REPORT A-2002-1

Object-Oriented Engineering of Visual Languages

Antti-Pekka Tuovinen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public critcism in Auditorium III, Porthania, on March 2nd, 2002, at 10 o'clock.

UNIVERSITY OF HELSINKI
FINLAND

Contact Information

Postal address:

Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: antti-pekka.tuovinen@{cs.helsinki.fi, nokia.com}

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

ISSN 1238-8645
ISBN 952-10-0375-8 (bound)
ISBN 952-10-0376-6 (PDF)
Computing Reviews (1998) Classification: D. 3. 4, F. 4 .2, D. 1. 7
Helsinki 2002
Helsinki University Printing House

Object-Oriented Engineering of Visual Languages

Antti-Pekka Tuovinen

Department of Computer Science
P.O.Box 26, FIN-00014 University of Helsinki, Finland
antti-pekka.tuovinen@{cs.helsinki.fi, nokia.com}
http://www.cs.helsinki.fi/antti-pekka.tuovinen/

PhD Thesis, Series of Publications A, Report A-2002-1
Helsinki, February 2002, 185 pages
ISSN 1238-8645, ISBN 952-10-0375-8 (bound)
ISBN 952-10-0376-6 (PDF)

Abstract

Visual languages are notations that employ graphics (icons, diagrams) to present information in a two or more dimensional space. This work focuses on diagrammatic visual languages, as found in software engineering, and their computer implementations. Implementation means the development of processors to automatically analyze diagrams and the development of graphical editors for constructing the diagrams. We propose a rigorous implementation technique that uses a formal grammar to specify the syntax of a visual language and that uses parsing to automatically analyze the visual sentences generated by the grammar.

The theoretical contributions of our work are an original treatment of error handling (error detection, reporting, and recovery) in off-line visual language parsing, and the source-to-source translation of visual languages. We have also substantially extended an existing grammatical model for multidimensional languages, called atomic relational grammars. We have added support for meta-language expressions that denote optional and repetitive right-hand-side elements. We have extended what basically is a context-free grammatical model to take into account a limited amount of contextual information in order to better represent general graphs. Futhermore, we have made the parsing algorithm of the grammatical model more deterministic to facilitate effective error handling.

The main product of the constructive part of our research is the VILPERT (VIsual Language exPERT) system. It is an object-oriented Java framework for implementing visual languages. Implementing a visual language with VILPERT means generating a language analyzer based on a formal syntactic specification and implementing a graphical editor for manipulating the visual programs. The framework has a language specification sub-framework that is based on our extended version of atomic relational grammars. The language specification framework providers a parser for recognizing the languages specificed by extended atomic relational grammars. The parser produces a parse tree from a correct input, and the semantics of the source program is defined operationally by operations on the parse tree.

In our system, the graphical editor of a visual language is derived from an open-source Java framework. In the editor framework, we have added support for the notion of composite figure containers that facilitate the drag-and-drop style of moving figures into and out of containers and the construction of deeply nested graphical structures.

Our system provides a clean separation of the concerns of the graphical editing and the interpretation of diagrams both from the architectural and the usability point of view. The user draws the diagram in free order (not dictated by a syntax directed editor) and then invokes the language analyzer to interpret the drawing. The analyzer informs the user about any errors it finds during parsing and semantic processing. This approach to visual language implementation makes it possible to combine the sketching and the checking of diagrams into an explorative style of constructing visual programs.

Separating the two concerns of editing and analyzing reduces the software complexity of the implementation. For example, the correctness of a diagram does not have to be constantly enforced during editing, syntactic rules do not have to be enforced by hand-coded checks, and it is natural to maintain a clear separation between representation (graphical objects) and meaning (semantic or domain objects).

We have validated our solution by implementing three visual languages that represent typical notations used in software engineering (UML structural diagrams, UML statecharts, and flowcharts) and other small experimental languages. Because VILPERT is a framework, tools produced from it can be open for extensions, modifications, and they can share a common pool of reusable software components. Our implementations of visual languages show a high degree of reuse: the language (application) specific parts of the implementations is less than 20% of the total size of the applications.

Computing Reviews (1998) Categories and Subject Descriptors:

D. 3. 4 [Programming Languages]: Processors—parsing, translator writing systems and compiler generators
F. 4 .2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—grammar types, parsing
D. 1. 7 [Programming Techniques]: Visual Programming

General Terms:

Languages, Algorithms

Additional Key Words and Phrases:

Grammatical modeling, visual language parsing, visual language translation, object-oriented frameworks, graphical editors, diagrammatic languages

Acknowledgements

I am grateful to my supervisor, Professor Jukka Paakki for guiding me through the long process of post-graduate studies and my thesis research. He has given me good advice on many aspects related to my studies, research, publications, and academic life in general. He always carefully read and commented my writings. We have also published papers together and I have very much enjoyed working with him.

I have carried out most of this research while working at the Department of Computer Science at the University of Helsinki. The department, headed by Professors Martti Tienari, Esko Ukkonen, Timo Alanko, and currently Jukka Paakki, has provided excellent working conditions and a supportive and friendly atmosphere. For example, the library of the department has been a very important source of information for my studies and my research work.

I have had the privilege to work with many colleagues during my years at the department. My thesis research was mostly solitaire work but I highly appreciate the co-operation with the colleagues in teaching and when working on other research topics. I have learnt a lot during the years from many people and I want to thank all the people of the department for this.

The National Technology Agency of Finland (TEKES), Helsinki Graduate School in Computer Science and Engineering (HeCSE), and the Academy of Finland have financially supported this work. Thanks to their support, I was able to concentrate on my thesis research almost full time during the first five years of my doctoral studies. I also thank my current employer Nokia for providing me the time needed to finish my thesis and for financing the publication of the thesis.

Espoo, January 27, 2002

Antti-Pekka Tuovinen

Contents

Chapter 1

Introduction

Graphical notations are important tools in a software engineer's toolbox. For instance, UML [RJB99][Obj99] diagrams are a common visual form of expressing and communicating design information; they are used for modeling, testing, specifying, and programming of software systems. This thesis proposes practical means for specifying and implementing diagrammatic graphical notations, or, visual languages, for software engineering.

In this chapter, we first introduce the concept of visual language. Then, we formulate the research problem, present our solution, and enumerate the contributions of this work to the field of visual language research. After that, we survey related work. Finally, we describe the structure of the rest of this thesis.

1.1 Visual Languages

With 'visual languages' we mean notations that employ graphics (icons, diagrams) to present information in a two or more dimensional space. The term 'textual language' is reserved for languages characterized as linear, one-dimensional streams of symbols. Of course, practical visual languages have both graphical and textual elements.

Visual languages are used in human-human and human-computer communication and interaction. In a broad sense, these languages include [NH98]:

- programming languages whose syntax is based on visual representations (visual programming),
- computer visual languages designed to convey aspects of underlying computation or its declarative specification (software visualization and algorithm animation), and

– human visual languages that seem amenable to formalization and computer implementation (diagrammatic representation and reasoning).

Several taxonomies have been developed to characterize and classify visual languages. For instance, Marriott & al. build a Chomsky-style grammar hierarchy of visual languages based on the expressiveness and the parsing complexity of the languages [MM98a]. Following the classical approach of language theory, they develop a hierarchy of progressively more expressive classes of constraint multiset grammars (CMGs) and show how other grammar formalisms for visual languages can be reduced to CMG grammars. Here the presumption is that the essential and distinctive characteristics of visual languages can be described grammatically and particularly by CMGs. The grammar-based classification emphasizes the *computational* properties of visual languages.

Narayanan & al. focus on the *human-computer interaction* perspective of visual languages [NH98]. They propose a conceptual framework for analyzing and developing visual languages usable by both computers and humans. The framework includes a model of visual languages and a taxonomy based on the different issues expressed in the model. Figure 1.1 shows the model that has three objects of interest: a computational system, a cognitive system, and the visual language. The language may have a formal specification and it is materialized in the visual representations used for communication. The visual display is the interface where the information encoded in visual representations appear. For communication to happen, three things are required: comprehension, inference, and feedback. On the computational side, communication implies processes like visual parsing, interpretation or compilation, and program execution. On the cognitive side, this means visual perception, comprehension, and reasoning with the information. Both systems construct and manipulate visual representations on the visual display to convey the results of their processing to each other. In this model, the success of a visual language depends on two things: the *computational tractability* and *cognitive effectiveness* of the language.

Based on the model, Narayanan & al. derive a taxonomy that has three major categories: (1) representation of information, (2) cycle of interaction, and (3) evaluation. The first category deals with the contents of the visual display. The central issues are what is to be represented, how to represent it, and how to associate the representation with the represented things (the application domain). The second category models the usage of a visual language by considering the cognitive and computational processes that take place in one cycle of activity during an episode of human-computer interaction. The third category addresses the issues of evaluating visual languages for their computational efficiency and cognitive effectiveness. Each major category has further subdivisions that can be used to elicitate a detailed characterization of a visual language. This human-computer interaction perspective gives a more holistic view of a visual language as a communication system than the grammar hierarchy -based taxonomy. It also acknowledges the usability aspects of visual languages and not just their computational properties.

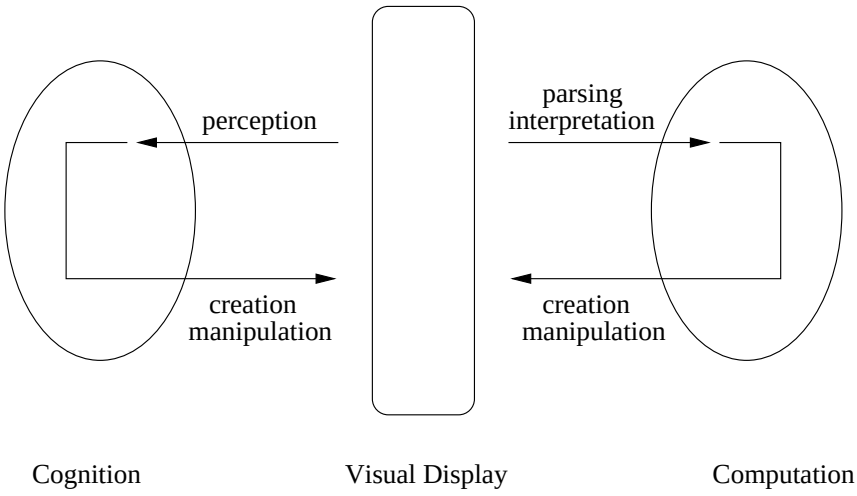


Figure 1.1: Model of Visual Language (from [NH98]).

Our work focuses on diagrammatic visual languages, as found in software engineering, and their computer implementations. With implementation we mean the development of processors to automatically analyze diagrams and the development of graphical editors for constructing the diagrams. In this section, we first describe the characteristics of graphical notation in more detail. Next, we discuss the role of visual languages in software engineering. Then, we survey the work done in the fields of specifying and implementing visual languages.

1.1.1 Characteristics of Graphical Notations

The power of graphical presentation lies in the ability to use two (or three) dimensional space for arranging graphical symbols to show relationships between the domain objects denoted by the symbols. For instance, in an engineering diagram, the symbols representing closely related domain objects may appear close to each other, contained within one or the other, or visually linked to each other by lines. The different ways of representing relationships can be used simultaneously in the same diagram so that each geometric or topological relation maps to a different semantic relation in the application domain. Also, other visual aids can be used: icons that appear as such in the application domain, color, lines in different styles, animation, and so on. In comparison with graphical notations, textual specifications are basically *linear* descriptions of the domain of interest. They rely on hierarchical structure, repetition, and symbolic linking (reference by name) of the domain objects for specifying the interesting relationships between them.

The effectiveness of graphical notations is based on the remarkable image processing and pattern recognition capabilities of the human brain. However, graphical

notations have also their limitations. Graphical representations generally suffer from low density of information content when compared to semantically equivalent textual presentations [Nic94, Whi97]. On the other hand, the complexity of the relationships that are displayed in a graphical presentation increases the density [Nic94] and effectiveness [Whi97, p. 124] of the presentation. Also, hybrid presentations that combine text and graphics can reach the density levels of pure textual presentations [Nic94].

Graphical representations seem to be the most effective when there is a direct mapping from the graphical symbols and the layout to the application domain [Ray91]. For instance, consider a tourist map of a city. The map is an example of a graphical presentation with a direct and a *semantically dense* mapping [Ray91] from the graphics to the application domain (the city). In the map, the domain objects (hotels, shopping areas, museums etc.) are represented by iconic symbols and the distances between the places on the map are directly related to the geographical distance of the actual places in the city.

The city map is an example of an *analog language*. The distances on the map translate into a continuous real-world metric. On the other hand, visual software engineering languages are largely *notational*: they deal with discrete values, they do not have the dense semantic mapping of analog languages, and the domain objects themselves are non-visual and therefore have no natural graphical representation. Notational languages are also called *diagrammatic languages* [NH98, p. 90]. Of course, a visual language can have both notational and analog features.

The analog—notational dimension cannot be used as the only factor when evaluating the effectiveness and suitability of a visual language for certain practical purposes [Ray91]. The classification framework by Narayanan & al. described above gives a more comprehensive basis for the evaluations of visual languages.

1.1.2 Visual Languages in Software Engineering

The two main categories of visual languages used in software engineering are visual *programming* languages and visual languages for *specifying* and designing software. Visual programming means constructing graphical representations that can be executed by a computer either directly (interpretation) or indirectly by a translation to a non-visual (textual) program. Visual specification languages are used to document the requirements and/or the design of a software system. The construction (drawing) of the visual specification can be an active part of the design process or it can take place as a reverse engineering activity after the design is stable.

Visual programming is a controversial issue. The following statement on the prospects of visual programming made over a decade ago by the distinguished software engineering authority Fred Brooks is often quoted:

> “A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graph-

> ics to software design[. . .] Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will." [Bro87, p. 15]

Indeed, fully visual general purpose programming languages have not been very successful. Experimental studies show that the benefits of visual programming languages over textual languages are limited at the best [Whi97]. On the other hand, visual tools for building GUI applications, like Visual Basic^{TM}, are used everywhere. An example of a truly visual and successful programming language is LabVIEW^{TM} which is a visual data-flow programming language for building graphical applications for controlling laboratory and manufacturing equipment [Nat99]. The common thing about the successful visual programming tools is that they have a rich graphical vocabulary that maps directly to a specific domain. The tools also employ the powerful metaphor of assembling a system from components. Furthermore, the transition from programming to running a system is smooth and quick which gives immediate feedback to the programmer.

The traditional data and algorithms -oriented programming does not lend itself naturally to graphical form [Bro87, p. 12]. After all, most of the computation is sequential and there are no natural graphical representations for symbolic computations (except mathematical and logical formulas). Also, the low density of a graphical representation is an issue. However, as argued above, domain specific visual languages can show complex semantic content concisely by representing domain specific high-level concepts in a visually compact form. For instance, Roberts & al. see a visual builder tool as the final state in the evolution of an object-oriented framework [RJ97]. The visual builder tool addresses one specific task: the configuration of an application derived from a black-box framework by instantiating and connecting the components that make up the application.

Brooks' skepticism on large scale visual programming is justified. However, visual representations are useful in conveying information on the design of software systems. It is rare to see software documentation without *any* pictures. Usually, figures are used to show structural relationships and interaction patterns between the components of a software system.

A prime example of a visual software engineering language is the Unified Modeling Language (UML) [RJB99, Obj99] which is a visual language for *modeling* and *specifying* software intensive systems. UML comprises eight different kinds of diagram notations, or, sublanguages. For instance, UML package diagrams are used to specify the decomposition of a software system into modules, and class diagrams are used to specify the structural relationships between the components in the modules. There are also notations for modeling the interactions of components and the physical deployment of the system into computing nodes. In addition to static structural diagrams, UML has sublanguages for specifying the dynamic properties of systems. For instance, statechart diagrams are used for specifying the event-driven behavior of system components and activity diagrams can describe the process flows in a system. Also, class diagram elements

can be adorned with textual constraints written in OCL (Object Constraint Language). The constraints specify restrictions on the attributes of classes, and the relationships between them. In the UML specification, OCL is used to express the well-formedness rules of UML models.

In addition to modeling software systems, UML is advocated as a visual language for *constructing* systems. The idea is that CASE tools can automatically generate software from UML models. In practice, however, UML is mainly used as an analysis and design tool during the development of software systems and/or for the post-development documentation of the design. Studies on general CASE tool usage [LC98, PC98, MI99] support this view of the role of visual software engineering languages.

Although UML is promoted as a general purpose modeling language, it still has a specific domain: modeling the architectural design of software systems. UML has a large graphical vocabulary for representing different aspects of software systems and it allows textual OCL expressions in addition to the graphics. Furthermore, it can express complex relationships between graphical elements. Hence, the popularity of UML is not surprising. However, the language does not impose rules on the layout of diagrams nor on the partitioning of large UML models into separate diagrams. In addition to the UML language reference, guidelines are needed on how to partition large diagrams, how to draw diagrams on different levels of abstractions, and how to order and organize diagrams according to the flow of the development process and according to the information needs of the different stakeholders of the system under development [BRJ99, McG99].

The UML language specification gives freedom for users and tool vendors concerning the visual representation of UML diagrams. The standard has rules for the general appearance of the diagram elements and even rules for font sizes and typefaces but the use of visual effects is mostly left to the discretion of users. The standard warns against overexploiting special visual effects and stereotyping (customization) in order to prevent users from inventing new languages on their own. However, in the light of the discussion above, users of UML should be encouraged to use semantically meaningful layout and other graphical effects for conveying domain specific information more effectively. For instance, Coad [CLL99] has suggested using color in the modeling of business systems in order to make the system-wide roles of the model elements clearly visible. Consequently, if layout and color are considered semantically meaningful properties of UML diagrams, the graphical properties in question should be part of the meta-model of UML.

1.1.3 Specifying Visual Languages

Graphical notations are languages in the same sense as textual notations. They have primitive graphical symbols, conventions for combining instances of the primitive symbols into more complex graphical constructs, and commonly accepted interpretations of the meaning of the pictures thus formed.

The bulk of the work done in visual language theory approaches the problem of specifying visual languages from the viewpoint of general language theory. The following classical definition of a visual language underlies most of the approaches:

> "[. . .] we will regard a visual language as some set of diagrams which are valid "sentences" in that language. Such a diagram is a collection of "symbols" in a two or three dimensional space. Which sentences are valid depends on spatial relationships between the symbols. The meaning of a sentence is, in general, constituted by the graphical symbols used in the sentence and by their spatial arrangement." [MM98b, p. 2]

When considering the model of a visual language depicted in Figure 1.1, the classical viewpoint concentrates mainly on the computational aspects of visual languages in order to develop methods for the automatic processing of visual languages. Here, the main problem is recognizing and parsing pictures efficiently. However, there are also approaches for specifying visual languages that try to formalize the interaction aspects of visual languages [BCLM98].

Marriott & al. provide an extensive survey of visual language specification and recognition in [MMW98]. They identify three main approaches to the specification of visual languages: the grammatical approach, the logical approach, and the algebraic approach. The grammatical approach extends one-dimensional string language grammars to multidimensional languages with spatial relations between primitive tokens. When compared to string languages, the generative methods of the grammatical formalisms for visual languages rewrite sets of objects rather than sequences of symbols; they also rewrite geometric and topological relationships between the objects. Consequently, parsing languages specified by such grammars has been a very active field of research. The grammatical approach has the longest history in visual language specification and covers now a variety of formalisms.

The logical approach uses first-order logic or other forms of mathematical logic with roots in artificial intelligence. The logical approaches are usually based on spatial logic which axiomatize the different possible topological and geometric relationships between objects. The logical approaches have the advantage that the same formalism can be used to specify both the syntax and the semantics of a diagram.

The third major approach to visual language specification is to use algebraic specifications. They consist of composition functions which construct complex pictures from more simple picture elements. The process of parsing means finding a function sequence that constructs the picture. Semantics are handled by defining algebraic specifications for both the diagrams and the application domain and by providing morphisms between the two algebras.

The number of visual language specification formalisms is surprising. As noted by Wittenburg, it is almost as every new researcher entering the field comes up

with a new specification formalism [Wit95]. Several reasons can be identified to understand this.

First, the field of visual communications is very broad and there are many different kinds of visual languages. Therefore, it seems to be very difficult to find a single formalism, a unified theory that would cover the vast range of visual languages. Also, until recently [MM98a], it has been difficult to compare the expressiveness of the existing specification formalisms.

Second, parsing pictures is computationally expensive; the 'naturally occurring' visual languages display a high degree of ambiguity and context-sensitivity as parsing is concerned [MM98a]. This has led to the development of many specification formalisms that are suitable for just a limited range of visual languages but that have practical parsing algorithms.

A third aspect (related to the second point) is that the distinction between the syntax and semantics in visual language specification is not as clear as with textual languages. For example, in a language for specifying object-oriented class hierarchies, the inheritance graph should be acyclic. In textual programming languages, constraints like this are typically semantic and not syntactic properties. On the other hand, in the specifications of visual languages, there is a tendency to express such rules on the syntactic level of the specification. The reason for this may be that the relationhips constrained by the rules have an explicit graphical representation (a connection line, for instance). Hence, the syntactic formalisms tend to be based on powerful declarative models of computation like constraint satisfaction and logic programming.

The focus of our work is on specifying and implementing artificial, or *formal* (as opposed to natural) diagramming languages. For example, a formal specification of a visual software engineering language is useful in two ways. First, it gives rules that help engineers to correctly map the diagrams made by others to the domain of the language. This reduces the need for textual explanations accompanying the graphics. Second, if the specification formalism includes practical methods for analyzing the expressions of the language, it helps the construction of computerized tools for creating correct diagrams and tools for automatically processing the information contained in the diagrams. When developing the language processing tools, it is easier to reuse declarative, high-level specifications (even in copy-paste -style) than program code. Additional argumentation for using formal specifications can be found in [MMW98, pp. 62–63].

Although several specification formalisms have been developed for visual languages, they have not found use outside of the visual language research community. In practice, most visual specification and programming languages lack *any* formal syntactic or semantic definitions [MMW98, p. 58]. The only exceptions are standardized industry-level visual languages like UML. The official specification document of UML [Obj99] describes the conceptual structure and meaning of models[1] that can be expressed in UML (semantics) and the graphical notations

[1] In UML parlance, *model* means a system description. A model can comprise several different kinds of diagrams on different levels of abstraction.

(diagram types) used to express the models (syntax). UML is not a simple nor a small language which can be seen from the size of the eight-hundred-page specification document.

The core of the UML specification is the semantic description of the sublanguages of UML. The semantic description of a sublanguage consists of three parts: the abstract syntax showing the conceptual structure (meta-model) of the language (expressed in the class-diagram notation of UML), a set of well-formedness rules (in OCL) to supplement the meta-model, and an explanation of the meaning (interpretation) of the meta-model in English prose.

The notational guide (syntactic specification) of the UML sublanguages relies on English prose and graphical examples that describe the primitive graphical elements of the diagrams and explains the rules for composing primitive elements. An important part of the syntactic specification is the mapping from the notation (graphics) to the meta-model (semantics) of the language.

The UML specification does not use any grammatical or other formalisms for the syntactic definitions of the notations. Because syntactic descriptions are given in prose and by graphical examples, they are often incomplete. Hence, the semantic specification must be consulted in order to understand the incomplete and confusing parts of the notational guide. For a person implementing the language, this means tedious mapping between the semantics expressed in UML and the graphics.

A more rigorous syntactic specification would make it easier to approach the UML standard when trying to implement the language. Like in the development of textual languages, having separate lexical, syntactic, and semantic specifications helps to divide the implementation work of a visual language into well-defined subtasks. Using this approach, the lexical and syntactic specification would define the graphical appearance completely and the semantic specification would add the well-formedness rules that cannot be conveniently expressed in the graphical syntax.

1.1.4 Implementing Visual Languages

The implementation of a visual language can mean a variety of things. A graphical drawing tool (editor) may support a visual language by providing the possibility to create, manipulate, and compose the primitive objects of the language on a drawing screen. This kind of tool is merely a dedicated editor for the visual language. For instance, most UML tools in the market belong to this category. More advanced tools provide ways to enforce the syntactic and semantic correctness of diagrams. For example, the Visio^{TM} drawing tool for business and engineering diagrams [Vis99] supports some of the sublanguages of UML and provides the possibility to check the semantics of UML drawings. Finally, there are true CASE tools that provide simulation and code generation based on the graphical models drawn by the user. Of course, visual programming tools must perform a full semantic analysis and interpretation of their graphical input.

As noted above, implementations of visual languages are usually not based on rigorous syntactic or semantic modeling. They do not use parsing techniques to analyze their input. Instead, the usual way to implement a visual language is to construct a dedicated graphical editor that enforces a syntax-directed way to construct diagrams. This means that the tool maintains an internal semantic model of a diagram being edited and at every editing step checks the consistency of the model. Editing actions leading to inconsistent states are rejected. In this way, the user of the tool cannot draw incorrect diagrams. For instance, the Rational Rose -tool prohibits the user from drawing generalization relationships between other than same kinds of generalizable types (syntactic rule). In class diagrams, the tool does not allow the user to enter two attributes with the same name in the same class (semantic rule).

The syntax-directed style of interaction is good for beginners who are learning a visual language and learning how to use a drawing tool for the language. Also, syntax-directed editing is acceptable for documenting a stable design because the order of entering the graphical input does not really matter. The problem is that syntax-directed editing is awkward when the user wants to radically restructure a diagram. This need occurs frequently during the actual design phase of the model represented by the diagram. As noted by Jarzabek & al., experienced users feel frustrated about design tools that push their own ways of doing things instead of providing an unconstrained environment for creative design work [JH98]. Hence, pen and paper are still favorite tools for many.

In unconstrained, free-order editing modes, error handling becomes of prime importance. If a tool allows incomplete sketches to be drawn, it should have the ability to detect and report any errors it finds when later checking the drawing. If a parsing-based approach is used to check diagrams, the parser should report as many errors as possible at one parse. Also, the design of the graphical interaction of error handling is important. The graphical environment of a visual language should provide possibilities for informative and highly interactive error reporting. Incremental parsing and analysis is one possible way to address error handling issues [CM95].

The lack of formal syntactic and semantic specifications has also other effects on tools. For instance, there has traditionally been great variation among UML tools in what they actually consider to be a “correct” UML diagram. Also, the completeness and depth of semantic checking varies considerably. The OMG standard of UML will hopefully help tool vendors to make their products to agree on the properties of the language.

Of course, a major reason for formally specifying visual languages is to facilitate the automatic generation of at least part of an implementation of a visual language. Currently, implementations of commercial products are based on ad-hoc solutions. More general techniques do exist, however. There are several object-oriented frameworks that address the issue of implementing graphical editors [Jin90, VL90, Bra95], and research prototypes of visual language generation systems have been developed. We will review existing visual language generation systems in Chapter 8 where we discuss the work most closely related to ours.

1.2 Research Problem and Contributions

The general goal of this research is to develop *a practical specification and implementation technology for diagrammatic visual languages used in software engineering*. The specific requirements of the technique are:

- the technology should support the development of diagramming languages (e.g. UML),
- it should be based on formal grammar,
- it should make unconstrained editing of diagrams possible, and,
- it should make language implementations open, extensible, and reusable.

In order to achieve the goal, several problems had to be solved. The main research problems have been:

- representing visual language grammars as object-oriented frameworks,
- choosing and adapting a grammatical model in order to represent the graphical syntax of typical diagramming languages, and
- error handling in visual language parsing.

Research has also been done on automatic *source-to-source translation of visual languages*, which is a closely related subject. In the following, we motivate the research, describe the research hypothesis and rationale, and summarize the results of the research.

1.2.1 Motivation

The initiative for this research came from the development of the communication protocol engineering language KANNEL [GHLP95] which has a visual syntax as an alternative for a purely textual representation. The early work by Järvinen [Jär92] on the implementation of visual languages had shown the field to be rather immature. Consequently, the implementation of the visual version of KANNEL was based on ad-hoc techniques. The development of visual KANNEL was in sharp contrast to the implementation of textual KANNEL which was based on the well-established compiler construction techniques. Clearly, the development of visual languages could benefit from a more scientific approach.

A study of the literature soon revealed the plethora of formal methods for the specification of visual languages. On the other hand, as noted in Section 1.1.3, the existing formal techniques have had little impact on engineering practices. In [Wit95] and [MMW98, p. 69], the authors identify possible reasons for this. First,

there is a mismatch between real-world problems and the proposed technology. That is, there is no empirical evidence of the suitability of the formal techniques to the implementation of real-world visual languages. Second, the literature suffers from high fragmentation which makes the field hard to approach for practitioners. Third, basic research does not pay enough attention to real-world engineering problems in implementing visual languages. So, it seemed as an interesting and a challenging task to try to apply one of the existing grammatical specification methods and the related parsing technique for the specification and implementation of large, widely used visual languages, e.g. UML. The work would have a clear focus on the engineering aspects. Indeed, there seemed to be no point in inventing yet another specification formalism.

Given the success of formal grammars in the implementation of textual programming languages and our experience in compiler construction, it seemed natural to concentrate on the grammatical approach for the specification and implementation of visual languages. Here, the technical challenge was in presenting a grammatical model as an object-oriented framework.

Recently, the visual language research community has also recognized the need for practically significant applications of formal visual language theory [CBL+99, MS99, p. 58]. Also, visual software engineering languages have been pointed out as a potential new application area for visual language research [MS99]. Our work is well in line with these directions.

1.2.2 Hypothesis and Rationale

Formal Specification of Visual Languages

In Section 1.1.3, we already elaborated on the reasons for formally specifying visual software engineering languages. In summary, the purpose of a formal specification of a visual language is to give an unambiguous syntactic/semantic description of the language which can be used to automate (at least part of) the implementation of the language. An implementation technique that is based on a formal grammar and parsing will add rigour and structure to the development of visual languages. It will help in keeping separate the concerns of editing a diagram and analyzing it. Also, free-order editing of visual programs (not dictated by some syntax-directed editor) is one of the main motivations for the use of grammars and parsing in implementing visual languages.

Free-order Editing by Visual Language Parsing

From the start of our research it was clear to aim at supporting free-order editing of diagrams. That is, an implementation of a visual language consists of a dedicated editor and an analyzer/parser. The editor supports the basic vocabulary of the language and it supports the construction of more complex expressions in any order the *user* wants; the analyzer then checks the drawing transforming it into an

internal representation (parse tree or graph) for additional processing. The idea was to do the parsing off-line (not incrementally) in order to limit the technical challenges involved.

The free-order approach is motivated by practical experience in implementing and using graphical diagramming tools. For instance, the general graphical editor Visio^{TM} [Vis99] is cheap and extensible, it has very good editing capabilities, and it supports a wide variety of diagramming notations. Dedicated CASE tools cannot provide the same level of flexibility in editing visual language expressions (programs). Paradoxically, in many organizations, object-oriented CASE tools are often used as mere drawing tools. The study by Lending and Chervany [LC98] indicates that the more advanced features of CASE tools like model analysis (checking) and model transformations (code generation) are seldom used. Hence, it seems reasonable to separate model construction (drawing) from model analysis and model transformation. This makes it possible to combine the flexible editing and the rigorous analysis of diagrams into an explorative design style which does not constrain the editing of diagrams but still offers a way to validate the diagrams according to the syntax and semantics of the modeling language. Also other researchers have recognized the value of free-order editing, see e.g. [RS97, p. 29], [Ser95], and [MV95, Min97].

Error Handling in Visual Language Parsing

An effective error handling technique is absolutely necessary for any visual language parser that is used to facilitate edit-and-compile style visual programming. Our early survey of visual language theory showed that little was known about error handling in visual language parsing (see also [MMW98, p. 66]). The parsing algorithms suggested for visual languages were mostly recognizers. The problem with recognizers is that if an input fails to satisfy the rules of the language, the algorithm cannot tell *why* it failed. For our application of visual language parsing, this is unacceptable. As a minimum requirement, the parser should be able to indicate the piece of input that caused the failure. Further, the parser must be able to recover from syntactic parse errors in order to process as much input as possible during one parse. Error handling is, or should be, one of the major concerns of any practical programming environment, visual or textual.

The work on error detection and recovery in parsing string (textual) languages has shown that general mechanisms that apply to all kinds of languages and error situations are hard, if not impossible, to develop. The problem of automatically correcting errors is even more difficult. Consequently, corrective error recovery techniques are heavily heuristic and language dependent. In practice, however, the techniques used by compilers are less ambitious. Our goal was to achieve a level of error recovery comparable to the standard compilers of the main-stream textual programming languages. Accordingly, we expect a typical programmer to be an experienced user rather than a newcomer. In our opinion, it is not the task of an error handling mechanism to teach software developers how to use

a language—it is the task of (human) trainers and (machine) wizards or other embedded mentoring agents.

Framework Technology

Object-oriented application frameworks are promoted as a technology that provides a high degree of reusability and extensibility of software assets [FSJ99a]. A framework captures the commonalities of a set of applications that belong to a certain domain in the form of an implementation skeleton. It embodies the most significant architectural design decisions that the perceived applications in the domain must conform to.

In many cases, the skeleton provides the main control of the application and provides extension points for configuring and adding the variable features of the applications. The user of the framework provides the configuration information and concrete implementations for the underspecified or missing parts in order to derive a working application from the framework.

From the engineering point of view, the grammar-based approach for specifying the syntax of a visual language and automatically producing (by a compiler-compiler) a language analyzer (parser) offers obvious benefits. Object-oriented frameworks have been successfully developed and used for implementing graphical editors for diagramming tools. Using these frameworks offers the chance to tap into the state-of-the-art in the implementation of graphical editors. Ideally, we would like to combine the benefits of both the framework- and grammar-based approaches in the development of visual languages.

The coupling of the editor part and the analyzer part is a central architectural issue in implementing a visual language. The (white-box) framework-based implementation of the editor means that the internal object structures of the editor that comprise the visual data (program) to be analyzed can be made directly accessible to the analyzer part. This makes it straightforward for the analyzer to get its input data and to provide feedback of the results of the analysis.

Source-to-Source Translation

The problem we address in this part of our research is the transformation between graphical diagrams. Current diagram editors for software engineering notations are usually implemented with ad hoc solutions on a weak methodological foundation. This makes it hard to develop sophisticated diagram manipulators, such as meaning-preserving transformators between two different styles of diagrams. For instance, consider transformations between class diagrams in UML [Obj99] and corresponding class diagrams in OMT [RBP$^+$91].

We consider diagram transformation as a translation process between two visual languages. By this interpretation, we can adopt the powerful toolset developed for

(source-to-source) translation of textual languages into use for the processing of visual languages.

Now, a transformation from a diagram given in a visual notation into a (corresponding) diagram in another visual notation can be considered as a syntax-directed translation, provided that both the source diagram and the target diagram can be represented as a tree over a source grammar and a target grammar, respectively.

To address this problem, we wanted to develop a solid method for the transformation between diagrams, or more generally, for the source-to-source translation between two visual languages. The main ingredients of our method are a mapping between grammars for the two languages, and considering translation as a parse tree transformation process. These are well-known techniques in the domain of textual languages.

1.2.3 Contributions

Our work has a theoretical and a constructive part. From the viewpoint of visual language theory, our work has two main contributions: an original treatment of error handling (error detection, reporting, and recovery) in off-line visual language parsing, and the source-to-source translation of visual languages. The latter is joint work with prof. Jukka Paakki, who is the designer of the actual translation algorithm.

We have substantially extended the powerful grammatical model for multidimensional languages called *atomic relational grammars* [Wit96]. We have added support for meta-language expressions that denote optional and repetitive right-hand-side elements. Also, we have extended what basically is a context-free grammatical model to take into account a limited amount of contextual information in order to better represent general graph structures at the syntactic level.

