropriate context, the others are context-free ones. Actually, one production is missing which is the same as the last one except the M hyperedge which has to be reversed.

The derivation structure and history of a hypergraph (cf. Figure 1.1) is represented quite similar to derivation trees for context-free grammars. Because of the embedding productions, we obtain a directed acyclic graph (DAG) instead of a tree: application of context-free productions extends the DAG in the same way as for derivation trees, but each application of an embedding production inserts a new node for the embedded hyperedge with DAG edges from all nodes representing the context to this new node. Derivation DAGs have a certain structure because of the restriction on derivation sequences: they consist of a "main" derivation tree and some additional derivation trees starting at embedded hyperedges which are pasted into the main derivation tree by DAG edges. Figure 1.7 shows the derivation DAG for the MSC hypergraph in Figure 1.4. DAG nodes are labeled with the label and the visited nodes of the represented hyperedge.

We can conclude from the structure of derivation DAGs that each hypergraph of the language of a $\mathrm{CF}_e\mathrm{HG}$ can be separated into a collection of subhypergraphs which are derived by the grammar's context-free productions. One of the subgraphs is derived from the axiom, the others are derived from hyperedges created by application of embedding productions.

We will see in the following subsection that even $\mathrm{CF}_e\mathrm{HG}$s do not have sufficient

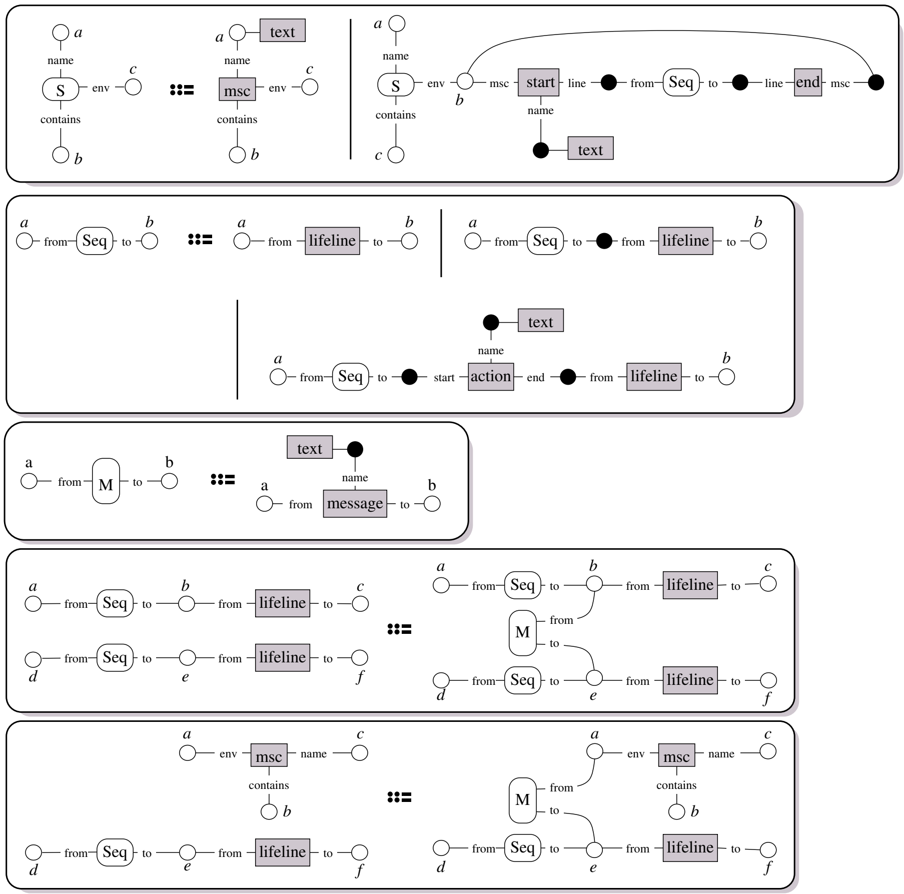


Figure 1.6: Context-free hypergraph grammar with embeddings for MSCs.

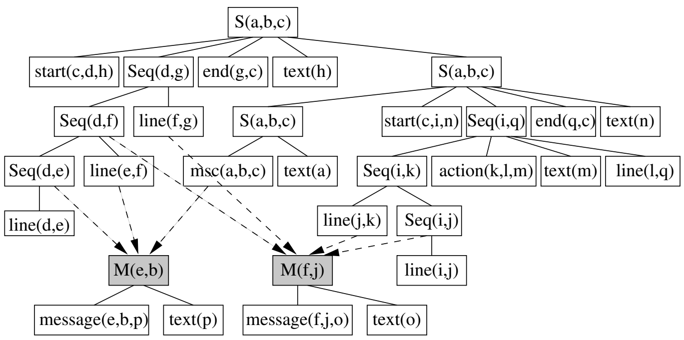


Figure 1.7: Derivation DAG of the MSC hypergraph in Figure 1.4.

expressive power for some VLs. Therefore, the following subsection considers unrestricted grammars, however not for hypergraphs as in this subsection, but for graph structures.

1.3.4 Graph Structure Representation of Visual Sentences

General attributed and labeled graph structures as introduced in [65] can be used to model all graph classes, e.g, hypergraphs as presented in the previous subsection as well as attributed graphs consisting of nodes, edges and (data) attributes. They are based on algebraic concepts as presented in [66], i.e., graph structures are algebras wrt. an algebraic signature, which consists of unary operation symbols only. This general graph structure approach is adapted for modeling VLs. Since connections between visual symbols require graphical constraints between them, these constraints are put into equations extending the signature and yielding a visual (algebraic) specification of a specific VL. We consider the visual specification as the VL-alphabet: visual symbols define the sort symbols with a certain layout, and connections define the operation symbols. On the other hand, connections are relations between visual symbols given by graphical constraints. These constraints define the equations which have to be satisfied by each visual sentence, i.e., by each algebra. As usual the VL-grammar consists of a start graph (axiom) and several productions. The start graph as well as each LHS and RHS of a production are algebras wrt. the VL-alphabet, i.e., the visual specification of a specific VL. Such VL-grammars are not restricted to be context-free nor context-free with embeddings.

An algebraic specification for hypergraphs modeling VLs can also be given according to the definition in [62] which is the basis for the hypergraph approach as presented in this chapter.[b] Although it is possible to model hypergraphs using the graph structure approach, the VL-definition will be different in this subsection. We have seen that hyperedges are used to define the pictorial objects, and nodes to define the connections. In our approach we do not distinguish between nodes and edges. Graph items are used to hold the pictorial objects, and operations between graph items define the connections. Additionally, we do not distinguish between nonterminal and terminal symbols.

In the previous subsections hypergraph grammars have been presented concerning the *concrete syntax* of NSDs and MSCs. Remember, that a SRG represents the pictorial objects, i.e., it is a reduction of the spatial relations, whereas the ASG represents the *abstract syntax* of a visual sentence. Within this subsection we discuss the graph structure approach according to the *concrete* as well as the *abstract syntax*. This is illustrated by a specific kind of UML class diagrams (cf. Figure 1.3 (c)). It is neither possible to define the *abstract syntax* of class diagrams by a context-free hypergraph grammar nor by a context-free one with embeddings. Therefore, we switch now to the graph structure approach to describe the syntax of this specific VL.

A visual sentence is an attributed graph structure [68], i.e., an algebra wrt. a visual specification (VL-alphabet) consisting of a logical part according to the *abstract syntax*, a graphical part according to the *concrete syntax* and suitable layout operations coupling both parts. The latter one is due to the *translation* in Figure 1.2 (cf. Page 14). The logical part of a visual specification is defined by a graph signature, a data signature and attribution operations. All sort symbols (sorts) of the logical part define the types occurring within a visual language, whereas the operation symbols (opns) define the connections between these types. The logical part of the visual specification for class diagrams is illustrated by Figure 1.8. It does not include the whole signature for the string data type which is built up over characters, because it is assumed to be well known.

An attributed graph structure is an algebra wrt. a visual specification, i.e., for each sort symbol exists a carrier set and for each operation symbol exists an operation, and equations (not described until now) are satisfied by each algebra. The graph part is given by total operations whereas the data and attribute parts are partial (cf. [69,70]). The graph-like logical part of the algebra, i.e., the visual sentence illustrated by Figure 1.3 (c), is presented in Figure 1.9. The carriers are visualized by rectangles with a leading number followed by its type inside. Each operation is visualized by an arrow.

The algebraic approach allows to define a kind of hierarchy on the types (sorts)

[b] An algebraic graph structure signature for hypergraphs supposing a slight different notation can be found by Def. 4.5.4 in [67].

ClassDiagram$_{\mathsf{Logical}}$ =

Graph Part	**sorts**	Class, Assoc, AssocClass
	opns	beg_assoc: Assoc → Class; end_assoc: Assoc → Class; assocclass: AssocClass → Class; beg_assocclass: AssocClass → Class; end_assocclass: AssocClass → Class;
Data Part	**sorts**	String
Attr Part	**opns**	classname: Class → String; assocname: Assoc → String; assocclassname: Assoc → String;

Figure 1.8: Logical signature for class diagrams.

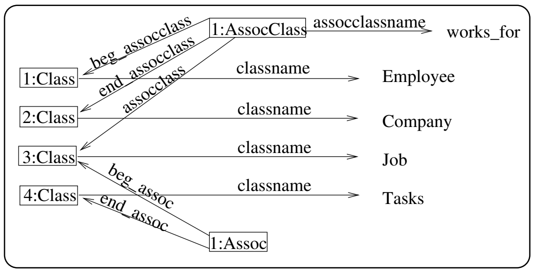


Figure 1.9: Graph structure representation (*abstract syntax*) of the class diagram in Fig. 1.3.

according to the operations. That means, if there is an operation symbol $op : Domain \rightarrow Codomain$ in the visual specification, and a graph item typed over the *Domain* should be inserted, it is required that at least one graph item typed over the *Codomain* must already exist. E.g. according to our example, this fact is given by the insertion of associations, where it is required that at least one class already exists.

Based on these considerations the VL-grammar for class diagrams can be presented as it is done by Figure 1.10. We assume that the start graph (the axiom) is given by the empty graph as illustrated by the LHS of the production *ins_Class*. Both, the productions *ins_Assoc* as well as *ins_AssocClass* illustrate the hierarchy on types. That means, the insertion of an association requires the existence of two classes according to the two operations *beg_assoc* and *end_assoc*. These two classes are put into the LHS of the production. Because it is desirable to have unique class names only, this requirement can be

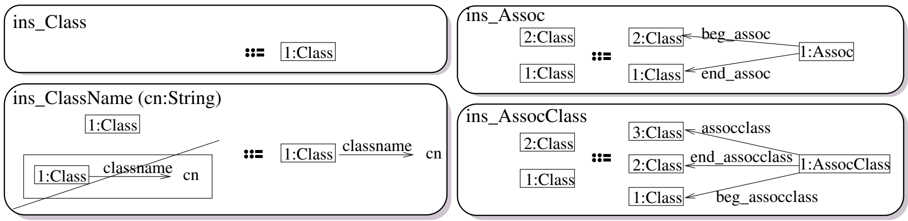


Figure 1.10: Abstract syntax of VL-productions for class diagrams.

expressed by the application condition as given by the canceled item of the *ins_ClassName* production's LHS. Such application conditions (cf. [71,70]) can be used for the adequate definition of the *abstract syntax*. Notice, the productions expressing the insertion of a binary association name as well as a name for an association class are analogous to the *ins_ClassName* production.

The VL-grammar of Figure 1.10 is an unrestricted graph structure grammar with empty start graph. The application of these productions yields the *abstract syntax* of the class diagram presented in Figure 1.3, respectively Figure 1.9:

ins_Class inserts 1:Class,
ins_ClassName(cn=Employee) inserts the string Employee,
ins_Class inserts 2:Class,
ins_ClassName(cn=Company) inserts the string Company,
ins_AssocClass inserts 1:AssocClass, 3:Class,
ins_ClassName(cn=Job) inserts the string Job,
ins_AssocClassName(an=works_for) inserts the string works_for,
ins_Class inserts 4:Class,
ins_ClassName(cn=Tasks) inserts the string Tasks, and
ins_Assoc inserts 1:Assoc.

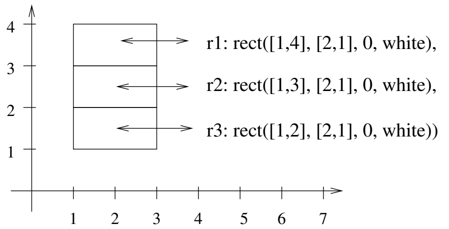
The *abstract syntax* of such class diagrams cannot be defined with a context-free hypergraph grammar with embeddings as presented in the previous subsection. Remember please, that embedded productions must not be applied to a context containing hyperedges which have been created by any previous rewrite step using an embedding production. This restriction is necessary for efficient parsing which is discussed in Subsection 1.5.3. Although it is conceivable in the hypergraph approach to define several association types as shown above the corresponding productions would be embedded productions. I.e., the application of production *ins_Assoc* (for 1:Assoc) is forbidden when the production inserting the association class (for 1:AssocClass) is already applied.

The logical part of a VL will be now extended by graphical attributes. As already mentioned, a visual specification consists of a logical part, a graphical part, suitable layout operations coupling both parts, and further equations. The *concrete syntax* of VLs is given by graphical expressions according to the language GVT [72]. Each visual means is defined by a box with variable parameters for its position and its size. These parameters are replaced by real values when instances are used, e.g., within a rewrite step. A possible algebraic signature for graphics is sketched in Figure 1.11.

```
graphic_Sig =
 sorts   Graphic, Point, Size, FillIntensity, FillColor, TextSize, TextFont, Bool, ...
 opns    /* Constructors */
         rect, roundrect: Point Size FillIntensity FillColor → Graphic;
         line, arrow: Point Size Point Point → Graphic;
         text: Point Size TextSize TextFont String → Graphic;
         box: Point Size ListOfGraphic → Graphic;
          ...
         /* Access Routines */
         refpoint: Graphic → Point;
         width, height: Graphic → Real; xpos, ypos: Point → Real;
         north, south, east, west, northwest, ... : Graphic → Point;
          ...
         /* Constraints */
         overlapping: Point → Point;      inside: Graphic → Graphic;
         begin, end: Point → Point;       equalPosition: Point → Point;
         equalWidth, equalHeight: Graphic → Graphic;
         fixedDistance: Graphic Graphic → Graphic;
          ...
```

Figure 1.11: Signature for graphics.

The concepts for graphical objects, i.e., graphical types, are now explained in more detail: We assume to have an implementation of a fixed graphic algebra G, which is an algebra wrt. graphic_Sig. In our model each graphical object is defined by a surrounding imaginary box with a reference point (refpoint), defined by the upper left corner, and its size (Width and Heigth). The elements of the graphic algebra G describe not only primitive graphical objects, as e.g. given by rect, line, etc. but also complex graphical objects which are defined by box-expressions. For illustration let us regard the graphical expression for a class presented by Figure 1.12. Such box-expressions describe the graphic types according to the *concrete syntax* of a VL's visual means. They will be taken into account for further progress, i.e., they define the default layout of each graph item. The offsets of each including primitive is set relatively to the reference point refpoint of the surrounding box. This reference point is given by a variable, i.e., it can be defined by any position. The sizes (Width, Height) are defined by variables, too. They are instantiated by concrete values when a production is applied. The size of the box depends on the size and relative positions of its parameters.



layout(Class) =
box([4,1], [2,3],
(rect([0,0], [2,1], 0, white),
rect([0,1], [2,1], 0, white),
rect([0,2], [2,1], 0, white))

Figure 1.12: Graphical expression (type) for a class.

On top of box-expressions graphical high-level constraints can be used to describe relations between graphics. Such relations are necessary according to the operations of the VL-alphabet's logical part including the attribution of graph items by data types. These constraint definitions establish equations, the signature for the VL-alphabet is enriched by. The equations for the operations assocclass, beg_assocclass and end_assocclass are presented in Figure 1.13.

∀ ac:AssocClass, class:Class;		
assocclass(ac) = class	$\Longrightarrow$	begin (layout(ac)) = layout(class).north;
∀ ac:AssocClass, class:Class;		
beg_assocclass(ac) = class	$\Longrightarrow$	begin (layout(ac)) = layout(class).east;
∀ ac:AssocClass, class:Class;		
end_assocclass(ac) = class	$\Longrightarrow$	end (layout(ac)) = layout(class).west;

Figure 1.13: Equations for the operations assocclass, beg_assocclass and end_assocclass.

According to the discussion about graph structures with unary operations only, the graphic algebra consists of n-ary operations. The same is true for the data type attributes. This is the reason why we do not distinguish between SRG and ASG. In our approach the SRG is the ASG with additional attributes, namely the graphics and data types. Two graph structures (LHS, RHS) are illustrated by the production describing the insertion of an association in Figure 1.14. The upper part of this figure presents the *abstract syntax* whereas the lower part shows the *concrete syntax* with the graphic's attachment points after rendering. These parts are coupled by layout operations visualized by dashed arrows. Note, both parts represent two algebras, one for the LHS and one for the RHS of the production. Additionally, because they are algebras, the equations as illustrated by Figure 1.13 are satisfied.

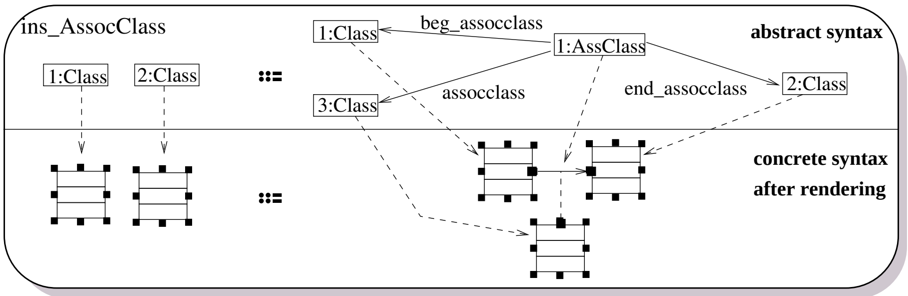


Figure 1.14: Production describing the insertion of an association class.

The application of VL-productions is done via graph transformation as described in Chapter ??. Additionally the attribute evaluation according to the data types is described there. The corresponding graphical attributes and the graphical constraints between them are handled in dependency of the evaluation for the logical part, i.e., the *abstract syntax*.

1.4 Generating Visual Language Editors

Visual language editors are needed when visual sentences have to be created and manipulated on the screen. Given a visual language, special purpose editors have to be provided since general graphics editors do not assist the user in creating correct visual sentences. This section discusses approaches how to generate visual language editors from a specification based on graph transformation.

1.4.1 Visual Editing Modes

Visual language editors may offer two different editing modes: *syntax-directed editing* and *free-hand editing*. Syntax-directed editors provide a set of editing operations. Each of these operations is geared to modify the meaning of the visual sentence in a certain aspect. Referencing Figure 1.1, editing operations modify the visual sentence's ASG; its SRG and thus its visual appearance are updated accordingly.

Syntax-directed editing is supported by two approaches: the *grammar-based* one uses an explicit grammar for defining a visual language and its ASGs independent of any editor. Editor operations modify directly a sentence's derivation history and, thereby, indirectly the generated (hyper)graph. A typical example is LOGGIE which permits generation of graphical, language-specific programming environments [73]. It uses context-free attribute grammars as the key data structure for specifying the language and derivation trees for representing the derivation history. The *transformation-based* approach omits an explicit grammar for specifying a visual language. Editing operations are defined by graph transformations on the ASG which also define the visual language at the same time. The visual language is thus defined by its editor. As usual such editors do not keep track of their visual sentences' derivation histories. Göttler's PAGG system (cf. Chapter ??) 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. Other examples of the transformation-based approach are PROGRES (cf. Chapter ?? and Subsection 1.6.1) and GenGEd (cf. Subsection 1.4.2).

The advantage of the grammar-based approach is to be based on a 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 graph transformations which can be checked against this grammar in order not to change the language. On the other hand, the transformation-based approach has the advantage of being able to easily model intermediate, temporally incorrect visual sentences in terms of the syntax of the visual language, 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 (e.g., PAGG and LOGGIE) requires the user to select an editing operation and some pictorial objects. As an effect 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 discussed in more detail in Subsection 1.4.3.

Visual language editors providing free-hand editing are low-level graphics editors directly manipulating the visual sentence. Referencing Figure 1.1, the editor modifies a visual sentence's SRG. The graphics editor becomes a visual language editor by manipulating only pictorial objects used by the visual language and by offering a parser. A parser is necessary for checking the correctness of visual sentences, for recreating the SRG's derivation history, and for updating the ASG accordingly. The advantage of free-hand editing over syntax-directed editing is to define a visual language 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. However, 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 parser and is thus restricted to visual languages and graph grammars which offer efficient parsers.

This section discusses syntax-directed and free-hand editors based on graph transformation only. Other visual language editors like the ones based on constraint multiset grammars [74] or description logic [75] mainly offer free-hand editing, but lack syntax-directed editing. The latter requires an explicit representation of objects and their relations which is the strength of the graph based approach.

A typical example of systems generating visual language editors providing free-hand editing and being based on some kind of graph grammar is VLCC by Costagliola et al. [76]. VLCC uses positional grammars [8] as a grammar formalism which is related to graph grammars in special cases and that provides efficient parsing. VLCC comes with a visual specification tool which is used to specify a set of pictorial objects that can be arbitrarily manipulated and connected by the generated low-level graphics editor. Syntax rules, which are also specified using this tool, use a notation quite similar to the concrete representation of VL sentences.

Similar to VLCC, GenGEd is a system for generating visual language editors which comes with a visual specification tool that allows to specify the VL's pictorial objects as well as its syntax rules in the same notation as used in the generated editor. Generated editors offer syntax-directed editing using the transformation-based approach and the select&apply-paradigm. GenGEd is discussed in more detail in the following Subsection 1.4.2. Subsection 1.4.3 then presents DiaGen as an example for an approach that supports syntax-directed editing using the grammar-based approach as well as free-hand editing. Syntax-directed editing is not restricted to the select&apply paradigm by permitting to specify complex interaction patterns. Visual parsers used by free-hand editing are described in Section 1.5. GenGEd and DiaGen disregard the distinction between ASGs and SRGs to a certain extent.

1.4.2 GenGEd

The main emphasis of the GenGEd-environment is to support the visual definition of a visual language and the generation of a visual language editor, i.e., a graphical editor for the specified VL.
As described in Subsection 1.3.4, a VL consists of a VL-alphabet, i.e., a visual specification, and VL-rules. According to this division the GenGEd environment as illustrated by Figure 1.15 is built up: The alphabet editor supports the definition of a VL-alphabet which is the input of the rule editor. Based on the VL-alphabet so-called alphabet rules are generated automatically. These alphabet rules usually are not the intended VL-rules. They are only used as the edit commands of the syntax-directed rule editor allowing the definition of VL-rules. The VL-rules defined using the rule editor are then the edit commands of the intended graphical editor, supporting syntax-directed manipulation of sentences over the defined VL.

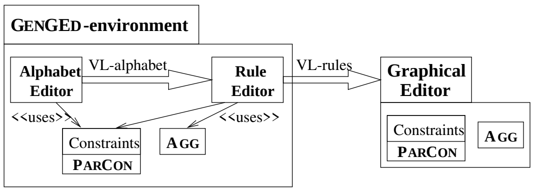


Figure 1.15: The GenGEd-environment.

In the following, the GenGEd-environment is explained along our running example of class diagrams as discussed in Subsection 1.3.4. We start with the constraint component because graphical constraints are used intensively defining a VL, whereas the integrated graph transformation system Agg is described in Chapter ??. Then, we present the alphabet editor supporting the definition of the VL-alphabet. Afterwards, the algorithm for the automatic generation of alphabet rules is discussed, followed by the presentation of the rule editor. At the end of this section, the generated graphical editor is sketched

allowing the syntax-directed manipulation of visual sentences which is done via the select&apply-paradigm.
Each editor consists of a frame with several top-down menus. Editor specific functions are put into the left side. The right area is used by each editor for its specific tasks. We have to mention, that for sake of space we cannot discuss the whole functionality of the environment, i.e., each editor.

Constraints The constraint component available within GenGEd offers a high-level constraint language. Each high-level constraint (HLC) is defined by suitable low-level constraints (LLCs) of the integrated constraint solver ParCon (cf. [77]).
The solver is used as a server. It is implemented in Objective C in contrast to all other components discussed in this subsection which are implemented in Java. The predefined HLCs are collected within a library which can be seen as a superset of the used ParCon constraint language. These predefined HLCs may not be sufficient enough to support the definition of all needed graphical symbols and their connections. Therefore, a user can extend the library by own constraints expressed in the HLC-language.

Alphabet Editor For the visual definition of a VL-alphabet two components are available: the *Graphical Symbol Editor* (GSE) and the *Connecting Editor* (CE). The GSE supports the drawing of graphical objects, i.e., the visual means of a VL, and the determination of implemented data type attributes. For each graphical object the user is asked to insert a name for it. Each name establishes one of the visual specification's sort symbols of the graph part, resp. data part, whereas each graphical object defines a graphic of the graphical part. Furthermore, naming the graphics establishes implicitly the layout operations. These layout operations couple the sort symbols with its graphical counterpart according to the *translation* in Figure 1.2 (cf. Page 14). The graphical objects connected with a name are further-on called graphical symbols. When the graphical symbols are determined, the CE can be used to determine the operations for the visual specification's logical part and the graphical constraints which define the equations.
The GSE is similar to common graphical editors. The main difference is given by graphical constraints, which are to be used for defining one graphical object: Each graphical object consists either of a primitive graphical object as given by a rectangle, circle, arrow, etc. or of several primitive graphical objects which are connected by graphical constraints yielding a box-expression as presented in Figure 1.12. According to our example, the graphical objects are determined

for a class (*Class*), a binary association (*Assoc*) and the association class (*AssocClass*).
The visual means for the *String* data type as used within our example for the class name, the association name and the association class name is given by *Text*. Note, the default properties of the graphical object *Text* can be changed. However, if different properties for names are desired, it is necessary to define several *String* data types. E.g. if all class names should be written in italics and all association names should be depicted in the normal style, then a user chooses *Text* twice and defines the desired properties.
The graphical symbols defined using the GSE are available within the sequently toggled CE. Their images are put onto buttons with their type names (sort symbols) below. They can be chosen and put onto the drawing area as illustrated in Figure 1.16, where the constraints according to the operations *beg_assocclass*, *end_assocclass* and *assocclass* are to be defined. Each operation definition is controlled by the CE, i.e., constraints can be defined and additionally the user has to define *Domain* and *Codomain* of the operation according to the logical part of a visual specification. The latter one is supported by a dialogue after pressing the *New Operation* button. If the operation and at least one constraint is defined this is visible by a connecting line as illustrated in Figure 1.16. When all operations, respectively constraints, are defined the visual specification as discussed in Subsection 1.3.4 is established.