In [MM98a, p. 167] Marriott and Meyer argue that the use of specification methods that have efficient parsing methods rules out context-sensitive visual languages. In the case of diagrammatic languages, this means that general graphs cannot be specified at the syntactic level. However, our work shows that these kinds of properties of diagrammatic languages are not a major issue and they can easily be dealt as semantic checks after the parsing phase. There are typically many kinds of semantic checks that have to be performed after parsing, anyway.

The main product of the constructive part of our research is the VILPERT (VIsual Language exPERT) system. It is an object-oriented Java framework for implementing visual languages. Implementing a visual language with VILPERT means generating a language analyzer based on a formal syntactic specification and implementing a graphical editor for manipulating the visual programs. The framework has a language specification sub-framework that is based on our extended version of atomic relational grammars. The model has a parsing algorithm for recognizing the sentences of a visual language according to its grammar. The parser

produces a parse tree from a correct input, and the semantics of the source program is defined operationally by operations on the parse tree. The graphical editor is derived from a Java version of the *HotDraw* framework [Bra95] [GE96][jho00] for general graphical editors.

In the editor side, we have added support for the notion of composite figure containers that facilitate the drag-and-drop style of moving figures into and out of containers and the construction of deeply nested graphical structures.

The VILPERT framework provides a clean separation of the concerns of the graphical editing and the interpretation of diagrams both from the architectural and the usability point of view. The user draws the diagram in free order (not dictated by a syntax directed editor) and then invokes the language analyzer to interpret the drawing. The analyzer informs the user about any errors it finds during parsing and semantic processing. This approach to visual language implementation makes it possible to combine the sketching and the checking of diagrams into an explorative style of constructing visual programs.

Separating the two concerns of editing and analyzing reduces the software complexity of a tool that implements a visual language. For example, the correctness of a diagram does not have to be constantly enforced during editing, syntactic rules do not have to be enforced by hand-coded checks, and it is natural to maintain a clear separation between representation (graphical objects) and meaning (semantic or domain objects). Also, the usability aspects of the editor are not compromised by the need of maintaining a consistent model during editing: the editor can provide all the freedom of graphical editing that users want. Furthermore, because VILPERT is a framework, tools produced from it can be open for extensions, modifications, and they can share a common pool of reusable software components.

We have validated our solution by implementing three visual languages that represent typical notations used in software engineering (UML structural diagrams, UML statecharts, and flowcharts) and other (toy) languages. The syntaxes of the languages have been specified by extended atomic relational grammars using the grammar framework of VILPERT and the editors for the languages have been derived from the editor framework of VILPERT. The editors provide syntax-free editing of diagrams that are analyzed by parsers produced automatically from the grammars of the languages. The implementations of the visual languages show a high degree of reuse: the language (application) specific parts of the implementations is less than 20% of the total size of the applications.

Publication of the Results

The initial design of the visual language analysis framework of VILPERT was published in [Tuo98b] and an overview of the whole system in [Tuo99]. Error handling was addressed first in [Tuo98a] and then in revised and deepened form in [Tuo00]. The work on source-to-source translation was published in [PT98],

where Jukka Paakki was the main author. All the other papers are single-author work by Antti-Pekka Tuovinen.

1.3 Thesis Outline

In Chapter 2, we introduce the formalism of atomic relational grammars for the purpose of reference. Then, in Chapter 3, we discuss the use of atomic relational grammars for specifying visual languages. We identify the limitations of the grammatical formalism and the parsing algorithm and propose several enhancements to both.

In Chapter 4, we describe our solution to the problems discussed in Chapter 2. We define the formalism of extended atomic relational grammars (EARG) and describe our changes to the parsing method. We continue the presentation of the extensions in Chapter 5, where we describe our technique of handling syntax errors in parsing visual languages that are specified by EARG grammars.

In Chapter 6 we present the VILPERT framework. We describe the design of the framework, explain how it is used, and report our experiences in using VILPERT in implementing visual languages.

Source-to-source translation is discussed in Chapter 7. In Chapter 8, we review the related work. Finally, in Chapter 9 we present our closing remarks and discuss further directions for the research. Readers who are not familiar with the implementation of visual languages may find it helpful to glance over Chapter 8 before reading Chapters 2– 5.

Chapter 2

Atomic Relational Grammars

Atomic relational grammars (ARG) provide a good compromise between the expressiveness of the specification formalism and the simplicity of the grammar formalism and the associated parsing algorithm. Therefore, we have chosen ARG as the grammar formalism used in VILPERT.

In this chapter, we describe ARGs as a reference to the reader. First, in Section 2.1, we present the grammatical formalism of atomic relational grammars. Then, in Section 2.2, we describe Wittenburg's parsing algorithm for ARGs. Our description of ARGs and the parsing algorithm are based on [Wit96].

2.1 The Grammatical Formalism

2.1.1 Relational Languages

Relational grammars (RG), a superclass of atomic relational grammars, belong to a family of constraint-based grammatical models for multidimensional, e.g. visual, languages. In [MMW98], the family is called *attributed multiset grammars*. In these approaches, grammar productions rewrite sets or multisets of symbols which have geometric and sometimes semantic attributes associated with them. Productions have constraints over the attributes of the symbols in the right-hand side and the constraints control rewriting of the symbol sets, that is, application of the productions.

The sets of expressions that can be generated (or recognized) by RGs are characterized as sets of *relation tuples* comprising references to a set of *objects*, according to the normal mathematical notion of relation. In the case of visual languages, the objects are graphical objects (terminals, icons) without discernible structure or composite objects (nonterminals) consisting of other objects. The relations denote geometric relationships (such as above, left-of) or some other basic form of relationship in the graphical language, for instance that two objects are associated

by a connecting line. The (infinite) set of the (finite) expressions generated by a relational grammar forms a *relational language.*

For example, the production

Block → *rectangle text*
inside(text,rectangle)

specifies that a *Block* nonterminal consists of the terminals *rectangle* and *text* that satisfy the relational constraint *inside(text,rectangle).* In other words, a *text* and a *rectangle* form a *Block* only if they are in the relation *inside.*

The relational constraints in the productions drive the generation (and parsing) of relational languages. The generating relations (like *inside* in the example above) are called *expander relations* and the relations must be binary[1].

In other words, expander relations are *syntactic relations.* In contrast to string-based grammars, where string adjacency is an implicitly assumed relation between the right-hand side elements of a grammar production, RG productions must explicitly state the syntactic relations between the right-hand side elements in the form of relational constraints.

The parsers for relational languages can be divided into two groups: *bottom-up enumeration* and *predictive top-down* parsing. A bottom-up parsing algorithm has the advantage that input objects can be composed into composite objects (and further) by the parser in any order [Wit92]. In other words, the parser is not directed to process (scan) the input objects in any specific order. With predictive parsing, however, some ordering over the input is needed to drive the scanning. The advantages of predictive parsing are that it is more efficient and it makes early error detection possible [Wit92, Wit96]. Also, the proposed predictive parsing schemes are conceptually and technically simpler than the bottom-up methods.

The characterization above of relational grammars and languages is very general and imposes no restrictions on the relations. In practice, however, some restrictions must be placed on the mathematical properties of the relations to develop a practical parsing strategy. For instance, in [Wit92], the relations over the input objects are required to be partial orders.

2.1.2 Atomic Relational Grammars and Languages

Atomic relational grammars form one of the less restrictive subclasses of relational grammars. For example, the syntactic (expander) relations can be symmetric, cyclic or nontransitive.

[1]Expander relations of greater arity would complicate parsing but there is no fundamental reason for having only binary expander relations.

ARGs are used to specify the syntactic structure of a visual language as compositions of graphical objects (terminals) in terms of syntactic relations. The formalism does not have a predefined set of graphical primitives and it does not provide facilities for specifying the structure of terminal symbols in terms of the primitives. That is, there is no concept of an alphabet (like in string grammars) and no concept of regular expressions (or other pattern matching rules) that could be used to specify terminals. So, ARG specifications do not deal with the low-level recognition of graphical primitives. Hence, the word 'icon' could be used in place of 'terminal', as well. On the other hand, this means that ARGs are not restricted to specifying only graphical languages.

ARGs do not provide any means for specifying the semantics of a visual language. That is, ARGs do not enforce any particular interpretation for the sentences of atomic relational languages (the languages generated by ARGs), and they don't have any specific way to attach semantic content to grammar symbols. In [Wit96], Wittenburg mentions the possibility of using semantic attributes with nonterminal symbols, but he does not state how the semantic attributes should be specified and used in the ARG model.

A fundamental issue in relational grammars is whether to allow nonterminals to appear as direct arguments to relational constraints. When using bottom-up parsing, including relational constraints directly on composites is reasonable, but it complicates the definition of RGs as generative systems since the composition-of relation must in principle be reversible. That is, there must be rules for rewriting constraints on nonterminals as constraints on the components. However, what constraints are produced may depend on the context, i.e. the production where the nonterminal appears on the right-hand side (see [TVC94] for examples of such rules). Further, significant problems are introduced for predictive parsing [Wit92]. The alternative is to write grammars that state relational constraints only on terminals in the input set and use syntactic attributes of nonterminals to pass up references to terminal objects in derivations. This is the approach adopted in *atomic* relational grammars.

Definition 2.1 *The class of atomic relational grammars is characterized by the restriction that the arguments of relational constraints must be atomic, i.e. non-composite (terminal) input objects.*

Consider Example 2.1, the productions of a flowchart ARG fragment. Here, nonterminals begin with an upper-case letter and terminals with a lower-case letter.

Example 2.1
Flowchart $\rightarrow$ $oval_1$ *ProcBlock* $oval_2$
connects($oval_1$*,ProcBlock.in)*
connects(ProcBlock.out,$oval_2$*)*
Flowchart.in = $oval_1$
Flowchart.out = $oval_2$

ProcBlock$_1$ $\rightarrow$ *choice ProcBlock*$_2$ *joint*
yesConnects(choice,ProcBlock$_2$*.in)*
connects(ProcBlock$_2$*.out,joint)*
connects(choice,joint)
ProcBlock$_1$*.in = choice*
ProcBlock$_1$*.out = joint*

ProcBlock $\rightarrow$ *rectangle*
ProcBlock.in = rectangle
ProcBlock.out = rectangle

The right-hand side elements of ARG productions are unordered; that is, the order in which the right-hand side elements are written in Example 2.1 is not significant. The right-hand side of a production can be thought as a graph with the symbols as nodes and the relations as edges between the nodes. An *ordering* of a production is a permutation of its right-hand side elements for which the following *connectedness constraint* holds.

Restriction 2.1 *For an ordering of the rhs elements* $x_1 \ldots x_n$ *in a production* $X \rightarrow x_1 \ldots x_n$*, there must exist at least one relational constraint,* $r_e(x_i, x_j)$ *or* $r_e(x_j, x_i)$ *for each element* x_j*,* $j > 1$*, such that* $i < j$*.*

That is, the right-hand side of a production must be connected. Also, when ordered with respect to parsing, each element must be connected to some other element earlier in the ordering (not necessarily the previous element). This requirement implies that ARGs can recognize only connected relation graphs since, for every production, there must be at least one ordering that meets Restriction 2.1.

Figure 2.1 depicts the grammar productions of Example 2.1. Nonterminals are represented by rectangles with rounded corners. The composition of nonterminals is represented by enclosing the constituents inside the rectangles. The arrows represent the spatial relations in the productions. For example, the *Flowchart* production in Figure 2.1 comprises three objects: two terminals of terminal type *oval* and a *ProcBlock* nonterminal. Note that the arrows appear as *relations* in the grammar and *not* as terminal objects. All relations in this example are constraints on individual members of the input set (the relation arcs connect only terminal objects). Consider, for example, the relational constraint *connects(oval*$_1$*, ProcBlock.in)* appearing in the *Flowchart* production. The first argument, *oval*$_1$, is a direct reference to a terminal object. The second argument, *ProcBlock.in*, is an indirect reference to the value of the *in* attribute of an object of (nonterminal) type *ProcBlock*. This indirect reference will eventually be replaced by a terminal object in a successful derivation; in other words, the attribute will be 'grounded'.

Definition 2.2 *The attributes appearing in any of the arguments of relational constraints in a grammar are called* expander attributes*.*

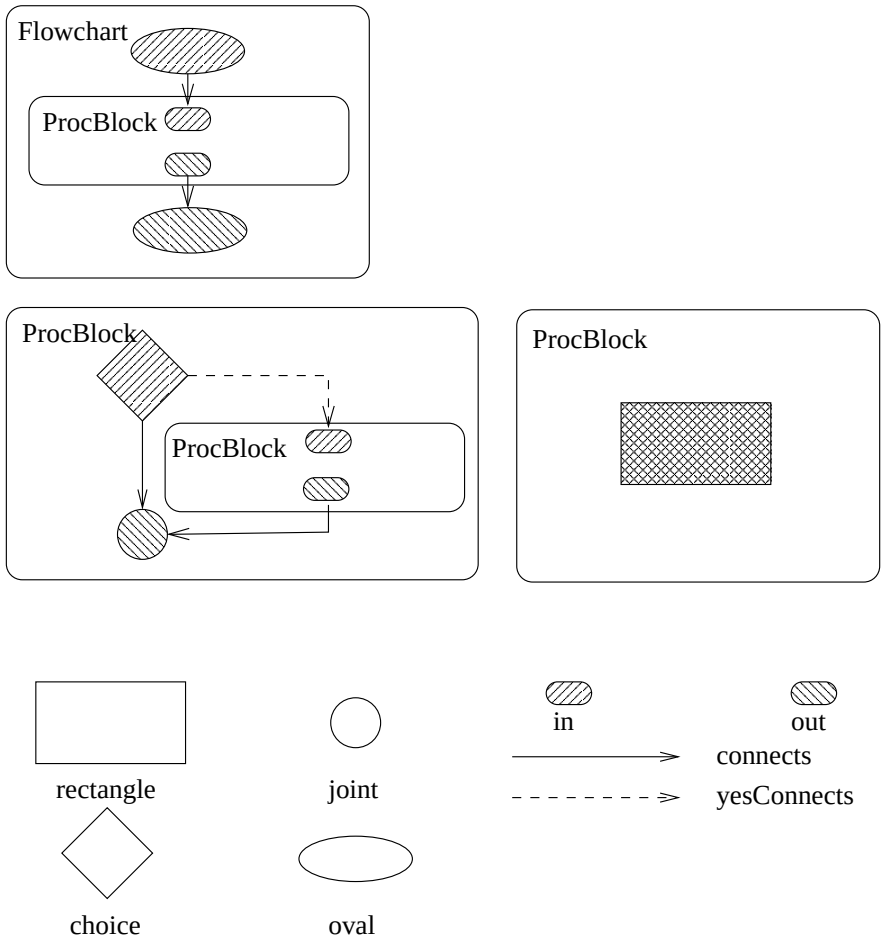


Figure 2.1: Graphical view of the flowchart grammar (from [Wit96]).

In contrast to normal attribute grammars [Knu68] where all the attributes usually have a semantic role, expander attributes are syntactic attributes. In relational grammars, the role of the expander attributes is to drive the scanning of the input, that is they are used to determine the order in which the input is examined during parsing. In the grammar of Example 2.1 *in* and *out* are both expander attributes.

To ensure that expander attributes eventually 'ground out', the productions must pass direct or indirect references to individual members of the input through assigments between the right-hand side elements and the expander attributes of the nonterminal on the left-hand side. Thus, similar to attribute value synthesis in normal attribute grammars, references to actual input objects are passed through chains of indirect references during parsing. In Figure 2.1, for each production, the graphical representation indicates (by fill patterns) which objects are bound to the attributes of the left-hand side nonterminal. For the nonterminals appearing on the right-hand side, the representation indicates the expander attributes referenced by the relational constraints.

The forms $B.attr_x = C.attr_y$ and $B.attr_x = b$ are used to represent attribute assignments in relational grammars. Here, $attr_x$ and $attr_y$ denote attributes, B and C denote nonterminals, and b denotes a terminal. B is the nonterminal on the left-hand side of a production. These assignment expressions are intended to be operationally equivalent to attribute assignment functions in attribute grammars. Unlike general attribute passing, however, arbitrary functions are not allowed on the right-hand side of the assignment operator. This is to ensure the context-free parsing of productions.

As an example of how (references to) terminal objects are passed as attribute values, consider the production *ProcBlock* $\rightarrow$ *rectangle* in Example 2.1. In the production the attributes *in* and *out* of *ProcBlock* are assigned (a reference to) an individual terminal input object; in this case, the terminal object is of lexical type *rectangle.*

Restriction 2.2 *In each ARG production, for every expander attribute used in the grammar, a value must be assigned from the right-hand side to the left-hand side.*

In the grammar of Example 2.1 this condition is met since *in* and *out* are the only expander attributes used in the grammar and every production associates the value of each of these attributes in its left-hand side with a value on its right-hand side.

A visual representation of a *derivation* of a sentence is shown in Figure 2.2. In Figure 2.2, rounded boxes depict nonterminal instances. The *cover* of a nonterminal instance includes all the terminal objects within the rounded box. For example, the cover of the innermost *ProcBlock* comprises of the single rectangle whereas the cover of the enclosing *ProcBlock* includes the rectangle, the diamond and the circle.

In addition to relational constraints, the productions of an atomic relational grammar may also include constraints called *predicates.* Predicates represent additional conditions that the right-hand side symbols of productions must satisfy. However,

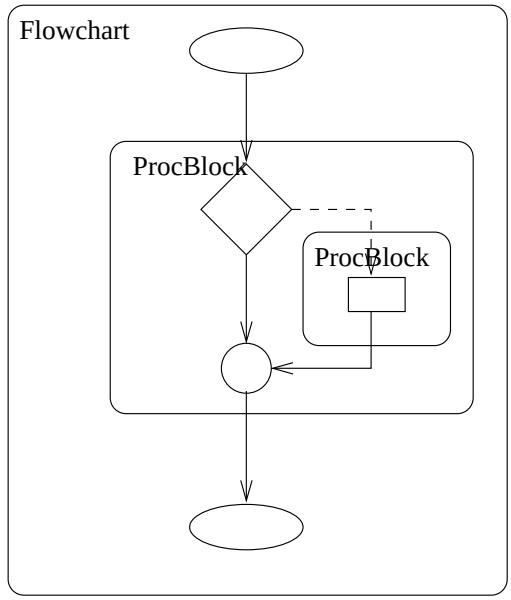


Figure 2.2: A derivation (from [Wit96]).

Wittenburg has not included predicates in the formal definition of ARGs. We also ignore predicates in this chapter, but we discuss them later in Section 3.2.3.

The expressions generable or recognizable by atomic relational grammars are defined as follows.

Definition 2.3 *An indexed multidimensional multiset (indexed md-set)* C *is an n-tuple* $(I, R_1 \ldots R_{n-1})$ *such that* $R_1 \ldots R_{n-1}$ *are binary relations on the indexed multiset of symbols* I.

Here, an index is a (partial) function from integers to members of the set. When writing indexed md-sets, the relations $R_1 \ldots R_{n-1}$ are usually written as a single set $\{r_i(x, y) \mid r_i = \text{relation identifier (name) and } x, y \in I\}$.

Definition 2.4 An *atomic relational grammar* (ARG) is a 6-tuple $G = (N, \Sigma, S, R_e, A, P)$, where

1. N is a finite set of *nonterminal symbols*.
2. Σ is a finite set of *terminal symbols* disjoint from N.
3. S is a distinguished symbol in N called the *start symbol*.
4. R_e is a finite set of relation symbols called the *expander relation symbols*.

5. A is a finite set of *expander attribute symbols*.

6. P is a finite set of *productions* of the form $B \rightarrow \alpha\ \beta\ F$, where

   $B \in N$;

   $\alpha \in (N \cup \Sigma)^+$;

   β is a set of *relational constraints* of the form $r_e(x, y)$ where $r_e \in R_e$ and x, y are either terminal members of α or expressions of the form $C.a$ where $a \in A$ and C is a nonterminal member x_i of α, $\alpha = x_1 \ldots x_i \ldots x_n$ ($x_i \in N$). Furthermore, Restriction 2.1 must hold for β.

   F is a set of *attribute assignment* statements of the form $B.a = x$ where $a \in A$ and x is either a terminal member of α or an expression of the form $C.a$ as above. Further, there must be exactly one attribute assignment statement $B.a_i = \ldots$ for each $a_i \in A$.

Definition 2.5 *The immediately derives relation "$\Rightarrow$" is defined over indexed md-sets: $(I_1, R_1, \ldots, R_n) \Rightarrow (I_2, R'_1, \ldots, R'_n)$ if there is a production $B \rightarrow \alpha\beta F$ such that B is a member of I_1, I_2 is equal to I_1 except for the replacement of a single occurrence of B with the elements of α, and $R'_1, \ldots, R'_n$ is equal to $R_1, \ldots, R_n$ except for the addition of tuples in β and the replacement of all arguments of tuples $r_e(x, y)$ as directed by the attribute assignments in F.*

The replacement of arguments in tuples $r_e(x, y)$ above means that each reference $B.a_i$ to an expander attribute of B, $a_i \in A$, is replaced by the value assigned to $B.a_i$ in F. Of course, the assigned values may be indirect references to the expander attributes of the nonterminals in α (to be further resolved in subsequent derivations).

Definition 2.6 *Given an atomic relational grammar G, $(I, R'_1, \ldots, R'_n)$ is a* sentence *of G if there exists a derivation $(\{S\}, R_1, \ldots, R_n) \Rightarrow \ldots \Rightarrow (I, R'_1, \ldots, R'_n)$ such that $R_i = \emptyset, i \in [1, \ldots, n]$, and $\forall s \in I : s \in \Sigma$.*

As an example, the derivation depicted in Figure 2.2 is presented in Example 2.2. In the example, the nonterminal to be rewritten in the next derivation step is underlined. The symbols of the same category are subscripted to distinguish between them in the sentential forms.

Example 2.2

$(\{\underline{Flowchart}\}, \emptyset) \Rightarrow$
$(\{oval_1, \underline{ProcBlock_1}, oval_2\},$
$\{connects(oval_1, ProcBlock_1.in),$
$connects(ProcBlock_1.out, oval_2)\}) \Rightarrow$

$(\{oval_1$,*choice*,$\underline{ProcBlock_2}$,*joint*,$oval_2\}$,
$\{$*connects*($oval_1$,*choice*),
yesConnects(*choice*,$ProcBlock_2$.*in*)
connects(*choice*,*joint*)
connects($ProcBlock_2$.*out*,*joint*)
connects(*joint*,$oval_2$)$\}) \Rightarrow$
$(\{oval_1$,*choice*,*rectangle*,*joint*,$oval_2\}$,
$\{$*connects*($oval_1$,*choice*),
yesConnects(*choice*,*rectangle*)
connects(*choice*,*joint*)
connects(*rectangle*,*joint*)
connects(*joint*,$oval_2$)$\})$

2.2 Earley-style Parsing for ARGs

The predictive parsing algorithm presented in [Wit96] is an extension of the original Earley's algorithm [Ear70]. First, we present Earley's basic algorithm and then, we describe Wittenburg's extensions to it for parsing relational languages.

2.2.1 Earley's Basic Algorithm

Earley's algorithm is a general parsing and recognition method for context-free languages. Earley's algorithm is in effect a top-down parser in which all possible parses are carried along simultaneously in such a way that common subparses can be combined. Thus, the algorithm can parse ambiguous grammars.

An informal description of Earley's algorithm as a recognizer is as follows [Ear70]: It scans an input string $X_1 \ldots X_n$ from left to right looking ahead some fixed number k of symbols. As each symbol X_i is scanned, a set of states S_i is constructed which represents the condition (overall state) of the recognition process at that point of the scan. Each state in the state S_i represents

1. a production such that a portion of the input string which is derived from its right side is currently being scanned,
2. a point in that production (the 'dot') which shows how much of the production's right side has been recognized so far,
3. a pointer back to the position in the input string[2] at which we began to look for that instance of the production, and
4. a k-symbol string which is a syntactically allowed successor to that instance of the production.

[2]The pointer is an index of a symbol in the input, say X_j, and hence, also a pointer to the corresponding state set S_j.

In general, the algorithm operates on a state set S_i as follows: the states in the set are processed in order, performing one of three operations on each one depending on the form of the state. The operations may add more states to S_i and may also put new states in a new state set S_{i+1}. The three operations are described next.

The *predictor* operation is applicable to a state when there is a nonterminal to the right of the dot. It causes the recognizer to add one new state to S_i for each alternative production for that nonterminal. The dot is put at the beginning of the production in each new state, since none of its symbols has been scanned yet. The pointer is set to i, since the state was created in S_i. Thus the predictor adds to S_i all productions which might generate substrings beginning at X_{i+1}.

The *scanner* is applicable just in case there is a terminal to the right of the dot. The scanner compares that symbol with X_{i+1} and if they match, it adds the state to S_{i+1} with the dot moved over the one in the state to indicate that that terminal symbol has been scanned.

The third operation, the *completer*, is applicable to a state if its dot is at the end of its production. For instance, when the completer is applied to a state set S_i having a subset of states representing a production P with the dot at the end of it, the completer compares the look-ahead string with $X_{i+1} \ldots X_{i+k}$. If they match, the completer goes back to the state set indicated by the pointer, say S_j where $j \leq i$, and collects all the states from S_j which have P to the right of the dot and adds them to S_i. Before the addition, the completer moves the dot over P in (the productions of) the states. Intuitively, S_j is the state set where the search for P was initiated. It has now been found, and the dot is moved over the P in these states to show that it has been successfully recognized.

The recognition process is initiated with the single state

$$\langle\, \phi \rightarrow .E \dashv \;\; \dashv \;\; 0 \,\rangle$$

in S_0. Here, $\phi \rightarrow .E \dashv$ is a 'dotted' production where E is the start symbol of the grammar, $\dashv$ is an end-of-input marker and ϕ is an artificial nonterminal symbol not used in the grammar. The production is followed by the lookahead string "$\dashv$" (for $k = 1$ in this case) and the pointer 0. A correct sentence of the language has been recognized if the algorithm ever produces a state set S_{n+1} consisting of the single state

$$\langle\, \phi \rightarrow E \dashv . \;\; \dashv \;\; 0 \,\rangle.$$

Note, that the states are stored in *sets*. Thus, any state will be added only once to a state set.

The algorithm as presented in [Ear70] requires no restrictions of any kind on the context free grammar to be successful. In the general case, the time requirement of the algorithm is n^3, where n is the size of the input (the number of terminals in a sentence). For unambiguous grammars and grammars with bounded ambiguity, the time requirement of the algorithm is n^2. Linear time is achieved for grammars which have a fixed bound on the size of the state sets, and using a proper lookahead, all $LR(k)$ grammars can be processed in time n.

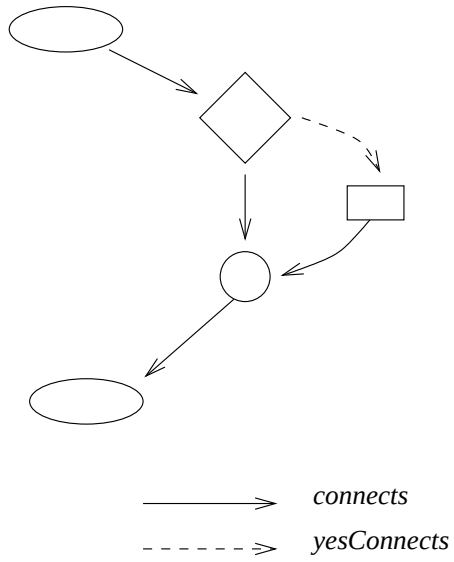


Figure 2.3: An input graph of terminal objects and relations.

2.2.2 Wittenburg's Extensions to Earley's Algorithm

The goal of Wittenburg's work was to develop a predictive Earley-style algorithm for parsing ARGs which can initiate a parse from an arbitrary input symbol. Like Earley's algorithm, the parser will build up state sets by applying the (extended) *predict, scan* and *complete* operations in order to match productions against the input. Wittenburg's algorithm employs also a fourth operation called *inverse-complete*. As opposed to complete, inverse-complete tries to extend an active state[3] with inactive states that already exist in the parse table.

As preliminaries to the parsing algorithm, Wittenburg makes the following observations. The existence of an Earley predictive state (an active state that covers no input) for a nonterminal A in a parse table, which holds the state sets $S_1, \ldots, S_n$, at position i implies that a derivation of an A may 'begin' at the input symbol denoted by i. That is, the input symbol may be part of the cover of an A.

To satisfy the any-start requirement, ordering variants of the right-hand sides of every production are required so that every right-hand side element appears first in at least one variant. After the starting point has been given, the parsing process is directed to scan the remaining input in orders (not necessarily *all* orders) consistent with Restriction 2.1. The requirement does *not* state that once the parser has chosen its first element, the next choice for scanning may arbitrarily be any of the remaining input elements. Instead, the remaining input will be scanned in the order determined by the relational constraints in the production variants.

[3] A state is active if its dot is not at the end of the production.

During the prediction operation, predictive states are created only for the appropriately ordered variants of the productions that expand the nonterminal in question. Appropriately ordered means that a production variant can provide a possible attribute assigment path (for attribute value synthesis) such that as the left-branch of an eventual derivation subtree bottoms out, the terminal elements scanned at that position can ground (give) the value of the expander attribute used in the prediction. For instance, consider the ARG in Example 2.1 with the input in Figure 2.3. Assume that an Earley-style parser has scanned the topmost oval. This implies the existence of a state with the dotted production:

Flowchart $\rightarrow$ *oval*$_1$. *ProcBlock oval*$_2$
connects(oval$_1$*,ProcBlock.in)*
connects(ProcBlock.out,oval$_2$*)*
Flowchart.in = oval$_1$
Flowchart.out = oval$_2$

The expander constraint *connects(oval*$_1$*,ProcBlock.in)* determines the next input objects to be scanned and the ordering variants of *ProcBlock* that may begin at those input objects. That is, the ordering variants of *ProcBlock* productions in which the first right-hand side element cannot serve to bind the *in* attribute need not to be considered.

Wittenburg uses a precompiled prediction table for storing for each nonterminal and for each expander attribute a_i the production variants where the first right-hand side element serves to bind the expander attribute a_i. As a first step, a production ordering algorithm generates for each production one ordering variant per right-hand side element such that the element appears first in the right-hand side. From these variants, the prediction table can easily be constructed. To facilitate the any-start requirement, the special attribute 'start' is added to prediction table for each nonterminal. For the 'start' attribute, the table entry includes one variant of all the productions expanding that nonterminal such that each right-hand side element of the productions appears first regardless of expander attribute bindings.

The *starts-by-binding* relation associates a nonterminal, an expander attribute, and a production variant as follows:

Definition 2.7 *A triple* (N, α, p) *, where* N *is a nonterminal,* α *is an expander attribute, and* $p: \ N \rightarrow X_1 \ldots X_n$ *is an ordered production variant of an atomic relational grammar G, is in the starts-by-binding relation, if and only if there is an attribute assignment of the form* $N.\alpha = X_1$ *or* $N.\alpha = X_1.\beta$ *in* p*.*

The Parsing Algorithm

To store the state sets S_i that represent the condition (state) of the parsing process, Wittenburg's algorithm uses a parse table. Like in Earley's algorithm, the parse

table is indexed with the input objects (terminal symbols). However, Wittenburg makes a distinction between two kinds of states: inactive and active states.

Definition 2.8 *An inactive state relative to an indexed md-set $C = \{I, R_1 \ldots R_n\}$ is a triple $[cat, f, c]$ where cat is a nonterminal or terminal symbol type*[4]*, f is the set of expander (and possibly other) attribute-value pairs, and c is a subset of I representing the state's terminal yield.*

Inactive states represent completely parsed productions (nonterminal instances). They are indexed in the parse table by the values of the expander attributes in f. Intuitively, we consider inactive states to *begin as well as end* at every terminal that is assigned to an expander attribute.

Active states represent partially parsed or predicted productions. As in Earley's algorithm, dotted productions are used with the dot representing a position in the ordered right-hand side elements.

Definition 2.9 *An active state is a triple $[p, i, (d_1 \ldots d_n)]$ where p is a production; i, the 'dot', is an integer ranging from 1 to the length of the right-hand side of p representing the next element to parse; and $(d_1 \ldots d_n)$ is an ordered list of pointers to inactive states of right-hand side elements parsed so far.*

The cover, or terminal yield, of an active state is derived from the covers of the inactive states that have already been recognized. The cover is computed as the union of the covers of the elements that have been parsed so far (the right-hand side elements of a production are called 'daughters' in [Wit96] like the descendants of a node in a derivation tree).

Like inactive states, active states are indexed by individual members of the set of input objects I. The intuition is, however, that active states are indexed by individuals in the input that are candidates to be used in the *next* advancement (parsing action) of that active state. For any other right-hand side elements except the first one, the relational constraint at the dot position is used to find such candidates. For active states that don't yet have any parsed daughter productions, the input indices are derived from higher predictions in the chain of predictions.

Wittenburg employs an *Agenda* as an intermediate storage for states that are to be potentially added to the parse table. The Agenda items are pairs consisting of a state and a *set* of keys. The keys are state indices identifying the state sets to which the state will be added. Note that on the Agenda, there are possibly several indices for a state. There are two reasons for this: (1) depending on the 'topology' of the relations, there may be multiple choices for extending an active state, and (2) the expander attributes of an inactive state may have different values.

To 'move the dot over' in a production, scan, complete and inverse-complete use the *Advance* procedure. The procedure receives as parameters an active state a

[4]The *instances* of nonterminals and terminals are represented by inactive states and by indexes of the md-set, respectively.

and an inactive state i and extends a with i, i.e. i is the right-hand side element of a expected next and the dot can be moved one step forward.

Procedure 2.1 *Advance (a, i)*

Input: An active state $a = [p, j, (d_1 \ldots d_n)]$ and an inactive state $i = [cat, f, c]$.

Output: A new agenda item or *null*.

Method:

If $j =$ the length of the right-hand side of p **then**

i is the last right-hand-side element of p and a new inactive state i' is created. The cover of i' is computed as union of the cover of i and the covers of the parsed right-hand side elements of a, and the expander attribute values of the nonterminal B in question (the left-hand side of p) are determined.

Return a new agenda item $[i', keys]$ where $keys$ is the list of inactive state indices of i' (the values of the expander attributes $A(B)$).

else

A new active state a' is created. The inactive state i is added with the parsed right-hand side elements of a as the daughters, $a' = [p, j + 1, (d_1 \ldots d_n i)]$, and a query is executed to find the input object candidates to drive the next expansion of the production. To launch the query, the expander constraint at the new dot position is resolved and, based on the expander attribute values of the right-hand side elements parsed so far, a subroutine identifies the already-bound value of the constraint used in the query over the expander relations $\{R_1, \ldots, R_n\}$. As a result of the query, new keys to be used as state indices are obtained for a'.

Return a new agenda item $[a', keys]$ where $keys$ is the non-emptylist of input objects returned by the query, or *null* if the result of the query was $\emptyset$.

end if

Note that an active state is created (and, eventually, added to the parse table) only *if there exists some tuple in the required relation in the input*, i.e. the result of the query is not $\emptyset$. Note that there can be *many* expander constraints applicable at the dot. The Advance procedure does not explicitly state how this situation is handled and we assume that one constraint is just randomly chosen.

Algorithm 2.1 is the main parsing algorithm. The parser is initialized with a recursively constructed set, *init-states*, consisting of the predictive states expanding the root symbol of the grammar at the given (arbitrarily chosen) input symbol. The set of initial states is added to Agenda, and the main loop of the algorithm is entered.