Generated Alphabet Rules The VL-alphabet, i.e., the visual specification, is the basis for the automatical generation of alphabet rules. These alphabet rules are used as the edit commands of the rule editor. Notice, that an alphabet rule may be already a VL-rule. Nevertheless, for most VL-alphabets not all generated rules express the intended VL-rules.
The alphabet rules are built up according to the graph part's sort symbols coupled with its graphic, i.e., for each graphical symbol an insertion rule is generated [c]: The LHS of such a rule is initially empty and the RHS comprises the graphical symbol. This generated rule is modified if an operation exists whose *Domain* is the sort of the symbol which should be inserted. Then, the LHS and the RHS comprise additionally the graphical symbol of the operation's *Codomain*. Furthermore, the RHS includes the graphical symbol for the operation's *Domain*. This handling is due to the hierarchy of operations and the algebraic approach as discussed in Subsection 1.3.4. That means, each side of a rule is a visual sentence and therefore an algebra wrt. the visual specification. Because the graph part is total all operations must be defined.

[c] Not only insertion rules are generated but also rules describing deletion, moving, etc.

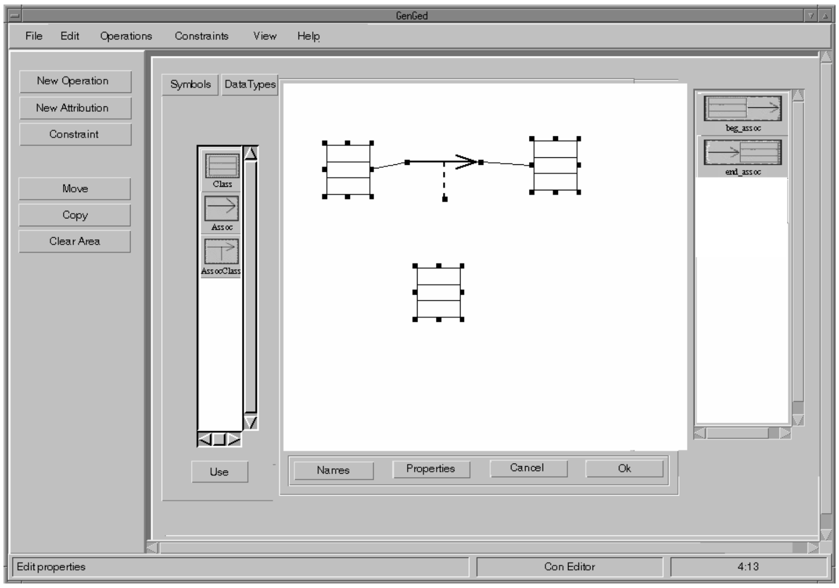


Figure 1.16: The GenGEd-connecting editor.

The upper part of Figure 1.17 presents the generated alphabet rule for the insertion of an *AssocClass*. Following the algorithm, first the LHS of this rule is empty, whereas the RHS includes the graphical symbol for *AssocClass*. The second step is to examine the operations. Three operations (*assocclass*, *beg_assocclass*, *end_assocclass*) are defined in the visual specification where *AssocClass* is the *Domain* of (cf. Figure 1.8). For each operation's *Codomain* a graphical symbol is generated, in this case three graphical symbols for *Class*. These three classes are now put into the LHS as well as into the RHS. The RHS includes additionally the graphical symbol for *AssocClass*. Then, the operations are defined according to the logical part of a visual sentence, and the graphical constraints, i.e., the equations, are satisfied.
Attribution rules are generated automatically, too. According to our example several attribution operations (and constraints) are defined, that are *classname*, *assocname*, and *assocclassname*, all given by the *String* data type. For each attribution an insertion rule is generated [d] where the attribute's value is denoted by the rule parameter. The algorithm for the generation of attribution rules is the same as described above. I.e., for each attribution a rule is gener-

[d] Not only insertion rules are generated but also rules describing deletion, moving, etc.

ated where the graphical symbol of the attribution's *Codomain* is included in the LHS and the RHS. The RHS includes in addition the rule parameter.

Rule Editor The rule editor (RE) illustrated by Figure 1.17 supports the definition of VL-rules. The edit commands of the rule editor are alphabet rules automatically generated from the defined VL-alphabet. Applying rules, the integrated graph transformation machine AGG (cf. Chapter ??) is responsible for the logical part of a VL-sentence, and the constraint solver PARCON completes the graphical layout wrt. the graphical constraints.

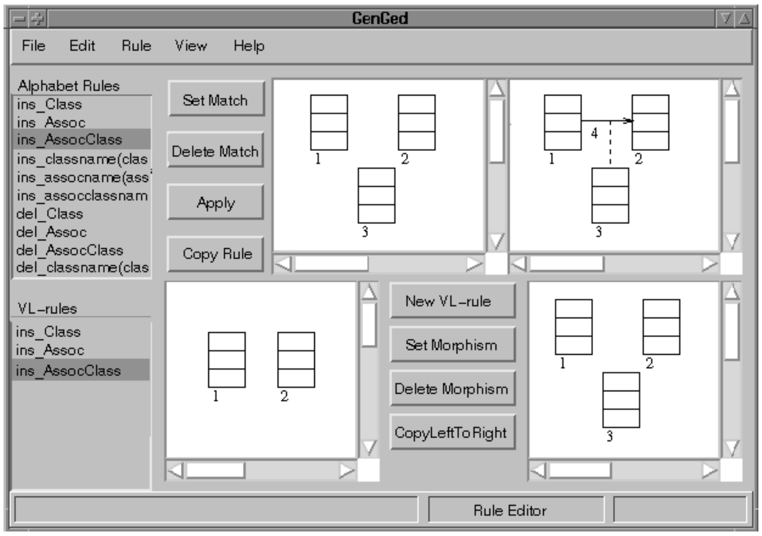


Figure 1.17: The GENGED-rule editor.

The rule editor comprises several components: a rule viewer and a rule builder. A rule manager is the control component of the rule editor. It is the component responsible for all rules, i.e., for the generated alphabet and VL-rules, as well as for the coordination of the match morphisms. The rule viewer (see the upper part of Figure 1.17) visualizes a selected rule (see rule names in the upper left corner of Figure 1.17). These rules cannot be modified. The rule builder (see lower part of Figure 1.17) consists of two host graphs[e], one for the LHS and one for the RHS of a VL-rule. Both sides can be edited in a syntax-directed mode by applying rules.

[e]The user only sees the graphics.

For rule application it is necessary to define the match morphism. I.e., after pressing the *Set Match* button, each graphical symbol of the rule viewer's LHS must be mapped onto exactly one component of the rule builder's LHS or RHS. Then, the visualized rule can be applied by pressing the *Apply* button. Applying a rule means, that the rule's LHS will be deleted in the host graph (graphic) and substituted by the RHS. If a VL-rule is complete the components of both sides must be related by a mapping (morphism). Such a mapping will be defined implicitely if a VL-rules RHS is constructed as a copy of its LHS: if first the VL-rule's LHS is edited, then, pressing the *CopyLeftToRight* button yields a copy of the LHS in the VL-rule's RHS including the mapping. Both, the VL-rule's LHS and RHS may be modified afterwards.

Let us illustrate the construction of the VL-rule presented by Figure 1.14 in Subsection 1.3.4 which expresses the proper semantical effect of inserting an *AssocClass* in contrast to the generated alphabet rule visualized in the upper part of Figure 1.17. Starting with empty LHS and RHS of the VL-rule, the alphabet rule *ins_class* is applied twice to the LHS. Pressing the *CopyLeftToRight* button yields both classes in the RHS and defines implicitly the necessarily mapping. Thereafter, the rule *ins_class* is applied again on the RHS yielding the third class in Figure 1.17. This RHS is the basis to apply the alphabet rule *ins_AssocClass* as shown by the rule viewer in Figure 1.17. This leads to the desired VL-rule.

Some automatically generated rules may already describe a VL-rule. In this case it is possible, to copy the generated rule into the list of VL-rules. For our example this was the case for the rule *ins_Class* and *ins_Assoc*.

Generated Graphical Editor A syntax-directed graphical editor (GE) as illustrated in Figure 1.18 is generated from the VL-rules, and supports the editing of visual sentences by application of these rules. The rule application of a GenGEd-generated graphical editor is performed by the graph transformation machine Agg. Agg is working on the logical graph structure. The constraint server ParCon solves the corresponding graphical constraints. In contrast to the rule editor, a graphical editor offers only one host graph. This host graph is the visual sentence VL-rules can be applied to. Apart from that, a graphical editor has the same behaviour as the rule editor.

VL-rules can be applied by mapping the graphical symbols of the VL-rule's LHS (visualized by the RuleViewer) onto type consistent graphical symbols of the working area. These objects are replaced by the corresponding objects of the VL-rule's RHS. Automatic layout adjustment of the adjacent graphical symbols is enforced if one graphical symbol is moved.

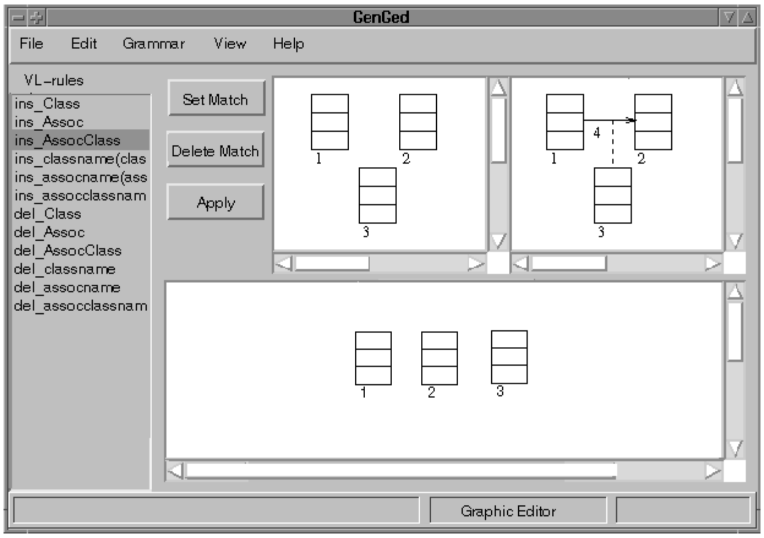


Figure 1.18: The graphic editor for class diagrams.

1.4.3 DiaGen

DiaGen is a tool consisting of a framework together with a generator that is used to create graphical editors from a formal specification. Editors employ hypergraphs as internal model of visual sentences and hypergraph grammars to define the VL's syntax. This subsection gives a brief survey of DiaGen and begins with DiaGen's grammar-based syntax-directed editing mode. Context-free VLs are considered first. DiaGen's extension to syntax-directed editing of non-context-free VLs are briefly described next. Later on, this subsection continues with DiaGen's free-hand editing mode which is made possible by visual parsers that check the correctness of visual sentences. As a running example of this subsection, we will use Nassi-Shneiderman diagrams (NSDs) as described in Section 1.3.

DiaGen uses the traditional compiler compiler approach to create visual editors from a (textual) specification. In the beginning, DiaGen was designed to generate syntax-directed editors for context-free VLs only. Editors of this DiaGen version consist of generated C++ code together with a C++-framework based on X Windows and Motif.

Figure 1.19 shows screenshots of a session with an NSD editor that has been generated by this DiaGen version. The left screenshot shows two NSDs on

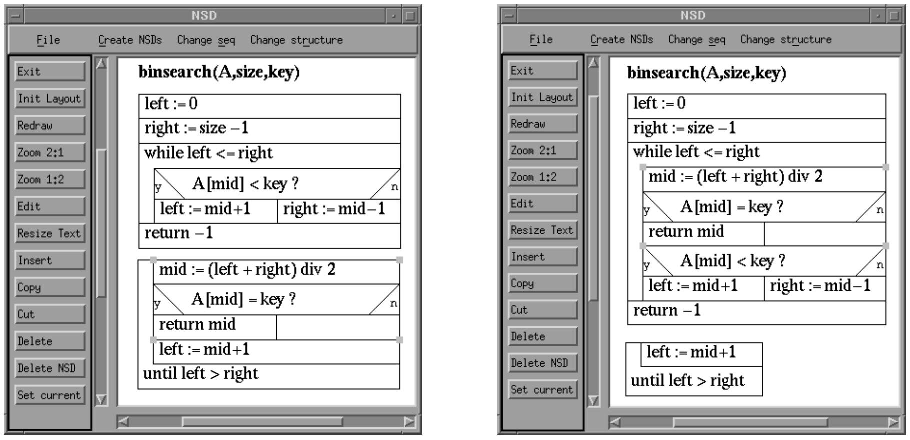


Figure 1.19: Syntax directed editing of Nassi-Shneiderman diagrams.

the editor canvas where a sequence of statements of the lower NSD is selected (marked by the small gray handles at the corners of the selected statements). The right screenshot shows the editor state after the selected statement sequence of the left screenshot has been moved into the upper NSD, which is again depicted as a selected statement sequence. The user can carry out this editing operation in a direct manipulation manner by clicking on the selected statement sequence in the left screenshot and dragging the sequence—an outline follows the mouse movement—to the new position, which is the line below the while condition, where the mouse button is released. The editor then updates both diagrams, i.e., it carries out the structural modification imposed by the user and then updates the diagram layout automatically.
This example illustrates the main features of syntax-directed editors generated by DiaGen:

- Editors allow to create syntactically correct diagrams only. Each editing operation preserves this property of any diagram edited on the screen.
- Editing operations are implicitly selected and carried out. Instead of the traditional select&apply paradigm, the user can interact in a very natural way with the editor by selecting diagram parts and dragging them to new positions in a direct manipulation manner.
- Editors are provided with a syntactic grouping mechanism, i.e., a collection of selected diagram components are considered as one component if

they form a syntactic whole which, of course, depends on the VL. For NSDs, statement sequences may be grouped.