Algorithm 2.1 *Membership in L(ARG)*

Input: An atomic relational grammar $G = (N, \Sigma, S, R_e, A, P)$, a set $C = (I, R_1, \ldots, R_n)$ to be parsed, and an object $o \in I$ from which to start parsing.

Output: A parse table T of state sets T_j.

Auxiliary data structures:

Agenda: A FIFO list of states to process, initally empty.

Init-states: The set of starting predictive states created as follows: for every production ordering variant p in starts-by-binding(S,start,p), add a state $[p, 1, \emptyset]$ to init-states. For every state $s = [q, 1, \emptyset]$ in init-states, if the right-hand side symbol x at position 1 of q is a nonterminal, then let

$$\text{init-states} = \text{init-states} \cup \{[t, 1, \emptyset] \mid \text{starts-by-binding}(x,\text{start},t)\}$$

Parse table: A hash table T of state sets T_j where $j \in I$.

Algorithm:

/ begin main loop */*
Add init-states to Agenda
While *Agenda is not empty* **do**

Remove one item [*state,keys*] from Agenda (assume FIFO management).

For each k *in keys*

If *an equivalent state is not already at* T_k **then**

add *state* to T_k; Then do one of the following:
Scanner: If *state* is active and the right-hand side symbol at the dot in the production is terminal and the input symbol y at k matches the terminal and does not intersect the cover of the state, then add all the items returned by *Advance(state,y)* to Agenda.
Predictor: If *state* is active and the right-hand side symbol x at the dot in the production is nonterminal, then for all attributes that are to be bound in the expander constraint at the dot, add to Agenda items with all the production variants that expand x and provide bindings for the attributes. The dot is positioned at the beginning of each variant and the key for the Agenda items is k.
Completer: If *state* is inactive, then for every active state $a_i \in T_k \cup$ *init-states*, if *cat* of *state* matches the right-hand side symbol at the dot of a_i and the covers of the states do not intersect, add all the items returned by *Advance(a_i,state)* to Agenda.
Inverse-Completer: If *state* is active, then for every inactive state i at T_k, if cat of i matches the symbol at the dot in *state*, and

the covers of *state* and i do not intersect, and the input symbol k, which is one of the indices of i because (a reference to) i is found at T_k, satisfies the expander constraint at the dot of *state*, then add all the items returned by *Advance(state,i)* to Agenda.

end if

end for

end do /* *end main loop* */
If there is an inactive state of the form $[S, f, u]$ in the parse table T and $u = I$ **then**

SUCCEED

else

FAIL

end if

Complexity of the Parser

Wittenburg has not given any results of the theorethical complexity of the parsing algorithm. Our analysis of the worst case complexity of the algorithm is presented in Section 3.3 in the next chapter.

Chapter 3

Problems in Using Atomic Relational Grammars

In this chapter, we discuss the problems in using ARGs for specifying and parsing visual languages. In Section 3.1, we discuss how to model the constructs of typical diagramming languages with ARGs. We point out the limitations of the original model and propose extensions. Then, in Section 3.2, we analyze the behavior of Wittenburg's parsing algorithm and suggest areas for improvement with respect to our needs. In Section 3.3, we analyze the worst-case time requirement of the algorithm. In Section 3.4, we summarize our findings.

3.1 Grammatical Problems

Visual diagramming languages have common, reoccurring syntactic constructs. The most typical ones are:

- *Connections* shown as lines or arrows starting from (inside or on the border) of one graphical object and ending to another; for instance, state transition arrows in finite state machines.
- *Topology* induced by connections; graph properties convey semantically significant information, for instance, the flow of execution control in flowcharts.
- *Hierarchical containment* of graphical objects; for instance, in the static structural diagrams of UML, classes have compartments that hold text items and packages can hold hierarchies of packages.
- *Labels* as text items attached to other graphical objects; for instance, conditions and actions associated with state transitions in Harel statecharts.
- *Other spatial relations*; for instance, left-to-right or top-to-bottom ordering of graphical objects such as the ordering of subtrees in yes-no decision tree diagrams.

In this section, the expressive power of atomic relational grammars is studied by presenting grammars that capture these basic constructs. Problems pertaining to the specification mechanism are pointed out and possible solutions are discussed.

The following discussion concentrates on the issues of representing graphs and hierarchical structures. We consider first structured graphs which can be specified with context-free syntactic rules. Then, we discuss how to specify general unstructured graphs. We use trees as examples of visual languages with hierarchical constructs.

3.1.1 Structured Graphs

In Example 2.1, an ARG for (a fragment of) the language of *structured* flowcharts was presented. When considered as a graph structure, the sentences of the flowchart language are directed graphs of bounded degree: the number of arcs leaving and entering each kind of node is fixed. Furthermore, in a well formed flowchart, each node is connected to some other node.

In the flowchart grammar, the arcs between nodes are modeled directly as relations. For example, consider the following production from the flowchart grammar in Example 2.1:

Flowchart $\rightarrow$ $oval_1$ *ProcBlock* $oval_2$
connects($oval_1$*,ProcBlock.in)*
connects(ProcBlock.out,$oval_2$*)*
Flowchart.in = $oval_1$
Flowchart.out = $oval_2$

The production defines a *Flowchart* to be a directed graph of two *ovals* connected to (some elements of a) subgraph defined by a *ProcBlock*. Note that the two expander attributes *in* and *out* of the *ProcBlock* provide the only possible connection points for the arcs starting from $oval_1$ and entering $oval_2$. According to this grammar, unstructured transfers of execution control (gotos) are prohibited, as demonstrated by the example flowchart in Figure 2.2.

As another example of structured graphs and as an example of hierarchical structures, consider the grammar of binary trees in Example 3.1. In the basic grammar, the nonterminal *Node* represents a labelled node of the tree and there are three productions for the nonterminal *Tree* to allow internal nodes to have zero, one, or two subtrees.

Example 3.1
Tree $\rightarrow$ *Node*
Tree.root = Node.root

$Tree_1$ → Node $Tree_2$
connected(Node.root,$Tree_2$.root)
$Tree_1$.root = Node.root

$Tree_1$ → Node $Tree_2$ $Tree_3$
connected(Node.root,$Tree_2$.root)
connected(Node.root,$Tree_3$.root)
$Tree_1$.root = Node.root

Node → circle text
inside(text,circle)
Node.root = circle

In the flowchart and binary tree grammars, the (sub)graphs have uniquely defined *access nodes* which link the subgraphs to their surroundings (e.g. the root of a *Tree*). That is, *arbitrary* nodes may not be connected with arcs. Thus, the parsing of a subgraph is independent of the context of the subgraph. In other words, such grammars are *context free*, which makes it simple to use the arc-relation as the relation that drives the parsing and determines the scanning order of the input.

Spatial Relations

The binary tree grammar in Example 3.1 makes no distinction between the left and right subtrees of the internal nodes. If the left-to-right spatial relation has semantic meaning and we want the order of the subtrees in a *Tree* (nonterminal instance) to reflect that, we can add a predicate (*left-of*) to the grammar. Further, we may want to enforce the condition that an internal node is *above* its descendants:

$Tree_1$ → Node $Tree_2$
connected(Node.root,$Tree_2$.root)
above(Node.root,$Tree_2$.root)
$Tree.root_1$ = Node.root

$Tree_1$ → Node $Tree_2$ $Tree_3$
connected(Node.root,$Tree_2$.root)
connected(Node.root,$Tree_3$.root)
left-of($Tree_2$.root,$Tree_3$.root)
above(Node.root,$Tree_2$.root)
above(Node.root,$Tree_3$.root)
$Tree.root_1$ = Node.root

In this case, the evaluation of the predicates could be based on the graphical properties of the input objects (the relative location in a co-ordinate space). If the

location property of the input objects can be accessed at parse time, the indexed md-set I used as input does not have to include relations *above* and *left-of*. That is, the predicates map to some external functions that are not part of the grammar specification.

3.1.2 Unstructured Graphs

If the underlying graph structure of a visual language has no upper bound on the degree of the nodes or the topological constructs are not context-free, it is not so easy to specify the language with an ARG as in previous examples. In order to support the specification of general graph languages, the ARG model needs to be extended.

In the following, we examine first how to specify general trees, a subclass of general (directed) graphs. Second, we discuss the modeling of the arcs of a graph. Third, we consider the representation of general graphs. Finally, we discuss how to extend the ARG model with context dependent remote references in productions.

General Trees

Consider a visual language of general labelled trees where the internal nodes may have any number of children. It would be convenient to be able to write the grammar in the following compact form:

Example 3.2
*$Tree_1$ → Node $Tree_2$**
connected(Node.root,$Tree_2$.root)
$Tree_1$.root = Node.root

Node → circle text
inside(text,circle)
Node.root = circle

Here, *Tree** denotes a sequence of zero or more *Trees.* The production defines a tree to consist of a labelled root node with zero or more subtrees connected to it by their roots. The order of the subtrees is arbitrary in this example. However, for some applications, it may be desirable to enforce an ordering of the sequential elements. For instance, an expression like:

order $Tree_2$.root by left

could be interpreted to enforce the parser to pre-sort the sequence of terminal symbols in relation *connected* with *Node.root* in the first production according to the ordering predicate *left.*

The introduction of sequential right-hand side elements allows a natural grammatical representation of languages that have nodes with an unlimited number of connections. Without iterative symbols, the iteration must be substituted with recursion as in the following tree grammar:

Example 3.3

Tree → *Node*
    Tree.root = Node.root

Tree → *Node SubTrees*
    connected(Node.root,SubTrees.root)
    Tree.root = Node.root

$SubTrees_1$ → *Tree* $SubTrees_2$
    sibling(Tree.root,$SubTrees_2$*.root)*
    $SubTrees_1$*.root = Tree.root*

SubTrees → *Tree*
    SubTrees.root = Tree.root

Node → *circle text*
    inside(text,circle)
    Node.root = circle

The grammar in Example 3.3 introduces a new relation, *sibling*, which is needed to link (sub)trees having the same parent (immediate ancestor). The nonterminal *SubTrees* is used to recursively collect the list of the descendants of an internal node of a tree. The iterative grammar in Example 3.2 is a much more concise and natural specification of the tree language than this grammar.

Modelling Arcs

In the grammars presented so far, the arcs (or connections) between nodes were represented directly by relations. This straightforward syntactic representation introduces two problems.

The first problem concerns the semantics of arcs. If the arcs have structure themselves, it is necessary to model arcs as nonterminal objects as well. For instance, the transition arrows of finite state machines have text objects attached as labels.

The second problem is less obvious, because it stems from the inherent tolerance for relation-based ambiguity of the Earley-style parsing algorithm. Consider the input graph describing a general tree in Figure 3.1. The tree is malformed because

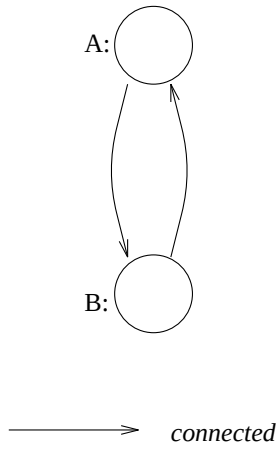


Figure 3.1: An ambiguous input graph.

there is a cycle in the *connected* relation. Also, the input is *ambiguous* because either the node labelled *A* or *B* may be interpreted as the root and the other as a child. In fact, this is exactly what Wittenburg's parsing algorithm does in this particular case: it creates both possible interpretations and accepts the input.

As another example of handling of ambiguous input, consider the flowchart in Figure 3.2. Although there are two relations connecting the topmost oval with the rectangle, Wittenburg's algorithm accepts the input simply because when the expander relation between the *oval* and the *P-block* is queried for, only one input object is returned (either the rectangle or the oval depending on the scanning order). So, in this case, the extra relation is automatically ignored.

The ARG model regards relations as second-class objects when compared to the actual input symbols. Thus, the ambiguity problem can be partly solved by modelling the connection lines or arrows as syntactic objects. This prevents the parser from accepting ambiguous inputs that contain extra connections. For instance with the input in Figure 3.2, the parser would recognize two flowcharts that both include one of the (duplicated) connections but neither will include both of the connections.

General Graphs

With tree structures it is easy to use the visible connections directly as the relations driving the parsing. As the grammar in Example 3.3 shows, even general trees can be modelled. This is due to the context-free topology of trees: they are connected, acyclic graph structures, where each subgraph has a *unique* access point, the root, which is a representative of the whole subgraph. What makes the description of general graphs different is that the arcs between nodes cannot be used as the relation that drives parsing. For instance, a graph may contain unconnected nodes.

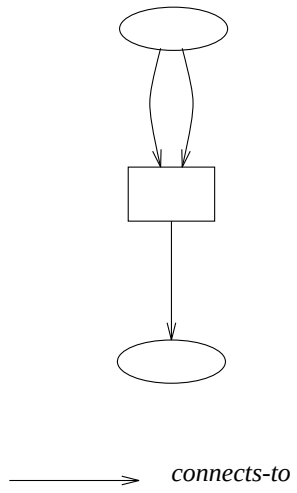


Figure 3.2: An erroneous flowchart.

To define general graphs with unbounded degree, the arcs or connections must be modeled as syntactic objects instead of relations. Consider the grammar in Example 3.4 that defines undirected general graphs. We use iterative constructs in the productions to achieve a compact specification: the expression $X*$ denotes zero or more symbols and $X+$ denotes one or more symbols.

Example 3.4
S → *graph Node+*

belongsTo(Node.connector,graph)
S.connector = graph

Node → *circle text arc**
startsFrom(arc,circle)
inside(text,circle)
Node.connector = circle

In the grammar of Example 3.4, the terminal *graph* represents a container (or drawing) that holds the nodes of the graph. The relation *belongsTo* associates the nodes of the graph with the container. Note that the container terminal may not have a graphical representation in the language. The grammar models arcs explicitly as terminal symbols (*arc*). The arcs are related to the *Nodes* by the relation *startsFrom* which connects each arc to exactly one *Node.* The arcs can be attached to *Nodes* arbitrarily via the *startsFrom* relation since the arcs are undirected. The relation *inside* denotes spatial inclusion. The expander attribute *connector* holds the terminal that provides access to a *Node* from outside.

In the grammar, an arc cannot be attached to both of its end-point *Nodes*. The reason for this lies in the parsing algorithm: if an arc would be allowed to belong to the covers of both of the productions generating the connected nodes, it would be scanned twice in two different subparses. This would prevent the parser from merging the covers of the subparses.

The grammar in Example 3.4 gives a rather simplistic definition of undirected general graphs: a graph is defined to be a collection of *circle*s including *text* with attached *arc*s. Nothing is said about the topology defined by the arcs connecting the nodes to each other. In fact, based on the syntactical definition, it is not even known which *Node* instances really are connected to each other.

However, in many visual languages, the topology is not very constrained. For example in the class diagrams of UML, the only topological restriction concerning the relations between classes depicted as (undirected and directed) arcs is that the inheritance relation is acyclic. Further, in the statecharts language of UML there are only few topological restrictions, for instance that final states cannot have outgoing arcs and initial pseudostates cannot have incoming arcs. The rules concern more about what kinds of objects can be connected by which kinds of arcs and what are the allowed labels on arcs.

On the other hand, languages like structured flowcharts or trees have topological rules which can be expressed by context free grammatical structures as have been illustrated by the earlier grammar examples. For example, in the statecharts language of UML, the inclusion hierarchy of superstates and substates can be used as a context-free driving relation of the grammar. If the transitions between states were used to direct the order of parsing, the transition arrows would have to be scanned in an order yielding a correct state hierarchy. That is, the transitions between the substates in a composite state (XOR, AND) are scanned first and the other transitions leaving or entering the composite state afterwards. As demonstrated by Tucci & al. in [TVC94], this approach may lead to an exponential search time for a correct parse.

Remote References

In [Gol91], Golin introduced the concept of remote objects in context-free productions. A remote object is a terminal object that is part of some other nonterminal instance outside the nonterminal instance currently being parsed. In productions, remote objects are used in constraints to give additional, context dependent conditions for recognizing nonterminals.

It is straightforward to add support for remote objects into the ARG model. For example, we can augment the *Node* production from the grammar in Example 3.4 with a predicate that uses a *remote reference* as shown in Example 3.5:

Example 3.5
Node $\rightarrow$ *circle*$_1$ *text arc*$*$

inside(text,circle$_1$*)*
startsFrom(arc,circle$_1$*)*
endsTo(arc,$\underline{\textit{circle}_2}$*)*

Node.connector = circle$_1$

The remote reference is underlined. The expression *endsTo(arc,*$\underline{\textit{circle}_2}$*)* means that in a correct instance of *Node*, every *arc* starting from *circle*$_1$ must also end to a *circle* somewhere in the input. The expression *endsTo(...)* must be evaluated as a predicate and not as an expander relation because the remote object is not *usually* one of the right-hand side symbols of the production. Note that this formulation does not forbid a remote object to actually be a part of the production: the object is only 'logically' remote (the same node may be both the source and the target of an arc). Also, there might be several types of lexical objects that can be the targets of an arc. The grammar notation should provide the means for expressing this kind of rule.

The remote reference provides a way to add expressive power to ARGs in the sense that it can further restrict the inputs that the parser will accept. For instance with the grammar in Example 3.4, the parsing algorithm will accept graphs where each *arc* is connected to a *Node* only by one of its ends. The production above ensures that an arc is connected by *both* of its ends with terminal symbols of correct types.

Looking from the semantic point of view, the arcs of a graph language usually denote relationships between instances of nonterminals rather than instances of terminals. A terminal symbol used in a remote reference is then a required part of some nonterminal instance. If a predicate on a remote reference fails, it implies that such a nonterminal cannot be parsed in the input because a required part of the nonterminal is missing. However, if the predicate evaluates to true, it does *not* imply that such a nonterminal has been or will be parsed. So, also when using remote references, the validity of arcs must be checked after parsing during a semantic processing phase.

3.2 Parsing Problems

In this section, we describe problems in the original parsing algorithm for ARGs. To set the following discussion in perspective, we recall that our aim is to support the implementation of off-line parsers for edit-compile style of visual programming. In such a setting, the goal of the parser is to produce an unambiguous interpretation of a visual program. This is different from the original goal of the

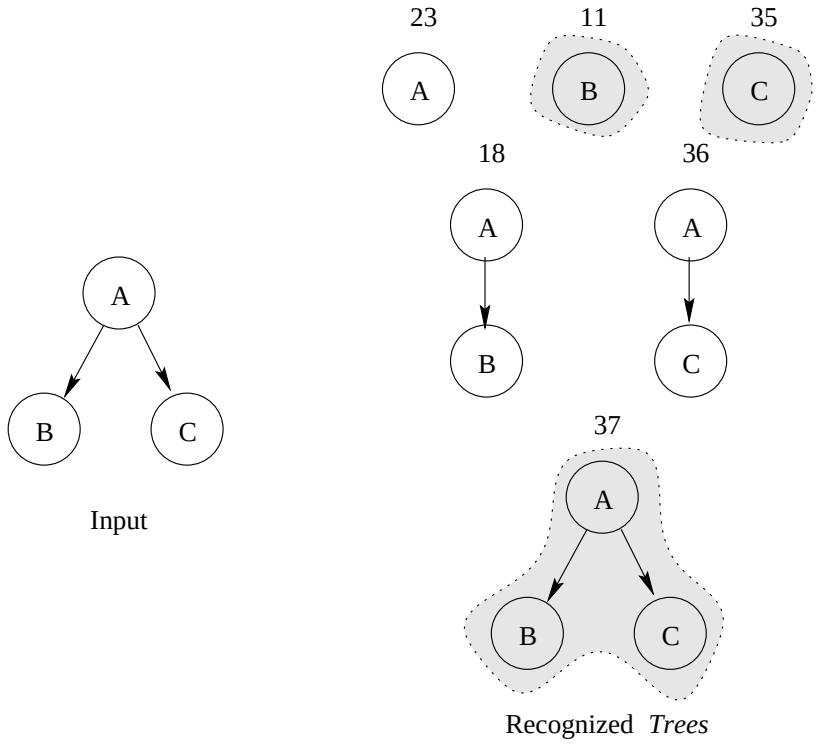


Figure 3.3: The parses (on the right) of the binary tree on the left, produced by Wittenburg's parser.

ARG model which was to support incremental parsing of possibly ambiguous visual languages.

We concentrate first on the problem of pruning redundant structural variants of nonterminals during parsing. Then, we discuss the effects of the any-start property on parsing, and finally, we briefly discuss the role of predicates in parsing.

3.2.1 Parsing Structural Variants

With the grammar in Example 3.1, Wittenburg's algorithm will produce many partial parses that cover only a section of the input. This is illustrated in Figure 3.3 that shows the *Trees* recognized by the parsing algorithm from the binary tree on the left-hand side of the figure. The *Trees* on the shaded background are the ones that are necessary for a parse that covers the whole input, that is they correspond to the instances of *Tree* in the derivation of the binary-tree used as input. The other *Trees* represent partial parses, or unnecessary reductions of grammar productions into nonterminal instances.

The numbers associated with the trees in Figure 3.3 refer to the (inactive) parse

states in Figure 3.4 that shows the corresponding parse table. The left-hand side column of the table shows the index symbol (input object) for a particular slot (the *circle* objects in the tree have been numbered in the breadth-first order). The right-hand side column shows the states in the parse table slots. Active states are depicted with dotted productions in brackets. The dot shows how much of the right-hand side of a production has been parsed so far. Inactive states are depicted as #(*nonterminal*). In front of each state there is a number that shows the order in which the states have been inserted into the table during parsing. In this case, the parsing has been started from the text object 'B', as indicated by the ordinals of the states in that slot.

In the binary tree grammar, the three productions for *Tree* represent *structural variants* that are in an inclusion relationship. This means that whenever the parser recognizes a *Tree* defined by production *Tree* → *Node Tree Tree*, the parser will also recognize the instances defined by *Tree* → *Node Tree* and *Tree* → *Node*. This is illustrated by the parse states 18, 23, 36, and 37 in the parse table in Figure 3.4 (Figure 3.3 shows the corresponding trees).

This behavior is analogous to the behavior of Earley's parser with a zero length lookahead. Because of the lack of a linear ordering of the input, Earley's lookahead mechanism cannot be applied to parsing visual languages.

However, the predicate mechanism can be used to achieve a form of lookahead (or *lookaround*) to distinguish between the structural variants during parsing. Consider the following productions:

Example 3.6
Tree → *Node*
    Tree.root = Node.root
    notConnected(Node.root)

$Tree_1$ → *Node* $Tree_2$
    connected(Node.root,$Tree_2$*.root)*
    onlyOneConnected(Node.root)
    $Tree_1$*.root = Node.root*

$Tree_1$ → *Node* $Tree_2$ $Tree_3$
    connected(Node.root,$Tree_2$*.root)*
    connected(Node.root,$Tree_3$*.root)*
    $Tree_1$*.root = Node.root*

Now, the interpretation of predicates *notConnected(Node.root)* and *onlyOneConnected(Node.root)* is that they evaluate to *true* if there are no input objects or there is just one object, respectively, in the *connected* relation with the input object bound to *Node.root*. In [WW98], Wittenburg and Weitzman use predicates like this to cut down unnecessary parsing in a flowchart grammar. Likewise, Chok and Marriott use *non-existence constraints* in CMGs [CM95] to distinguish structural

A	**16**	[Node -> circle . text]
B	**1**	[Node -> . circle text]
	2	[Node -> . text circle]
	3	[Tree -> . Node]
	4	[Tree -> . Node Tree]
	5	[Tree -> . Tree Node]
	6	[Tree -> . Node Tree Tree]
	7	[Tree -> . Tree Node Tree]
	8	[Tree -> . Tree Node Tree]
C	**33**	[Node -> circle . text]
◯ *1*	**12**	[Tree -> Tree . Node]
	13	[Tree -> Tree . Node Tree]
	14	[Tree -> Tree . Node Tree]
	15	[Node -> . circle text]
	17	#(Node)
	18	#(Tree)
	23	#(Tree)
	36	#(Tree)
	37	#(Tree)
◯ *2*	**9**	[Node -> text . circle]
	10	#(Node)
	11	#(Tree)
	19	[Tree -> Tree Node . Tree]
	21	[Tree -> Tree Node . Tree]
	24	[Tree -> Node . Tree]
	26	[Tree -> Node . Tree Tree]

◯ *3*	**20**	[Tree -> Tree Node . Tree]
	22	[Tree -> Tree Node . Tree]
	25	[Tree -> Node . Tree]
	27	[Tree -> Node . Tree Tree]
	28	[Tree -> Node Tree . Tree]
	29	[Tree -> . Node]
	30	[Tree -> . Node Tree]
	31	[Tree -> . Node Tree Tree]
	32	[Node -> . circle text]
	34	#(Node)
	35	#(Tree)

Figure 3.4: Parse table after parsing a binary tree.

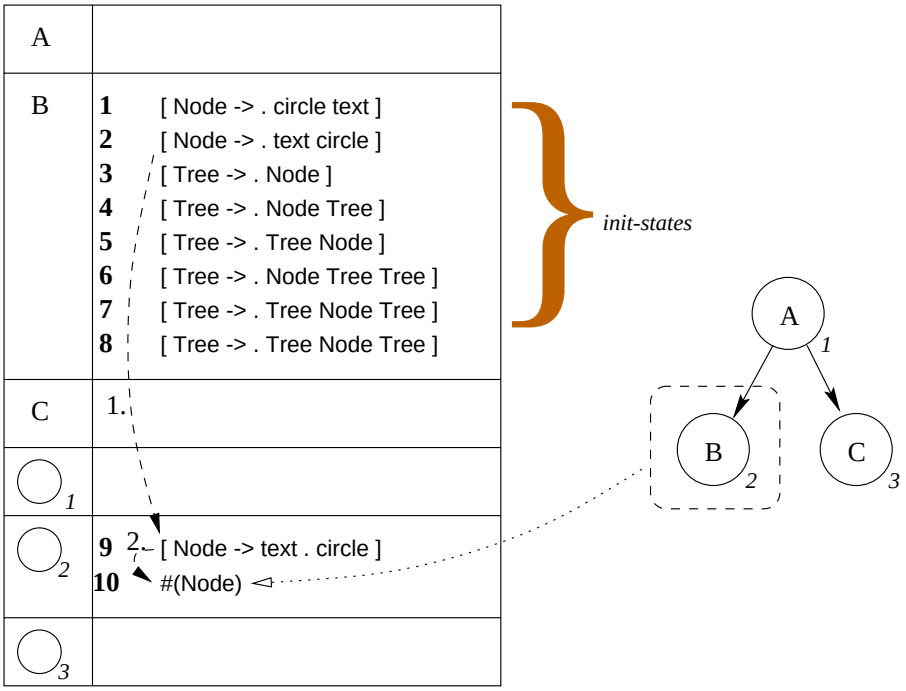


Figure 3.5: Parsing a binary tree starting from one of the leaves.

variants from each other; their main concern is to make parsing deterministic. The writer of a grammar has to add the lookahead rules manually when using CMGs and ARGs.

3.2.2 Any-Start

In Wittenburg's parser (Algorithm 2.1), the any-start property is achieved by using the set *init-states* in the completion operation. Init-states contains predictions for every nonterminal in the grammar to provide missing predictions of nonterminals when starting parsing from somewhere in the 'middle' of the input (with respect to the derivation that created the input). The role of init-states is illustrated by the follwing example.

Consider the situation in Figure 3.5 where parsing of the binary tree on the right has been started from the object 'B'. In the parse table shown on the left, the slot indexed by 'B' has been initialized with the init-states (the two active states $[Tree \rightarrow . \, Tree\, Node\, Tree]$ are ordering variants of the same production). The figure shows a snapshot of a parse where after the successful scanning of 'B' (arrow 1) and *circle*$_2$ (arrow 2) a *Node* has been recognized. This is indicated with the inactive state *#(Node)* in the slot indexed by *circle*$_2$. However, this slot contains no states that would be waiting for a *Node* to be parsed. That is, there is

no prediction for the *Node*. Therefore, the complete operation in Algorithm 2.1 takes a union of the states in the slot and *init-states* to advance the parse. As result of the complete, the states 11, 12, 13, and 14 are created (see Figure 3.4) and the parsing continues.

Any-start is a powerful property of the parsing algorithm which is important in incremental parsing applications. However, in an offline parsing setting it is not necessary. The combined effect of any-start and the lack of lookahead is clearly shown when comparing the parse tables in Figures 3.4 and 3.6. Figure 3.6 shows a parse table for a complete parse of the input in Figure 3.3 with a minimum number of parse states. By giving up the any-start property and by pruning structural variants during parsing it is possible to achieve an unambiguous parse like in Figure 3.6. Note that in Figure 3.6, the parsing has been started from the *circle* of the root node of the tree instead of the text object 'B', as in the parse in Figure 3.4.

Giving up the any-start property means that the grammar of the visual language has to be written so that it is possible to define an auxiliary function that can decide from which input object (terminal) to start parsing. For the binary tree language, this function would just search (one of) the smallest input object in the *connected* relation. For the languages that we have been studying in our work, it has been a simple matter to define such a function.

3.2.3 Semantics and Evaluation of Predicates

Predicates can be used to enforce *local* conditions between the right-hand side elements of productions. For instance, recall the binary tree example form page 37, where the predicates *left-of* and *above* were used to enforce the conditions that the subtrees of an internal node are below it and that they are ordered from left to right. If the location property of the input objects can be accessed at parse time, the indexed md-set I used as input does not even have to include the relations *above* and *left-of* because the relations do not drive the scanning of the input.

However, predicates open the possibility to perform context-sensitive checks. For instance, in Example 3.6 above, predicates were used to prune structural variants by checking the presence of input objects outside the current parse context in a relation with a local object. Also, in Example 3.5, a predicate was used to implement a reference to a remote object outside the current parsing context.

Wittenburg and Weitzman have used even more powerful predicates: in [WW98] a predicate searches the parse table to find similar kinds of states that have a larger cover than the one being parsed. Their idea is to suppress parses that eventually lead to the creation of the same states over and over again (like in the binary tree parsing example above).

If predicates can be arbitrarily complex, the expressive power of the grammatical model is increased significantly. On the other hand, formal reasoning about the expressive power and the complexity of the parsing becomes more difficult. The difficulties created by allowing arbitrarily complex computations in addition

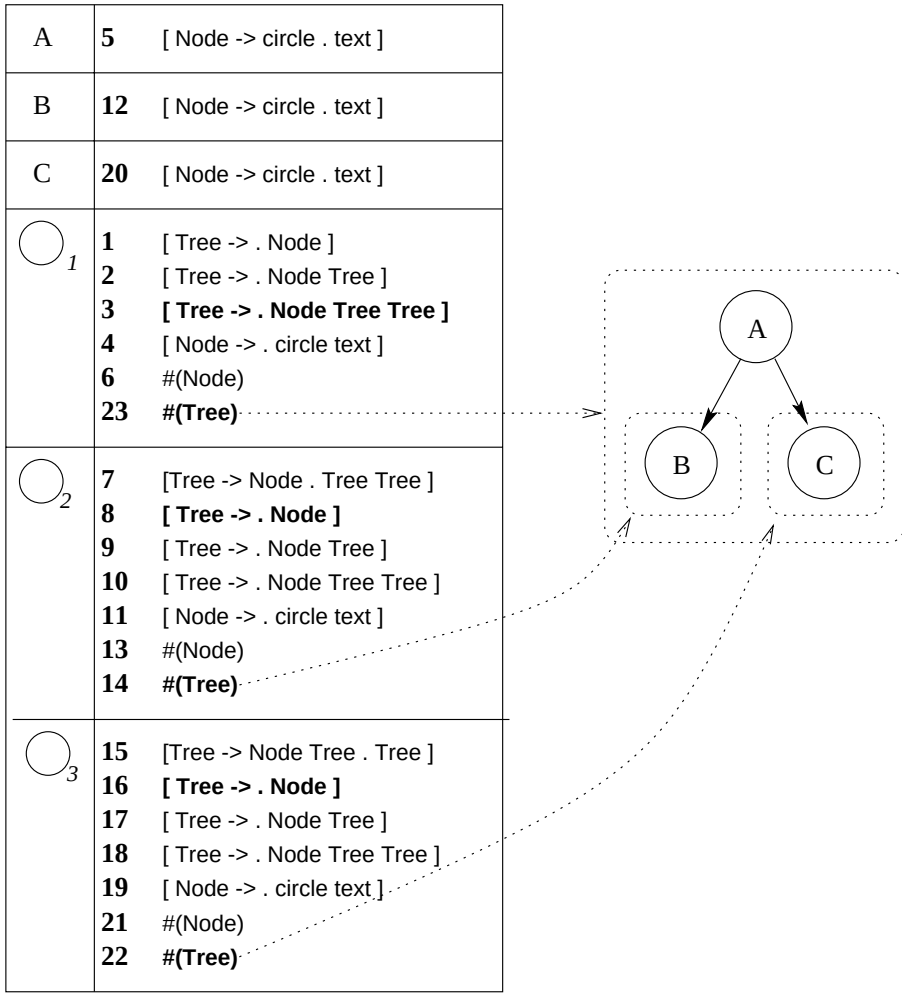


Figure 3.6: A parse of the binary tree on the right-hand side with the minimum number of parse states.

to normal parsing actions are illustrated in [MM98a] where Marriott and Meyer use a restricted form of constraint multiset grammars as the basis of the CCMG hierarchy of visual languages. The restriction is that attribute values can only be copied from the right-hand side elements to the attributes of left-hand side elements. That is, complex computations on attributes are not allowed. Marriott and Meyer argue that a formal treatment of the complexity of parsing is possible only with this restriction.

3.3 Complexity of Parsing

Wittenburg has not given any analysis of the complexity of Algorithm 2.1. In the following, we analyze the theorethical worst case time requirement of the algorithm and show that it is $O(2^n)$ where n is the number of symbols in the indexed md-set I used as input. The analysis follows the reasoning by Earley on the complexity of the original algorithm [Ear70].

We have analyzed the worst case complexity as a function of the number of the input objects and not as a function of the number of the relation tuples. We did this because it is a necessary condition of a successful parse that all the input objects have been processed (see the success condition of Algorithm 2.1 on page 34). However, Algorithm 2.1 does *not* require that all the relation tuples have been used in a successful parse.

For the analysis, we first recall that parse states are stored in sets in the parse table. The equality between parse states is based on comparing the equality of the parts of the states. Two active states are equal if they have the same production variant, the same dot position and equal sets of parsed daughters. The equality of inactive states is based on nonterminal type, the values of expander attributes, and the equality of the cover sets.

In the analysis, we first determine how the size of the parse state sets T_i in the slots of the parse table T depend on the number of input objects. Then, we can analyze the number of steps executed by the parsing operations of the algorithm. The analysis follows the worst case scenario.

3.3.1 Analysis

The Number of Parse States in T_i

In Earley's algorithm, the number of states in any state set T_i in the parse table T for the input symbol X_i is proportional to i ($\sim i$) because only the value of the back pointer depends on i; the ranges of the other elements of a state tuple are bounded (see Section 2.2.1). For Algorithm 2.1, we state:

Lemma 3.1 *In the worst case, the number of states in any set T_i is proportional to 2^{n-1}, where n is the number of symbols in the indexed md-set I used as input.*

Proof: The covers of inactive states (and the covers of the parsed daughters of active states) represent selections of symbols from I. With certain grammars and inputs (see below), it is possible that the parser generates for each symbol X_i the set W_{X_i} of nonterminal instances that has the following properties:

- The states in W_{X_i} cover together all possible selections of input symbols that include X_i (the order of the symbols in the selection is not significant). The size of the set is
$$|W_{X_i}| = \sum_{j=0}^{n-1} \binom{n-1}{j} = 2^{n-1} .$$
- Each state in W_{X_i} has X_i as the value of one of the expander attributes.
- Because of the second property above, $W_{X_i} \subset T_i$.

It can be thought that the nonterminal instances in W_{X_i} represent all the possible paths through the input symbols that 'begin' at X_i and that follow the expander relations in the input.

It is possible that the same symbol appears as the expander attribute value of nonterminal instances that represent different types. However, the number of nonterminal types is bounded by the (constant) number of nonterminals in the grammar. Therefore, the number of inactive states in T_i is $\sim 2^{n-1}$. The set T_i may also contain active states that represent parse paths through the input that include X_i at some point. The number of such paths is subject to the same combinatorial constraint as the number of paths that begin at X_i. So, we conclude that the number of parse states in T_i is $\sim 2^{n-1}$. ■

For example, in Section 3.2 we parsed binary trees with the ambiguous grammar of Example 3.1. Figure 3.3 and the corresponding parse table in Figure 3.4 show that the parser creates all the *Trees* that begin from *circle*$_1$ and that have only this symbol in common. If the *circles* in the input were completely connected by the *connected* relation so that each *circle* would be *connected* to every other *circle*, the parse table would contain also all the possible trees rooted at *circle*$_2$ and *circle*$_3$ in the parse table slots for *circle*$_2$ and *circle*$_3$, respectively.

The Number of Steps per T_i

The predict operation executes a bounded number of steps per state in any state set. The complete, inverse-complete, and scan operations execute $\sim n \cdot 2^{n-1}$ steps for each state they process in the worst case because they may have to add new states into every other table slot (in case of a completely connected input). Adding new states into a set involves testing that an equal state is not already in the set. In the case of inactive states, this requires testing if the covers of the states are the same, which may take up to n^2 steps. In each parsing operation, there is also the test that the intersection of the covers of two parse states is empty and this may take up to n^2 steps per test. So, these operations take up to $\sim n \cdot 2^{n-1} \cdot n^2 \cdot n^2 \cdot 2^{n-1} = n^5 \cdot 2^{2n-2}$ steps in T_i.

The Number of Steps per I

For all the sets $T_1, \ldots, T_n$, the number of steps executed by the algorithm is $\sim n \cdot n^5 \cdot 2^{2n-2} = O(2^n)$. So, we get the result in Lemma 3.2:

Lemma 3.2 *The worst case time requirement of Algorithm 2.1 is $O(2^n)$.*

3.3.2 The Causes of the High Complexity

In the worst case, Algorithm 2.1 is very inefficient. Also, Earley's original algorithm has a relatively high time requirement, $O(n^3)$, in the general case. However, for unambiguous grammars and grammars with bounded ambiguity (every sentence has only $\leq k$ derivation trees for some fixed k), the time requirement of Earley's algorithm is $O(n^2)$. Further, linear time can be achieved for grammars that have a fixed bound on the size of the state sets, and using a proper lookahead, all $LR(k)$ grammars can be processed in time $O(n)$.

The ambiguity of a grammar and the ambiguity in the input are the causes of the high worst case complexity of Wittenburg's algorithm. The main issue here is the number of states in the slots (state sets) of the parse table. If the size of the state sets is bounded by some constant (or even a polynomial function over n), it is possible to have a polynomial time requirement.

Figure 3.6 suggests that by pruning structural variants during parsing and with an unambiguous input, it is possible to have a bounded number of states in the state sets. This would remove the exponential term from the complexity calculation. In practice, this is what has happened with the grammars that we have implemented with VILPERT.

There are other factors than ambiguity that contribute to the high polynomial terms in the overall complexity. Computing the set intersection of the covers of inactive states and computing the equality of the covers can be expensive operations if naïve implementations are used. For example, Wittenburg suggests using a bit vector representation for the cover sets to reduce the cost of these set operations at runtime. This solution saves time with the expense of space since for an input set of size n, an n-bit vector is required to represent a cover set.

The representation of the input relations and the cost of executing expander queries has been ignored so far. Many kinds of underlying data strcuctures or database technologies can be used to achieve an optimized solution for the representation and the querying problems.

3.4 Discussion

Atomic relational grammars provide a good compromise between the expressiveness of the specification formalism and the simplicity of the grammar formalism

and the associated parsing algorithm. One nice feature is that the grammar formalism does not require the productions to be in any kind of normal form because of the Earley-style parsing algorithm. With the extensions suggested in Section 3.1, ARGs can be used to specify and implement syntax analyzers for a large class of diagramming languages.

The worst-case time requirement of the parsing algorithm seems prohibitive. In practice, however, it is possible to achieve much better behavior with unambiguous grammars and inputs.

The any-start property makes Wittenburg's original model suitable for specifying incremental parsing interfaces to visual language applications. However, edit-compile style interfaces to visual languages do not benefit much from this feature. In fact, the any-start property introduces unnecessary complexity to syntax analysis. Fortunately, the context-free backbone of the grammar formalism and the parsing method make it possible to achieve more deterministic parsing behavior. Predicates can be used to disambiguate the parsing of structural variants of nonterminals that are in an inclusion relationship.

Wittenburg's algorithm is a recognizer rather than a parser. This means that the algorithm can decide whether a relational sentence belongs to a relational language but it does not impose any phrase structure on the sentence. The parsing table does contain enough information for constructing a parse tree for the sentence but the presence of many unnecessary states in the table makes the construction difficult. However, for visual programming applications like CASE-tools, it is necessary to obtain a parse tree for recognized sentences.

There is also another consequence due to the nature of the recognizer. In case of erroneous input, Wittenburg's algorithm just fails giving no information about the error. When considering the edit-compile style of visual programming, this is a serious weakness. As a minimum requirement, the parser should be able to indicate the piece of input that caused the failure. Further, the parser must be able to recover from syntactic errors in order to process as much input as possible during one parse.

Chapter 4

Extended Atomic Relational Grammars

In this chapter, we present our additions and changes to atomic relational grammars. The changes address the problems presented in Chapter 3 and they concern both the form of the productions and the parsing algorithm. The changes to the form of productions make it easier to express typical syntactic structures in visual diagramming languages. The changes to the parser make parsing deterministic which enables the effective handling of syntax errors. The changes limit the set of languages that can be recognized by the parser. However, the limitations are minor. We call the modified formalism *extended* atomic relational grammars (EARG).

First, in Section 4.1, we present the specification of extended atomic relational grammars. Next, in Section 4.2, we describe our predictive lookahead method that makes parsing EARG languages more deterministic than parsing ARG languages. Then, in Section 4.3, we discuss the changes to the parser: we describe how to handle the parsing of iterative symbols in productions, discuss the implementation of the predictive lookahead method, and show how to construct a parse tree. Finally, in Section 4.4, we make remarks of the complexity of parsing and the expressive power of extended ARGs.

4.1 Specification of Extended ARGs

Definition 4.1 A *extended atomic relational grammar* (EARG) is a 7-tuple $G = (N, \Sigma, S, R_e, Q, A, P)$, where

1. N is a finite set of *nonterminal symbols*.
2. Σ is a finite set of *terminal symbols* disjoint from N.
3. S is a distinguished symbol in N called the *start symbol*.

4. R_e is a finite set of relation symbols called the *expander relation symbols*.

5. Q is a finite set of *predicate symbols*.

6. A is a finite set of *expander attribute symbols* such that each nonterminal $X \in N$ is associated with a subset $A(X)$ of expander attributes, $A(X) \subset A$.

7. P is a finite set of *productions* of the form $B \rightarrow \alpha\ \beta\ \delta\ \gamma\ \omega\ F$, where

   $B \in N$;

   $\alpha \in ((N \cup \Sigma)(?| + | * |\epsilon))^+$ where

   $x?$ means that x is optional (zero or one),

   $x+$ means one or more, and

   $x*$ means zero or more.

   There must be at least one non-optional symbol in α (x or $x+$, where $x \in N \cup \Sigma$).

   β is a set of *relational constraints* of the form $r_e(y, z)$ where $r_e \in R_e$ and y, z are either terminal members of α or expressions of the form $C.a$ where $a \in A(C)$ and C is a nonterminal member x_i of α, $\alpha = x_1 \ldots x_i \ldots x_n$ ($x_i \in N$). Furthermore, Restriction 2.1 must hold for β.

   δ is a set of *predicates* of the form

   (a) $q(y_1, ..., y_n)$, or
   (b) *not exists* $v \in D : r_e(v, z)$ or *not exists* $v \in D : r_e(z, v)$, or
   (c) *exists some* $v \in D : r_e(v, z)$ or *exists some* $v \in D : r_e(z, v)$, or
   (d) *exists* k $v \in D : r_e(v, z)$ or *exists* k $v \in D : r_e(z, v)$

   where $q \in Q$, $D \subseteq \Sigma$, $r_e \in R_e$, z and y_i are references to α as in β, and $n, k \geq 1$. The predicates of type (b), (c), and (d) are remote references.

   γ is a set of *disambiguation constraints* of the form

   (a) *not exists* $v \in D : r_e(v, z)$ or *not exists* $v \in D : r_e(z, v)$, or
   (b) *exists* k $v \in D : r_e(v, z)$ or *exists* k $v \in D : r_e(z, v)$

   where $D \subseteq \Sigma$, $r_e \in R_e$, z is a reference to α as in β, and $k \geq 1$.

   ω is a set of *ordering expressions* of the form $order\ z_1, \ldots, z_n\ by\ f$ where $n \geq 1$ and z_i are references to α as in β such that:

   (a) if $n = 1$, the symbol referenced by z is iterative, and
   (b) if $n > 1$, all the symbols referenced by z_i are non-iterative and non-optional.

   $f(u, v)$ is a function $f : J \times J \rightarrow \{true, false\}$, where $J \subseteq I$, I is an indexed multiset of input symbols, and f implies a total order on J.

F is a set of *attribute assignment* statements of the form $B.a = x$ where $a \in A$ and x is either a terminal member of α or an expression of the form $C.a$ as in β. Further, there must be exactly one attribute assignment statement $B.a_i = \ldots$ for each $a_i \in A(B)$.

When compared with the original definition of ARGs (Definition 2.4), Definition 4.1 above introduces several differences. The definition distinguishes predicates from expander constraints and introduces iterative right-hand symbols, remote references, disambiguation constraints, and ordering expressions. In the following, we discuss the changes.

Iterative Right-Hand Side Symbols

The first change in the definition of the productions P is the addition of standard *iteration markers* in α. A right-hand side symbol can be followed by exactly one of the markers '?', '$*$', or '$+$'. The marker '?' means that the symbol is optional (zero or one), the marker '$*$' means a sequence of zero or more, and the marker '$+$' denotes a sequence of one or more. The introduction of iterative symbols has several consequences.

Relational constraints in productions may refer to symbols followed by an iteration marker. These constraints are interpreted as follows. Let G be an EARG and let p be a production of G, $p \in P_G$, Let $r_e(x, y)$ be a relational constraint of p, $r_e(x, y) \in p_\beta$, such that either x or y (or both) refers to an iterative symbol. In the context of p, let I_x be the set of input objects represented by x and let I_y be the set of input objects represented by y. Then, $r_e(x, y)$ is evaluated as true if and only if $r_e(a, b)$ for each $(a, b) \in I_x \times I_y$.

The iteration markers '?' and '$*$' mean that the marked symbol is optional. That is, the symbol may not be present in some instance of p. However, Restriction 2.1 (p. 22) states the connectedness constraint that must hold also when there are optional symbols in a production. So, even if an optional right-hand side symbol $x_i \in p_\alpha$ of some ordering variant of p is not present, the next unprocessed symbol x_{i+1} must be 'connected' by some relational constraint to a non-optional symbol x_j, $j < i$, in the already processed part $x_1 \ldots x_{i-1}$ of p_α.

Predicates

The second addition to the definition of productions is the set of predicates, δ. In addition to the constraints β, also the predicates represent necessary conditions that must hold for the parser to recognize a nonterminal.

The definition includes the set Q of predicate symbols that are distinct from the relation symbols R_e. A predicate $q \in Q$ can be of any arity greater than 0; q denotes an external function that can be invoked with a list of arguments and that returns either *true* or *false*. The predicates of type (a) in the definition of δ are these kind of predicates.

The predicates of types (b), (c), and (d) are remote references (see Section 3.1.2). They are used to enforce the existence of some or a fixed number of terminal symbols in the given relations with the terminals that can be referenced within the production. That is, given

- production p,
- remote reference $r_e(v, z)$ ($r_e(z, v)$),
- finite indexed multiset I with an index set $L = \{1, \ldots, |I|\}$ that is input to the parser, and
- object (constant) $a \in I$ that is bound to the right-hand side reference z in $r_e(v, z)$ ($r_e(z, v)$)

the following formulas determine the Boolean value of $r_e(v, z)$ ($r_e(z, v)$),:

- case (b)

$$\nexists v \in I : v \in D \wedge r_e(v, a)$$

$$(\ \nexists v \in I : v \in D \wedge r_e(a, v)\ )$$

- case (c)

$$\exists v \in I : v \in D \wedge r_e(v, a)$$

$$(\ \exists v \in I : v \in D \wedge r_e(a, v)\ )$$

- case (d)

$$\exists v_{i_1}, \ldots, v_{i_k} \in I : ((v_{i_1} \in D \wedge \ldots \wedge v_{i_k} \in D)$$

$$\wedge (v_{i_1} \neq v_{i_2} \wedge \ldots \wedge v_{i_{k-1}} \neq v_{i_k}) \wedge (r_e(v_{i_1}, a) \wedge \ldots \wedge r_e(v_{i_k}, a))$$

$$\wedge (\forall j \in L : (r_e(v_j, a) \rightarrow (v_j = v_{i_1} \vee \ldots \vee v_j = v_{i_k})))$$

$$\Big(\ \exists v_{i_1}, \ldots, v_{i_k} \in I : ((v_{i_1} \in D \wedge \ldots \wedge v_{i_k} \in D)$$

$$\wedge (v_{i_1} \neq v_{i_2} \wedge \ldots \wedge v_{i_{k-1}} \neq v_{i_k}) \wedge (r_e(a, v_{i_1}) \wedge \ldots \wedge r_e(a, v_{i_k}))$$

$$\wedge (\forall j \in L : (r_e(a, v_j) \rightarrow (v_j = v_{i_1} \vee \ldots \vee v_j = v_{i_k})))\ \Big)$$

Disambiguation Constraints

The third addition is the set of disambiguation constraints, γ. These constraints are used to prune out structural variants during parsing (see Section 3.2.1). Let productions p_1 and p_2 be in an inclusion relationship. That is, there is an injective, one-to-one mapping from the symbols, constraints, predicates, and attribute assignments of p_1 to p_2. In other words, p_1 is parsed whenever p_2 is parsed (see the grammar in Example 3.1). Then, p_1 should be annotated with disambiguation constraints to ensure that if p_2 is parsed, then p_1 will not be parsed. Given p, I, L, and a as above, the following formulas determine the value of a disambiguation constraint in p:

– case (b)

$$\nexists v \in I : v \in D \wedge r_e(v, a)$$
$$(\nexists v \in I : v \in D \wedge r_e(a, v))$$

– case (c)

$$\exists v_{i_1}, \ldots, v_{i_k} \in I : ((v_{i_1} \in D \wedge \ldots \wedge v_{i_k} \in D)$$
$$\wedge (v_{i_1} \neq v_{i_2} \wedge \ldots \wedge v_{i_{k-1}} \neq v_{i_k}) \wedge (r_e(v_{i_1}, a) \wedge \ldots \wedge r_e(v_{i_k}, a))$$
$$\wedge (\forall j \in L : (r_e(v_j, a) \rightarrow (v_j = v_{i_1} \vee \ldots \vee v_j = v_{i_k})))$$
$$\Big(\exists v_{i_1}, \ldots, v_{i_k} \in I : ((v_{i_1} \in D \wedge \ldots \wedge v_{i_k} \in D)$$
$$\wedge (v_{i_1} \neq v_{i_2} \wedge \ldots \wedge v_{i_{k-1}} \neq v_{i_k}) \wedge (r_e(a, v_{i_1}) \wedge \ldots \wedge r_e(a, v_{i_k}))$$
$$\wedge (\forall j \in L : (r_e(a, v_j) \rightarrow (v_j = v_{i_1} \vee \ldots \vee v_j = v_{i_k}))) \Big)$$

For example, we can now write the binary tree grammar from Example 3.1 as follows:

Example 4.1

Tree $\rightarrow$ *Node*
  not exists $c \in \{circle\} : connected(Node.root, c)$
  Tree.root = Node.root

Tree$_1$ $\rightarrow$ *Node Tree*$_2$
  connected(Node.root,Tree$_2$*.root)*
  exists 1 $c \in \{circle\} : connected(Node.root, c)$
  Tree$_1$*.root = Node.root*

Tree$_1$ $\rightarrow$ *Node Tree*$_2$ *Tree*$_3$
  connected(Node.root,Tree$_2$*.root)*
  connected(Node.root,Tree$_3$*.root)*
  order Tree$_2$*.root,Tree*$_3$*.root by left*
  Tree$_1$*.root = Node.root*

Node $\rightarrow$ *circle text*
inside(text,circle)
Node.root = circle

The predicates of types (b) and (d) and the disambiguation constraints of types (a) and (b) have the same form. However, there is an important difference between predicates and disambiguation constraints. The predicates (and remote references) are required conditions for recognizing valid syntactic structures. The disambiguation constraints are used to make parsing deterministic in order to avoid partial parses that cannot cover the whole input. Therefore, it is a potential syntax error if a predicate fails in production p. On the other hand, an unsatisfied disambiguation constraint in p indicates that parsing can continue normally although p must be discarded because some production q, which includes p, should be parsed instead.

Ordering Expressions

The fourth addition is the set of ordering expressions, ω. These expressions have two distinct but related purposes. Basically, they provide the parser a mechanism to unambiguously choose the next input object to scan in situations where an expander query returns multiple input objects.

We recognize two situations where we need ordering expressions. In the first case, there is an iterative symbol on the right-hand side of a production as shown by the example below. In this example, an ordering expression is used to obtain a left-to-right ordering of subtrees in a general tree:

Tree$_1$ $\rightarrow$ *Node Tree*$_2*$
connected(Node.root,Tree$_2$*.root)*
order Tree$_2$*.root by left*
Tree$_1$*.root = Node.root*

That is, when querying input symbols (see Procedure 2.1, *Advance*) to start parsing the iterative terminal or nonterminal, the parser sorts the input objects returned by the query using the sorting function f as declared by the ordering expression. The parser will then process the objects in this order (see Section 4.3.1). Therefore, the sorting function must imply a total order on those input objects that may be returned by the query (this could mean all of the input objects but not necessarily).

The second case does not involve iterative symbols. The syntax of the ordering expressions makes it possible to declare the same ordering functions for many non-iterative right-hand side symbols. For instance, in the following production

from the grammar in Example 4.1, an ordering expression is used to enforce a left-to-right ordering of the subtrees in a binary tree:

Tree$_1$ $\rightarrow$ *Node Tree*$_2$ *Tree*$_3$
connected(Node.root,Tree$_2$*.root)*
connected(Node.root,Tree$_3$*.root)*
order Tree$_2$*.root,Tree*$_3$*.root by left*
Tree$_1$*.root = Node.root*

In the example above, there are two symbols in the *connected* relation with the symbol *Node*. For instance, let us assume that the parser is processing input that represents a binary tree such as in Figure 3.3 (page 44) according to the production above. After parsing the topmost *Node* (consisting of *circle*$_1$ and the text object 'A') and when starting to parse *Tree*$_2$, the next expander query, say ψ, will return two input objects (*circle*$_2$ and *circle*$_3$) that represent the root nodes of the two subtrees. Without the ordering expression, the parser could not determine which *circle* should be associated with *Tree*$_2$ and which one with *Tree*$_3$. So, the parser would, in this case, start two separate subparses for *Tree*$_2$ from both *circle*$_2$ and *circle*$_3$. A similar sequence of events would then occur when starting to parse *Tree*$_3$. Eventually, this would lead to the situation where the parser would have produced two different parses for the same input. However, if the ordering expression is specified for the production (as above), the parser is able to associate the input objects returned by ψ with the right subtrees: the parser sorts first *circle*$_2$ and *circle*$_3$ using function *left* and then it associates the ordered input objects with the symbols in the order in which the symbols appear in the expression. That is, the first object is associated with *Tree*$_2$ and the second object with *Tree*$_3$. So, the order of the (references to) symbols in the ordering expression is also significant.

Predicates can also be used to enforce certain orderings of right-hand side symbols as shown by the example in Section 3.2.3 (page 48). However, this adds overhead to the parsing and makes error recovery more difficult because the parser may create ambiguous and redundant subparses that represent all the possible orderings of the multiplicated right-hand side symbols, and when the parser finally can evaluate the predicates, it only then discards the redundant subparses. But if ordering expressions are used, the parser will not create the redundant subparses, in the first place.

In the definition of ω (in Defintion 4.1), we stated that the form of ordering expressions with a list of references to right-hand side symbols can be used only for non-iterative symbols. The reason for this will be explained below in Section 4.2, where we give also some additional restrictions for the ordering expressions in a production.

Attribute Assignments

The form of the attribute assignments F has not been changed. However, to guarantee that attributes are always assigned a value and that the value is unambiguous, we give Restriction 4.1.

Restriction 4.1 *Repetitive or optional right-hand side symbols cannot be used in the attribute assignments in F.*

4.2 Predictive Lookahead

One of our goals in extending atomic relational grammars was to make parsing more deterministic. With deterministic parsing we mean:

- pruning redundant structural variants of nonterminals (see Section 3.2.1) with the help of disambiguation constraints,
- using filtering based on the terminal type of input objects to avoid starting parsing nonterminals from input objects that cannot possibly 'begin' such nonterminals, and
- using the ordering expressions to deterministically select the next input object to be scanned from a set of candidate objects.

The last two items in the list above comprise our *predictive lookahead* method. The method uses three properties of an EARG grammar G to filter and order the input objects returned by an expander query ψ for an active state a. Let active state $a = [p, i, (d_1, \ldots, d_n), \ldots]$, where c is the expander constraint used as the basis of ψ and X_i is the symbol at the dot i in p. In the method, we use the following information :

1. the expected types of terminals returned by the query,
2. the number of terminals expected to be returned by the query *in the context of p*, and
3. the ordering expression $\phi \in \omega$ for X_i in p (if specified).

In the following, we explain how the lookahead method uses this information; the actual implementation of the method is described later in Procedure 4.6. First, we describe how we filter input objects returned by the query ψ based on the expected (terminal) types of the objects. Then, we describe how we use the ordering expressions to deterministically select the next input object to be scanned from a set of candidates.

Expected Types of Terminals

Let z be a reference to a symbol X in a constraint, a predicate, a remote reference, or an ordering expression. We define $terminals(z)$ as the set of the (types of) terminals that z can possibly represent:

$$terminals(z) = \begin{cases} \{X\} & : \quad z \text{ is of form } X, X \in \Sigma \\ First(X, \alpha) & : \quad z \text{ is of form } X.\alpha, X \in N, \alpha \in A(X) \end{cases}$$

First(X, α) is the set of terminals that can be bound to the expander attribute α of the nonterminal X. We construct the set for each pair $(X, \alpha) \in N \times A$ based on the attribute assignments for $X.\alpha$ in the productions by:

1. For each $X \in N$ and each $\alpha \in A(X)$, let *First*$(X, \alpha) = \emptyset$.
2. For each assignment $X.\alpha = z$ in G, where $z \in \Sigma$, add z to *First*(X, α).
3. For each assignment $X.\alpha = z$ in G, where $z = Y.\beta$ and $Y \in N$, add *First*(Y, β) to *First*(X, α).
4. Repeat step 3 until no more new items are added to any of the sets *First*(X, α).

This algorithm is a modification of the method presented in [ASU86, p. 189] for computing the FIRST sets for string grammars. The differences are that we have to compute the sets for nonterminal-attribute pairs and that there are no ϵ-productions in EARGs.

In our lookahead method, we use the *terminals* sets as follows. Let ψ be an expander query, let c be the relational constraint used as the basis of the query, and let z be the reference to the next object to be parsed in c. Now, when examining the set J of objects returned by ψ, we can discard all the objects in J that do not belong to $terminals(z)$ because those objects cannot possibly 'begin' the symbol referenced by z. So, we get a new query result

$$J' = J \cap terminals(z).$$

Note that with the help of the *First* sets, we can also check that each nonterminal reference actually 'grounds out' in an EARG grammar. That is, $First(X, \alpha) \neq \emptyset$ for all $X \in N$ and all $\alpha \in A(X)$.

Using Ordering Expressions

We mentioned earlier on page 60 two situations where an expander query might return multiple input objects. In the following, we analyze these situations in more detail. In the analysis, we consider only the local context of one production (see the discussion below on page 66 about this restriction).

Let $c = R_j(u, v)$ (or $c = R_j(v, u)$) be the constraint used in the expander query ψ, as above. Let v be the reference to X_i (the symbol to be parsed next) and let u be a reference to a symbol X_k, $k < i$ (a symbol that has already been parsed). We call X_k the *anchor* of c in ψ. Let r_c be the expander relation in the constraint c. The result set J returned by ψ will initially hold all the input objects that are in relation r_c with the anchor object.

The first thing to do is to remove from J all the objects that have a wrong terminal type:

$$J' = \{o | o \in J \wedge term(o) \in terminals(X_i)\},$$

where the function *term* returns the terminal type of an input object. Then, if $|J'| > 1$, we have two possibilities. First, if X_i is iterative and there is an ordering expression ϕ specified for X_i, we sort J' with f, and continue as explained in Section 4.3.1. If there is no ordering expression for X_i, an arbitrary order is assumed.

Second, the situation may be as shown by the binary tree example on page 61. That is, there are other constraints in the procution, $b_1, \ldots, b_l$, such that they have a reference to X_k (the anchor object) at the same position as in c and they have the same expander relation, $r_{b_j} = r_c$ for all b_j. Now, if the production p has an ordering expression ϕ specified for exactly those symbols that are referenced by the constraints $\{c, b_1, \ldots, b_l\}$, the lookahead method can unambiguosly choose an object from J' that corresponds to X_i *iff* the additional restrictions, which are given below, hold. First, the input objects are sorted with the ordering function f. Then, if m is the index of the reference to X_k in the list of references in ϕ, let the final result be:

$$J'' = \{o_m\},$$

where the o_m is the m^{th} object in the sorted sequence of input objects, or

$$J'' = \emptyset$$

if $|J'| < m$ (incorrect input for p).

Let D be the set of right-hand side symbols that are referenced by the constraints $\{c, b_1, \ldots, b_l\}$, excluding X_k. The selection of o_m is unambiguous only if all the symbols in D are *non-iterative* and *non-optional* because the method expects there to be a one-to-one mapping between the input objects in J' and the list of references in ϕ. Otherwise If $|J'| \neq |D|$, there is no way to know, by looking at the input objects in J', which optional symbols in D might be missing or which iterative symbols map to some subset in J'.

Furthermore, in the second case above, the number of (distinct) symbols in D gives directly an upper limit to the number of expected objects in J' in the context of p. If only the local parsing context of p needs to be taken into consideration (see Restriction 4.5 below), it can be regarded as a syntax error if $|J'| > |D|$.

If there is no ordering expression, there is no way to construct a consistent one-to-one mapping between the input objects in J' and the right-hand side symbols of p. If the objects in J' are ordered arbitrarily, the ordering may change every time

the objects are ordered ($\leq |\{c, b_1, \ldots, b_l\}|$ times). In this case, one possibility would be to first remove from J' those objects that already belong to the cover of a (the active state that represent the part of the production that has already been parsed, see page 62); then, if all the other symbols except X_k that are referenced by $\{c, b_1, \ldots, b_l\}$ are non-iterative and non-optional, any one of the remaining objects in J' could be returned as the result. However, in the current implementation of EARG grammars, we fall back to the default mode of the original ARG parser and return J' as the final result of the query ϕ.

Additional Restrictions for Ordering Expressions

In addition to the restriction given in the specification of ω in Definition 4.1, the ordering and selection scheme described above works only under the following restrictions for an EARG grammar G.

To make the selection of the ordering expression unambiguous, we give the following restriction:

Restriction 4.2 *Let X be a right-hand side symbol in a production p. There can be only one ordering expression in ω for X, $X \in \Sigma$, or for $X.\alpha$, $\alpha \in A(X)$ and $X \in N$.*

Given an ordering expression ϕ and the references $z_1, \ldots, z_n$ to the right-hand side symbols of the production p, the following must hold:

Restriction 4.3

$$terminals(z_1) = terminals(z_2) \wedge \cdots \wedge terminals(z_{n-1}) = terminals(z_n)$$

Restriction 4.3 makes sure that the filtering based on just one of the references z_i is valid for all $z_1, \ldots, z_n$.

In the following, we assume for simplicity that X, Y, and W are terminals:

Restriction 4.4 *Let X and $Y_1, \ldots, Y_n$, be distinct right-hand side symbols of production p such that $n > 1$ and that there exist in p a constraint $r_e(X, Y_i)$ ($r_e(Y_i, X)$) for all $Y_i \in [Y_1, \ldots, Y_n]$ where $r_e \in R$. If $\bigcap_{i=1}^{n} terminals(Y_i) \neq \emptyset$, there must be an ordering expression $\phi \in p$ such that all Y_i appear in the list of references of ϕ.*

Note that the restriction in the specification of ω implies that in the case above, Y_i have to be non-iterative symbols. Furthermore, Restriction 4.3 actually implies that:

$$terminals(Y_1) = terminals(Y_2) \wedge \cdots \wedge terminals(Y_{n-1}) = terminals(Y_n).$$

Also, Restrictions 4.2 and 4.4 together imply that all the symbols Y_i (or symbol-attribute pairs $Y.\alpha$, $Y \in N$ and $\alpha \in A(Y)$) that fulfill the criterion in Restriction 4.4 must appear in exactly one ordering expression of p.

All the restrictions above can be checked statically.

Lookahead Context

The predictive lookahead method described above depends on the property of G that only the *local context* in a production p needs to be considered when computing the expected size of a query result J and when mapping multiple query objects to right-hand side symbols. That is, we need to consider only the constraints in the *current production* to map query objects to right-hand side symbols. For example, the following grammar does not have this property. In production $A \rightarrow d\ b$, if we look at the local context only, a query over r with d as the bound argument is expected to return exactly one object, a b. However, because the same d is assigned to $A.\alpha$, it is also constrained to be in relation r with another b in production $S \rightarrow A\ b\ c$. Therefore, the query will return two b terminals.

S → *A b c*
  r(A.α,b)
  r(b,c)
  S.α = A.α

A → *d b*
  r(d,b)
  A.α = d

Of course, it would be possible to compute the closure of the contexts to be used in determining the size of J by following the propagation of references to terminals through the attribute assignments in G. However, in the example above, it would still be impossible to decide which one of the two b objects in the result set belongs to the context of A without more information; the decision is context-dependent.

For enabling lookahead, we therefore give the following contextual restriction to EARG grammars:

Restriction 4.5 *For each production $p \in P$ of an EARG grammar G, it must be possible to determine the mapping of input objects returned by an expander query to right-hand side symbols by considering only the local context.*

It is possible to make a static check to enforce this condition. This is because nonterminals can be referenced only through their syntactic expander attributes in the right-hand side of production. Therefore, it is possible to analyze the propagation of references to terminals via the expander attributes (i.e. to build a data flow graph) to check all the contexts where a terminal may be constrained by a relational constraint.

Disambiguation Constraints and Ordering Expressions

When multiple input objects returned by expander queries are processed by the predictive lookahead method, disambiguation constraints are not needed in certain situations to make parsing deterministic. This is because we can now set a limit for the number of expected objects in a query result in the local context (current production) of parsing. For example, compare the following binary tree grammar with the grammar in Example 4.1 on page 59.

Example 4.2
Tree → *Node*
not exists $c \in \{circle\} : connected(Node.root, c)$
Tree.root = Node.root

$Tree_1$ → *Node* $Tree_2$
connected(Node.root,$Tree_2$*.root)*
$Tree_1$*.root = Node.root*

$Tree_1$ → *Node* $Tree_2$ $Tree_3$
connected(Node.root,$Tree_2$*.root)*
connected(Node.root,$Tree_3$*.root)*
order $Tree_2$*.root,*$Tree_3$*.root by left*
$Tree_1$*.root = Node.root*

Node → *circle text*
inside(text,circle)
Node.root = circle

The difference is in production $Tree_1$ → *Node* $Tree_2$. In the grammar in Example 4.2 above, there is no disambiguation constraint to distinguish it from production $Tree_1$ → *Node* $Tree_2$ $Tree_3$. The constraint is not needed because after parsing the *Node*, result of the query *connected(Node.root,?)* is expected to hold only one input object of terminal type *circle*. If there are more objects in the result, this is considered as a syntax error and the parsing of this production is not continued. However, the disambiguation constraint in production *Tree* → *Node* is still necessary.

The grammars in Examples 4.1 and 4.2 fulfill Restrictions 4.2, 4.3, 4.4, and 4.5.

4.3 Parsing Extended ARGs

The new features in extended ARGs imply changes to the parsing algorithm. However, the changes are extensions to the parser rather than changes to its fundamentals. The extensions are:

1. support for parsing iterative right-hand side symbols,
2. the use of predictive lookahead,
3. the use of disambiguation constraints,
4. the construction of a parse graph and a parse tree, and
5. the recovery from syntax errors.

Iterative right-hand side symbols make it easy to write compact productions that include list-like substructures. The second and the third extension make parsing more deterministic than in the original algorithm (Algorithm 2.1). The predictive lookahead method (described above in Section 4.2) uses lexical filtering (based on terminal types), the local parse context (production), and the ordering expressions of a production to associate unambiguously multiple input objects (returned by a single expander query) with the right-hand side symbols of the production. This removes many unnecessary steps from parses. The usage of disambiguation constraints reduces the number of steps even further.

Deterministic parsing is more efficient in terms of the steps taken by the parser when compared with the original parsing method. It is also an enabling property for the effective handling of syntax errors. The construction of a parse graph is also fundamental for error recovery whereas the construction of a parse tree as an intermediate representation of a visual program makes it possible to apply many kinds of conventional post-parse transformations (translation, code generation) to the program.

In this section, we describe how the extensions are integrated into Algorithm 2.1. However, because of the complexity of error handling and recovery, we will treat it separately in Chapter 5 where we present the complete EARG parsing algorithm (Algorithm 5.1, p. 98).

As a general observation, we have been able to retain the overall design of the original algorithm; the extensions are isolated in a few procedures and data structures. The support for iterative right-hand side symbols and the predictive lookahead method imply the biggest changes.

4.3.1 Parsing Iterative Symbols

The parsing of iterative right-hand side symbols imply extensions to the Advance procedure (Procedure 2.1) of the parsing algorithm. Iterative symbols require also changes in the representation of the active states.

Iterative right-hand symbols are supported in active states by a queue that holds the input objects that can be bound to an iterative symbol at the dot. The idea is that when the dot is first moved to an iterative symbol, a query is launched to find all the input objects that can be bound to the iterative symbol and the objects are placed in the queue. If an ordering expression is given for the iterative symbol, the

objects will be ordered with the first object at the head of the queue; otherwise, the order is arbitrary. Then, subsequent steps to advance the parse (to move the dot over to the next symbol) will consume an object from the head of the queue instead of launching an expander query. Based on the type of the symbol, either a terminal is scanned or a parse for a new nonterminal instance is started. Only when the queue of pending objects is exhausted, the *Advance* procedure will move the dot over to the next symbol.

An active state also holds a list of those optional symbols that are not present in the current parse. With the list, the constraints and predicates that refer to the missing optional symbols can be excluded from the evaluation of constraints and predicates.