- Editors have an automatic layout mechanism which computes a valid diagram layout after structural modifications, but also after explicit layout modifications by the user (e.g., when the user repositions some diagram components).

Each of these features have to be described in the editor specification. The following presents a brief survey of their concepts.

Diagrams are internally represented by hypergraphs as described in Section 1.3: each pictorial object is modeled by a hyperedge (for a hypergraph representation of NSDs see Figure 1.4); nodes and hyperedges carry attributes which define the diagram layout. An attributed context-free hypergraph grammar defines the VL syntax in terms of its hypergraph models. Figure 1.5 shows the grammar for NSDs which also contains constraints on attributes which define valid layouts. The main data structure representing a diagram, however, is the derivation tree of its hypergraph representation. It is this data structure that is primarily modified by editing operations; the visual appearance of a diagram is the image when mapping the hypergraph specified by its derivation tree to the screen. The mapping is defined by the editor specification, which contains descriptions how each hyperedge type has to be visualized on the screen, and by attribute values defining the layout, i.e, position, geometry etc., of each hyperedge image. A constraint solver makes use of the derivation tree and computes and recomputes valid attribute values according to the constraints of the attributed hypergraph grammar after each structural modification to the derivation tree and after each layout modification by the user [64].

Syntax-directed editing operations are internally transformations on these derivation trees. They have to be defined in the specification used by the generator. However, they are not made visible to the editor user. Their application is controlled by *interaction automata* which are part of the specification, too [78]. Interaction automata have been motivated by Statecharts [79] and are specialized for describing interactions. Each interaction automaton has an initial state and a set of final states. An automaton recognizes an interaction pattern, i.e., a sequence of external or internal events (e.g., mouse clicks or mouse movements respectively satisfaction of some predicates on the editor's state), if there is a path from the initial to a final state such that the transitions along the path are subsequently carried out. Transitions are labeled by conditions and actions. Conditions may be references to external events, references to predicates written as (C++) procedures representing internal events, or references to other interaction automata. A transition is carried out when the state at the transi-

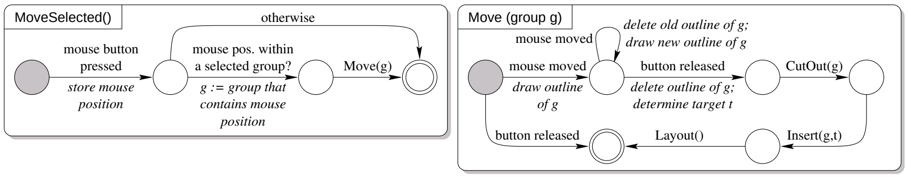


Figure 1.20: Two of the interaction automata describing the interaction depicted in Figure 1.19. Actions (shown in italics) and conditions are presented using an informal syntax. Initial states are marked by gray fill pattern, final states by double circles.

tion source is active and the associated condition is satisfied. This means that either an appropriate external event has happened, or the predicate evaluates to true, or the referenced automaton has recognized an interaction pattern. By using interaction automata as conditions within other interaction automata, one obtains a hierarchical description of interaction patterns which the editor reacts on. Additionally to conditions, transitions are labeled by actions, e.g., transformations on derivation trees, which are carried out together with the transitions. Figure 1.20 shows the a simplified version of interaction automata responsible for the editing operation depicted in Figure 1.19, i.e., detecting a mouse click over the selected statement sequence in the left screenshot, dragging an outline of the sequence following the mouse movement, and inserting the sequence at the new position.

As the example also shows, editors employ a syntactic grouping mechanism. When diagram components get selected, the corresponding hyperedges are labeled as selected. A transformation system which is described in the editor specification, too, then replaces appropriately labeled subtrees of derivation trees by special hyperedges that represent the selected group. This whole process is again controlled by interaction automata. For NSDs, subtrees that represent sequences of selected statements are replaced by such group hyperedges. Finally, editing operations are also specified as transformation rules on derivation trees that may contain group hyperedges as replacement of a whole (selected) subtree. Interaction automata take care of selecting the right rules which have to be applied. Figure 1.20 first applies a rule *CutOut(g)* which removes the selected group (g) from its original position. *Insert(g,t)* then inserts this group at the target position (t).

Since context-freeness turned out to be too restrictive, DiaGen has been extended to non-context-free VLs in a limited way that preserves the mechanisms already available for editing of context-free VLs: each non-context-free diagram has to consist of a context-free skeleton, which is represented by a derivation

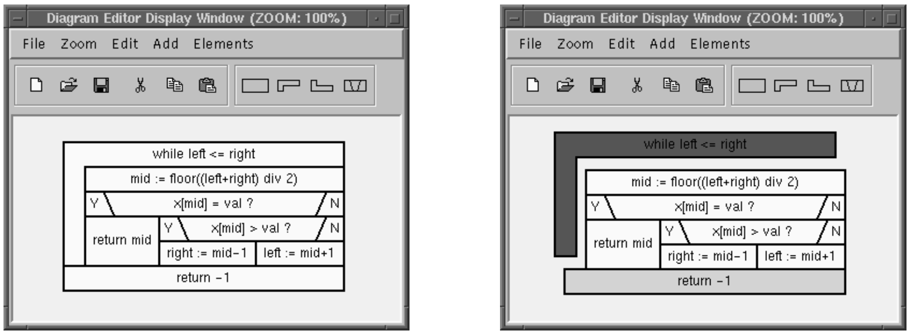


Figure 1.21: Free-hand editing of Nassi-Shneiderman diagrams.

tree, and a layer of additional hyperedges. The editor specification then has to contain a context-free hypergraph grammar for the context-free skeletons, but also additional transformation rules that add and remove these additional hyperedges. Again, interaction automata control editing operations that modify derivation trees or the additional hypergraph layer.

However, syntax-directed editing as described so far turned out to be quite restrictive for the editor user. The user had to use predefined editing operations. Furthermore, visual sentences have to be correct all the time. Intermediate syntactically incorrect editing states that are sometimes convenient are prohibited by this approach. Therefore, the syntax-directed editing mode has been combined with a free-hand editing mode supported by visual parsers that check whether diagrams are syntactically correct and recreate their derivation DAG (c.f. Section 1.3). Actually, a new version of DiaGen (DiaGen II) for this combined approach has been implemented using Java instead of C++ since the previous DiaGen version happened to be less portable.

Figure 1.21 shows two screenshots of a session of an NSD editor created by DiaGen II that also supports free-hand editing. The left screenshot shows a syntactically correct diagram which is now modified by the user who moves the top-most component representing the while condition to a new position. The visual parser recognizes that the resulting diagram is not syntactically correct. It consists of a diagram component that does not belong to any correct subdiagram (the while condition) and two correct subdiagrams[f] (the remaining two rectangular subdiagrams). The following Section 1.5 explains visual parsers that are used by DiaGen for this task.

[f] On the computer screen, the color red is used to mark such incorrect diagram parts; bright colors like white and yellow characterize correct subdiagrams.

For syntax-directed editing, editing operations are carried out on the hypergraph model; the visual appearance of the diagram is an image when mapping the hypergraph to the screen. For free-hand editing, however, the situation is different: a low-level graphic editor allows the user to modify pictorial objects directly. The hypergraph has to be kept up-to-date accordingly. This task is simplified by the use of the hypergraph model as described in Subsection 1.3.3: hyperedge tentacles represent attaching areas of the corresponding pictorial object. Pictorial objects that are related by their attachment areas are modeled by the corresponding hyperedge tentacles being connected to common nodes in the hypergraph model. DiaGen creates a low-level graphic editor that allows to manipulate pictorial objects of the specified VL; the internal hypergraph model is modified accordingly by watching the pictorial objects' attaching areas [80].
Free-hand editing with parser support relaxes the need to specify a full set of transformations on diagrams for syntax-directed editing since free-hand editing can be used for (yet) unspecified diagram operations. Therefore, this editing mode enhances usability of editors and also makes rapid prototyping of diagram editors possible because—as an extreme case—specification of diagram operations can be omitted completely.

1.5 Generating Visual Language Parsers

The previous section pointed out that visual editors which represent visual sentences by graphs and allow for free-hand editing have to make use of a graph parser. It is the parser's task to distinguish syntactically correct graphs—and therefore visual sentences—from incorrect ones and to translate a graph into its syntactic structure. The syntactic structure may then be used for further processing, e.g., to create an abstract syntax graph (cf. Figure 1.1). This section discusses how visual languages defined by graph grammars may be parsed and how visual editors can make use of parsing information.

1.5.1 The Visual Language Parsing Problem

A graph parser tries to decide the membership problem for graph grammars: for a given graph and a graph grammar, it tries to find a derivation sequence from the grammar's axiom to the given graph. This problem is undecidable for general grammars [81]; hence, restricted classes of graph grammars are examined for which this problem is more or less efficiently decidable. In the following we will consider context-free hypergraph grammars, context-free ones with embeddings, as well as layered graph grammars, which are introduced in

Section 1.5.4. In general, the derivation sequence is the only way to represent a graph's syntactic structure if it is a member of the grammar's language. For these (hyper)graph grammar classes, however, derivation trees resp. DAGs (cf. Section 1.3) can be used for a more structured syntax representation.

Visual parsers, i.e., graph parsers used for parsing graph representations of visual sentences, have to provide additional features. A simple graph parser just classifying an assembly of pictorial elements as correct or incorrect is unsatisfying for the user. Similar to programming language compilers, the visual parser has to identify the errors of the visual sentence, i.e, it has to find a minimal subset of pictorial objects that is responsible for being an incorrect visual sentence and the maximal subsets being correct visual sentences. Low-level graphics editors may then give feedback and guide the user in correcting errors. Furthermore, visual parsers should work incrementally in order to keep informations gathered in earlier parser runs. This is especially important in the situation described in Section 1.3 where the parser is used to keep an abstract representation (ASG) of the visual sentence up-to-date with its concrete representation (SRG). Since it is this ASG that is used for further processing of the visual sentence, it is necessary to maintain those parts of an ASG that have not been modified by editing operations. Parsers always starting from scratch would rebuild the abstract representation anew after each editing operation. Parsers working incrementally, however, can meet this requirement.

The next Subsections 1.5.2 and 1.5.3 outline visual parsers for context-free hypergraph grammars and context-free ones with embeddings which run offline as well as incrementally and which offer error feedback. These hypergraph grammar classes already cover a wide range of practical problems. Finally, in Subsection 1.5.4, a parser for *layered graph grammars*, an even more general class of graph grammars is described. However, graph and hypergraph grammars are not the only grammar type for which visual language parsers have been proposed. Table 1.1 summarizes parsing approaches for *constraint multiset grammars*, *relational grammars*, *picture grammars* (consisting of picture layout grammars and positional grammars), and *graph* as well as *hypergraph grammars* including the ones discussed in more detail in the next subsections. The table answers the following questions for each of the parsing approaches:

- How does a production's *LHS* and *RHS* look like?
- Does the formalism allow references to additional *context elements*, which have to be present, but remain unmodified when applying a production?
- Does the grammar type of this parsing approach have more or less complex *embedding rules*, which establish connections between new elements (created by a production) and the surrounding structure?

Type	Authors	LHS	RHS	Context elements	Embedding rules	Add. parsing restrictions	Space/Time complexity
Graph	Kaul [82,83]	one non-terminal	directed nonempty graph	no	yes	implicitly def. vertex ordering by edges	linear
	Rozenberg/ Welzl [84]	one non-terminal	directed graph without nonterminal neighbours	no	yes	(bounded degree, connected graph)	exponential (polynomial)
	Rekers/ Schürr [85]	directed graph	directed connected, nonempty graph	directed graph	extended context	global layering condition for grammar	exponential
	Zhang/ Zhang [86]	directed graph	directed connected, nonempty graph	directed graph	extended context	confluent grammar	polynomial
Hyper-graph	Bunke/ Haller [87]	one non-terminal	nonempty plex structure	no	fixed set of connection points	no	exponential
	Minas [88]	one non-terminal	hypergraph	no	fixed set of connection points	(k-separability [62])	exponential (polynomial)
		one non-terminal	hypergraph	hypergraph	fixed set of connection points	embedding restrictions (k-sep. [62])	exponential (polynomial)
relational	Wittenburg [56]	one non-terminal	nonempty relational structure	no	yes	explicitly def. vertex ordering by relations	exponential
	Ferrucci et al. [89]	one non-terminal	rel. structure without nonterminal neighbours	no	yes	(bounded degree, connected structure)	exponential (polynomial)
Picture	Golin [4]	one non-terminal	one or two (non) terminals	one nonterminal	implicit	finite set of possible attribute values	polynomial
	Costagliola et al. [90]	one non-terminal	positional sentential form	no	implicit or fixed set of connection points	LR-condition	linear
Multiset	Marriott [6]	one non-terminal	arbitrary nonempty multiset	(yes)	implicit	acyclic grammar, no context elements	exponential

Table 1.1: Visual language parsing approaches.

- Are there *additional restrictions* for the set of productions or for the form of the parsed structures?
- Is the time and space complexity of this parsing approach linear, polynomial, or even exponential with respect to the size of the input structure?

The question of *embedding rules* deals with a specific aspect of graph grammars as opposed to string grammars since it is concerned with the problem of how new elements (created by a production) are connected to its surrounding structure. Connections are either *implicitly* established as with picture layout or constraint multiset grammars since relations between objects are only implicitly defined. Or the LHS and RHS of a productions are extended by a common *context*, which is not modified when applying the production. Or there are complex *embedding rules* that describe the redirection of arbitrary sets of relationships from replaced objects to replacing objects.
The precedence graph grammar parser of Kaul [82,83] is an attempt to generalize the idea of parsing based on operator precedence and has a linear time and space complexity. The parsing process is a kind of handle rewriting, where graph handles (subgraphs of the input graph) are identified by analysing vertex and edge labels of their direct context. Unfortunately, this approach works only for a very restricted class of graph languages.
Other approaches use Earley-style parsers [91]: Bunke and Haller [87] use plex grammars, which are essentially context-free hypergraph grammars with restricted forms of embedding rules. Any nonterminal has only a fixed number of connection points to its context. Wittenburg's parser [56] uses dotted rules to organize the parsing process for relational grammars, but without presenting any heuristics how to select "good" dotted rules. Furthermore, it is restricted to the case of relational structures, where relationships of the same type define partial orders. Finally, the approach of Ferrucci et al. [89] with so-called 1NS-RG grammars is a translation of the graph grammar approach of Rozenberg/Welzl [84] into the terminology of relational languages. In this approach RHSs of productions may not contain nonterminals as neighbours, thereby guaranteeing local confluence of graph rewriting (parsing) steps. Furthermore, polynomial complexity is guaranteed as long as generated graphs are connected and an upper boundary for the number of adjacent edges at a single vertex is known.
Golin's and Marriott's parsers are based on bottom-up parsing similar to the parsers by Tomita [92] and Cocke-Younger-Kasami [93,94] for context-free string grammars. Marriott's constraint multiset grammars [6] furthermore allow rather general forms of context elements (preserved common elements of LHS and RHS). Golin's parsing algorithm for picture layout grammars [4]

supports terminal context elements, too, but has a main focus on productions with one nonterminal on the LHS and at most two nonterminals on the RHS with predefined spatial relationships between them.
Minas's parsers [88] for hypergraph grammars are also inspired by the Cocke-Younger-Kasami parser. The first one is restricted to purely context-free hypergraph grammars, the second allows to parse context-free grammars with embeddings. Rekers and Schürr's parser [85] for layered graph grammars is able to deal with nearly unrestricted graph grammars. The only restriction is a *layering restriction* which ensures that parsing is alway terminating. Minas's as well as Rekers and Schürr's parsers are discussed in more detail in the following subsections.
Zhang and Zhang [86] have recently proposed a simple parser for a further restricted kind of layered graph grammars. They reverse each grammar production and thus obtain a new graph transformation system. As long as the reverse graph transformation system is confluent, a graph can be parsed by reducing it to an initial graph.
Finally, Costagliola et al. have proposed a parser for *positional grammars* [90] which basically extends string grammars by 2D-relationships. Their parser is based on LR-parsers used for context-free string grammars.
However, with the exception of the two parsing algorithms for hypergraph grammars which will be outlined in the next subsections, none of these parsers supports incremental parsing which is necessary for maintaining as much information from earlier parser runs as possible. Parsing of partially correct structures is only possible with the parsers outlined in the next subsections as well as with Bunke/Haller's and Wittenburg's parser.

1.5.2 Parsing for Context-Free Hypergraph Grammars

The idea of a parser for context-free hypergraph grammars (cf. Section 1.3.1) has been inspired by the Cocke-Younger-Kasami parser (CYK) [93,94] for context-free string grammars. The original CYK parser can be used with any context-free grammar that has Chomsky normal-form (CNF), i.e., there are only productions with the RHS consisting of either exactly one terminal or exactly two nonterminal symbols. This is no restriction since every context-free grammar can be transformed into CNF.[g] The special case of the empty word in the grammar's language is omitted here. CNF for context-free hypergraph grammars is similarly defined: the RHS of each production consists of a hypergraph with either exactly one terminal hyperedge (*terminal production*) or

[g] Table 1.1 does not mention the restriction to CNF grammars because every context-free grammar can be transformed into CNF.

exactly two nonterminal hyperedges (*nonterminal production*).
The hypergraph parser constructs syntactic information bottom-up similar to the original CYK parser: it starts with the terminal hyperedges and creates all nonterminal hyperedges that can be derived using terminal productions. Thereafter, the parser creates additional hyperedges that are derivable to hyperedges created previously. Derivation trees for syntactically correct subgraphs are inherently constructed when storing the directly derivable hyperedges for each nonterminal hyperedge.
An efficient organization of the bottom-up construction of nonterminal hyperedges is to successively create sets S_i ($i = 1, 2, \ldots, n$ with n the number of terminal hyperedges) of nonterminal hyperedges that can be derived to a terminal subgraph with exactly i hyperedges.[h] Thus S_1 has to be filled with hyperedges derivable with terminal productions. Then the sets $S_2, S_3, \ldots, S_n$ are successively created. A new nonterminal hyperedge e together with references to its visited nodes is inserted into S_k if there exist an i ($1 \leq i < k$) and hyperedges $e_1 \in S_i$ and $e_2 \in S_{k-i}$ such that

- e can be derived to the hypergraph consisting of e_1 and e_2 (together with the corresponding nodes) using a nonterminal production.
- the terminal subgraphs of e_1 and e_2 have disjoint hyperedge sets.

After termination, the sets S_i of hyperedges together with references to their derived child hyperedges contain the complete syntactic information of the hypergraph. The entire hypergraph is syntactically correct if S_n contains a hyperedge labeled with the grammar's axiom (*axiom hyperedge*). The grammar is ambiguous if more than one of such hyperedges is contained. In the following, we assume non-ambiguous grammars.
The parsing algorithm outlined above is a refinement of the parsing procedure presented in [62]. The algorithm makes use of the grammar's CNF and an explicit organization of parsing results in its structure of sets S_i which make the algorithm well-suited for its application with low-level graphics editors. The parsing algorithm has to have exponential complexity for general context-free grammars, but it has polynomial complexity for certain grammars as pointed out in [62]. The NSD grammar is an example of such a grammar.
The hypergraph parser furthermore meets the criteria for visual language parsers: it can also be used for partially correct hypergraphs, and it may be used incrementally, i.e., subsequent runs of the parser can make use of the results of previous runs. These features are considered next.

[h] In the following, we call the terminal subgraph that can be derived from a hyperedge e simply *terminal subgraph of* e.

In general, the hypergraph as entirety is not syntactically correct, but contains a number of correct subgraphs. The sets S_i make it easy to find a set of maximal correct subgraphs (i.e., consisting of as many hyperedges as possible and being syntactically correct): the sets $S_n, S_{n-1}, S_{n-2}, \ldots$ are searched for occurrences of axiom hyperedges. Such a hyperedge is selected if there are no conflicts with any previously selected one. Two hyperedges are said to have conflicts if their terminal subgraphs have common terminal hyperedges. After termination, the selected axiom hyperedges are the derivation tree roots of correct subgraphs, thus specifying correct (sub)sentences. Note that this classification is in general ambiguous: if a set S_i contains axiom hyperedges with overlapping terminal subgraphs, at most one of them is selected; therefore, the sequence of checking axiom hyperedges in S_i may influence the selection.

The parser algorithm described so far constructs a hypergraph's syntactic information from scratch. When keeping the syntactic information, i.e., the sets S_i, up-to-date even after hypergraph modifications, we obtain an incremental parsing algorithm. Each hypergraph modification can be divided into basic modifications of three kinds:[i]

- *Splitting the hypergraph into two unconnected subgraphs*: Nonterminal hyperedges become invalid if their terminal subgraphs are also split when splitting the entire hypergraph. Therefore updating the sets S_i means to check each nonterminal hyperedge and to remove it if its terminal subgraph is split.

- *Joining two previously unconnected subgraphs by merging nodes*: The existing nonterminal hyperedges remain valid, and the parser has to create additional ones. This is done like parsing from scratch, but pairs of already processed nonterminal hyperedges may remain unconsidered.

- *Adding an isolated hyperedge to the hypergraph*: An additional, but empty set S_{n+1} has to be established, and S_1 is extended by nonterminal hyperedges that can be derived to the new terminal one.

The remaining task of the compiler is to search for maximal correct subgraphs and to reconstruct derivation trees. Again, the parser can save work here by keeping track of unchanged portions of the sets S_i.

[i] We assume that syntactically correct subgraphs have to be connected like in the NSD example. This corresponds with a grammar where each production has a connected RHS. Otherwise a similar argumentation applies.

1.5.3 Parsing for Context-Free Hypergraph Grammars with Embeddings

Context-free grammars have limited expressiveness. Therefore, Subsection 1.3.3 has introduced *context-free hypergraph grammars with embeddings* (CF_eHGs) which consist of context-free as well as embedding productions. Please remember, that embedding productions must not be applied to a context that contains hyperedges which have been created by any previous rewrite step using an embedding production. Their definition and the restriction of permitted derivation sequences has been motivated by the wish for more expressiveness and yet efficient parsing capabilities. The following shows that only a slight extension of the parser for context-free grammars is necessary in order to obtain a parser for CF_eHGs.
Derivation structures for CF_eHGs are DAGs as explained in Subsection 1.3.3. Each of these DAGs consists of a main (derivation) tree and some additional (derivation) trees that are pasted into the main tree by applying embedding productions. When used with an CF_eHG's context-free productions only, the parser outlined for context-free grammars will just create these trees, however, without the DAG edges embedding the additional trees into the main tree. An extension of the parser is then responsible for selecting context hyperedges within the main tree and appropriately pasting in these additional trees rooted at embedded nonterminal hyperedges. The overall procedure of parsing for CF_eHGs is as follows:

- *Transformation step:* The grammar is first transformed into a purely context-free one. Each embedding production is replaced by a context-free production with the same RHS, but with a hypergraph as LHS which consists of a single hyperedge with a new and unique nonterminal label, but without any tentacles. This new context-free hypergraph grammar is then transformed into CNF. Each of the original embedding productions is thus transformed into a set of context-free productions which derive the embedding context and the embedded hyperedge of the original embedding production from a new nonterminal. Whenever such a nonterminal is created in the *parsing step*, a candidate for an application of an embedding production has been found.

  Figure 1.22a shows the new context-free production as a result of transforming the last (embedding) production of the CF_eHG for MSC shown in Figure 1.6. E is the new and unique nonterminal label. The derivable context consists of the hyperedges labeled by *msc*, *Seq*, and *lifeline*, the embedded hyperedge is an M-hyperedge. Figure 1.22b shows the CNF of this new production.

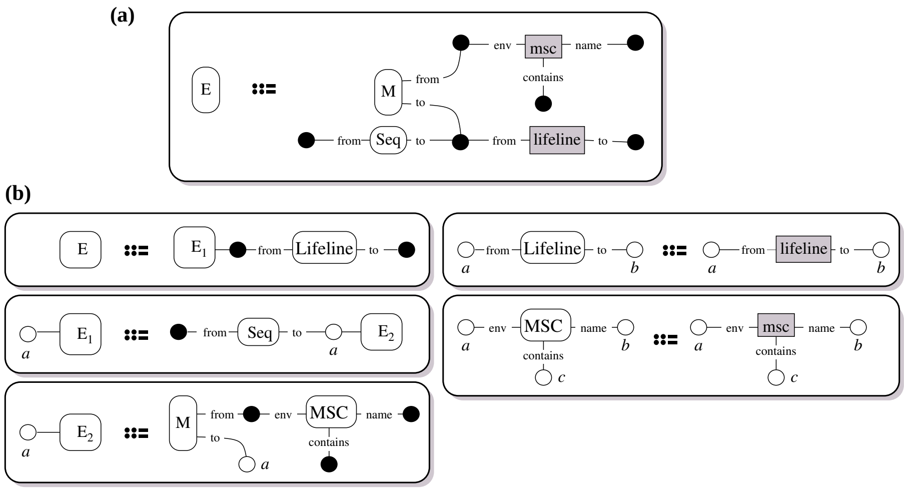


Figure 1.22: Translation of the last embedding production of Figure 1.6 into a context-free production (a) and then into CNF (b). Please note that E is a new and unique nonterminal label for hyperedges without any tentacle.

- *Parsing step:* Using the parser for this CNF grammar will create the main (context-free) derivation tree as mentioned above. Furthermore, all the other (derivation) trees which have to be pasted into this main tree are stored in the parsers structure of sets S_i. Occurrences of RHSs of embedding productions, i.e., embedding context and embedded hyperedge, can be found by searching in the sets S_i for new nonterminal labels introduced in the transformation step. Embedded hyperedges are connected to their corresponding context by DAG edges if all of these context hyperedges belong to the same main derivation tree. Please note, that new nonterminals introduced in the *transformation step* are not included into the derivation DAG.

This outline shows that the parser is essentially the same as the parser for context-free grammars. The extension consists of searching for certain nonterminals in the sets S_i and checking whether they can be derived to certain hyperedges contained in the main derivation tree for the context-free portion of the hypergraph.
The description of the parsing algorithm for $\mathrm{CF}_e\mathrm{HG}$s assumed that only one syntactically correct hypergraph is parsed. But it is easily extended to parsing of hypergraphs that are only partially correct, i.e., which contain correct

subgraphs and some hyperedges not belonging to any correct subgraph. The procedure is also an extension of the parsing for context-free hypergraph grammars: The context-free parser searches for maximal main derivation trees in the structure of sets S_i as described before. In the following step, i.e., when searching how to paste in embedded derivation trees, the parser has to verify for each embedding that its context's hyperedges belong to the same main derivation tree.