Our new definition of active state is:

Definition 4.2 *An active state is a quintuple*

$$[\, p,\, i,\, (d_1, \ldots, d_n),\, (o_1, \ldots, o_k),\, (X_1, \ldots, X_m)\,]$$

where p is a production; i is the 'dot', $i \in [1, \ldots, |$right-hand side of $p|]$, that represents the next element to parse; $(d_1 \ldots d_n)$, $n \leq |$right-hand side of $p|$, is an ordered list of pointers to inactive states or sets of pointers to inactive states (for repetitive symbols) of right-hand side elements parsed so far; $(o_1, \ldots, o_k)$ is an ordered list of pending input objects that can be bound to the iterative symbol at i; and $(X_1, \ldots, X_m)$, $m < |$right-hand side of $p|$, is a list of those optional right-hand side symbols of p that are not present in the parse represented by the state.

The semantics of Advance has not been changed: it either creates an inactive state representing a parsed nonterminal instance or it advances a current sub-parse to consume new input objects. The extensions are the handling of iterative symbols and the evaluation of predicates and constraints.

The idea is to evalutate predicates and disambiguation constraints as soon as all their arguments (references to the right-hand side symbols) can be resolved. This means that we evaluate the predicates and the constraints immediately when all the symbols in their arguments have been parsed. In the following procedures, 'constraints' mean both expander and disambiguation constraints.

Procedure 4.1 *Advance (a, i)*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$ and an inactive state $i = [\, cat\,, f, c]$.

Output: A new agenda item or *null*.

Method:

If $j = |$*right-hand side of* $p|$ **and** the queue of pending input objects of a is empty **then**

Return $closeParse(a, i)$

else

Create a new unitialized active state a'.
If the symbol at the dot in a is repetitive **and** there are objects in the queue of pending input objects of a **then**
let $t = consumePendingInput(a, i, a')$
else
let $t = queryForInput(a, i, a')$.
end if
Return t.

end if
/* *end Advance* */

Procedure *closeParse* creates a new inactive state that represents a recognized nonterminal instance. It also checks that the predicates and constraints that refer to the last right-hand side symbol of p (represented by i) evaluate to true.

Procedure 4.2 *closeParse (a,i)*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$ and an inactive state $i = [cat, f, c]$

Output: A new agenda item or *null*.

Method:

i is the last right-hand side element of p and a new inactive state i' is created. If the symbol at j is repetitive, the daughters of i', $D_{i'}$, is determined by the following rule: if i is the first instance of the repetitive nonterminal, $D_{i'} = (d_1, \ldots, d_n, \{i\})$; else $D_{i'} = (d_1, \ldots, d_n \cup \{i\})$. The cover of i' is computed as union of the cover of i and the covers of the parsed right-hand side elements of a, and the expander attribute values of the nonterminal B in question (the left-hand side of p) are determined.

If all the predicates and constraints in i' that refer to the symbol at j evaluate to $true$ **then**

Return a new agenda item $[i', keys]$ where $keys$ is the list of inactive state indices of i' (the values of the expander attributes $A(B)$).

else

Return $null$

end if

/* *end closeParse* */

Procedure *consumePendingInput* consumes the object from the head of the queue of pending input of active state a. It initializes the new active state a' and evaluates the predicates and constraints of a' pertaining to the right-hand side symbol of p at position j that i represents. The procedure returns a new agenda item or *null* if the predicates and constraints are not satisfied.

Procedure 4.3 *consumePendingInput (a,i,a')*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$, an inactive state $i = [cat, f, c]$, and a new unitialized active state a'.

Output: A new agenda item or *null*.

Method:

Set the dot position of a' to j (the same position as a). The inactive state i is merged with the parsed right-hand side elements of a as the daughters of a'; repetitive daughters are represented as an ordered set of pointers. If i is the first instance of the repetitive nonterminal, the daughters of a' $D_{a'} = (d_1, \ldots, d_n, \{i\})$; else $D_{a'} = (d_1, \ldots, d_n \cup \{i\})$. The input object o_1 is removed from the head of the queue and the remaining queue is assigned to a',

$$a' = [p, j, D_{a'}, (o_2, \ldots, o_k), (X_1, \ldots, X_m)].$$

If all the predicates and constraints in a' that refer to the symbol at j and that can be evaluated are $true$ **then**
    Return a new agenda item $[a', \{o_1\}]$
else
    Return $null$
end if
/ end consumePendingInput */*

Procedure *queryForInput* finds the next input object to be scanned. The main issue is the handling of iterative symbols and missing optional symbols. The procedure uses the predictive lookahead function *filter* (page 73) to reduce the number and the types of terminals returned by an expander query. If there is an ordering expression for the symbol at $j + 1$, *filter* will also order the reduced set accordingly.

Procedure 4.4 *queryForInput (a,i,a')*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$, an inactive state $i = [cat, f, c]$, and a new unitialized active state a'.

Output: A new agenda item or *null*.

Method:

Initialize a'; the inactive state i is added with the parsed right-hand side elements of a as the daughters of a', $a' = [p, j+1, (d_1, \ldots, d_n, i), \ldots]$.

If all the predicates and constraints in a' that refer to the symbol at j and that can be evaluated are $true$ **then**

Execute a query to find the set of input objects J to be parsed next by using any one of the expander constraints at the new dot position and the right-hand side elements parsed so far. Let $J' = filter(a, i, a', J)$.
If $J' \neq \emptyset$ **then**
Return a new agenda item $[a', J']$
else
If the symbol at the dot of a' is optional **then**
Return $skipOptionalSymbol(a')$
end if
end if
else
Return $null$
end if
/* *end queryForInput* */

Procedure *skipOptionalSymbols* parses a sequence of missing optional symbols in a production. It creates intermediate active states for each advancement and adds the missing symbol to the list of missing optional symbols in each state. The procedure terminates when the next symbol in the production is not optional, an input object is found that may begin the next optional symbol, or when the production ends.

Procedure 4.5 *skipOptionalSymbols (a)*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$.

Output: A new agenda item or *null*.

Method:

Let $t = null$
loop
  Let $a' = [p, j_a + 1, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m, Y)]$, where Y is the symbol at the dot (position j_a) in a;
  Let $e =$ a dummy inactive state that represents a missing optional symbol.
  If $j_{a'} = |$right-hand side of $p|$ **then**
    Return $closeParse(a', e)$
  end if
  Launch a query to find the input objects J to be parsed next by using one of the expander constraints at the new dot position and the right-hand side elements parsed so far. Let $J' = filter(a, e, a', J)$.
  If $J' \neq \emptyset$ **then**
    let $t = [a', J']$
  end if
  Let $a = a'$
until $t \neq null$ **or** the symbol at the dot of a is not optional
Return t
/* *end skipOptionalSymbols* */

4.3.2 Implementation of Predictive Lookahead

The *filter* procedure, called by *queryForInput* and *skipOptionalSymbol*, integrates the processing techniques described above in Section 4.2 into one lookahead function. In the procedure, function *term* returns the terminal type of an input object, and function *cover* returns the input objects that belong to the cover of an active state (the objects that have already been parsed).

Procedure 4.6 *filter (a,i,a',J)*

Input: An active state $a = [p, j, (d_1, \ldots, d_n), (o_1, \ldots, o_k), (X_1, \ldots, X_m)]$, an inactive state $i = [cat, f, c]$ used to advance a, the next active state a' advanced from a, and a set J of input objects.

Output: An ordered subset of J.

Method:

Let c be the constraint used in the expander query that returned the set J and let z be the reference in c to the symbol $X_{j_{a'}}$ at the dot $j_{a'}$ in a'.
If $X_{j_{a'}}$ is iterative **then**
  Let $J' = \{o \mid o \in J \wedge term(o) \in terminals(z) \wedge o \notin cover(a')\}$.
  If $J' = \emptyset$ **then**
    Return J'
  else
    If $X_{j_{a'}}$ has an ordering expression ϕ, then sort the objects of J' into list $L = \{o_1, \ldots, o_k\}$ using the sorting function f in ϕ. Else, insert the objects of J' into L in an arbitrary order. Let $J'' = \{o_1\}$ and let $\{o_2, \ldots, o_k\}$ as the pending input of a'.
    Return J''
  end if
else
  Let $J' = \{o \mid o \in J \wedge term(o) \in terminals(z)\}$.
  If $|J'| > 1$ **then**
    If $X_{j_{a'}}$ has an ordering expression ϕ **then**
      Sort the objects of J' into list $L = \{o_1, \ldots, o_k\}$ using the sorting function f in ϕ. Let l be the ordinal position of $X_{j_{a'}}$ in ϕ. If $l \leq k$, let $J'' = \{o_l\}$.
    else
      Let $J'' = J'$
    end if
    Let $J''' = \{o \mid o \in J'' \wedge o \notin cover(a')\}$
    Return J'''
  else
    Let $J'' = \{o \mid o \in J' \wedge o \notin cover(a')\}$
    Return J''
  end if
end if
/* *end filter* */

In the procedure above, we remove from the result set those input objects that are already part of the cover of the active state a. This is because such input objects would eventually be discarded anyway in subsequent parse actions (see Algorithm 2.1 on page 33).

4.3.3 Building a Parse Tree

Wittenburg's parser is a recognizer rather than a "real" parser in the sense that it does not impose any explicit phrase structure on relational sentences. In other words, it does not produce a parse tree. Also in general, the concepts of parse trees and parse traces are somewhat unclear in the context of visual languages, and their central role as universal intermediate representations of language processing has not been fully recognized.

However, as shown in [Tuo98a], an explicit parse graph can be constructed by linking the parse states in the parse table to each other to represent the creational relationships induced by parse actions. Further, as described in [Tuo98b], inactive states can be linked to each other to form a parse tree that imposes a phrase structure on the parsed input. In the following, we summarize these techniques.

In the parse table, we recognize two kinds of relationships between parse states. States s_i and s_j are in the *succession* relationship, $Succeeds(s_j, s_i)$, if s_j was created from s_i as a result of a scan, complete, or inverse-complete operation applied to s_i. States s_i and s_j are in the *prediction* relationship, $Predicts(s_i, s_j)$, if s_j was created from s_i as a result of the prediction operation. That is, the *prediction* relation links the state(s) where a prediction for a nonterminal instance was made to the predictive states that begin the parses of the instance according to all the alternative productions. The states that are linked by the *Succeeds* and *Predicts* relationships form a *parse graph* that captures the trace of the parsing process.

An active state s has a collection of pointers to *daughters* that are inactive states representing the instances of the nonterminals to the left of the dot. A predictive state has no daughters whereas a state with the dot at the end has all its constituents as daughters. It is straightforward to store the daughters also by the inactive states to create a parse tree that captures explicitly the implicit phrase structure imposed on the input by the parser.

Parse Graph

In Figure 4.1, the parse states are explicitly linked to form a directed *parse graph* with multiple roots (states 1, 2, and 3). The solid edges between states represent the *succession* relationship, advancing a parse for a nonterminal instance as a result of a scan, complete, or inverse-complete operation. Dashed edges depict the *prediction* relationship, i.e. they link the state where a prediction for a nonterminal was made to the predictive states that begin the parses of the nonterminal instance according to all the alternative productions. Inactive states are indicated by a frame around the state number. Active states with no outgoing edges represent “dead-ends”, that is, they terminate an unsuccessful parse path for some nonterminal instance.

The parse graph makes it possible to trace the parsing process *a posteriori*. For example, it can be seen from the parse graph that the parses starting from states 1 and 2 have failed completely. The parse path starting from 3, on the other hand, leads to a successful parse of the input: the scanning of the *circle* object creates the successor link from 3 to 4, completing 3 with 6 (a *Node*) leads to state 7, completing 7 with 14 (a *Tree*) leads to state 15, and, finally completing 15 with 22 (a *Tree*) leads to state 23 that covers the whole input. Note that during the traversal of the parse graph, if a state has successors, the possible prediction links can be ignored because the successors indicate the advancement of parsing.

The parse graph clearly shows how different parse paths converge. For instance, state 4 has three originators, 1, 2, and 3, so it was predicted by all three states.

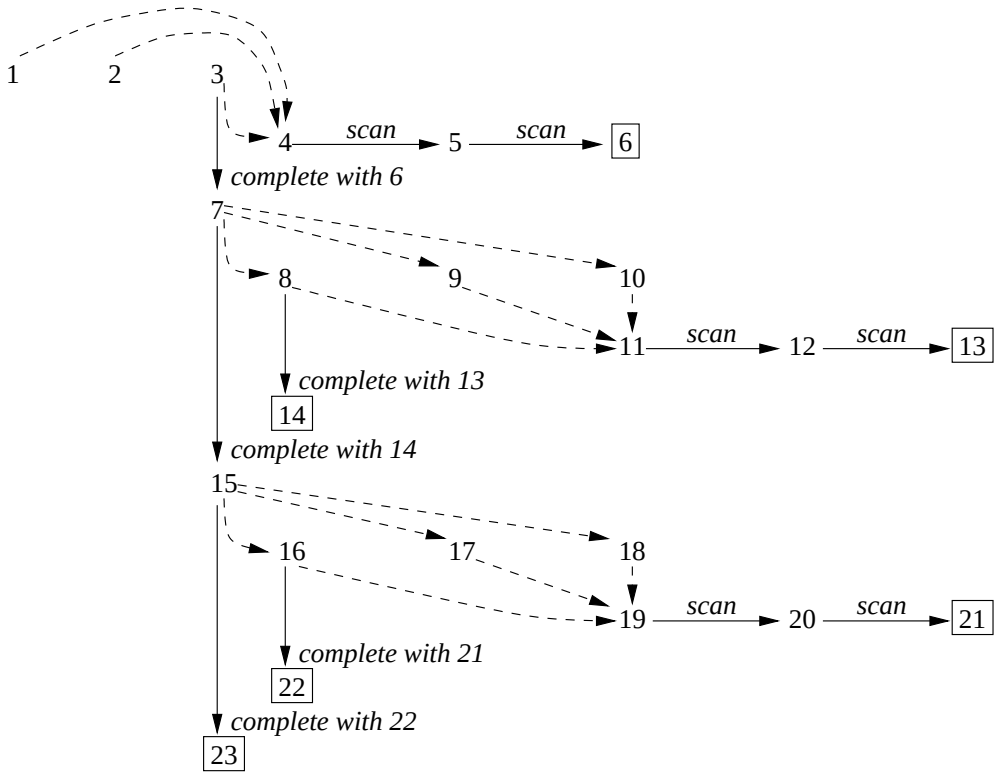


Figure 4.1: The parse graph for the parse table and input in Fig. 3.6.

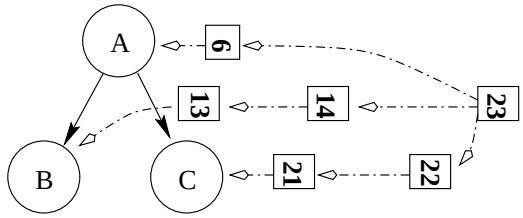


Figure 4.2: The parse tree extracted from the parse graph in Fig. 4.1.

Parse Tree

Figure 4.2 depicts a parse tree extracted from the parse graph. The parse tree represents the phrase structure imposed on the input by the parser, or, the immediate dominance relationships between the nonterminal instances recognized during parsing. We pick as the root of the parse tree state 23 which is the *Tree* covering the whole input. We can determine the constituents of 23 by following the parse graph backwards from 23 along the path of successor links to state 3 where the *Tree* was predicted. During the traversal of the path, we collect all the input objects and inactive states used in advancing the parse by either the scan or the complete operations. So, we conclude that the *Tree* represented by 23 is composed of (6,14,22) which is depicted in Figure 4.2 by arrows starting from 23 and leading to those objects. In similar fashion, we can build the parse tree for the whole input.

4.4 Additional Remarks

Complexity of Parsing

The disambiguation constraints, the support for iterative right-hand symbols, and the predictive lookahead method with ordering expressions make parsing more deterministic than with the original parser. The determinism can also reduce the number of steps taken by the parser.

For example, with the new features, the parser will produce the parse shown in Figure 3.6 (page 49) for the input in that figure. So, in this case, the parser will perform only the minimum number of steps that are necessary to obtain a correct parse of the input.

If the new features are not used in a grammar and if the input contains ambiguous relations, the parser still has the high worst-case complexity analysed in Section 3.3. However, the new features ensure efficient parsing of context-free productions with support for context-dependent syntactic structures (via remote references).

The Expressive Power of Extended ARGs

In general, the changes we have introduced to atomic relational grammars aim at making parsing deterministic and they discourage the use of ambiguous syntactic structures. This is reflected clearly in the introduction of the disambiguation constraints and ordering expressions. Furthermore, Restriction 4.5 reflects the strong context-free nature of EARG grammars when compared to ARGs.

From the grammatical point of view, iterative and optional right-hand side symbols and remote references increase the expressive power of the formalism. They give a grammar writer strong tools to express complex syntactic relations in a concise manner. They are also important because they make it possible to write grammars where the relational constraints reflect the 'natural' topological structure of many visual languages.

In terms of the languages generated by EARG grammars, the usage of disambiguation constraints has one clear limitation: cyclic relations cannot be used as the relation that drives parsing. Instead, some other driving relation must be used that yields a cycle-free order of parsing. The cyclic dependencies can be expressed through remote references, but the cyclic structure of the dependencies cannot be enforced on the level of syntax. It is therefore likely that the languages defined by EARG grammars form a proper subset of the languages defined by ARGs. However, we have not verified that formally. On the other hand, this has not been a problem with the visual languages that we have implemented with EARG grammars.

Any-Start

The any-start property makes it difficult to perform error diagnosis and to construct an unambiguous parse tree (see Section 3.2.2). Therefore, we give the following restriction:

Restriction 4.6 We limit the parser to begin parsing a start nonterminal instance of an EARG grammar only from an input object (terminal symbol) that will be eventually bound to one of the expander attributes of an inactive state representing the recognized start nonterminal instance.

This restriction affects only the selection of the starting point in the input: parsing must be started from an input object that will be bound to one of the expander attributes of an inactive state representing an instance of the start nonterminal of the grammar (if the input is correct). After the starting point has been established, the scanning order of input objects is determined by the relations.

Consequently, the grammar of the visual language has to be written so that it is possible to write an auxiliary function that can decide from which input object (terminal) to start parsing. For the binary tree language in Example 4.2, this function would just search an input object in the *connected* relation that has no other

object connected to it (the least object in the relation). The flowchart language in Example 2.1 (page 21) has a distinct terminal type for the start and stop symbols in the language, which makes it simple to find the starting point for parsing. For general graphs, the problem of finding the starting point can be solved by introducing to the grammar a container terminal that holds the nodes of the graph (see Example 3.4 on page 41). This container terminal represents the whole visual program (diagram), then.

For parsing, Restriction 4.6 implies that we do not have to use the set *init-states* in the initialization of parsing nor in subsequent completions. Instead, we initialize parsing by putting on the agenda all the predictive states for the start nonterminal such that the first right-hand-side symbol binds (gives a value) to some syntactic expander attribute. Consequently, we only need ordering variants of each production such that every right-hand symbol that binds an expander attribute appears first. Thus, all the 'clutter' caused by the non-determinism of the original parsing method is removed from the parse trace. As the following Lemma shows, this does not affect the correctness of parsing if parsing begins from an input object that meets Restriction 4.6.

Lemma 4.1 *An inactive state s_k representing an instance of nonterminal A and having the expander attribute values $\alpha_0 = x_0, \ldots, \alpha_n = x_n$ will always have a corresponding prediction in parse table T.*

Proof: Let us assume that the parser has created the inactive state s_k that represents an instance of A. Then, there must be a predictive active state s_j for a nonterminal instance of A in some slot $T[i]$ such that the parse initiated from s_j has lead to the creation of s_k. By the definition of the predict operation [Wit96], there must be an active state s_i in $T[i]$ such that *Predicts*(s_i, s_j), i.e. $s_i : X \rightarrow \cdots . A \cdots$ (the only exception are the predictive states for the start nonterminal inserted first into the table). Because s_j is in slot i and because of Restriction 4.6, the input object denoted by i gives a binding to some expander attribute of the nonterminal instance s_k. Hence, there will be an attribute α_l such that $\alpha_l = i$ in s_k. Thus, state s_k will be inserted also into the slot $T[i]$ and parsing may be continued by completing s_i with s_k fulfilling the prediction for an instance of A. If s_j was one of the first predictive states created in the initialization phase, s_k does not need to have a prediction. ■

Chapter 5

Error Handling in Parsing Relational Languages

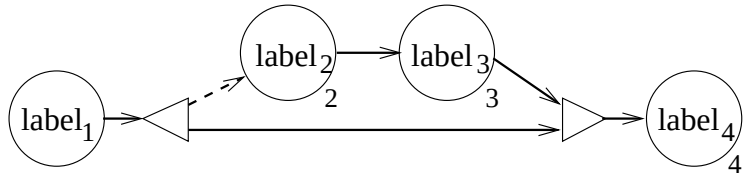
In this chapter, we describe our error handling strategy for the parsing algorithm for extended atomic relational grammars (EARG). With error handling, we mean the detection of syntax errors, reporting them, and the ability of the parser to recover from syntax errors to continue parsing the rest of the input.

First, in Section 5.1, we present a definition of parser-defined syntax errors for the parser by analyzing the possible parsing action failures and by showing how an explicit parse trace can be constructed to locate the errors. Next, in Section 5.2, we explain how different parsing failures occur and how they are detected. Then, in Section 5.3, we present two error recovery techniques. In Section 5.4, we describe how the error recovery techniques are integrated to the parser and, in Section 5.5, we present the EARG parsing algorithm. Finally, we end the chapter by discussing the effectiveness of our error recovery techniques in Section 5.6.

The error handling techniques were originally developed for atomic relational grammars [Tuo98a, Tuo00] without all the new features introduced in Chapter 4. However, the technical challenges in creating an effective error handling strategy were also driving the development of extended ARGs. Therefore, the error handling techniques described in this chapter have been integrated into the implementation of EARGs in the VILPERT framework with only small changes.

An Example Language

In the following, we use examples based on Grammar 5.1 that defines a language of lists with a branching structure. The grammar has the terminals {*circle, text, switch, junction*}, the nonterminals {*List, Node*}, the relations {*next, inside, branch*}, and the expander attributes {*in, out*}. Figure 5.1 shows a sentence of this language. Solid arrows represent the relation *next*, dashed arrows represent the relation *branch*, and the relation *inside* is represented by spatial enclosure.



next = {(circle$_1$,switch),(circle$_2$,circle$_3$),(circle$_3$,junction)
(switch,junction),(junction,circle$_4$)}
inside = {(label$_1$,circle$_1$),(label$_2$,circle$_2$),(label$_3$,circle$_3$),
(label$_4$,circle$_4$)}
branch = {(switch,circle$_2$)}

Figure 5.1: A *List* in graphical form and the corresponding relations

Grammar 5.1

List$_1$ → Node List$_2$ (1)
next(Node.out,List$_2$.in)
List$_1$.in = Node.in
List$_1$.out = List$_2$.out

List → Node (2)
not exists $x \in \{switch, circle\}$: next(Node.out,x)
List.in = Node.in
List.out = Node.out

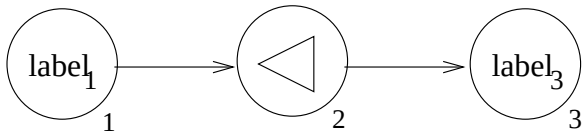
Node → circle text (3)
inside(text,circle)
Node.in = circle
Node.out = circle

Node → switch List junction (4)
branch(switch,List.in)
next(switch,junction)
next(List.out,junction)
Node.in = switch
Node.out = junction

Note that this is a toy visual language that serves the purpose of illustrating the error handling techniques.

5.1 Defining Syntax Errors

The syntax errors in relational languages are anomalies in the object-relation graph constituting the input. As with string languages [SSS90, Chap. 9], we could



○1	[List -> . Node List] [List -> . Node] [Node -> . circle text] [Node -> . switch List junction] #(Node)	1. 2. 3. 4. 6.
label1	[Node -> circle . text]	5.
○2	[List -> Node . List] [List -> . Node List] [List -> . Node] [Node -> . circle text] [Node -> . switch List junction]	7. 8. 9. 10. 11.
◁	[Node -> circle . text]	12.
○3	-	
label3	-	

Figure 5.2: The parse table for the invalid list shown on the top.

give a definition of *actual* syntax errors based on the shortest editing distance between indexed md-sets by defining that what was really meant with an erroneous sentence are the nearest (correct) sentences of the language. The editing operations needed to correct an incorrect sentence would then determine what was wrong in it.

The definition of actual syntax errors is not very practical. Instead, we must be contented with reporting *parser-defined* syntax errors that reflect how the parser might fail with certain inputs. With Wittenburg's parser, the definition of parser-defined errors involves analyzing the conditions that cause parsing operations to fail. Error situations can then be described in terms of the failed conditions and the input involved.

However, parsing operations may also fail in the normal course of action because the parser runs the alternative parses for a predicted nonterminal in parallel and independent of each other. If the grammar and the input are unambiguous, only one of the parallel parses will succeed and the others will fail. For instance, in Grammar 5.1 there are two alternative productions for *Node* but they cannot both match against the same piece of input.

A *global parsing failure* means that (1) all the parallel parses initiated by the first

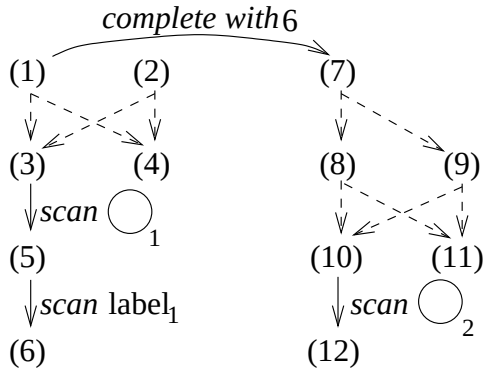


Figure 5.3: A parse graph.

predictive states for the start nonterminal failed, or that (2) at least one of the parses succeeded but there is unprocessed input left. In case (1), the actual causes of failure are found by examining the reasons why each parallel parse failed. This means that parse states must be linked to form an explicit parse graph that can be traversed during error diagnosis. A successful parse path for a nonterminal leads from a predictive state to an inactive state whereas a failed path is terminated by (one or more) errors. Furthermore, it is natural to use the number of input objects scanned along a parse path as a measure of the *relative success* of the path.

Definition 5.1 *In the case of global failure (1), the parser-defined error is the set of input objects causing the parse action failures at the end of the most successful parse paths starting from the first predictive states. In the case of failure (2), the parser-defined error is the set of extra input objects.*

Note that there can be several equally successful parse paths. Also, *ambiguities* detected by the parser are usually considered errors. For instance, every sentence of UML should have an unambiguous interpretation [BRJ99, p. 15].

Example

An invalid *List* and the corresponding parse table are shown in Figure 5.2. The leftmost column of the table shows the index symbol for a particular slot, the states in the slot are in the center column, and the rightmost column numbers the states in the order they were inserted. Active states are depicted with dotted productions in brackets. The dot shows how much of the right-hand-side of a production has been parsed so far. Inactive states are denoted by #(*nonterminal*). In Figure 5.3, the parse states are explicitly linked to form a directed parse graph with two roots: states 1 and 2 (see Section 4.3.3 about constructing a parse graph).

The parse was started by inserting into the parse table the predictive states 1 and 2 at circle_1. Then, states 3 and 4 were created by prediction from 1 and 2. In 4, the scanning of a *switch* against circle_1 failed but in 3 the scanning of a *circle* succeeds. Then, because there is an object *inside* circle_1, the parse according to the production of state 3 was advanced. In 5, which is the successor of state 3, a *text* was scanned leading to the recognition of a *Node* (state 6). The path $(3, 5, 6)$ represents thus a successful parse of a *Node* instance.

Next, states 1 and 2 were completed with state 6. No successor was created for 2 because the predicate in production (2) of the grammar prevents it (there is a circle *next* to circle_1). However, state 7 was created as a successor for 1. This initiated a parse for a new list at circle_2 (states 8 and 9). Now, both of the parses for a *Node* at circle_2 (started from 10 and 11) failed. In 11, circle_2 is seen when a *switch* was expected. In 10, circle_2 was scanned leading to state 12 but there, a *switch*-triangle is seen when a *text* was expected. Now, there is no state in the parse table where parsing could be continued (the parsing agenda is empty). Because the most successful parse path led to state 12, we report as the parser-defined error the *switch* ($\triangleleft$) object that caused the scanning failure at state 12. The complete paths that start from a root and lead to the error are $(1, 7, 8, 10, 12)$ and $(1, 7, 9, 10, 12)$.

The example above is for atomic relational grammars. When parsing extended atomic relational grammars, as described in Section 4.3, the only difference to the example above is that the parser would not create state 12 at all. The reason is that when querying for the next input object after scanning circle_2 in state 10, the *filter* procedure (Procedure 4.6 on page 73) would return an empty set because the terminal type of the object ($\triangleleft$) inside circle_2 is wrong ($term(\triangleleft) \notin terminals(text)$). Therefore, in this case, the paths leading to the error would be $(1, 7, 8, 10)$ and $(1, 7, 9, 10)$. However, as will be explained below, the parser will attach to state 10 an error descriptor that identifies the direct cause of the failure and the input objects involved.

5.2 Parsing Failures

In the following, we describe how different parsing failures occur and how they are detected. We make a distinction between failures that occur *during* parsing (parse action failures) and failures that can be detected only *after* parsing (ambiguities and extra input). The idea is to represent failures as *error descriptor* objects that can be associated directly with the states in the parse graph or kept in global lists depending on their type.

Parse Action Failures

The actions of Wittenburg's parser (Algrorithm 2.1, p. 33) and the EARG parser (Algrorithm 5.1, p. 98) consist of the *scan*, *predict*, *complete*, and *inverse-complete*

operations. Scan, complete, and inverse-complete call the Advance procedure (Procedure 2.1, p. 32, for ARGs and Procedure 4.1 , p. 69, for EARGs) to advance a parse for a nonterminal. Errors cannot occur during prediction because no input is involved. In the following we list the causes of failure for the other actions. In the listing, we declare the version of atomic relational grammars to which the causes apply.

Scan

1. (ARG only) The lexical classes of the symbol at the dot and the current input object are different (Figure 5.4 a) (this was the reason for parsing failure in the previous example).
2. There are missing relation tuples, which can be detected in two ways:
   - (ARG and EARG) there are unsatisfied relational constraints between the symbol at the dot (current input object) and the recognized right-hand-side elements to the left of the dot (Figure 5.4 b), or,
   - (EARG only) there are unsatisfied remote references for the symbol at the dot.
3. (ARG and EARG) There are failed predicates for the symbol (current input object) at the dot and the recognized right-hand-side elements to the left of the dot.
4. (ARG only) The current input object is already part of the cover of the active state.

Complete/Inverse-Complete

1. There are unsatisfied relational constraints (ARG and EARG) or remote references (EARG only) like in *scan*.
2. (ARG and EARG) There are failed predicates.
3. (ARG and EARG) The covers of two states overlap, i.e. the intersection of the covers is not empty (Figure 5.4 c).

Advance

The procedure is responsible for querying the input for the next symbol to be scanned, based on the constraints at the dot position of a newly created active state. The error that can occur is manifested by

1. (ARG and EARG) missing relation tuples (Figure 5.4 d), i.e. an empty query result.
2. (EARG only) There are more than the exptected number of objects in the result set of the query (see the discussion on page 64).

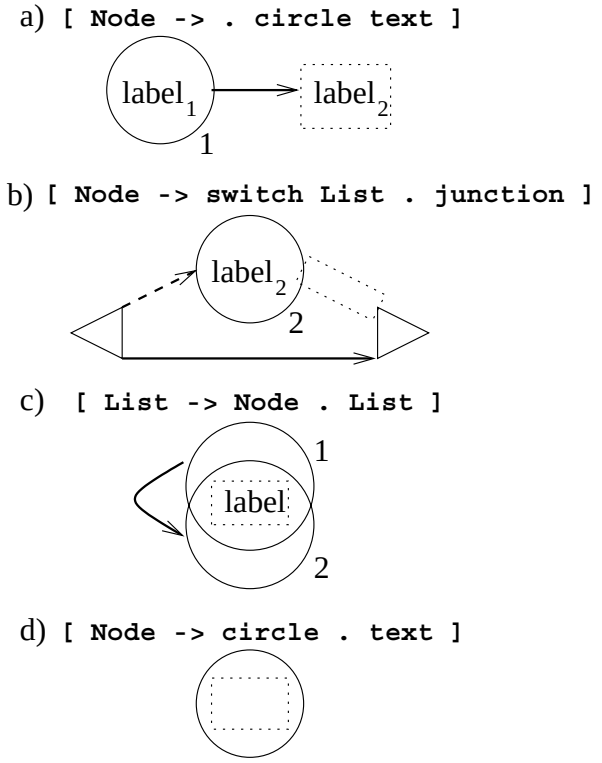


Figure 5.4: Examples of syntax errors detected by the parser. The location of the error is shown by a dotted box in the input and the position of the dot in the corresponding parse state.

> Note that a query may return several candidate objects either by design of the (ARG) grammar or due to an error (ambiguity) in the input (EARG).

The list of parse action failures above reflects the principle that we test all the (relational and disambiguation) constraints, predicates, and remote references as soon as it is possible during parsing. Note that failed disambiguation constraints are *not* considered as errors (see the discussion about the difference of predicates and disambiguation constraints on page 60).

The predictive lookahead method introduced for extended ARGs in Section 4.3.2 prevents scanning errors of type 1 and 4 to actually happen. Instead, the parser will detect a *missing relation* error while executing the Advance procedure.

Ambiguities

Like Earley's parser, Wittenburg's parser can process ambiguous grammars. However, in many applications of parsing ambiguities are considered as errors.

Ambiguities can come from two sources. *Grammar-induced* ambiguities depend only on the properties of the grammar whereas *input-induced* ambiguities can arise even with unambiguous grammars. Figure 5.5 shows an example of the latter (with respect to Grammar 5.1): two different *List*s are recognized because of the two distinct objects *next* to circle_1.

Ambiguities can be detected from the parse graph in the following situations:

1. more than one of the alternative parses for a nonterminal instance has succeeded, or
2. there is a state with more than one successor, or
3. there is a state with more than one predecessor.

There is one exception to rule 2: left-recursive productions cause multiple successors for some active states. However, left-recursive structures can be detected to prevent the creation of wrong ambiguity-error descriptors. In our implementation of extended atomic relational grammars, each active state in the parse table (with a nonterminal at the dot) keeps a list of (references to) the inactive states that the *completion* parse action has used to advance the parse represented by that active state. Then, upon a completion, the parser checks that all the inactive states used in previous completions are part of the inactive state (or the parse tree represented by the inactive state, see page 77) used in the current completion. This condition holds only for left-recursive syntactic structures. Otherwise, the grammar reports an ambiguity error.

Extra Input Objects and Relation Tuples

The union of the terminal covers of all the inactive states in the parse table forms the set of successfully scanned objects, that is, objects that are part of some nonterminal instance. The set of extra objects is then the set difference of all input objects and the covered input.

Extra relation tuples must be considered as errors when they are represented by *explicit graphical objects* in the visual language, such as relation *next* in our list language. This means considering the tuples of at least some relations as first class objects similar to the terminals. In our implementation of atomic relational grammars [Tuo99], we have a mechanism for marking relations to be treated as graphical objects in the parser.

The verification of constraints when recognizing nonterminal instances provides a way to distinguish between expected and extraneous relation tuples. By tagging the relation tuples that are used to verify constraints, all untagged relations can be declared as extras after parsing.