Incremental parsing is also a simple extension of the context-free parser. Since the CF$_e$HG has been transformed into a context-free hypergraph grammar, the structure of sets S_i can be kept up-to-date as described before. Main derivation trees are created incrementally by the context-free parser as described before. The parser can save work in the following step, i.e., when trying to paste in embedded derivation trees, by keeping track which derivation trees have not been changed.

1.5.4 Parsing for Layered Graph Grammars

Following the structure of Section 1.3 we now have to explain how it is possible to parse more general classes of (hyper)graph grammars than those discussed until now. Therefore, we will return to the language of class diagrams of Subsection 1.3.4, which could not be characterized precisely by means of context-free hypergraph grammars with embeddings. Subsection 1.3.4 introduced for that purpose unrestricted graph grammars for which the parsing problem becomes undecidable in the general case. Furthermore, this subsection switched from hypergraphs to graph structures which were depicted as simple directed graphs. Here we will also exchange the employed graph data model and switch from hypergraphs to directed graphs, too. But we have to emphasize that the parsing algorithm presented on the following pages is more or less independent of the underlying graph model and works, therefore, also for hypergraphs. It was introduced in [95] and accepts a rather general subclass of unrestricted graph grammars as input, so-called *layered graph grammars* (LGGs). As a consequence, it has an exponential worst-case runtime and space complexity.

Talking about parsing algorithms one has to distinguish two tasks which have to be performed:

1. The searching in the host graph for matches of a production's RHS. This is an expensive process which works at the graph element level. One standard solution for improving the efficiency of the needed graph pattern matching algorithm is based on so-called search plans. These search plans are lists of graph traversal instructions, which find as a whole all possible matches for a given rule's LHS or RHS in a graph.

For further details concerning the computation of almost optimal search plans the reader is referred to [96].

2. Each completed RHS match corresponds to one possible reduction step, i.e., to the reverse application of a production. These reverse applications, called *production instances*, have to be combined into a derivation. In the case of ambiguities it might however happen that more than one derivation exists, or it might happen that a recognized production instance is not part of any derivation of the complete input graph.

During the development of the LGG parsing algorithm it became evident that dealing with these two tasks at the same time results in very complex algorithms. These algorithms would even perform a lot of work which often turns out to be useless afterwards. Therefore, the LGG parsing algorithm consists of two phases, a bottom-up phase similar to the one of the presented hypergraph grammar parsers which is followed by an additional top-down phase.

The *bottom-up phase* searches the graph for matches of right-hand sides of productions. On the recognition of such a RHS, a production instance *pi* is created and copies of its LHS elements without corresponding RHS elements are added to the graph. The additions might in turn lead to the recognition of other right-hand sides and thereby to the creation of new production instances that create in turn .... The result of the bottom-up phase is the collection *PI* of all production instances discovered.

The production instances created have *dependency relations* among each other, such as *above(pi, pi')*, which means that *pi* should occur before *pi'* in a derivation of the regarded input graph. This happens whenever one production instance creates a graph element which is then matched by the RHS of another production instance. Another example of a dependency relation is *exclude(pi,pi')*, which states that *pi* and *pi'* may not occur in the same derivation. Two productions exclude each other, whenever they reduce (with their RHS) the same graph elements. These dependency relations can be computed during the bottom-up phase.

The dependency relations are used to direct the second phase of the parsing process, the *top-down phase*. It starts with the graph grammar's axiom graph and applies production instances of *PI* in such a way that the *above* and *exclude* relations are respected. By knowing all possible production instances and their dependencies in advance, the top-down phase is able to postpone exploration of alternative derivation branches as long as possible. When necessary, these alternative derivations are developed in a pseudo-parallel fashion, with a preference for depth-first development.

Please note that the bottom-up phase does not terminate in the general case for a given unrestricted graph grammar. We, therefore, have to introduce the following *layering restriction* on the grammar: every edge and node label is assigned a layer number, such that terminal labels belong to layer 0, and non-terminal labels belong to a layer greater than 0. This layer assignment has to be such that every production respects the following order for its LHS and RHS:

$$\text{LHS} < \text{RHS} \quad \Longleftrightarrow \quad \exists_i |\text{LHS}|_i < |\text{RHS}|_i \wedge \forall_{j<i} |\text{LHS}|_j = |\text{RHS}|_j$$

with $|G|_k$ the number of elements in G which have a label of layer k. The layering restriction guarantees that the reduction step of any production instance covers more graph elements of lower layers with its RHS than it adds to these layers as copies of its LHS. It enforces thereby the termination of the bottom-up phase by disallowing "cyclic" grammars.
The class diagram graph grammar of Figure 1.10 fulfills the layering restriction in a trivial way. The LHS of each of its productions is a proper subgraph of its RHS, i.e., each LHS contains simply less graph elements than the corresponding RHS. Even better, there is no LHS graph element which does not have a corresponding RHS graph element. As a consequence, the bottom-up phase does not add a single node or edge to the given input graph, but just computes all possible RHS matches and the dependency relationships between thereby generated production instances.
For further technical details concerning the presented parsing algorithm the reader is referred to [95]. A considerably simplified parsing algorithm for confluent layered graph grammars, sometimes called reserved graph grammars (RGG), may be found in [86]. The parsing algorithm for RGGs has a polynomial worst-case behavior but is restricted because of the requirement of confluence.

1.6 Visual Graph Transformation Languages

Graph transformation languages—as presented in the preceding sections as well as in Chapters **??**, **??**, and **??** of this book—constitute a well-established branch of rule-based visual specification or programming languages. They offer solutions for a number of open problems mentioned at the end of Subsection 1.2.4 about rule-based VLs:

- a tighter integration of data modeling by means of class and object diagrams and operation specification by means of rules,

- a (more) precise definition of the semantics of rules, including non-obvious cases like "two left-hand side items match the same graph item",

- and a controlled application of rules instead of the standard "recognize-select-execute" cycle of pure rule-based systems.

It is not the purpose of this section to explain in detail how today available graph transformation languages and their programming environments look like and how they solve the presented list of problems. On the contrary, this section complements in-depth presentations of the general purpose graph transformation languages AGG and PROGRES in Chapters ?? and ??, respectively.
This section starts with short subsections about the history and the most important properties of these languages and then compares main features of them and of the VL environment generating graph transformation languages GENGED and DIAGEN presented in the preceding sections.

1.6.1 The Graph Transformation Language PROGRES

One of the biggest projects that uses graph transformation systems as an executable specification or visual programming language was launched about 18 years ago [97]. Its central idea was to use graph transformation systems instead of attribute grammars for mechanizing the development of Integrated Programming Support (software engineering) ENvironments [98]. This lead to the project's name IPSEN. Such an environment offers tools for all phases of the software life cycle from analysis over design to programming, and it supports accompanying administration tasks like configuration and project management as well as reuse of software.
In the beginning pure graph transformation systems with rather simple forms of rules were used in IPSEN [99]. But soon it became obvious that additional concepts were needed to make the description of complex software engineering tools feasible. One of the most important extensions was the introduction of new means for controlling the application of graph transformation rules. This lead to the invention of *PROgrammed Graph REwriting Systems*, with PROGRES being one example of this kind (cf. Chapter ??). PROGRES is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. It was developed having the following design goals in mind:

- Use graphical syntax where appropriate, but do not exclude textual syntax when it is more natural and concise.

- Distinguish between data definition and data manipulation as database languages do and use graph type definitions in the form of class diagrams (graph schemata) to typecheck graph manipulating operations.
- Offer declarative language elements for the (visual) definition of graph views, derived attributes, and integrity constraints (based on incremental attribute evaluation algorithms and ECA-rules of active database systems).
- Refrain users from the task to guarantee confluence of defined graph transformation systems by keeping track of rewriting conflicts and backtracking out of dead-end derivations.
- Complement the visual rule-oriented programming paradigm with the imperative programming paradigm of textual procedural programming languages.
- Finally, do not incorporate elements into the language, which are either not (efficiently) executable or come without a precisely defined syntax and semantics definition.

The development of PROGRES and its formal definition in [100,101] would not have been possible without the existence of a number of predecessors. We have to mention its direct predecessors developed by Engels/Schäfer [98] and Lewerentz [102] as well as another language based on Programmed Attributed Graph Grammars (PAGG), developed by Herbert Göttler [103,104]. All these languages use the same data model of directed, attributed, node- and edge labeled graphs as PROGRES, offer a mixed visual/textual notation for the definition of graph transformation rules, and regulate the application of their rules by means of imperative control structures, too.
Comparing PROGRES with those rule-based VLs introduced in Subsection 1.2.4 we have to state that languages like BITPICT [48] or ChemTrains [49] focus on manipulation of data only. Even Vampire [51] with its class hierarchies and icon rewriting rules comes without a rigid type concept and without any type checking tools. Therefore, all these systems postpone recognition of programming errors to runtime and suffer from the same disadvantages as any untyped or weakly typed programming language. With respect to its graph type definition capabilities, PROGRES is more similar to visual database programming languages such as GOOD [105]. But these languages have less expressive pattern matching and replacing constructs; some of them are even not computationally complete (no recursion). Furthermore, neither the above mentioned languages nor—to the best of our knowledge—any other visual rule-oriented

language offers nondeterministically working control structures together with the ability to backtrack out of dead-ends of locally failing graph transformation steps.

1.6.2 The Graph Transformation Language AGG

Compared to visual programming languages, AGG may be understood as a hybrid programming language which offers visually as well as textually represented elements. Graph transformation on the visual side is combined with Java[j], the textual object-oriented programming language recently developed by SUN. Both concepts are tightly coupled, thus, the visual program part is scalable. The advantage of combining visual concepts with a common textual programming language lies in the broad availability of programming know-how concerning concepts, libraries and tools for textual programming languages. We use Java for this purpose, because Java is a modern, interpreted language which is platform independent. In AGG we also chose the interpretative approach for graph transformation, which is very natural in this context. Since AGG itself is written in Java we get platform independence for the whole application developed with AGG for free.

The development of the graph transformation system AGG at TU Berlin started with the realization of a flexible graph editor plus an accompanying small graph transformation component programmed in Eiffel [106]. AGG has been reimplemented in Java to open it for flexible attribute computations. In this way, AGG becomes a general purpose language where a new programming paradigm, i.e., graph transformation, is incorporated into the object-oriented paradigm. A further aim of AGG is to make graph transformation fit for distributed applications. Especially distributed algorithms are well described by graph transformation. Independent and concurrent actions within such systems are better formulated in a rule-based manner to avoid global control flow as far as possible. The whole development of AGG is headed towards a visual design tool for distributed systems. Such a tool should be able to at least prototypically realize really distributed systems with graph transformation.

As PROGRES, AGG is designed as a general purpose graph transformation based language. But both languages differ in various concepts, especially concerning the underlying graph model and the graph transformation concepts. These differences will be discussed in Subsection 1.6.3 in more detail. Concerning the graph transformation-based concepts, the comparison of PROGRES to other visual languages holds also for AGG. Concerning attribution and rule application control, the way how graph transformation is embedded into other

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

programming paradigms is similar to PROGRES. The main differences are: PROGRES has some basic built-in data types. All other types have to be defined in a so-called host language, which may be Modula-2 or C, and which have to be imported to PROGRES. In AGG no built-in types exist and Java is used for this purpose. In PROGRES, custom-defined imperative control constructs are used to formulate programmed graph rewriting, i.e., the control constructs are not taken from an established programming language. Programmed graph rewriting in PROGRES is based on transactions which are designed to have the same characteristics as rules. Using Java to control rule application as it is planned for AGG such a transaction concept is not automatically provided, but may be programmed in Java.

AGG has a formal foundation based on the single-pushout approach to graph transformation [67]. Its underlying graph model is the graph structure of labeled and attributed hierarchical graphs. This graph model is used inside to manage and transform the graphs. For the user interface typed and attributed directed graphs are chosen as graph data model. It is planned to make hierarchical graphs also available at the user interface. The overview on abstraction levels as well as their inspection will be supported by several views.

Arbitrary graph rules are allowed to be formulated including insertion, deletion and gluing of graph parts as well as complex attribute computations formulated in Java. Moreover, negative application conditions can be added to restrict the applicability to certain situations. It is planned to further control rule applications by formulating an explicit control flow again in Java. A change to hierarchical graphs would also allow these graphs in the rules to formulate actions on several abstraction levels.

Since the theoretical concepts are implemented as directly as possible – not leaving out necessary efficiency considerations – AGG offers clear concepts and a sound behavior concerning the graph transformation part. Furthermore, the formal foundation of AGG offers the possibility for software validation. Validation features are not yet available, but will be one of the next extensions of AGG.

The AGG environment supports the visual administration of graphs and rules, structured into graph grammars. Visual editing, interpretation, and debugging tools are integrated in the AGG environment. The visual layout of graphs is quite simple yet showing nodes by some closed form (rectangle, circle,etc.) and edges by two-colored lines. More sophisticated user-defined layouts of nodes and edges will be available by exchanging the graph and rule editors of AGG by new ones generated by GENGED. Furthermore, layout constraints for pretty printing transformed graphs are then supported.

1.6.3 Comparison of Graph Transformation Languages

Having presented the two general purpose graph transformation languages AGG and PROGRES in the preceding sections and the special purpose languages GENGED and DIAGEN,[k] in Subsections 1.4.2 and 1.4.3, it is now time to present the promised comparison of these languages, their accompanying (meta) programming tools, and their generated VL tools (the output of their meta programming tools). The comparison is organized as two tables, where the first one deals with language issues and the second one with tool issues.

Table 1.23 about graph transformation languages consists of four parts, where the first one deals with the language's data model, the second one with the related topic of derived data elements, the third one with the central topic of graph transformation rules, and the last one with additional means to regulate the application of graph transformation rules.

The first nine rows about the *data model* of the presented graph transformation languages start with some overall properties of a graph data model and discuss then more specific properties of nodes, edges, and attributes, respectively. As already explained beforehand, the presented languages employ rather different data models, varying from directed graphs in the case of AGG and PROGRES to hypergraphs in the case of DIAGEN (plus derivation DAGs generated by a parser) and to arbitrary graph structures in the case of GENGED. All four languages have a kind of type definition, which declares first the components of a regarded class of graphs, before instances of this class may be manipulated by means of graph transformations. These type definitions have rather different forms, ranging from algebraic signatures in the case of GENGED to extended Entity Relationship diagrams (class diagrams) in the case of PROGRES. It is worth-while to notice that not a single of the four languages offers proper support for the definition of graph hierarchies, i.e., the well-known concept of aggregation is missing. This is partly due to the fact that it is very difficult to come up with a precise definition of the differences between regular relationships (edges) and aggregations in the case of arbitrary graph structures. AGG uses hierarchical graphs internally, but they are not yet available at the user's interface.

Concerning the properties of the *basic elements* of the four language's data models the differences are less important. All languages know some kind of attributed object. These objects, the *nodes*, play the central role in the case of AGG, GENGED, and PROGRES, but are of minor importance in the case of DIAGEN. Nodes of DIAGEN are only used as connection points between

[k] In the following, DIAGEN actually refers to DIAGEN II if DIAGEN I is not explicitly referred to.

Language Props.			AGG	DiaGen	GenGEd	PROGRES
Graph Data Model	graph	kind	directed graph	hypergraph, deriv. DAG	graph structure	directed graph
		type definition	node and edge types	hyperedge types	graph signature	graph schema (diagram)
		graph hierarchy	(internal)	—	—	—
	node	kind	labeled, attributed object	unlabeled attr. object	labeled, attributed object	labeled, attributed object
		label	form, color, text	—	graphic	node type of schema
	edge	kind	dir., labeled, attr. object	labeled attributed object	dir., labeled, attr. object	directed, labeled tuple.
		label	color, text	graphics	graphics	edge type of schema
	attr.	value	Java object/value	Java {C++} object/value	Java object/value	C values, built-in values
		language	Java expressions	Java expr.essions	visual, constraints	C or PROGRES expr.
Derived Elems		graph views	[planned]	—	[planned]	def. with ECA rules
		derived edges	—	grouping	—	defined with path expr.
		derived attributes	—	defined with constraints	defined with constraints	def. with directed equ.
		integrity constraints	[visual definition]	—	—	visual, 1st order logic exp
Rules (Productions)		notation	visual	textual, [visual]	exchangeable visual	visual & textual
	matches	homo/isomorphic	both	both	both	both
		multiple matches	[subrules]	—	—	set node, star rule
	conditions	attr. conditions	[Java expression]	Java expressions	[graphical constraints]	1st order logic expr.
		pos. LHS context	LHS-subgraphs	LHS-subgraphs	LHS-subgraphs	paths, LHS-subgraphs
		neg. LHS context	LHS-related graphs	LHS-related graphs	LHS-related graphs	neg. LHS nodes, edges
	conflicts	delete/preserve	delete	forbidden	delete	forbidden
		dangling edges	delete	forbidden	delete	delete
		graph gluing for	graph objects	graph objects	graph objects	—
	embed	visual notation	—	—	—	set nodes
		text notation	—	—	—	embedding rules
Control		programming language	[Java methods]	Java {C++} & automata	—	transactions
		programming style	[object-oriented]	object-oriented	—	imperative & backtrack

Figure 1.23: Comparison of four graph transformation languages (entries in square brackets are not implemented, entries in curly braces were part of an earlier version).

hyperedges and do not have any additional label as in all other cases. *Edges*, on the other hand, model

- labeled, attributed binary relationships (directed edges) with identifiers in AGG (parallel and m:n relationships are supported),

- labeled attributed n-ary relationships (hyperedges) with labeled tentacles in DiaGen (which have about the same properties as edges in GenGEd)
- labeled, attributed binary relationships with identifiers in GenGEd (parallel relationships as well as m:n or 1:n relationships are supported),
- and labeled, attributed binary relationships without identifiers in the case of PROGRES (parallel relationships are not supported, but m:n relationships are).

All languages except GenGEd use a host language for the definition of attribute types and the manipulation of attribute values. This is Java in the case of Agg and DiaGen II, it is C in the case of PROGRES, and C++ in the case of Diagen I. Both GenGEd and PROGRES offer (furthermore) a number of built-in data types which are part of the language itself and not imported from a host programming language.
Until now we had our main focus on those data elements which are explicitly manipulated by graph transformation rules. But there is another category of so-called *derived* data elements which are automatically updated whenever some related data elements are modified. First of all PROGRES offers some help for the definition of *graph views* based on the concept of active integrity constraints, which is closely related to the Event-Condition-Action (ECA) rules of active database systems. Furthermore, PROGRES allows one to define derived edges and attributes as well as normal integrity constraints based on so-called path expressions, directed equations, and graphical patterns. DiaGen and GenGEd, on the other hand, use the more general concept of arbitrary constraints (undirected equations, inequalities, ...) instead of directed equations for the definition of derived attributes. Finally DiaGen has some grouping mechanism for viewing and handling collections of hyperedges selected by the user as composite derived hyperedges.
Concerning the definition of *graph transformation rules* Table 1.23 distinguishes four groups of properties plus two additional properties. They concern the employed notation, some properties of LHS matches, additional possibilities for the definition of application conditions which prohibit certain LHS matches, the properties of the rewriting mechanism itself, and the existence of additional means for the manipulation of embedding edges (those edges which connect the replaced/replacing subgraph with the rest graph).
It is remarkable that all discussed languages have a switch which 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. Furthermore,

not a single of the presented languages supports parallel graph transformation in the general case, where several rules are applied in parallel to some or all occurrences (matches) of their LHS. PROGRES offers some limited support for manipulating sets of nodes of arbitrary size in a single graph transformation step as well as for rewriting all matches of a single rule in parallel (with certain restrictions). The PROGRES language constructs for these purposes are a special case of so-called subrules with *multiple matches*, as they are planned for AGG.

Additional *conditions*, which prohibit rule applications to otherwise legal LHS matches, are treated very similar in all four cases. There is some textual language for the definition of attribute conditions, it is possible to require the existence of certain nodes and edges as part of the LHS, which are not affected by the specified graph transformation, as well as to prohibit the existence of certain nodes and edges. All languages, except PROGRES, support the definition of negative graphs (which may not exist as extensions of a determined LHS occurrence), whereas PROGRES relies on the simpler concept of single negative LHS nodes and edges.

The graph transformation mechanisms of our four languages, 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 rule's LHS and RHS, and another graph element, which is only part of the rule's LHS, match the same host graph element. AGG and GENGED vote for deletion in the case of such a *delete/preserve conflict*, whereas DIAGEN and PROGRES simply forbid 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 rule's match. DIAGEN favours again the conservative approach and forbids such a graph transformation step, whereas all other languages simply delete the otherwise dangling edges, too.

The following table row about *graph gluing* shows that all languages except PROGRES allow one to glue (merge) some distinct host graph elements in the course of a graph transformation step. This concept is not supported by PROGRES, forcing its users to model gluing as the deletion of all graph elements except of one. This is possible due to the fact that PROGRES supports the redirection of all incidents edges of deleted (or preserved) nodes to other nodes. First of all the visual notation of the above mentioned set nodes may be used for this purpose. Furthermore, there are textually denoted *embedding rules*, which follow the tradition of the expression-oriented algorithmic graph transformation approach.

Finally, the last rows of Table 1.23 sketch the available means for controlling the application of rules. DiaGen uses the imperative programming means of Java (C++ in the case of DiaGen I) for this purpose as well as so-called interaction automata. PROGRES, on the other hand, offers so-called transactions, which combine an imperative programming style with a more rule-oriented depth-first-search and backtracking programming style.

Switching from languages to tools (systems) we have to regard Table 1.24. This table consists of two subtables, where the first one characterizes the tools for editing, analyzing, and 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*. The table shows that Agg is not used for the development of visual tools, whereas the three remaining systems are able to generate different parts of VL programming environments. GenGEd is rather strong concerning the definition of graphical representations and their layout (based on constraints); it generates syntax-directed graphical editors only. DiaGen, on the other hand, lies its main emphasis on free-hand editing and parsing of diagrams. PROGRES, finally, has its strong points when complex analysis operations or animations have to be specified.

Concerning the available tool support it is clearly visible that PROGRES is the "eldest" system with the, therefore, most complete support for editing, analysing, and execution activities. It possesses a sophisticated type checker and supports direct interpretative execution as well as compilation of specifications. DiaGen follows the traditional compiler compiler approach, i.e., it processes a pure textual definition 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.

The remaining two systems may be described as follows: Agg has the closest integration of a graph transformation machinery with the Java execution machinery and follows the interpretative execution approach only. GenGEd is famous for its integration with the layout constraint solver ParCon and its flexibility concerning the notation of LHS and RHS of its rules. It is the only graph transformation tool which allows one to adapt its rule editor such that it uses directly the visual constructs of a studied application domain.

Tools		AGG	DiaGen	GenGEd	PROGRES
(Meta) Programming Tools	syntax-dir. editor for	visual elements	—	visual/texual elements	visual/textual elements
	text editor for	Java elements	all language elements	high-level constraints	all language elements
	text parser for	Java elements	all language elements	high-level constraints	all language elements
	analyzer	—	{consistency checker}	—	type checker
	interpreter	Java Byte Code	—	ParCon & AGG	PROGRES Byte Code
	debugger		—	—	part of interpreter
	code generator	[Java]	Java {C++}	—	C, Modula-2, tcl/tk
	graph display	graph editor	—	graphics editors	browser, text file
	graph layout	manual	—	constraint-based	Sugiyama, …
Generated VL Tools	VL representation	—	visual, API	visual	diagram & text [table]
	syntax-dir. editor	—	{+}	graphical (rule) editors	diagram editors
	free-hand editor	—	+	—	—
	parser	—	restr. context-sensitve	—	—
	analyzer	—	—	—	+
	animation tools	—	+	[planned]	+
	layout algorithms	—	constraints, plug-in	constraint-based	Sugiyama, …

Figure 1.24: Comparison of four graph transformation programming tools (entries in square brackets are not implemented, entries in curly braces were part of an earlier version).

1.7 Conclusions

Graph transformation approaches and visual languages are two research topics which are close to each and which may profit from each other considerably. As we have seen in this chapter, graph transformation languages are a special brand of rule-based visual languages. They have some properties which are rather unusual for rule-based visual languages in general, such as a very precise formal definition of their rewriting semantics or the combination of rules with imperative control structures.

Visual rule-based languages, on the other hand, offer some concepts which are usually not present in graph transformation languages. Examples of this kind are a very tight integration of layout modifications and logical operations (in most cases the layout of a visual programming language rules LHS and RHS is meaningful and affects the layout of the manipulated workspace), the construction of rules based on programming-by-example, the generation of families of similar rules (based on some examples), and so forth.

Furthermore, we have seen that graph grammars and graph transformation based programming languages are very useful for constructing precise definitions of visual languages as well as for generating visual programming tools. Within this chapter we laid the main emphasis on syntax definition issues. We presented a number of graph grammar parsing algorithms of increasing complexity as well as some techniques and meta tools for generating syntax-directed editors and free-hand editors. Nevertheless, we have to point out that the graph transformation approach is also well prepared for defining the static as well as the dynamic semantics of a visual language. These topics are addressed by different chapters of this book, which discuss for instance the construction of software engineering tools (cf. Chapter ??), the realization of compiler-compiler components (cf. Chapter ??), or the development of process modeling tools (cf. Chapter ??). In all these cases graph transformation rules are used to analyze and animate (the internal graph representations of) the sentences of a regarded language.

Being a survey about the relationships between visual languages and graph transformation approaches, this chapter presented and compared four different graph transformation languages. Two of them, the visual programming tool generating languages DiaGen and GenGEd, were presented in more detail. The remaining two more general purpose graph transformation (programming) languages Agg and PROGRES, which are the main subjects of Chapter ?? and Chapter ?? in this book, were only introduced for comparison purposes.

Agg, DiaGen, GenGEd, and PROGRES are the results of a decade of application-oriented graph transformation research activities. They were built based on the experiences of with a long list of predecessors, which starts with some common ancestors of all visual and graph transformation languages: AMBIT/G and PLAN2D.

Nevertheless, our list of important future research topics is still rather long: First of all unnecessary differences of different approaches should be eliminated. Furthermore, guidelines have to be developed which explain the remaining differences between hypergraphs and directed graphs or graph structures, distributed graph transformation and controlled sequential graph transformation, context-free hypergraph grammars with embeddings and layered graph grammars, and so forth from a users point of view. Last but not least we have to attack some common problems with the visual language community: we have to solve the "scaling-up problem" by offering appropriate module concepts, the "frustrated user problem" by developing more efficient tools which make better use of available screen space, produce less unacceptable diagram layouts, and so forth.