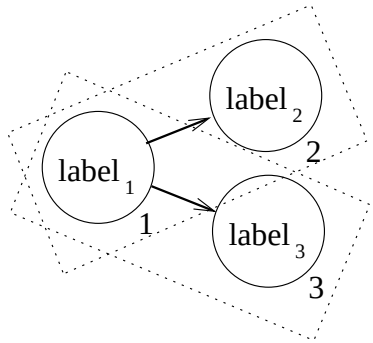


Figure 5.5: An ambiguous *List*.

5.3 Error Recovery

In this section, we describe an error recovery strategy to be embedded in the parser for atomic relational grammars. The strategy aims at enabling the parser to continue processing the input *in spite of syntactic errors* rather than by actually *correcting* the errors.

In the following, we first present two state-level error recovery techniques that can be applied *locally*. Second, we describe a backtracking strategy that employs the two local techniques to perform *global* recovery. Then, we describe the integration of the recovery strategy to the parser and present the resulting parsing algorithm. Finally, we discuss ideas about more effective recovery.

5.3.1 Local Recovery

The following two techniques provide the basic mechanisms for creating new parse items (states) from the states representing the dead-ends on a parse path. That is, the scope of recovery is the parse of the last predicted nonterminal instance on the parse path terminated by a dead-end. Because of the limited scope, we call these techniques local. The techniques are conservative in the sense that there is no heuristic guessing involved.

Attribute Patching

This technique is based on the fact that the expander attributes determine the only possible 'connection points' of nonterminal instances in derivations. Simply, the idea is that an inactive state representing a completely parsed nonterminal B can be safely created from a partial parse of production p, $p = B \rightarrow \alpha\ \beta\ \delta\ \gamma\ \omega\ F$, if the right-hand-side symbols in α that have already been parsed provide bindings

○$_2$	[List -> Node . List] [List -> . Node List] [List -> . Node] [Node -> . circle text] [Node -> . switch List junction] #(Node)	7. 8. 9. 10. 11. 13.
◁	[Node -> circle . text]	12.
○$_3$	[List -> Node . List]	14.
label$_3$	-	

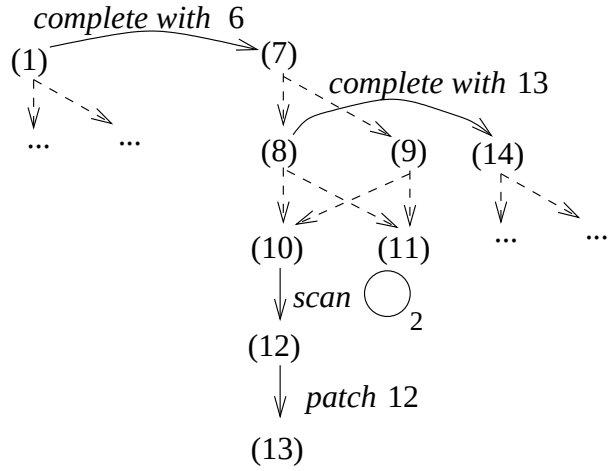


Figure 5.6: Patching state 12.

for all the expander attributes in $A(B)$. That is, all the assignments in F can be made. Then, it is guaranteed that subsequent parsing actions operating on the state won't fail because of undefined expander attribute values of the instance of B. Of course, all the constraints (in β and γ) and predicates (in δ) that can be evaluated in the partially parsed p must be evaluated when creating the instance of B (inactive state).

For instance, consider the input in Figure 5.2 and state 12 in the parsing table. According to Grammar 5.1 (production 3), the *circle* on the right-hand-side of the production provides the value for both of the expander attributes of *Node*, *in* and *out*. So, we can create state 13 (Figure 5.6) representing a *Node* that covers only $circle_2$. In this case, the incorrect input symbol (the triangle) is completely discarded. Then, state 8 would be completed with state 13, move to state 14, and finally lead to successful parsing of the rest of the input.

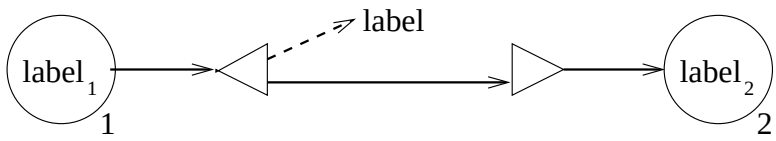


Figure 5.7: Bypassing a fault.

Finding a Detour

Figure 5.7 shows a situation where attribute patching is not possible. Consider the branching *Node*; assuming that the parsing of the node has reached the point $[Node \rightarrow switch\ .\ List\ junction]$, the parse of the *List* fails. Attribute patching cannot be done because the unseen portion of the production (the *junction*) binds one of the expander attributes (*out*) of *Node* (see production 4 in Grammar 5.1). However, when looking at the constraint topology of the production, we see that the *junction* can be reached also from the *switch* bypassing the *List*. Parsing may be resumed by putting on the agenda the active state $[Node \rightarrow switch\ List\ .\ junction]$ with *junction* as the key. However, the unparsed portion of the production (the *List*) and the constraints concerning it must be masked out from constraint verification in the subsequent parsing steps.

In the general case, finding a detour is a little more complicated. For instance, more than one right-hand-side element may have to be bypassed to find a detour. Also, only symbols that do not provide values for the expander attributes of the left-hand-side nonterminal in the production may be bypassed.

5.3.2 Global Recovery

When developing a global recovery strategy based on the local techniques, two questions arise. First, given a parse state representing a dead-end, which local recovery action (attribute patching or detour) should be chosen? Second, the scope of local recovery actions is limited to the parse of the last nonterminal instance on the parse path leading to a dead-end. What can be done if both local actions fail on the dead-end?

Choosing the Local Action

If both actions are applicable to the (active) state associated with an error, we prefer attribute patching over detour finding. The reason for this is that we try to avoid introducing additional ambiguities by recovery actions in situations where there are multiple errors, and with attribute patching it is easier to achieve. Because patching produces inactive states with fixed expander attribute values, we can check that all inactive states created from the parses for the same nonterminal

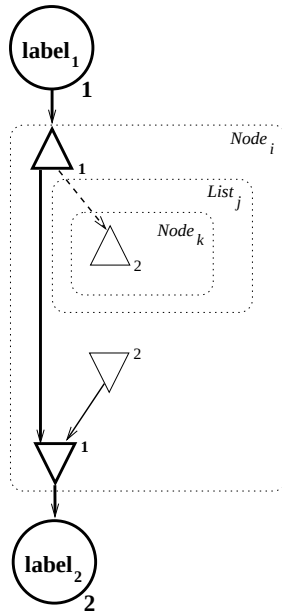


Figure 5.8: Backtracking error recovery over an invalid list.

instance have the same attribute values. Detouring, on the other hand, creates active states and, if the bypassed parts of productions used in the parses provide the only way to distinguish between alternative parses, ambiguities may arise.

Backtracking Recovery

The focus of recovery can be extended by following the parse path *backwards* from the parse of the last predicted nonterminal to the parse of the previous predicted nonterminal. Then, the local recovery actions can be reapplied in a new context with, hopefully, better results. This process may be repeated until a recovery action succeeds or the roots of the graph are reached in which case *error recovery fails*.

For example, with the input in Figure 5.8, neither of the local recovery actions are applicable to the parse of $Node_k$ (see production 4 in Grammar 5.1) starting from the $switch_2$ triangle. Because there are no arrows standing for the relations next and branch starting from the *switch*, the parser cannot determine the next symbol to be scanned and, for the same reason, no detour can be found by just skipping the erroneous part (the *List* of production 4). Expander attributes cannot be patched because only the symbol giving the value for the attribute *in* of $Node_k$ has been scanned so far. Accordingly, no recovery actions can be applied to the parse of $List_j$. However, a detour can be found in the parse of $Node_i$ where the parse of $List_j$ was predicted. Parsing can then be resumed from $Node_i$ and the rest

of the input is successfully parsed; the parser will finally recognize three *Node*s at $circle_1$, $switch_1$ and $junction_1$, and $circle_2$, respectively. The remaining input symbols and relations are ignored.

In some sense, our backtracking technique is analogous to the error recovery mechanism for recursive descent string language parsers presented by Welsh and McKeag [WM80]. The actual mechanisms are of course different but the idea of unwinding the parse stack or path until synchronization between the input and the state of parsing is achieved is the same.

5.3.3 Error Recovery in EARG Parsing

The local and global error recovery techniques described above can be applied almost as such in the parsing of EARG languages. Attribute pathcing is not affected by the extensions of the grammatical formalism in EARGs in any way. However, iterative right-hand side symbols need special handling in the recovery procedure that implements detour finding.

The detour finding procedure for EARGS constructs relational (expander) queries like Advance (Procedure 4.1) and uses the Filter procedure (Procedure 4.6) for predictive lookahead. Optional right-hand side symbols of productions do not provide any additional complexity because detour finding treats, by default, every right-hand side symbol as potentially missing. However, if the error happens in the *middle* of parsing an iterative sequence of symbols, the next input object in the queue of pending input is always a viable starting point for parsing the next instance of the iterative symbol. So, in this case, finding a detour means just removing the next pending input object from the queue and using that as the key of a new parse state to be put on Agenda.

Figure 5.9 shows a snapshot of an implementation of the binary tree language in Example 4.1 (p. 59) that was done with VILPERT. The upper window is the graphical editor and the lower window shows the syntax errors reported by the parser after parsing the binary tree in the editor. We can see that the parser has found three errors in the binary tree:

- a missing *text* symbol inside the left child of the node labelled with ‘A’,
- two *text* symbols instead of one in the node labelled with ‘B’ and ‘C’, and
- three *circles* instead of two connected to the node ‘B C’.

However, because of the last error, the parser has not been able to analyze the three nodes below the node ‘B C’. Therefore, it has missed the error (missing *text*) in the last node of the three.

Figure 5.10 shows the same input as in Figure 5.9 parsed with the following general tree grammar:

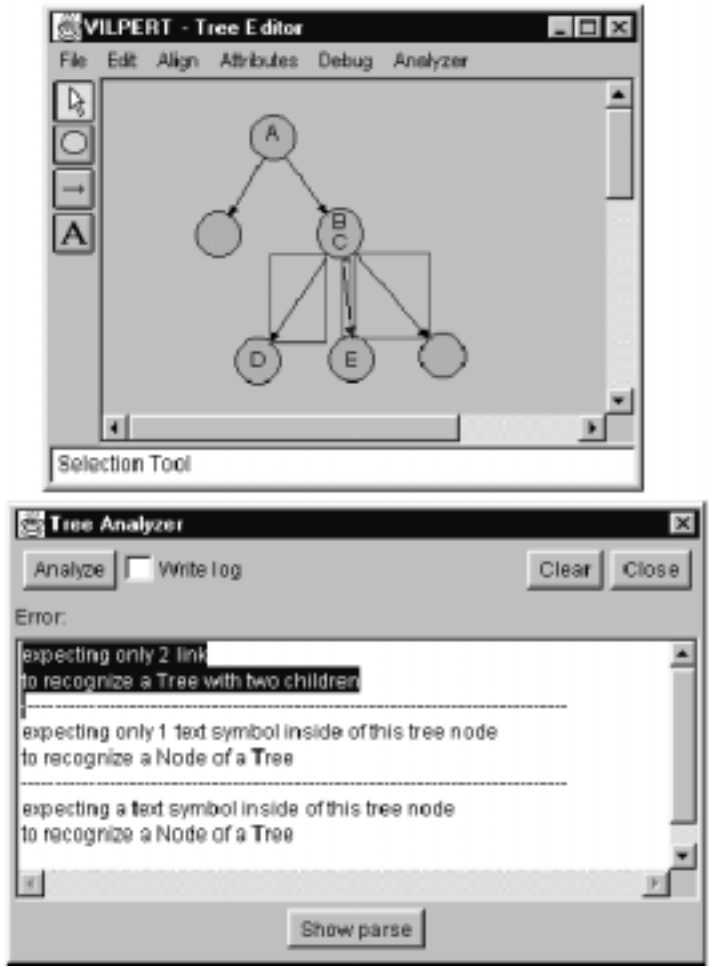


Figure 5.9: An incorrect binary tree (above) and the syntax errors reported by the EARG parser (below).

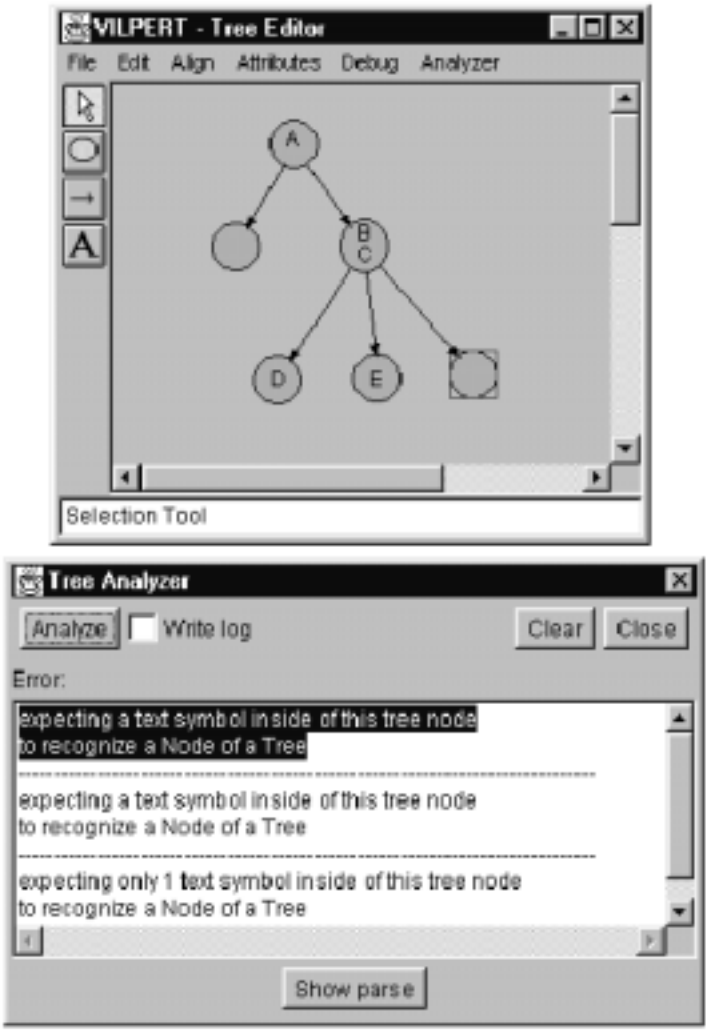


Figure 5.10: An incorrect general tree (above) and the syntax errors reported by the EARG parser (below).

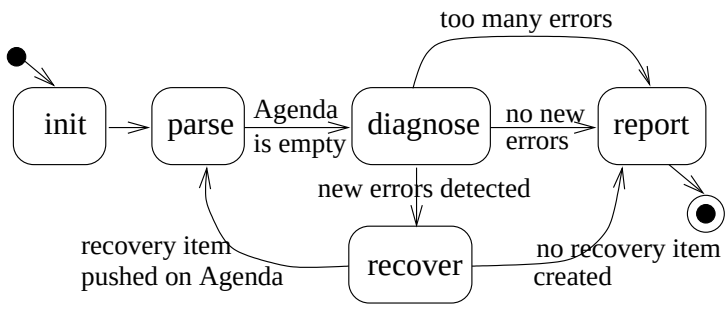


Figure 5.11: Parsing with error handling.

$Tree_1 \rightarrow$ Node $Tree_2$*
    connected(Node.root,$Tree_2$.root)
    order $Tree_2$.root by left
    $Tree_1$.root = Node.root

Node $\rightarrow$ circle text
    inside(text,circle)
    Node.root = circle

In this case, the parser is able to report also the error in the node on lower right. That is, the parser processes all of the input unlike in Figure 5.9.

5.4 Integration to the Parser

The basic parsing cycle of Wittenburg's parser needs only small changes to accommodate our error handling procedures. The main modification is to separate the initialization of parsing from the actual parsing process. Parsing is initialized by putting on agenda the predictive states for each production expanding the start nonterminal, with a given input symbol as the table index key. Then, the actual parsing procedure is invoked and continues until the agenda is empty.

After the parsing procedure *halts*, an *error diagnosis* procedure examines the parse table to determine the result of the parse. If there are errors, a recovery procedure is launched to create new parse states by applying *recovery actions*. If new states are created, they are put on the agenda and the parsing procedure can be restarted. This cycle is continued until no new errors are detected or some predefined error limit is exceeded. Then, all the detected errors are reported. Figure 5.11 shows the modified parse process. In the following we present it in more detail.

Error Diagnosis

The error diagnosis routine analyzes the success of a parse according to Definition 1. In the case of global parse failure (case 1 in Definition 1), the routine traverses the parse graph by going through all paths starting from the roots and collecting all relevant dead-ends. Otherwise, the set of extra objects is computed. Also, the absence of ambiguities in the parse graph must be checked.

If a state has successors, the traversal ignores any possible prediction links and follows the successor links in a depth-first search. There is no need to examine the predicted parses, because the successors indicate the advancement of parsing.

There are four ways for a path to terminate at a state:

1. the state is an inactive state representing a recognized nonterminal, or,
2. an error descriptor is attached to an active state meaning that a parse action failed, or,
3. the path contains a cycle.

Only the second case stands for a dead-end. Note that the same dead-end can appear on many paths like on those leading to state 12 in Figure 5.3. However, cyclic relations cause cycles in the parse path and the graph traversal routine must notice them. Whether or not this is an error depends on the language that is being parsed. Cycles may be caused by circularities in the nonterminal references between the productions of the grammar, as well, but these can be detected statically.

The idea is to collect the error descriptors of all dead-ends and rank them according to the number of input objects scanned along the paths leading to the dead-ends. After the traversal is complete, the first-ranking error descriptors are reported as the parser-detected syntax errors.

Ambiguities are detected during parsing when parse states are inserted to the parse table and the parse graph is constructed. The parser collects the ambiguity descriptors in a global list during parsing. Then, during the post-parse diagnosis, the parser checks whether there are ambiguity descriptors on the list.

5.5 The EARG Parsing Algorithm

We present here the main routine of the modified parsing algorithm for extended atomic relational grammars that implements also the global error recovery strategy. On the surface, the major modification is separating the initialization of parsing from the actual parsing process. This is done to support the integration of the error recovery strategy as explained above.

The other changes are the following:

- Because the any-start property of Algorithm 2.1 is not used, we do not need the special syntactic attribute *start* neither all the possible ordering variants of productions (see Restriction 4.6 and the related discussion on page 78).
- The parse actions *scan*, *complete*, and *inverse-complete* use Procedure 4.1 Advance instead of Wittenburg's Procedure 2.1. Advance uses Procedure 4.6 Filter that implements the predictive lookahead method of EARGs described in Section 4.2.
- Whenever the parse of some production is advanced (by *scan*, *complete*, or *inverse-complete*), all the relational constraints, predicates, remote references, and disambiguation constraints that can be evaluated are evaluated at that point.
- The parser links the inactive states representing recognized nonterminals into a parse tree (see Section 4.3.3). The parse tree is built incrementally during the Advance operation as part of Procedure 4.2 closeParse. The ordering of the subtrees in the internal nodes of the parse tree reflects the order of the right-hand side symbols in the corresponding productions.

Parsing is initialized by putting on agenda the predictive states for each production expanding the start nonterminal, with a given input symbol as the table index key.

In the algorithm, the constant *MAX* defines an upper limit for the number of errors to be reported and, thus, for the number of times that error recovery is to be attempted.

Algorithm 5.1 *Parsing with error recovery*

Input: An extended atomic relational grammar G, a set $C = (I, R_1, \ldots, R_n)$ to be parsed, and an object $o \in I$ from which to start parsing.

Output: A set of error descriptors (errors) representing the syntax errors found by the parser. An empty set indicates a successful parse.

Auxiliary data structures:

Agenda: A FIFO list of states to process, initally empty.

Init-states: The set of predictive states for the start nonterminal $S \in G$ obtained by generating only the ordering variants of the productions $p_S = \{p | p \in G \wedge p : S \rightarrow \ldots\}$ such that for all $p \in p_S$, every right-hand-side symbol of p that binds an expander attribute $\alpha \in A(S)$ appears first in some ordering of the right-hand-side symbols of p.

Parse table: A hash table T of state sets T_j where $j \in I$.

Algorithm:

1. **for each** $s \in$ Init-states **do**
2. add an item $[s, \{o\}]$ to Agenda
3. **end do**
4. $errors := \emptyset$
5. $ambiguities := \emptyset$
6. $success :=$ **false**
7. **while** Agenda is not empty **do**
8. remove an item $[state, keys]$ from Agenda
9. **For each** $k \in keys$ **do**
10. **if** T_k does not contain *state* **then**
11. Add *state* to T_k, perform the applicable parsing actions (*scan, complete, inverse-complete* and Procedure 4.1 Advance, or *predict*) on it like in Algorithm 2.1, and put the resulting $[state', keys']$ items on Agenda.
12. **end if**
13. **end do**
14. **end do**
15. **if** T contains an inactive state t that is an instance of S and t covers the whole input **and** *ambiguities* $= \emptyset$ **then**
16. $success :=$ **true**
17. **else**
18. Perform the error diagnosis routine by traversing the parse graph and collecting the highest ranking dead-ends. Add the error descriptors associated with the dead-ends to *errors*. If no dead-ends are found, check for ambiguities and extra input and add the correspondig descriptors to *errors*.
19. **for each** new error $e \in errors$ **do**
20. **if** e is not an ambiguity error or an extra input error **then**
21. Apply the recovery strategy to the parse state associated with e and add all resulting recovery items to Agenda.
22. **end if**
23. **end do**
24. **end if**
25. **if** *success* = **false and** $|errors| \leq$ *MAX* **and** Agenda is not empty **then**
26. **goto** 6
27. **end if**
28. **return** *errors*

For a discussion about the theoretical complexity of parsing in terms of the O notation, see Section 4.4.

5.6 Discussion

Patching and detouring do not actually perform error correction. They both just ignore some erroneous part of input like the panic mode error recovery techniques that are common in string language parsers [ASU86, p. 164]. For instance, in Figure 5.2 (p. 83) the *switch* is not interpreted as a *text* but simply discarded. In general, detouring attempts to minimize the number of discarded input objects at the cost of introducing ambiguities.

However, both patching and detouring depend on the connectedness of the input: missing relations can cause both methods to fail. For instance, with the input in Figure 5.12, only one of the four nodes would be parsed according to Grammar 5.1 no matter from where the parsing starts, and the three other nodes would be considered as extra.

To increase the effectiveness of recovery in case of missing relations, the following scheme (adapted from the strategy suggested in [SSS90, Chap. 9] for LL(1)-parsers) could be quite easily implemented to deal with extra input. The idea is that the parser initiates new parses from some unprocessed input symbol that could be bound (based on their terminal/lexical type) to an expander attribute of some nonterminal instance in the parse table and repeats the process until no more input is consumed. Nonterminals could be ranked by their grammatical distance from the start symbol of the grammar, which would have the highest rank. The amount of input to be parsed could then be maximized by selecting always the highest ranking nonterminal that could possibly be parsed. The selection of the input symbol could be simply based on the lexical type of the object. However, a more sophisticated scheme that takes into account even the immediate neighborhood of the candidate object in the input graph could yield still better results.

In order to deal with extra relations like in Figure 5.9 (p. 94), the grammar of the visual language can be extended with productions that actually allow erroneous syntactic structures. This grammatical trick is well-known in string language parsing [ASU86, p. 165]. For example, the binary tree grammar in Example 4.1 (p. 59) could be extended with the following two productions that are recognized when there are three or four subtrees, respectively, for a *Node*:

$\text{Tree}_1 \rightarrow$ Node Tree_2 Tree_3 Tree_4
    connected(Node.root,Tree_2.root)
    connected(Node.root,Tree_3.root)
    connected(Node.root,Tree_4.root)
    order Tree_2.root,Tree_3.root,Tree_4.root by left
    Tree_1.root = Node.root

$\text{Tree}_1 \rightarrow$ Node Tree_2 Tree_3 Tree_4 Tree_5
    connected(Node.root,Tree_2.root)
    connected(Node.root,Tree_3.root)

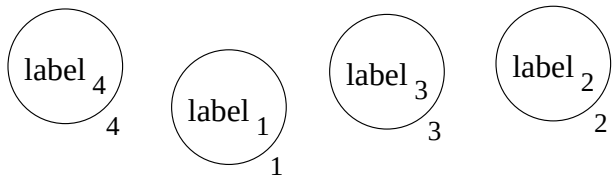


Figure 5.12: Unrecoverable errors.

connected(Node.root,$Tree_4$.root)
connected(Node.root,$Tree_5$.root)
order $Tree_2$.root,$Tree_3$.root,$Tree_4$.root,$Tree_5$.root by left
$Tree_1$.root = Node.root

Adding these productions would make the parser to process the whole input in Figure 5.9. That is, the parser would also notice the error shown in Figure 5.10. The drawback of this trick is that the parser would not report as a syntax error the fact that there are three subtrees for one internal node of a binary tree. This problem could be solved, for instance, by marking the productions (or the recognized nonterminal instances) as error productions and by changing the error recovery strategy to be able to handle also this kind of errors.

Chapter 6

The VILPERT Framework

In this chapter, we present VILPERT, an object-oriented application framework for implementing visual languages. The framework implements the extended atomic relational grammatical formalism presented in Chapter 4 and the error recovery techniques presented in Chapter 5.

First, we present the concept of object-oriented application frameworks in Section 6.1. In Section 6.2, we describe the JHotDraw framework that we have used in VILPERT. Then, we introduce VILPERT in Section 6.3. In Section 6.4, we present an overview of the architecture of the framework and explain through an example how to derive an application from the framework. In Section 6.5, we describe the user interaction of editing and analyzing visual programs with a tool produced by VILPERT. Finally, in Section 6.6, we characterize the visual languages that we have implemented and discuss areas of improvement and future work.

6.1 Object-Oriented Application Frameworks

Object-oriented application frameworks are promoted as a technology that provides a high degree of reusability and extensibility of software assets [FSJ99b]. According to [JF88],

> a framework is a set of classes that embodies an abstract design for solutions to a family of related problems.

A framework captures the commonalities of a set of applications that belong to a certain domain in the form of an implementation skeleton. It embodies the most significant architectural design decisions that the perceived applications in the domain must conform to. The skeleton captures the most stable concepts (structure)

and collaborations (behavior) of the perceived applications as a mixture of abstract and concrete implementation elements (e.g. classes). Usually, the skeleton provides the main control of the application (inversion of control)[1].

In addition to the common properties of applications and the main control (i.e. the event loop), a framework provides extension points for configuring and adding the variable features of the applications. The user of the framework provides the configuration information and concrete implementations for the underspecified or missing parts (i.e. callbacks to user implemented components) in order to derive a working application from the framework.

A framework may cover only a subsystem in the application domain instead of a skeleton of a complete application. Fayad & al. give the following classification of frameworks by their scope [FSJ99a]:

System infrastructure frameworks These include system infrastructure frameworks (operating systems), communication frameworks, and frameworks for user interfaces and language processing tools.

Middleware integration frameworks These frameworks are used to integrate distributed applications or componets. Examples are ORB frameworks, message-oriented middleware, and transactional databases.

Enterprise application frameworks These frameworks address broad application domains (telecom, avionics, manufacturing, financing etc.). They capture extensive domain knowledge and are the cornerstone of enterprise business activities. Therefore, they can provide substantial return on investment. They are also expensive to develop and are usually developed in-house compared to infrastructure and middleware frameworks that are often purchased.

Frameworks can also be classified by the extension technique used to derive applications from a framework:

White-box The framework relies heavily on object-oriented language features (inheritance, dynamic binding) in order to facilitate the extension and reuse of existining functionality. The framework exposes its internal structure in a transparent manner for the application developer.

Black-box The framework supports extensibility by defining interfaces for components that can be plugged into it. The framework is opaque in the sense that the application developer can not see the internals of the framework.

Gray-box This kind of frameworks try to provide a reasonable compromise between the flexibility and complexity of white-box frameworks and the ease of use of black-box frameworks.

[1] Also known as *The Hollywood Principle*: “Don’t call us—we’ll call you.”

Roberts & al. describe the typical life-cycle of an object-oriented framework as gradual evolution from a white-box framework towards a black-box framework by an iterative process where applications are derived from the framework and the framework is extended to support faster derivation and more features [RJ97]. They see a visual (sic.) builder tool as the final state in the evolution of a framework. The visual tool addresses one specific task: the configuration of an application derived from a black-box framework by instantiating and connecting the components that make up the application.

There has been strong belief in the potential of object-oriented frameworks in terms of reuse. Intuitively it is clear that implementing the common parts of applications only once and the variable parts per each application should lead to savings in development effort for a family of related products. Frameworks do not facilitate only code reuse—they facilitate also the reuse of the domain knowledge (analysis) and the design incorporated into the architecture and the provisional components of the framework.

On the other hand, developing a framework is more expensive than developing a single application: it is difficult to estimate the needs of future applications and to find the right abstractions to support the expected variability [Mat00]. Hard facts about the economical benefits of using the framework technology have not been published until recently. The study by Mattsson [Mat99] shows clear economic gain from using frameworks, however.

6.2 HotDraw and JHotDraw

There exists a few object-oriented frameworks for the implementation of graphical editors [Jin90, VL90, Bra95]. One of them is *HotDraw*, which dates back to late 1980's. Originally developed by Kent Beck and Ward Cunningham in the Smalltalk language, the framework has been further developed as a Smalltalk framework [Joh92, Bra95] and as two different Java implementations by Erich Gamma (JHotDraw [GE96]) and Ken Auer (Drawlets [Rol00]). Currently, JHotDraw is being developed as an open source project [jho00].

With JHotDraw, users can implement simple graphical editors for 'node-and-arrow' kind of visual languages. Figure 6.1 shows the main concepts of JHotDraw. A JHotDraw document is a *Drawing* composed of *Figures* which can themselves be composite. A drawing is composed in a *DrawingView* through *Tools* that manipulate figures through *Handles* owned by the figures. Specialized interactions can be realized by implementing special tools and handles. *DrawWindow* is the base class of the editor application and it is derived from the Frame class of the host GUI framework (Java AWT in this case). The JHotDraw features include:

– Animated manipulation of figures. That is, when moving or resizing a figure, the changes are immediately reflected in the drawing.

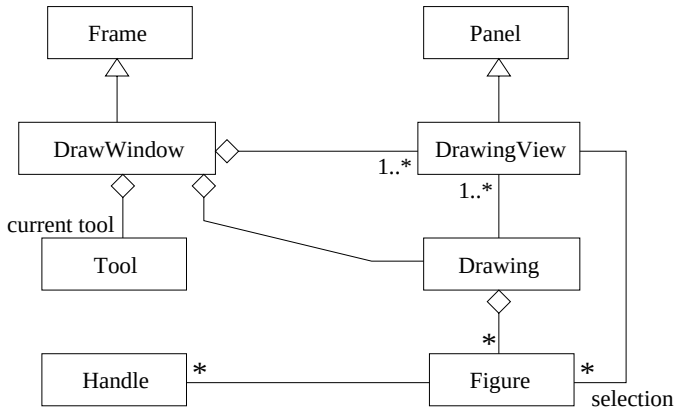


Figure 6.1: The main framework classes of JHotDraw.

- Connecting figures with lines. Figures provide locator objects that decide where and how the connection line intersects the boundary of the figure. The connections are maintained during manipulation (i.e. moving) of the connected figures. Connections and figures can have text objects attached to them.
- Grouping/ungrouping of figures into flat (one-level) composite objects.
- Multiple views on the same drawing.

JHotDraw provides a base editor class for both standalone applications and web applets. From the technical point of view, the distinctive property is the extensive use of design patterns [GHJV95] in the design and implementation of the framework (see the documentation in the JHotDraw package [GE96]).

The derivation of an application from JHotDraw entails writing the classes for any particular figures not provided by the framework (only basic shapes included). Special tools may also be needed (the basic selection and creation tools are usually enough). The semantics of the drawings are implemented by adding semantic attibutes and methods to figures, and the dialogs for accessing them (if needed). Finally, the application is configured by specifying the tools and figures provided the editor, and by defining the additional menus (and corresponding actions) needed by the applications.

According to the classification above, JHotDraw is a white-box system infrastructure framework. Despite the lack of some common features (no undo and no zooming) and despite the limited features of the underlying Java AWT graphics (e.g. AWT supports only one line style), JHotDraw is a good base framework for the graphical editor part of VILPERT. However, we have made extensions to the framework itself to better support some typical syntactic structures of the visual languages used in software engineering (see Section 6.4.2).

6.3 Introduction to VILPERT

6.3.1 General

VILPERT combines an implementation of the EARG formalism and the JHotDraw framework into an object-oriented framework for implementing visual languages. VILPERT comprises two separate frameworks for specifying the syntax of a visual language and for deriving a graphical editor for the language. The main benefits of the approach are a separation of the concerns of editing and automatically analyzing visual programs, and a rigorous implementation methodology based on a powerful syntactic model which does not compromise the usability of the resulting tools.

From the engineering point of view, the grammar-based approach for specifying the syntax of a visual language and automatically producing (by a compiler-compiler) a language analyzer offers obvious benefits when compared with an *ad-hoc* implementation of syntax checking. Compiler-compilers are established tools in the implementation of textual languages, and hand-coding of parsers is rarely done, except in the case of very simple language processors. However, in the case of visual languages, which are usually special purpose high-level languages, there is a much closer relationship between the language environment (e. g. editor) and the language analyzer than in the case of textual languages. Often, the editor and the language are inseparable. For instance, UML CASE tools support typically syntax directed editing of UML diagrams where the tool checks constantly the validity of the diagrams during editing. State-of-the-art UML tools support also the division of large models into many separate diagrams in different sublanguages and the sharing of models between individual developers in collaborative mode. It is naive to think that such complex language environments could be generated based on a grammatical description of the target language. On the other hand, object-oriented frameworks have been successfully developed and used for implementing graphical editors for diagramming tools. Using these frameworks offers the chance to tap into the state-of-the-art in the implementation of graphical editors. In VILPERT, we aim at combining the benefits of both the framework- and grammar-based approaches in the development of visual languages.

The VILPERT framework provides a clean separation of the concerns of the graphical editing and the interpretation of diagrams both from the architectural and the usability point of view. The user draws the diagram in free order (not dictated by a syntax directed editor) and then invokes the language analyzer to interpret the drawing. The analyzer informs the user about any errors it finds during parsing and semantic processing. This approach to visual language implementation makes it possible to combine the sketching and the checking of diagrams into an explorative design style.

Separating the two concerns of editing and analyzing reduces the software complexity of a tool that implements a visual language because the correctness of a diagram does not have to be constantly enforced during editing. Also, the us-

ability aspects of the editor are not compromised by the need of maintaining a consistent model during editing: the editor can provide all the freedom of graphical editing that users want. Furthermore, because VILPERT is a framework, tools produced with it are open for extensions and modifications. Also, the (white-box) framework-based implementation of the editor means that the internal object structures of the editor, which comprise the visual data (program) to be analyzed, can be made directly accessible to the analyzer part.

6.3.2 Object-Oriented representation of EARGs

In the following, we present the main concepts of our implementation of EARGs. In VILPERT, an EARG grammar is represented as an explicit object structure. The UML class diagram in Figure 6.2 shows a conceptual view of the main classes of the grammar framework and the associations between them.[2] The specialization interface of the framework consists of two extendable (but not fully abstract) classes, *RelationalGrammar* and *SyntaxTreeNode*, and a concrete class, *Production*, which is the main vehicle for specifying the grammar of the target language.

The interface of *RelationalGrammar* consists of methods for defining the elements of the grammar. To build a grammar, the method *buildGrammar* in *RelationalGrammar* calls user-defined methods in concrete subclasses for particular languages to first build the symbol sets and then to build the productions of the language. This is an instance of the *Template Method* design pattern [GHJV95].

The methods for specifying the symbol sets (nonterminals, terminals, relation names, and attributes) return a list of the symbols of the category (see the example below). However, the bulk of the grammar is in specifying the productions. For each production of the grammar, a method specifying the structure of the production must be defined in a concrete grammar class. Then, the grammar object must know which methods to call to actually create the productions. This could be implemented by overriding the *buildProductions* method in *RelationalGrammar* to call each named method in succession. However, in the Java implementation of VILPERT, we use the reflection mechanism of the language to automate the creation of productions. The grammar writer names each method that defines a production with B_x where $B \in N$ is the left-hand side nonterminal and x is an arbitrary string such that all the methods for the same nonterminal have a different name. Furthermore, the signature of the methods is restricted so that each method receives one instance of *Production* as an argument and returns a *Production*. Then, the *buildProductions* method simply checks for each nonterminal whether methods for that nonterminal are defined in the grammar class and calls them.

[2] The diagram has been simplified and some details have been left out. For instance, the sets of nonterminal, terminal, attribute, and relation symbols are also part of the grammar but this is not shown in the diagram. Also, the different types of constraints form a class hierarchy under the *Constraint* class.

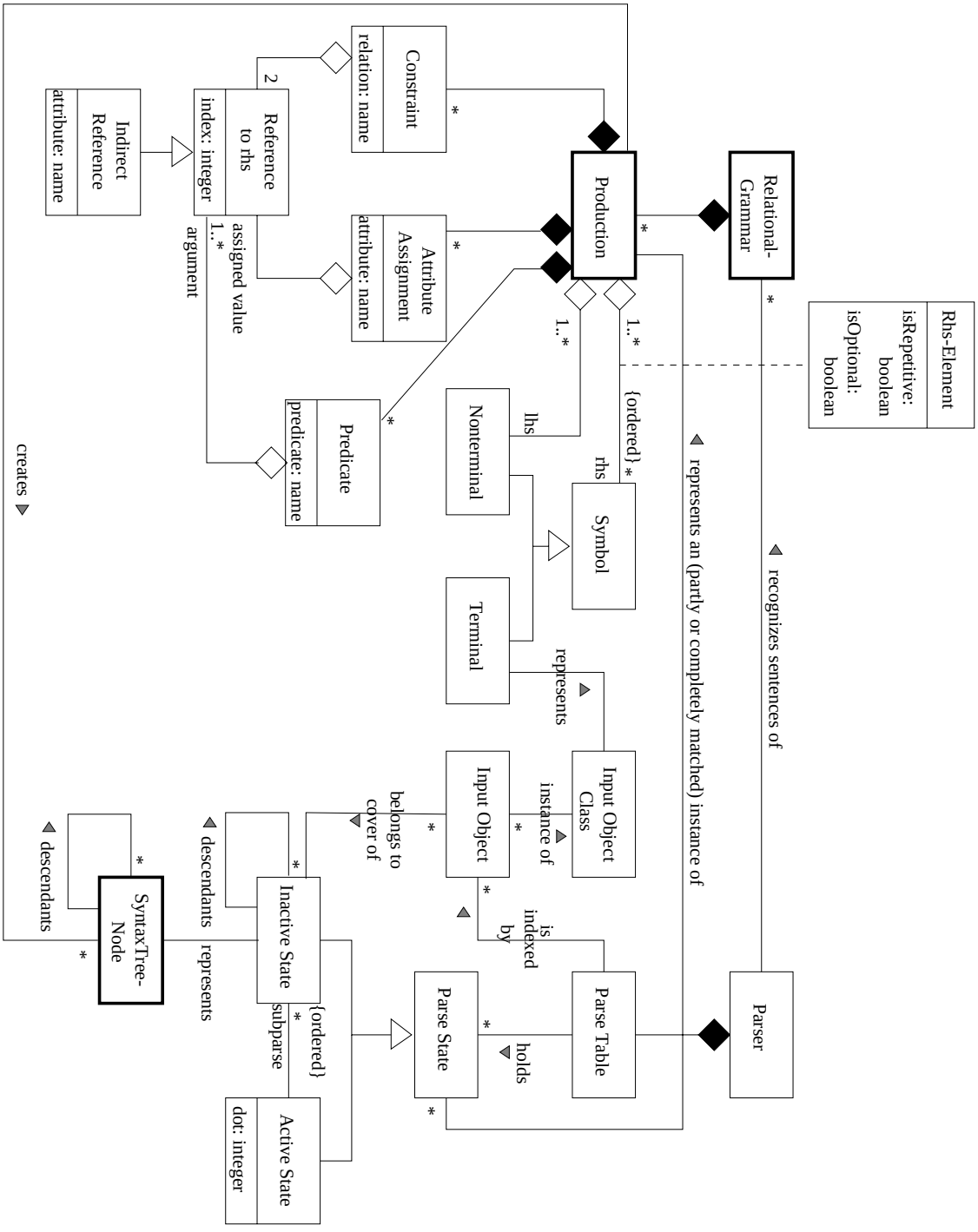


Figure 6.2: The EARG grammar framework.

The interface of *Production* consists of methods for defining the parts of a production: the left-hand side nonterminal, the right-hand side symbols, constraints, predicates, ordering expressions, expander attribute assignments, and the class of parse tree node that represents the production. The parts are defined simply as string arguments to the methods. *Production* objects delegate the parsing of the strings to a *GrammarReader* (class) object that then fills the slots in the productions by creating all necessary parts. The *GrammarReader* checks the lexical and syntactic validity of the strings that define the parts of a production and raises exceptions if the strings are not valid. Furthermore, each grammatical object has a method for checking the semantic validity of themselves. For instance, a production object checks that all the right-hand side symbols have been defined, and a constraint object checks that the references to the right-hand side symbols of its production are correct.

A predicate q of type (a) in δ in Definition 4.1 (page 55) is specified by declaring a method with the name q, the return type *java.lang.Boolean*, and exactly the same number of arguments of type *java.lang.Object* as in all usages of that predicate in the productions. Note that it is possible to have many methods with the same profile except that the number of arguments may vary. In a similar fashion, the ordering function f in an ordering expression $\phi \in \omega$ (see Definition 4.1) is specified by declaring a method with the same name, *java.lang.Boolean* as the return type, and an argument list of two objects of type *java.lang.Object*. These methods are declared in an analyzer class associated with a grammar and not in the grammar class itself (see the example in Section 6.4.3).

All the grammar building operations involve checking to verify the validity of the declarations. That is, when building a grammar, the framework executes the actions of a typical metacompiler and raises exceptions if it encounters invalid constructs.

To define the (operational) semantics of the language, the grammar writer can subclass *SyntaxTreeNode* and define additional (semantic) methods and attributes. The *semantics* method of *Production* is then used to set the actual class of *SyntaxTreeNode* to be created when complete productions have been parsed (see Section 4.3.3). Then, these methods can be invoked on the parse tree after parsing.

The interface of *Parser* is simple. The parser receives as parameters of the *analyze* method the grammar object and an object representing the input object-relation network. If the parse is successful, the *SyntaxTreeNode* representing the root of the resulting parse tree is returned.

An Example The following grammar fragment is part of the specification of the UML statechart language. Figure 6.7 shows an example of this language. Appendix A contains the full Java code for this grammar.

In VILPERT, a grammar is defined by deriving a concrete class from *RelationalGrammarImplementation* (derived from *RelationaGrammar*) that is the base class for all grammars. The Java code in Grammar 6.1 below defines first the symbols

of the language (terminals, nonterminals, relation names, and attributes) as Java strings. Then, the productions of the grammar are specified by defininining methods that construct the productions when invoked (the fragment shows only the declaration of the first production). The actual construction processs is explained in Section 6.4.3.

Grammar 6.1

```
package CH.ifa.draw.samples.statechart;
import relap.LanguageModel.*;
import java.io.*;
import com.objectspace.jgl.*;

public class StateChart extends RelationalGrammarImplementation {

 public String terminalDeclarations () {
   return  "rrect text arrow initial final statePanel namePanel "+
           "itPanel pseudoPanel labelPanel";
 }
 public String nonTerminalDeclarations () {
   return  "StateChart Initial Final State StateSymbol Trans "+
           "NameCompartment StateCompartment ITCompartment Label";
 }
 public String startSymbolDeclaration () {
   return  "StateChart";
 }
 public String relationDeclarations () {
   return  "inside enters exits attached ";
 }
 public String attributeDeclarations () {
   return  "root ";
 }

 public GrammarProduction StateChart_(GrammarProduction p)
     throws InvalidGrammarException {
   p.description(
     "State machine"
   );
   p.rightHandSide(
     "pseudoPanel Initial State+ Final?"
   );
   p.constraints(
     "inside(2:root,1) inside(3:root,1) inside(4:root,1)"
   );
   p.assignments(
     "0:root = 1"
   );
   p.semantics(
```

```
      "CH.ifa.draw.samples.statechart.StateChartRep"
    );
    return p;
  }
  ...
}
```

The production shown in the grammar above declares a *StateChart* to consist of a *pseudoPanel* (an implicit object representing the whole drawing) containing an *Initial* pseudo state, one or more *State*s, and an optional *Final* pseudo state. The right-hand side symbols are referenced by their position (starting from 1) in the constraint expressions. The constraints declare that the input objects bound to the *root* attribute of the nonterminal instances are *inside* of the *pseudoPanel*. The attribute assignment expression assigns the *pseudoPanel* object as the value of the *root* attribute of the left-hand side nonterminal (index 0). The last expression defines the class of the parse tree node to be created when a nonterminal instance matching this production has been recognized.

Reuse of Grammar Specifications

Because grammars are represented as Java classes, the framework presented above allows for incremental language develompent. That is, when developing a different version of a language, the methods defining parts of the grammar can be overridden in subclasses.

The smallest practical unit of reuse of EARG grammars is a production, however. That is, a new grammar A' derived from an grammar A would typically only add new symbols (terminals, nonterminals, and relations) and productions for existing and new nonterminals. One reason for this is that in the case of EARGs, there are many elements in the grammar productions and many interrelationships and restrictions within and between productions (e.g. disambiguation constraints).

The only reason for overriding a part of a production in a subclass is to change the (operational) semantics of the language by defining a different parse tree class for the production. Consider, for instance, developing different versions of a Flowchart language: we could define the syntactic structure of the language in a superclass and then subclass it to define the semantics of the language. One subclass could translate the flowchart into a textual programming language while another subclass could define the operational semantics for interactive, animated execution of the visual program.

So, in our framework, reuse is confined rather within a language family than between languages of different ancestry. That is, reuse and incremental language development is a planned process rather than *ad hoc* reuse of implementation. As an example, Figure 6.3 shows a family of grammars for trees. The root grammar of the hierarchy is *TreeBase* that specifies the structure of a *Node* of a tree and specifies a *Tree* to consist of one *Node* only (i.e. a leaf with no subtrees).

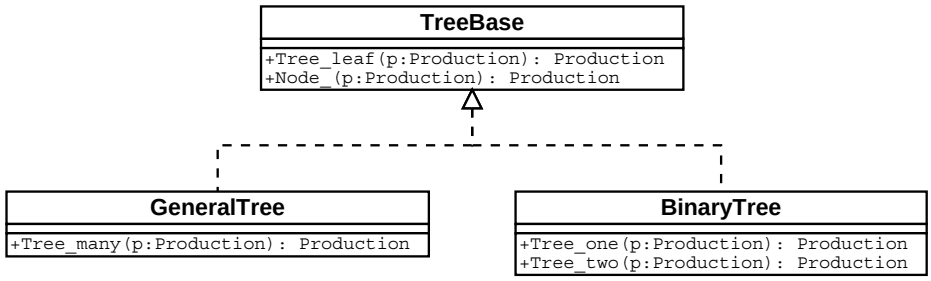


Figure 6.3: A family of tree grammars.

Then, the subclass *GeneralTree* adds a production for internal nodes with one or more subtrees, and class *BinaryTree* adds two productions for internal nodes with one and two subtrees. Further possible variations would be to add new types of nodes, for instance. This would require redefinition of the *terminalDeclarations* and *nonTerminalDeclarations* methods in a subclass. Note, that Java does not allow multiple inheritance, so there is not complete freedom in mixing different base grammars.

6.4 Architecture of VILPERT

As shown in Figure 6.4, the framework consists of two subpackages, *Relap* and *Draw*. The former provides the language specification and analysis framework (grammar and parsing) and the latter the graphical editor framework. In the following, we take a brief look at each of the packages. The example in Sect. 6.4.3 provides a more detailed view of using the framework.

6.4.1 The *Relap* Package

The three main subpackages of the *Relap* (RElational LAnguage Processor) package are shown in Figure 6.4. The *Language Model* package provides an object-oriented model of atomic relational grammars. The model includes abstractions for all the basic concepts of a grammar, such as *RelationalGrammar*, *GrammarProduction*, *Constraint*, *AttributeAssignment*, *Predicate*, etc. The package (subframework) relies on the reflective properties of the Java language in the construction of grammar instances. For the user of the VILPERT framework, the class *RelationalGrammarImplementation* provides the access point to the framework. That is, this is the one and only (abstract) class that needs to be subclassed when deriving a new grammar.

The *Analyzer* package contains the classes needed in parsing. The framework user needs not to be concerned with them.

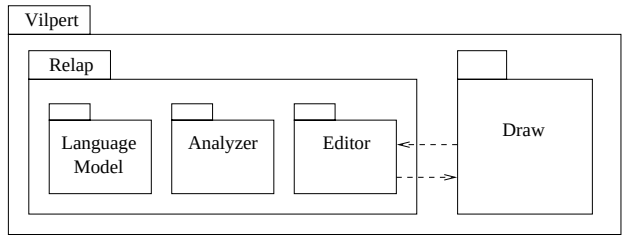


Figure 6.4: The package organization.

However, the *Editor* package provides the mechanisms for glueing the grammar and parser part of a visual language implementation to the graphical editor (derived from the *Draw* framework). This is represented by the dependencies between the packages in Figure 6.4. The package has abstractions for binding the graphical objects manipulated by the editor of a visual language to the terminal symbol instances and relations of the grammar of the language. Other features of the package address the issues of displaying errors and translating the parser defined errors into comprehensible messages. So, the framework user needs rather detailed knowledge of the features and abstractions contained in this package.

6.4.2 The *Draw* Package

This package provides an extended version of the JHotDraw framework introduced in Section 6.2. In addition to connecting figures with lines, semantically meaningful containment is a prevalent feature in visual modeling languages. However, we found that the management of deep figure containment hierarchies was only partially implemented in JHotDraw.

We introduced to the framework the concept of structured graphics with truly hierarchical composite figures and Java AWT-style layout managers. This makes it possible to construct panel-like figures onto which other figures can be dragged and dropped. Each panel class or object is free to implement its own layout policy for arranging the subfigures. Also, panels can be nested to an arbitrary depth just like AWT GUI-components (see also *Composite* design pattern [GHJV95]).

Panels can have a frame figure (*FrameComposite*) that gives them a tangible form or they can be rectangular areas with no visible borders (*FigurePanel*). *FigurePanel*s are used, for instance, to create compartments with specific layout policies within a *FrameComposite*. This is analogous to windows and panels in GUI frameworks. Combined with drag and drop, hierarchical composites provide a powerful and highly usable interaction paradigm. The popularity of GUI building tools is clear testimony of this.

With the features described above, the *Draw* package provides a good base for the implementation of UML-like modeling languages used in software engineer-

ing. The structural abstractions of the package are already close to the syntactic constructs of the intended target languages.

6.4.3 An Example – The UML Statechart Language

Structural View

Figure 6.5 shows an UML class diagram of the implemententation of the statecharter tool shown in Figure 6.7. The diagram presents the classes of the statechart language and their relationships to each other and to the VILPERT framework. The classes belonging to the statechart implementation are shown grayed in the figure. These are the classes that the user has written when specializing the statechart language from the framework. Note that Figure 6.5 shows an abstract view of the static system architecture concentrating on the user view of the system.

The top half of Figure 6.5 shows the classes of the editor. The main component is *StateChartEditor* where the tools, figures, and views of the editor are defined and the application is configured. *StateChartAnalysisView* is a separate window that shows the interface of the analyzer of the language. The view has controls for performing analysis and synthesis (i. e. code generation) and for displaying error messages.

The lower half of Figure 6.5 is concerned with the parsing and the semantic processing of statecharts. The central component is *StateChartAnalyzer* that controls the analysis process. The (concrete) analyzer component defines methods that

- return the name of the grammar class of the visual language (as a string),
- return an instance of the semantic processor of the language,
- return instances of classes that translate the graphical objects into grammatical objects (see below),
- create and initialize the relations that hold (part of) the input to the parser,
- find the start object of parsing, and
- define the predicates and the ordering functions that are part of the EARG grammar of the language.

The analyzer is the glue and mediator object between the editor, parser, semantic processor, and the analysis view. The analyzer defines also methods related to generation of error messages from the errors reported by the parser and the semantic processor. Furthermore, it is possible to define pre- and post-parse actions that take effect immediately before and after a parse is launched.

The classes *StateChartInputGeneratorMap* and *StateChartEditorMap* define translation mappings and actions that convert the *Drawing* and the *Figure*s it holds to

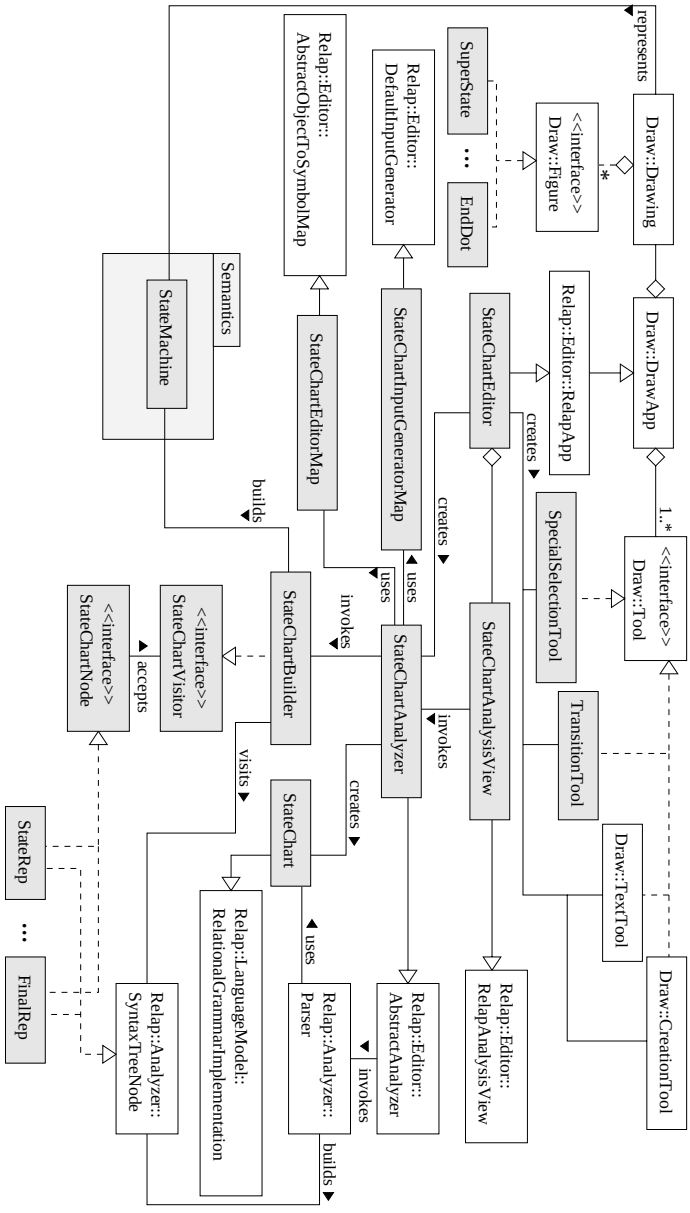


Figure 6.5: The classes of the statechart implementation and their relationships to the framework.

the relational form of input that the parser requires. *StateChart* defines the relational grammar of the language and it is used by the parser.

The parser builds a parse tree from a correct input. The tree is processed by a *StateChartBuilder* that constructs a *StateMachine* from the tree. In the construction process, the mapping from the semantic objects to the graphical objects (the figures in the drawing) is maintained making the representation relationship between the *Drawing* and the *StateMachine* concrete. The contents of the *Semantics* package specifying the structure of *StateMachine*s is not shown. The structure is essentially the same as the semantic model of state machines given in the UML reference documentation.

Behavioral View

Figure 6.6 shows as a UML activity diagram the process of constructing a state machine from a drawing. The drawing is first edited as a collection of figures contained by the *Drawing d*. Then, the analyzer is invoked to process *d*.

The first phase of processing converts the diagram into the set of relations specified in Grammar 6.1. *StateChartEditorMap* defines a mapping from figure classes to strings representing the terminals. *StateChartInputGenerator* is a dynamic visitor that visits the drawing and converts the figure containment hierarchy and the connections between figures to relation tuples and stores them into the IndexedMDSet *i* that is the result of the conversion. Because extended atomic relational grammars support the notion of iterative right-hand side symbols in productions, it is straightforward to transform the containment (parent-child) relationship of the figures into tuples of the *inside* relations of the Statechart grammar (Grammar 6.1). It is also straightforward to generate the tuples of the *attached* relation because the text objects know which connection figures they adorn. Furthermore, the connection figures know the figures that they connect.

The dynamic visitor holds a map (configured by the grammar writer) from the classes (types) of input object to visitor objects. So, the concrete visitor that handles a certain kind of object is determined at run time instead of compile time like in the basic *Visitor* design pattern [GHJV95]. *StateChartInputGenerator* also stores the terminal symbol class of each figure within *i*. The input objects in *i* are references to the figure objects in *d*.

Then, the grammar object *g* is constructed. The grammar class relies on reflection to call the methods in its body that specify the grammar. First, it calls the methods that define the symbol sets and then, for each nonterminal, it calls all the methods that have that nonterminal (plus the underscore character) as a suffix of the method name. The production building methods receive as an argument a skeleton production with empty slots that are filled in the method body as shown in Grammar 6.1.

As explained above in Section 6.3, grammars are classes and, therefore, inheritance and polymorphism can be used to extend and modify existing grammars.

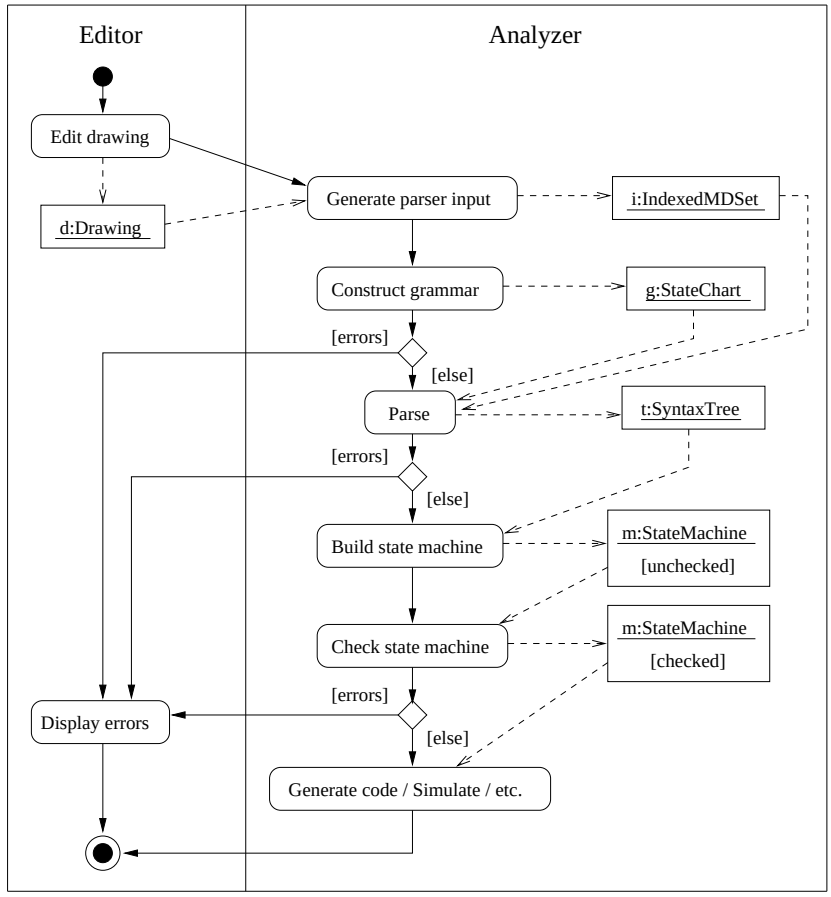


Figure 6.6: An activity diagram showing the construction of a state machine from a statechart drawing.

Also, during the construction, checks are made to ensure that a valid grammar is being constructed. The checks are the same as a metacompiler would do. If there are any errors, they are reported in the analysis view and the analysis process terminates.

The parser receives the grammar *g* and the input *i* and produces the parse tree *t* as the result. Again, if there were any syntax errors, the error messages are displayed and the process terminates.

The last phase of the analysis process is to transform the concrete syntax tree into the abstract semantic representation of a state machine (depicted as the instance *m* in Figure 6.6). That is, the parser hands the parse tree over to *StateChart-Builder* (see Figure 6.5) that visits (*Visitor* design pattern [GHJV95]) the parse tree and transforms it into a state machine (like a *Builder* [GHJV95]). The parse tree nodes (e.g. *StateRep* in Figure 6.5) created by the parser are simple extension objects of generic parse tree node classes (Relap.Analyzer.MiddleNode and Relap.Analyzer.LeafNode) provided by the *Relap* framework. These base classes provide methods for traversing the nodes and accessing the parts of nodes. The extended parse tree nodes of the Statechart implementation specify convenience methods for accessing the parts of nodes and the methods required by *StateChart-Builder* to visit the parse tree.

The main tasks in building the state machine is (1) creating the object structure that is compliant with the OMG metamodel of UML Statecharts and (2) linking the states according to the transitions in the input. The parse tree created by the parser does not contain expicit links between nodes that would represent the topological structure induced by the transitions between states (see the discussion about remote references on p. 42). During the linking phase, *StateChartBuilder* builds a map that associates the created state objects with the graphical input objects representing the figures physically connected by the transition arrows (figures). Then, when visiting a transition, the builder can connect the state objects because a transition figure knows the figures it connects. The builder also assigns the text objects associated with states and transitions as the attributes of the corresponding objects of the state machine.

After the transformation, the state machine then performs a self check ensuring that it is well formed and valid. The transformation routines and the checks are coded by hand.

After the analysis, other actions can be performed on the constructed state machine. These can include code generation or interactive animated simulation, for instance.

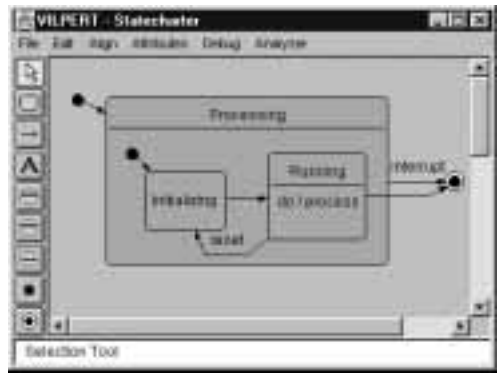


Figure 6.7: A nested statechart.

6.5 User Interaction

6.5.1 General

Figure 6.7 shows an UML statechart editor implemented with VILPERT. The window titled *VILPERT - Statecharter* is the editing view that provides the basic tools for creating and manipulating the graphical objects (terminals) of the statechart language (arbitrary shapes cannot be drawn). As explained above, the editor is a structured drawing tool in the sense that it supports features like persistently connecting figures with lines and managing hierarchies of figures. Also, moving and resizing of figures is animated in real time which provides immediate feedback of the editing actions.

In the statechart editor shown in Figure 6.7, nested substates can be individually selected, moved, and resized and the containing state adjusts its shape accordingly. Moving a superstate moves all its substates. There is no limit on the level of nesting.

Text figures can be edited *in place*, which removes the need for clumsy dialogs for filling in the textual properties of diagram elements. Also, text objects can be dragged and dropped between states and transitions. A transition has drop zones (*FigurePanels*) near both of its end-points and the mid-points of each of its segments. A region is highlighted when the center point of a text object enters the region to indicate that the text object can be dropped there. The layout managers within each *FigurePanel* take care of the automatic alignment of the figures dropped onto them.

6.5.2 Error Handling

We illustrate the practicality of our error handling technique described in Chapter 5 by using the statechart diagrams of UML as an example. In Figure 6.8, the

lower window titled *Statechart Analyzer* shows the interface of the language analyzer (combined parser and semantic analyzer). The figure shows the situation, where the user has a moment ago analyzed the drawing by pressing the *Analyze* button. The errors found by the analyzer are listed in the window. The 'locations' of the detected errors are shown by a numbering superimposed on the actual screen shot of the editor window—the ordinal numbers are not part of the drawing.

When the user selects an error message from the list in the analyzer window, the input objects involved in the error are highlighted in the drawing by a thick frame. In Figure 6.8, the user has selected the first error from the list. The corresponding input object, a filled dot representing an intial state, is automatically highlighted in the editor window. The statechart grammar requires that an initial pseudo state must have exactly one transition exiting the state and no transitions entering the state (see production `Initial_` in Appendix A).

Pressing the *Clear* button empties the list and restores highlighted objects in the editor window to their normal state. The *Context* button launches a dialog where the user can set the source for event and action signatures of the statechart. Because there are errors, the *Simulate* and *Generate* buttons are disabled.

The analyzer provides also facilities for debugging the grammar of the visual language. The *Show parse* button activates a parse graph browser. With the browser the user can inspect the parse paths leading to errors. The browser view shows one parse state at a time and the user navigates in the parse graph by following the (bi-directional) links between the parse states that correspond to the *succession* and *prediction* relationships between parse states (see Section 4.3.3 on page 74). The button is enabled only if errors occur during parsing. If the *Write log* box is checked, the parser writes a log file that contains the full parse table of each parse.

For example, Figure 6.9 shows the parse graph browser launched from the third error shown in Figure 6.8 (missing text symbol in the name compartment of a state). The browser displays in the center of the dialog the parse state that holds the descriptor for the error. In this case, the error is a missing relation (see Section 5.2). The @ symbol in the right-hand side of the production shows the position of the dot. The parse action that created the current parse state is shown under the state.

The parse states are numbered in the order they are inserted into the parse table. The numbered buttons move the focus of the browser to the parse states that are adjacent to the current parse state. The states on the left and on the right are in the succession relationship (black arrowhead) with the current state and the states above and below are in the prediction relationship (white arrowhead) with the current state.

Figure 6.10 shows the inactive state 10 (an instance of *NameCompartment*) that has been created by the error recovery routine *patch* from state 9 (see attribute patching in Section 5.3). Figure 6.11 shows state 8 where the *NameCompartment* instance represented by state 10 was predicted.

The parse graph browser and the logging feature are designed to aid the development of a visual language rather than to help using its implementation. These

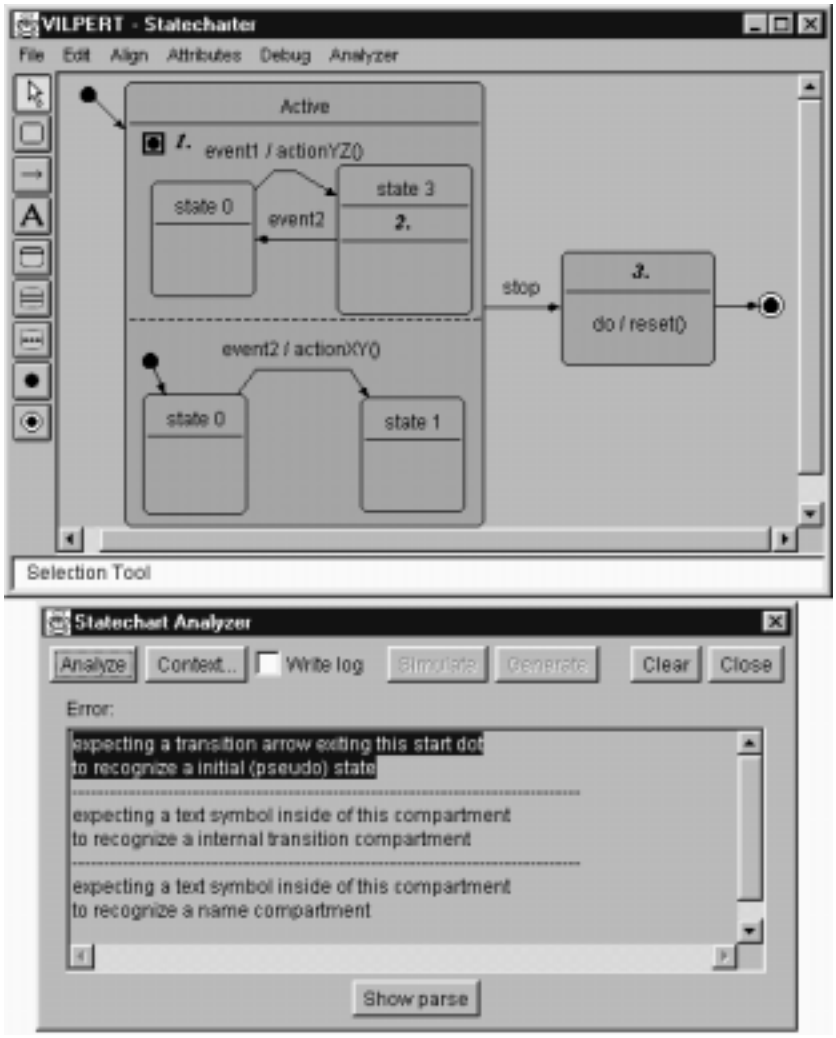


Figure 6.8: Reporting the errors that the parser has found in a statechart.

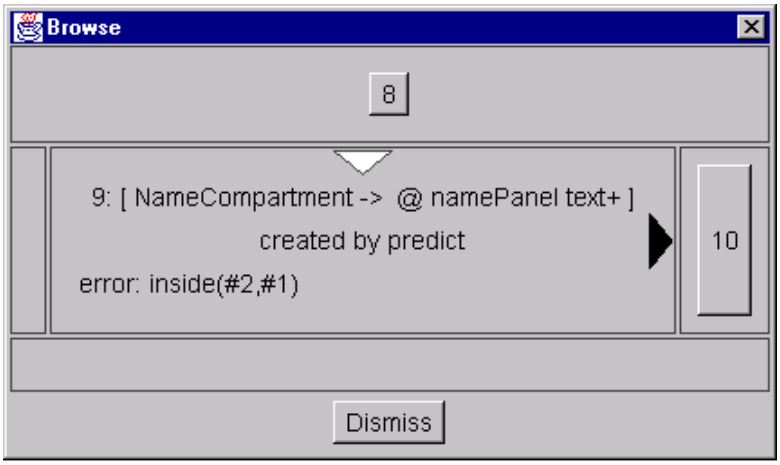


Figure 6.9: Parse graph browser indicating the error at the parse state corresponding to the third error in Figure 6.8.

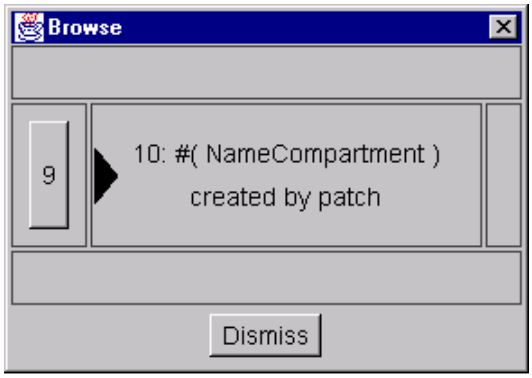


Figure 6.10: The inactive parse state produced by the error recovery mechanism of the parser from state 9 in Figure 6.9.