References

1. *Proc. IEEE Symposium on Visual Languages (VL'95)*, Los Alamitos, CA, 1995. IEEE Computer Society Press.
2. *Proc. IEEE Symposium on Visual Languages (VL'96)*, Los Alamitos, CA, 1996. IEEE Computer Society Press.
3. *Proc. IEEE Symposium on Visual Languages (VL'97)*, Los Alamitos, CA, 1997. IEEE Computer Society Press.
4. E. Golin. *A method for the specification and parsing of visual languages.* PhD thesis, Brown University, 1991.
5. E. Golin. Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2(4):371–394, 1991.
6. K. Marriott. Constraint multiset grammars. In *[107]*, pages 118–125, 1994.
7. G. Rozenberg, editor. *Handbook on Graph Grammars: Foundations*, volume 1. World Scientific, Singapore, 1997.
8. 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.
9. K. Wittenburg and L. Weitzman. Visual grammars and incremental parsing for interface languages. In *[108]*, pages 111–118, 1990.
10. 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.
11. S. Chok and K. Marriott. Parsing visual languages. In *18th Australasian Computer Science Conference*, Glenolg, South Australia, 1995.
12. A. B. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *[32]*, pages 33–40. 1990. Reprint from [109].
13. V. Dock. *Cobol: American National Standard.* Reston Publ. Company, 1973.
14. I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), 1973.
15. M. Halverson. *Microsoft visual basic 4 step by step.* Microsoft Press, Redmond, WA, 1995.
16. I. E. Sutherland. Sketchpad: A man-machine graphical communications system. In *Proc. 23rd AFIPS Spring Joint Computer Conference*, 1963.
17. S.-K. Chang. Visual languages: A tutorial and survey. In *[32]*, pages 7–18. 1990. Reprint from [110].

18. A. L. Ambler and M. M. Burnett. Influence of visual technology on the evolution of language environments. In *[32]*, pages 19–32. 1990. Reprint from [111].
19. N. C. Shu. Visual programming languages: A perspective and a dimensional analysis. In *[32]*. 1990. Reprint from [112].
20. C. Christensen. An example of the manipulation of directed graphs in the ambit/g programming language. In *Interactive Systems for Experimental and Applied Mathematics*, New York, 1968. Academic Press.
21. E. Denert, R. Franck, and W. Streng. Plan2d - towards a two-dimensional programming language. volume 26 of *Lecture Notes in Computer Science*, pages 202–213. Springer Verlag, 1974.
22. D. C. Smith. *Pygmalion - A Computer Program to Model and Stimulate Creative Thought.* PhD thesis, Stanford University, Stanford, CA, 1975.
23. D. C. Smith. Pygmalion: A computer program to model and stimulate creative thought. In *[32]*, pages 216–251. 1990.
24. 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.
25. T. O. Ellis, J. F. Heafner, and W. L. Sibley. The grail project: An experiment in man-machine communication. Technical Report Report RM-5999-Arpa, RAND, 1969.
26. M. C. Pong and N. Ng. Pigs - a system for programming with interactive graphical support. *Software, Practice and Experience*, 13(9):847–855, 1983.
27. J. F. Sowa. *Conceptual Structures: Information Processing in Minds and Machines.* Addison-Wesley, Reading, Mass., 1984.
28. J. Rumbaugh, M. Blaha, W. P. F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.
29. Rational Software Corporation. UML semantics, version 1.1. http://www.rational.com, 1997.
30. M. M. Burnett and M. J. Baker. A classification system for visual programming. *Journal of Visual Languages and Computing*, 5(3):287–300, 1994.
31. *Proc. IEEE Symposium/Workshop on Visual Languages (VL'XX)*, Los Alamitos, CA, 19XX. IEEE Computer Society Press.
32. E. P. Glinert, editor. *Visual Programming Environments: Paradigms and Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1990.
33. M. M. Burnett, A. Goldberg, and T. G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments.* Manning Publica-

tions Co., Greenwich, 1995.

34. R. B. Grafton and T. Ichikawa, editors. *Computer: Special Issue on Visual Programming*, volume 18 (8). IEEE Computer Society Press, Los Alamitos, CA, 1985.
35. M. M. Burnet and D. W. McIntyre, editors. *Computer: Special Issue on Visual Programming*, volume 28 (3). IEEE Computer Society Press, Los Alamitos, CA, 1995.
36. D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *Proc. 1968 Fall Joint Computer Conf.*, volume 33-1, pages 395–410, Montvale, NJ, 1968. AFIPS Press.
37. V. Bush. As we may think. *Atlantic Monthly*, pages 101–108, 1945.
38. V. Bush. As we may think. In *[113]*, pages 85–110. 1991. Reprint of [37].
39. A. Goldberg. *Smalltalk 80 - The Interactive Programming Environment.* Addison Wesley, Reading, MA, 1984.
40. D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann. The structure of cedar. In *[114]*, pages 230–244. 1985.
41. M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.
42. S. P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11(3):276–285, 1985.
43. S. P. Reiss. Garden tools: Support for graphical programming. In *[115]*. 1986.
44. T. Teitelbaum and T. Reps. The cornell program synthesizer. *Communications of the ACM*, 24(9):563–573, 1981.
45. B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
46. E. H. Shortliffe. *Computer Based Medical Consultations: MYCIN*. Elsevier Science Publ., Amsterdam, 1976.
47. C. L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, 1981.
48. G. W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proc. of Conf. on Human Factors in Computer Systems - CHI'91*, pages 71–78, New York, 1991. ACM Press.
49. B. Bell and C. Lewis. ChemTrains: A language for creating behaving pictures. In *[116]*, pages 188–195, 1993.
50. D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, 1994.
51. D. M. McIntyre. Design and implementation with vampire. In *[33]*, pages

129–160. 1995.

52. K. M. Kahn and V. Saraswat. Complete visualizations of concurrent programs and their executions. In *[108]*, pages 7–15, 1990.
53. L. Bronwston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5 – An introduction to rule-based programming*. Addison-Wesley, New York, 1986.
54. M. Costabile and T. Catarci, editors. *Special Issue on Visual Query Systems*, volume 6 (1). Academic Press, New York, 1995.
55. J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *[2]*, pages 148–155, 1996.
56. K. Wittenburg. Earley-style parsing for relational grammars. In *[117]*, pages 192–199, 1992.
57. A. Grbavec and D. Blostein. Mathematics recognition using graph rewriting. In *Third Int. Conf. on Document Image Analysis and Recognition*, pages 417–421, Montreal, Canada, 1995.
58. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
59. A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, number 904 in Lecture Notes in Computer Science, pages 151–163, Berlin, 1994. Springer Verlag.
60. ITU-T, Geneva. *Recommendation Z.120: Message Sequence Chart (MSC)*, 1993. See also: *http://www.win.tue.nl/win/cs/fm/sjouke/msc.html*.
61. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Errata 5:1 (1971) 95–96.
62. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In *[7]*.
63. A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
64. M. Minas and G. Viehstaedt. Specification of diagram editors providing layout adjustments. In *[116]*, pages 324–329, 1993.
65. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, New York, 1993.
66. 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.
67. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, and A. Wagner.

Algebraic approaches to graph transformation – part ii: Single pushout approach and comparison with double pushout approach. In *chapter 4 in [7]*, pages 247–312. 1997.
68. R. Bardohl and G. Taentzer. Defining visual languages by algebraic specification techniques and graph grammars. In *[118]*, pages 27–42.
69. G. Taentzer, I. Fischer, M. Koch, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In *[119]*.
70. I. Fischer, M. Koch, and V. Volle. Conditinal attributed graph transformations as unifying formalism for neural networks. Technical report, Technical University Berlin, 1998.
71. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.
72. 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.
73. B. Backlund, O. Hagsand, and B. Pehrson. Generation of visual language-oriented design environments. *Journal of Visual Languages and Computing*, 1:333–354, 1990.
74. S. S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *[1]*, pages 242–249, 1995.
75. V. Haarslev and M. Wessel. GenEd – an editor with generic semantics for formal reasoning about visual notation. In *[2]*, pages 204–211, 1996.
76. G. Costagliola, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *IEEE computer*, pages 56–66, 1995.
77. P. Griebel. *Parcon - Paralleles Lösen von grafischen Constraints.* PhD thesis, Paderborn University, February 1996.
78. G. Viehstaedt and M. Minas. Interaction in really graphical user interfaces. In *[107]*, pages 270–277, 1994.
79. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
80. M. Minas. Hypergraphs as a uniform diagram representation model. In *Prelim. Proc. 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98)*, pages 24–31. Universität-Gesamthochschule Paderborn, Technical Report tr-ri-98-201, Reihe Informatik, November 1998.
81. A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.
82. M. Kaul. Parsing of graphs in linear time. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Com-*

puter Science, volume 153 of *Lecture Notes in Computer Science*, pages 206–218, 1983.

83. M. Kaul. *Syntaxanalyse für Präzedenzgraphgrammatiken.* PhD thesis, Universität Passau, 1985.
84. G. Rozenberg and E. Welzl. Boundary NLC graph grammars – Basic definitions, normal forms, and complexity. *Information and Control*, 69:136–167, 1986.
85. J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *[1]*, pages 195–202, 1995. Available from *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1995/tr95-15.ps.gz*.
86. D.-Q. Zhang and K. Zhang. Reserved graph grammar: A specification tool for diagrammatic VPLs. In *[3]*, pages 288–295, 1997.
87. H. Bunke and B. Haller. A parser for context-free plex grammars. In M. Nagl, editor, *Proc. 15th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'89)*, LNCS 411, pages 136–150. Springer-Verlag, 1989.
88. M. Minas. Diagram editing with hypergraph parser support. In *[3]*, pages 226–233, 1997.
89. F. Ferrucci, G. Tortora, M. Tucci, and G. Vitellio. A predictive parser for visual languages specified by relation grammars. In *[107]*, pages 245–252, 1994.
90. G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M. Tucci. Automatic parser generation for pictorial languages. In *[116]*, pages 306–313, 1993.
91. J. Earley. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery*, 13(2):94–102, 1970.
92. M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.
93. D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.
94. T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford Mass., 1965.
95. J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
96. A. Zündorf. Graph pattern matching in PROGRES. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 454–468. Springer, 1996.

97. M. Nagl. An incremental compiler as component of a system for software development. In *Informatik Fachberichte*, Berlin, 1980. Springer Verlag.
98. G. Engels and W. Schäfer. *Programmentwicklungsumgebungen: Konzepte und Realisierung*. Teubner, Stuttgart, 1989.
99. M. Nagl, G. Engels, R. Gall, and W. Schäfer. Software specification by graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 267–287, 1983.
100. A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen Anwendungsbeispiele und Werkzeugunterstützung*. Deutscher Universitäts-Verlag, Wiesbaden, 1991.
101. A. Schürr. Developing graphical (software engineering) tools with PROGRES. In *Proc. 19th Int. Conf. on Software Engineering (ICSE'97)*, pages 618–619, Los Alamitos, CA, 1997. IEEE Computer Society Press. Formal Demonstration.
102. C. Lewerentz. *Interaktives Entwerfen großer Programmsysteme: Konzepte und Werkzeuge*, volume 194 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, 1988. Doctoral dissertation.
103. 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*, volume 291 of *Lecture Notes in Computer Science*, pages 216–231, 1987.
104. H. Göttler. *Graphgrammatiken in der Softwaretechnik*, volume 178 of *Informatik-Fachberichte*. Springer-Verlag, Berlin, 1988.
105. J. Paredaens, J. Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. van Gucht, V. Sarathy, and L. Saxton. An overview of good. *SIGMOD Record*, 21(1):25–31, 1992.
106. M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In *Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456. Springer-Verlag, 1993.
107. *Proc. IEEE Symposium on Visual Languages (VL'94)*, Los Alamitos, CA, 1994. IEEE Computer Society Press.
108. *Proc. IEEE Workshop on Visual Languages (VL'90)*, Los Alamitos, CA, 1990. IEEE Computer Society Press.
109. 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.
110. S.-K. Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, 1987.

111. A. L. Ambler and M. M. Burnett. Influence of visual technology on the evolution of language environments. *Computer*, 22(10):9–22, 1989.
112. N. C. Shu. Visual programming languages: A perspective and a dimensional analysis. In *[120]*. 1986.
113. J. M. Nyce and P. Kahn, editors. *From Memex to Hypertext: Vannevar Bush and the Mind's Machine.* Academic Press, London, 1991.
114. L. P. Deutsch, editor. *ACM SIGPLAN '85 Symp. on Language Issues in Programming Environments, ACM SIGPLAN Notices*, volume 20. ACM Press, New York, 1985.
115. R. Conradi, T. Didriksen, and D. Wanvik, editors. *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1986.
116. *Proc. IEEE Symposium on Visual Languages (VL'93)*, Los Alamitos, CA, 1993. IEEE Computer Society Press.
117. *Proc. IEEE Workshop on Visual Languages (VL'92)*, Los Alamitos, CA, 1992. IEEE Computer Society Press.
118. *Proc. Workshop on Theory of Visual Languages (TVL'97)*, Capri, Italy, 1997.
119. G. Rozenberg, editor. *The Handbook of Graph Grammars, volume 3.* World Scientific, 1999.
120. S.-K. Chang, T. Ichikawa, and P. Ligomenides, editors. *Visual Languages.* Management and Information Systems Series. Plenum Press, New York, 1986.

Index