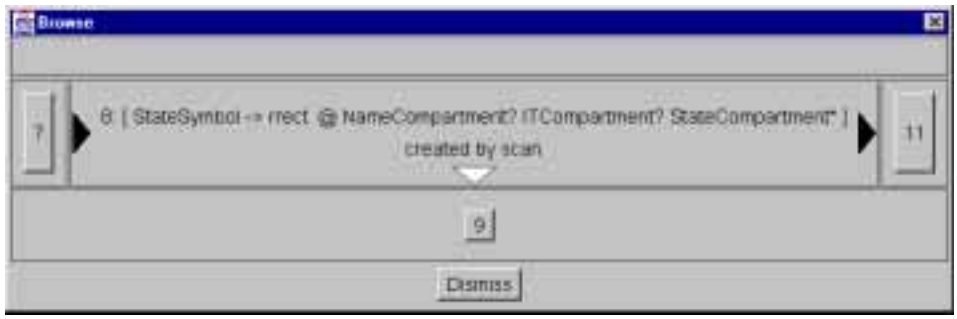


Figure 6.11: The parse state where state 9 in Figure 6.9 was predicted.

kinds of features would probably just confuse the end user of a visual language. When browsing large parse graphs, it is easy to get lost. Therefore, the browser could be extended to show also a map view of the parse graph to aid navigation.

The statechart in Figure 6.8 contains also semantic errors undetected by the parser. After the parser has successfully checked the syntax, the errors are caught in the semantic processing phase. That is, the self-check of the *StateMachine* created by the *StateChartBuilder* fails and the analyzer catches the failures reported by the *StateMachine* (see Figures 6.5 and 6.6).

The interaction of reporting syntax errors, as described above, is used to report semantic errors, also. Figure 6.12 shows the semantic errors reported by the analyzer after the syntax errors in Figure 6.8 have been corrected. The highlighted error message refers to the entry action of state 'stopping': the action 'reset()' is not specified in the action signatures of this statechart that are defined in the dialog launched from the 'Context' button.

Finally, as an example of the interpretation of the input diagram, Figure 6.13 shows the textual description of a state machine produced by the analyzer from the corrected input. The text panel in the lower part of the screen has been scrolled down to show the first lines of the description of the state labelled *Active* in the input.

The default semantic processor used by *AbstractAnalyzer* (see Figure 6.5) is a graphical browser that makes it possible to interactively traverse the parse tree created by the parser. Figure 6.14 shows a correct general tree (grammar *GeneralTree* in Figure 6.3) and the corresponding parse tree browser view.

Creating Error Messages

One requirement of an effective error handling strategy is that the parser can issue informative error messages. In the VILPERT framework, this is achieved by adding descriptive comment strings to the classes of graphical objects, grammatical objects (terminals and nonterminals), and relations. The strings are part of the

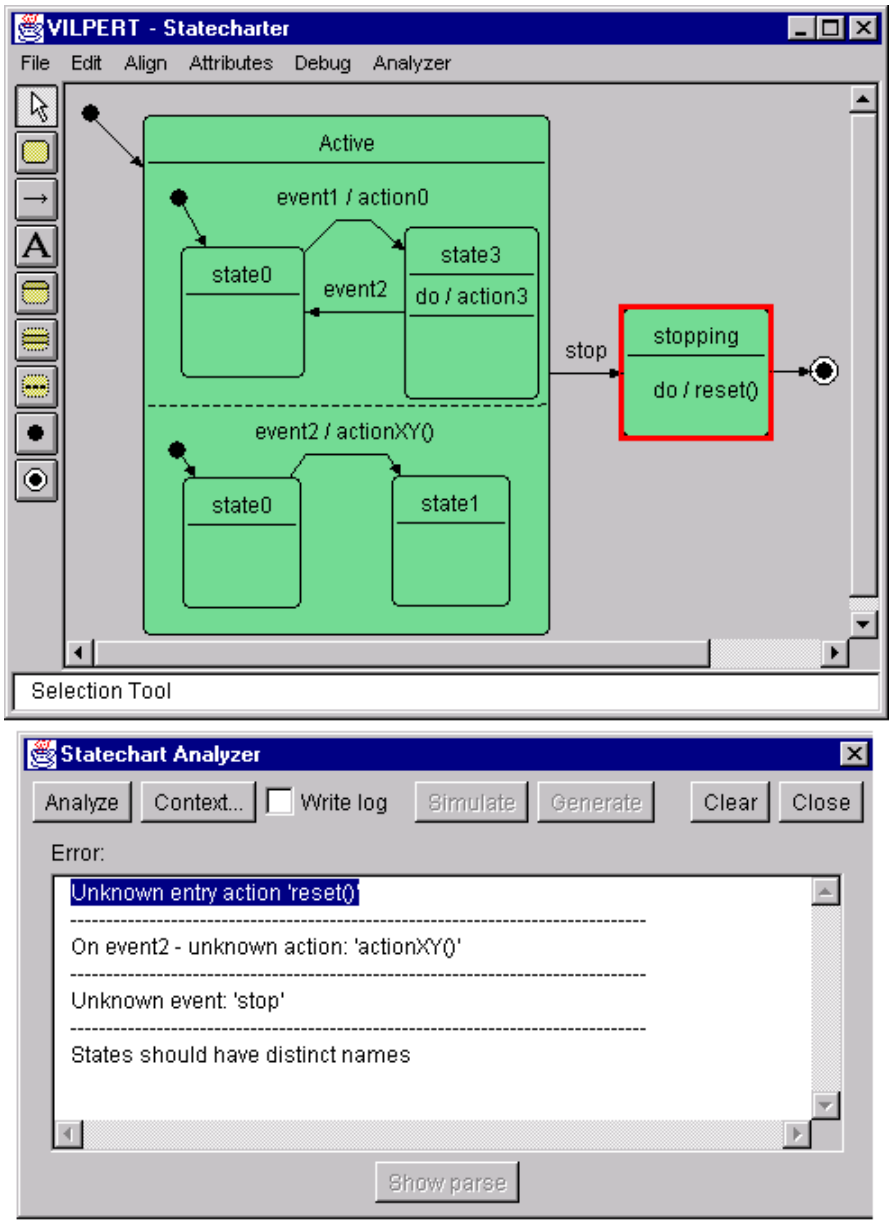


Figure 6.12: Semantic errors detected during the post-parse self-check by the state machine.

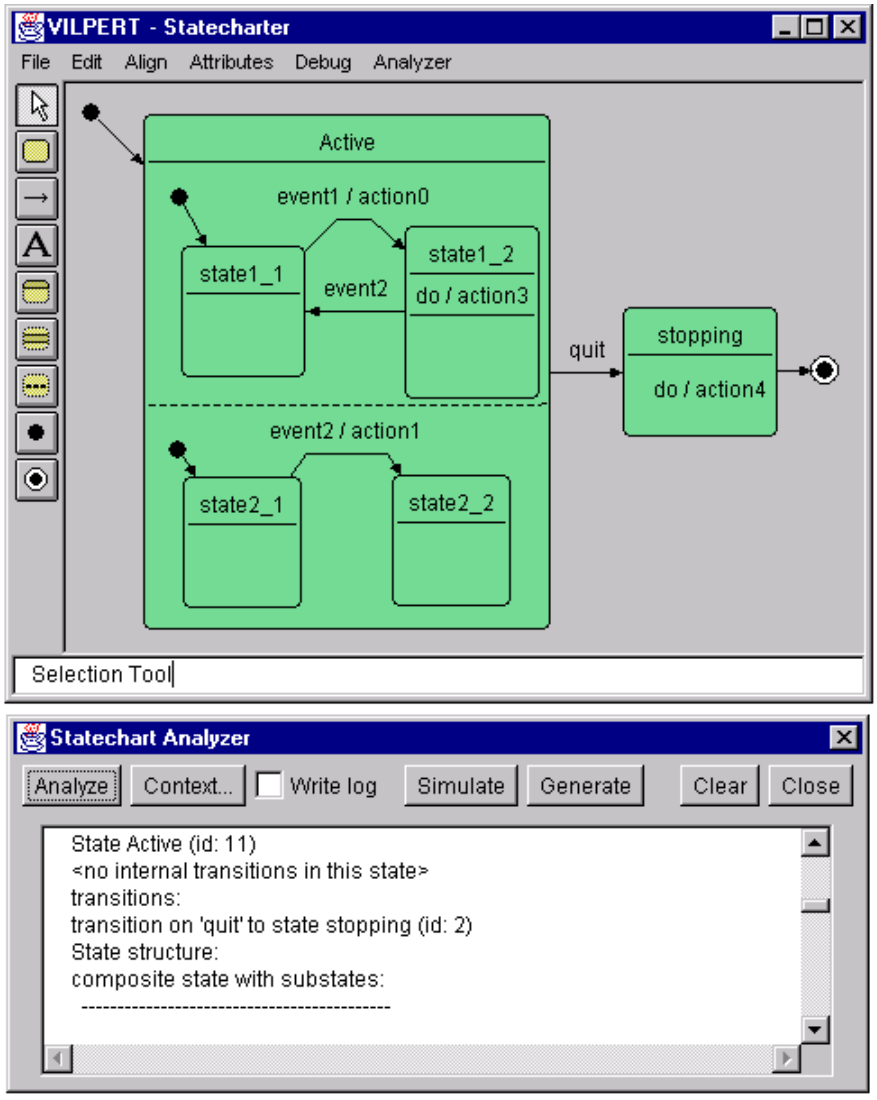


Figure 6.13: A textual representation of the statemachine generated from a correct input.

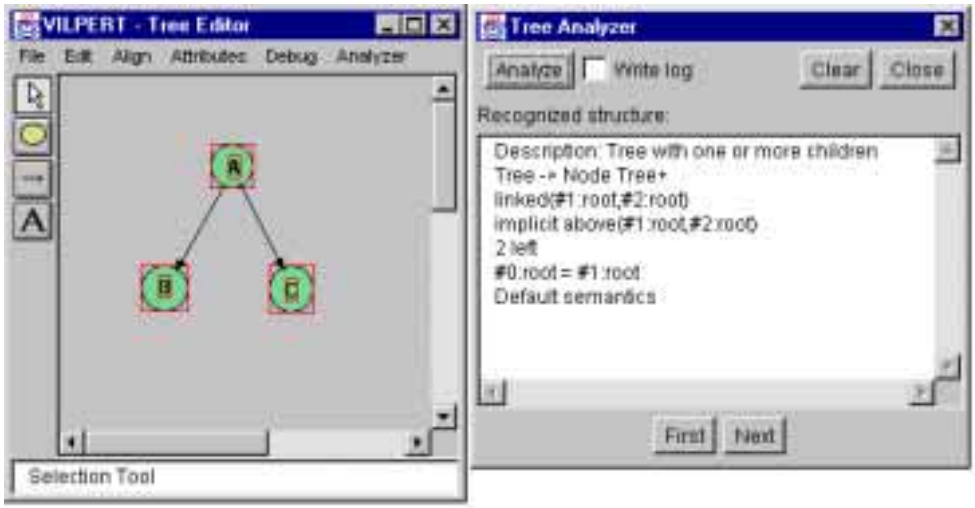


Figure 6.14: The parse tree browser view for a general tree.

map derived from *AbstractObjectToSymbolMap* (see Figure 6.5) that maps editor objects to grammatical objects. Also, the *Production* class allows a descriptive string to be attached to each production.

From these strings, the error handling system is able to compose meaningful messages as shown in Figure 6.8. Of course, being able to directly indicate the input involved in errors is a powerful way to draw the immediate attention of the user to the problems in the input.

6.6 Experiences with VILPERT

6.6.1 About the Implementation

The first version of the EARG framework was implemented in Smalltalk [Tuo98b]. The framework was then rewritten in VILPERT that has been implemented in Java using the Java Development Kit version 1.1.8 and JHotDraw version 5.1.

The *Relap* package comprises 166 package-level classes and about 13 000 lines of code. The extended JHotDraw package contains 185 package-level classes and a total of about 22 000 lines of code. Our extensions to JHotDraw comprise about 45 classes and about 5000 lines of code (included in the previous figure). The sizes of the implementations of our sample visual languages are listed below.

The implementation has not been optimized for performance. On the development machine (PIII 350 MHz processor, 64 MB of memory) and with the JIT compiler by IBM (part of IBM's JDK 1.1.8 distribution), the real-time editing performance

is usually adequate. However, when moving tens of objects simultaneously, the update of the display becomes too slow for comfortable editing, especially with the UML structural diagramming tool (see below). The bottleneck seems to be the drawing of association lines that have a lot of (invisible) panels for holding the decorations attached to them (name, roles, arity, etc.). Also, because the 1.1.8 Java graphics supports only the solid line style, the drawing of dashed lines had to be separately implemented in a suboptimal way.

The VILPERT distribution is available for free[3]. The author has programmed himself the *Relap* package, the JHotDraw extensions, and the sample visual languages.

6.6.2 Visual Languages Implemented with VILPERT

We have implemented with VILPERT the following visual languages (included in the VILPERT distribution)

- (a subset of) UML statecharts
- UML static structural diagrams (object diagrams excluded)
- Structured flowcharts
- Binary trees and general trees

We have already described above the statecharter tool. The implementation of the statechart language does not include forks and joins of transitions, history states, or synch states [Obj99]. However, there is no technical reason why these features could not also be implemented.

Figure 6.15 shows the UML static structural diagramming tool in action. The only features not included in the implementation are the template class construct and qualified associations. Again, there is no technical reason why these features could not be implemented. The tool uses the default semantic processor, the parse tree browser; the tool does not generate any external representation from a static structural diagram.

Figure 6.16 shows the flowcharter tool (Grammar 7.1 on page 134). The flowcharter tool translates the structured flowchart into a Java program (as Java source code). The procedure boxes and the branching diamonds can hold several text objects that the tool concatenates in top-down order. The texts should be Java expressions, if the generated Java source is to be compiled and run.

Table 6.1 shows statistical figures about the implementations of the languages. The table lists the number of nonterminals, terminals, productions, expander relations, predicates, and syntactic (expander) attributes in the grammars. The last

[3] http://www.cs.helsinki.fi/antti-pekka.tuovinen/vilpert

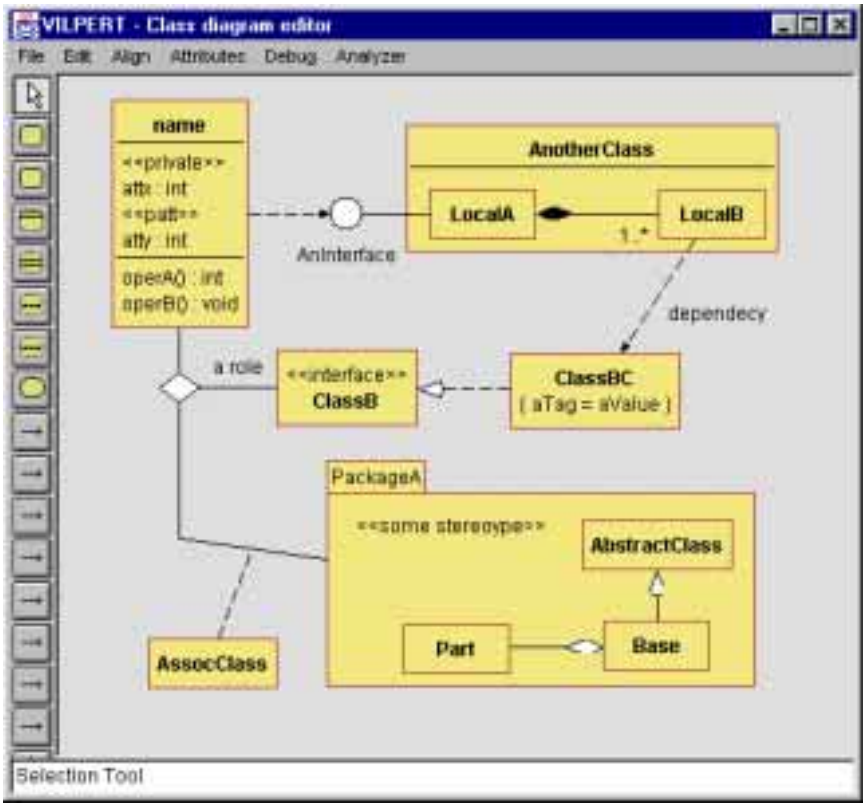


Figure 6.15: The tool for creating UML structural diagrams produced with VILPERT.

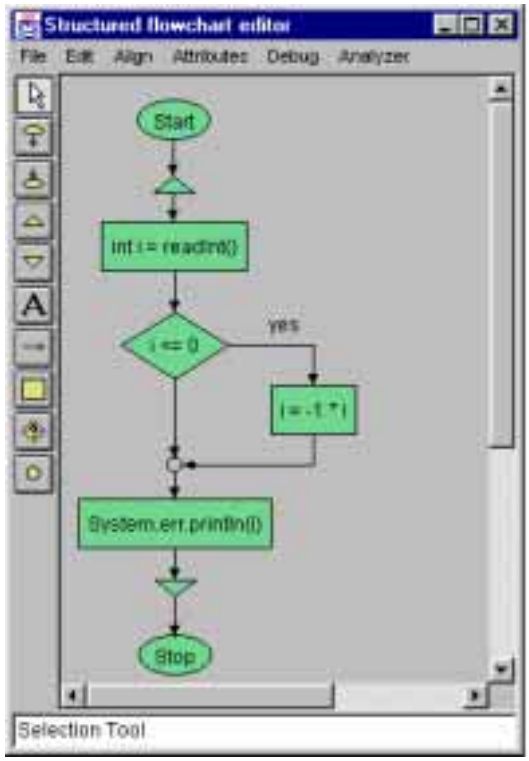


Figure 6.16: The flowcharter tool produced with VILPERT.

Table 6.1: Implementations of Sample Languages

	UML Statechart	UML Structural	Flowchart	Binary tree
$\|N\|$	10	13	3	2
$\|\Sigma\|$	10	24	8	2
$\|P\|$	12	18	9	4
$\|R_e\|$	4	7	3	2
$\|Q\|$	0	1	0	0
$\|A\|$	1	1	2	1
$\sim$ LOC	3700	3700	1900	700

row reports the lines of Java code in each implementation. The figure covers all the code that was spefically written for each language implementation as extensions to the code in the two (sub-) frameworks. However, most of the framework classes of VILPERT are used by the implementations. This implies a reuse ratio of at least 8:2 even for the two largest implementations.

Although the UML Structural diagrams is the largest language, it comprises the same amount of code as the UML Statechart language. This is because the UML Statechart language has a more elaborate semantic processing phase that includes generation of a UML compliant semantic model (state machine) from the input and the generation of a textual representation from the input state machine.

A striking property of the language implementations described in Table 6.1 is the small size of each grammar specification. For instance, the UML statechart grammar comprises 12 productions only and the size of the grammar specification class *StateChart* is about 200 LOC. In addition to this, the classes that translate a drawing as the input to the parser and the parse tree node classes comprise about 400 LOC. The rest of the 3700 lines written by the implementor are divided between the editor part and the semantic analysis part of the implementation. Table 6.2 lists the relative sizes of the three main parts of the UML statechart implementation. Indeed, in our experience, writing the grammar specification of a visual language is the smallest of the subtasks in implementing the language with VILPERT. So, the figures in Table 6.2 correspond to the actual development effort.

Table 6.2: Relative Sizes of Main Parts

Editor	32%
Syntax analysis	17%
Semantic analysis and code generation	51%

6.6.3 Further Remarks

The graphics support in Java 1.1 is relatively poor. Porting the JHotDraw framework to Java 1.3 would make it possible to use the powerful features of the Java2D graphics package.

In the framework, the role of the analyzer object (see Section 6.4.3) could be made more clear. Also, the generation of the input for the parser from the graphical objects (the drawing) in the editor is tedious. Better abstractions are needed there.

Chapter 7

Source-to-Source Translation of Visual Languages

In this chapter we study the problem of translation between visual languages. We present a solid method for the transformation between diagrams, or more generally, for the source-to-source translation between two visual languages. The method is based on a mapping between grammars for the two languages, and on considering translation as a parse tree transformation process.

The method was originally developed for atomic relational grammars augmented with disambiguation constraints only. We have not yet adapted the method to cover the full formalism of extended atomic relational grammars.

We proceed as follows. In Section 7.1, we introduce the language of structured flowcharts that we use in our examples. In Section 7.2, a syntax-directed tree-transformation scheme for visual languages is presented. We give a definition of *relational tree transformation grammars* that are used as a formal mapping between the atomic relational grammars for the involved languages, and an algorithm for the transformation between parse trees over the source and target program. Our technique is illustrated by an example where structured flowcharts are translated into corresponding box diagrams (Nassi-Shneiderman charts). Finally, in Section 7.3, we discuss the issues in extending the method for full extended atomic relational grammars.

7.1 The Structured Flowchart Language

In the translation examples of this chapter, we use Grammar 7.1 that defines a language of structured flowcharts. Structured flowcharts have no “go-tos” which means that every language structure has well defined entry and exit points for directed lines depicting control flow. Figure 7.1 shows the productions in graphical form, and Figure 7.2 shows a flowchart for computing the absolute value of an integer.

Grammar 7.1

$N = \{\text{Flowchart, ProcBlock, RestBlock}\}$
$\Sigma = \{\text{start, stop, text, joint, choice, rect, begin, end}\}$
$S = \text{Flowchart}$
$R_e = \{\text{connects, yesConnects, inside}\}$
$A = \{\text{in, out}\}$
P :

Flowchart → start ProcBlock stop (1)
    connects(start,ProcBlock.in)
    connects(ProcBlock.out,stop)
    Flowchart.in = start
    Flowchart.out = stop

ProcBlock → begin RestBlock end (2)
    connects(begin,RestBlock.in)
    connects(RestBlock.out,end)
    ProcBlock.in = begin
    ProcBlock.out = end

RestBlock_1 → ProcBlock RestBlock_2 (3)
    connects(ProcBlock.out,RestBlock_2.in)
    RestBlock_1.in = ProcBlock.in
    RestBlock_1.out = RestBlock_2.out

RestBlock → ProcBlock (4)
    not exists $x \in \{\text{joint, choice, rect, begin}\}$: connects(ProcBlock.out,x)
    RestBlock.in = ProcBlock.in
    RestBlock.out = ProcBlock.out

ProcBlock → rect text (5)
    inside(text,rect)
    ProcBlock.in = rect
    ProcBlock.out = rect

ProcBlock_1 → choice text ProcBlock_2 joint (6)
    inside(text,choice)
    yesConnects(choice,ProcBlock_2.in)
    connects(ProcBlock_2.out,joint)
    connects(choice,joint)
    ProcBlock_1.in = choice
    ProcBlock_1.out = joint

ProcBlock_1 → choice text ProcBlock_2 ProcBlock_3 joint (7)
    inside(text,choice)
    yesConnects(choice,ProcBlock_2.in)
    connects(choice,ProcBlock_3.in)
    connects(ProcBlock_2.out,joint)
    connects(ProcBlock_3.out,joint)
    ProcBlock_1.in = choice
    ProcBlock_1.out = joint

ProcBlock_1 → joint choice text ProcBlock_2 (8)
    inside(text,choice)
    yesConnects(choice,ProcBlock_2.in)
    connects(ProcBlock_2.out,joint)

connects(joint,choice)
ProcBlock$_1$.in = joint
ProcBlock$_1$.out = choice
ProcBlock$_1$ → joint ProcBlock$_2$ choice text (9)
inside(text,choice)
yesConnects(choice,joint)
connects(ProcBlock$_2$.out,choice)
connects(joint,ProcBlock$_2$.in)
ProcBlock$_1$.in = joint
ProcBlock$_1$.out = choice

Grammar 7.1 has the relation *yesConnects* but not the complementary relation *noConnects*. Instead, we use the relation *connects* in its place. This is due to the fact that the *no* or *false* branch out of the *choice* of the looping constructs (while-do, do-while) appears in other contexts, i.e. not inside the loops. The *noConnects* relation could be used if there were two alternate productions for each context referring a *ProcBlock* (or a *RestBlock*): one with *connects* and the other with *noConnects* on the *out* expander attribute of the *ProcBlock* (or the *RestBlock*) instance. Remote references introduced in Chapter 4 could then be used to enforce the restriction that the *noConnects* relation exists only between a *choice* and some other terminal. However, this would increase the size of the grammar and, for the sake of simplicity, we therefore use Grammar 7.1 instead.

7.2 Syntax-Directed Source-to-Source Translation

In Section 4.3.3 (p. 74), we have shown how a parse tree can be constructed for a visual program, as specified by an atomic relational grammar for the language. In this section we formulate a syntax-directed tree transformation method between two visual languages that are both specified by an atomic relational grammar. The general core of our method is based on techniques originally developed for textual programming languages and structured documents, such as those described in [KPPM84] and [Lin97]. However, while the original idea of syntax-directed translation has been retained in our work, the special characteristics of visual languages have made it necessary to significantly revise the original formalisms and techniques.

7.2.1 Flow of Syntax-Directed Translation

Syntax-directed translation involves two grammars, $G(S)$ for the source language S and $G(T)$ for the target language T, and a mapping $m(G(S), G(T))$ from $G(S)$ to $G(T)$. In our case both $G(S)$ and $G(T)$ are atomic relational grammars. The transformation is made on the level of trees, from the parse tree $T(S)$ for the source program to the parse tree $T(T)$ for the target program. $T(S)$ captures the

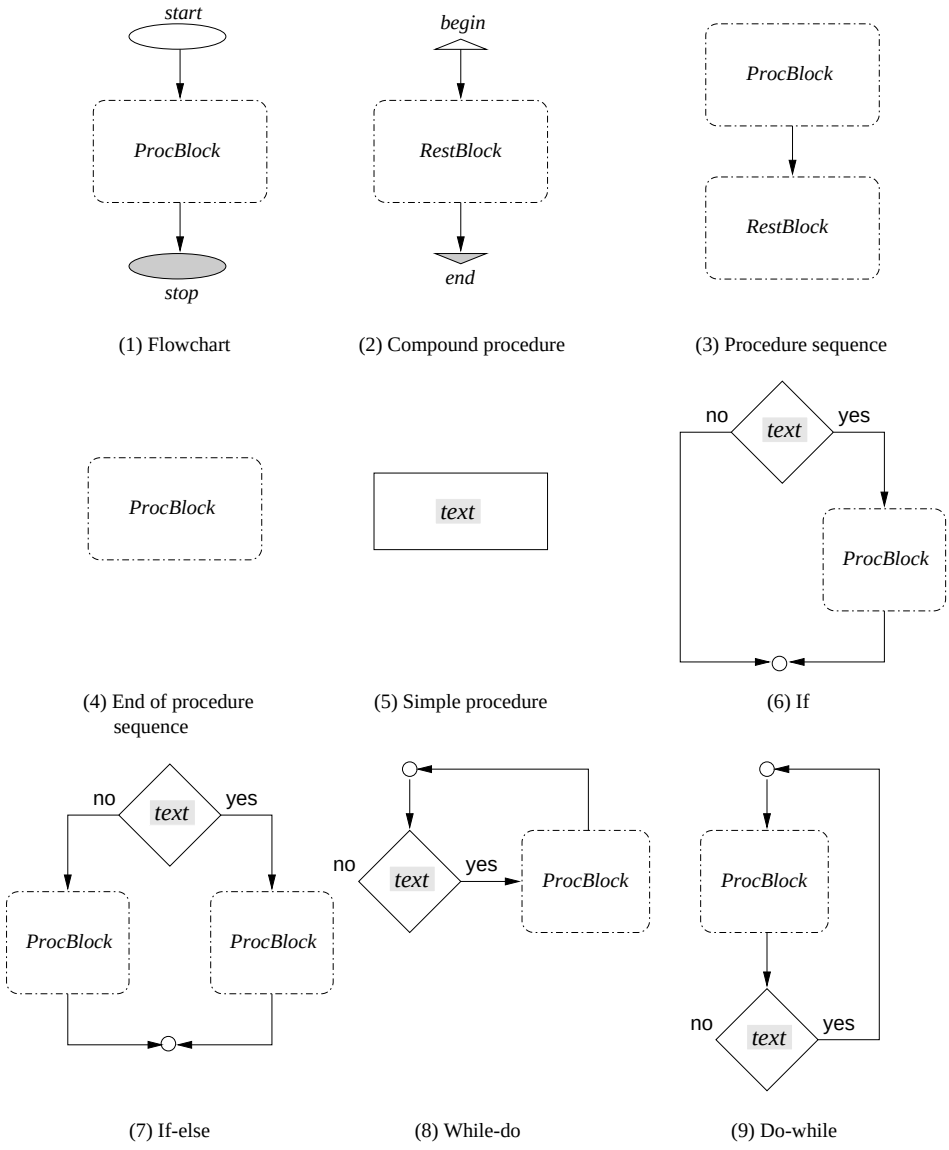


Figure 7.1: The productions of the structured flowchart grammar.

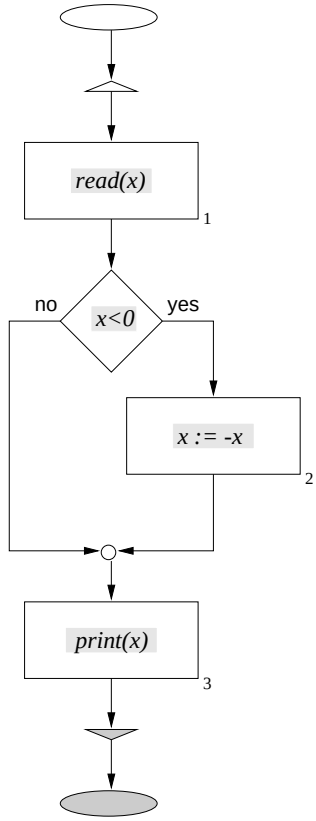


Figure 7.2: A visual program for computing the absolute value of an integer.

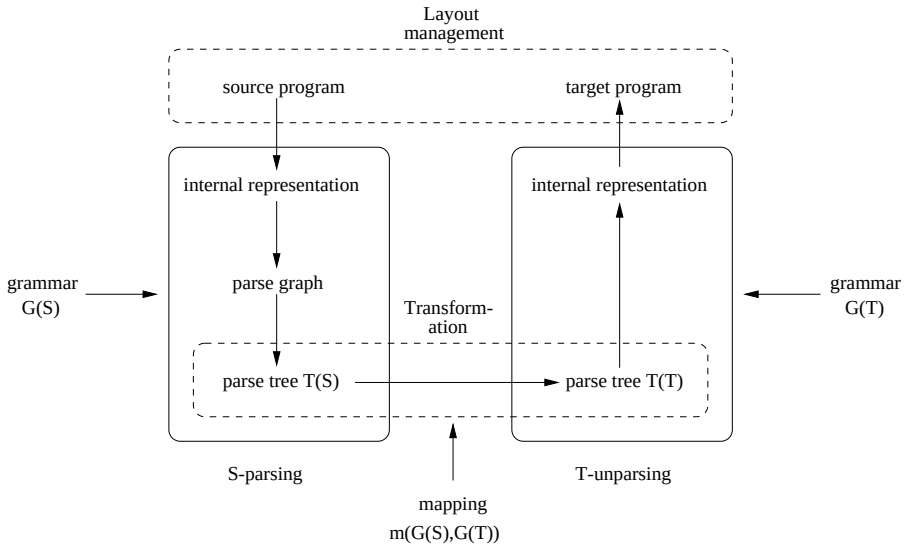


Figure 7.3: Program translation process.

syntactic structure of the source program in terms of grammar $G(S)$, and $T(T)$ captures the syntactic structure of the target program in terms of grammar $G(T)$. The actual program layouts, as visually seen by the user, are not directly involved in the transformation but instead hidden behind parsing and unparsing processes. The complete flow of the translation from a visual source program into a visual target program is depicted in Figure 7.3.

The left-hand side in the figure stands for the processing of the source program, transforming the visual layout of the program into a parse tree which is consistent with the grammar of the source language. The right-hand side illustrates the processing of the target side according to the grammar $G(T)$ of the target language. The task of this phase is to transform the parse tree into the actual concrete program. Since the flow of processing is reverse when compared to the source side, the syntactic structure of the target program with respect to the grammar $G(T)$ has already been coded in the parse tree $T(T)$ and no complementary parse graph is needed.

The source and target processes are integrated by a tree transformation specified by the grammar mapping $m(G(S), G(T))$. This phase transforms the source parse tree $T(S)$, produced by the source parser, into the target parse tree $T(T)$ which is then given as input to the target unparser. The grammar mapping is the central component in our source-to-source translation scheme and will be formally defined below.

We concentrate on the *S-parsing* and *Transformation* phases of the translation (see Figure 7.3). Most notably, the immediate handling of a program's visual layout is not addressed, mainly because it falls beyond formal grammatical modeling

and parsing of visual languages. Of course, in practical systems the program layout is most important. On the source side the layout is typically managed by a (syntax-directed) visual program editor, such as provided by VILPERT, and on the target side by a pretty-printer that generates an optimal spatial representation for a program from its logical description (the parse tree $T(T)$ in our scheme). There exists a number of applicable algorithms and tools for generating graphical layout of diagrams; see, e.g., [DBT88] [GKNV93] [PSTS91].

7.2.2 Relational Tree Transformation Grammars

The task of the transformation phase is to convert a parse tree (with respect to an atomic relational grammar) into another parse tree (with respect to another relational grammar). Recall from Section 4.3.3, that the parse tree of a visual program with respect to an atomic relational grammar represents the grammatical phrase structure imposed on the program. Figure 7.4 shows the parse tree of the input flowchart in Figure 7.2; the framed numbers in Figure 7.4 refer to the corresponding parse states in a parse table that is not shown here.

As usual, each level in the parse tree with n as the root node and $n_1, \ldots, n_k$ as its children corresponds to the syntactic part $s(n) \rightarrow s(n_1) \ldots s(n_k)$ of a production in the grammar, where $s(x)$ denotes the nonterminal or terminal symbol of the grammar that labels node x in the tree. Moreover, the leaf nodes of the parse tree (standing for terminal symbols) are connected by relations that are consistent with the expander constraints in the grammar and reflect the spatial layout of the program. Finally, each interior node in the tree (standing for a nonterminal symbol) is associated with a set of references to leaf nodes in its terminal cover, standing for the values of the nonterminal's expander attribute instances.

Several tree transformation techniques have been developed for textual programming languages. Some of the techniques are rather restricted by just providing removal and insertion of terminal symbols or reordering of subtrees. More powerful methods make it possible to transform the tree quite extensively, for instance by removing and inserting complete subtrees, by moving subtrees into a completely new context, or by removing and adding intermediate levels in the tree. Since our aim is to support transformations between diagrams that may be radically different in their syntax, the transformational grammar class defined below, *relational tree transformation grammars* or *RTT-grammars*, is rather general. RTT-grammars are based on the notion of *TT-grammars* [KPPM84] that were originally introduced for the specification of syntactic tree transformations over context-free grammars. RTT-grammars extend TT-grammars in several ways, most notably by including relational constraints that are irrelevant for textual languages but essential for the modeling and processing of visual languages.

An RTT-grammar describes a relationship between a parse tree over an atomic relational grammar G_S and a parse tree over another atomic relational grammar G_T. In principle the relationship is purely declarative and could be constructively utilized in both directions, either from trees over G_S to trees over G_T or vice versa,

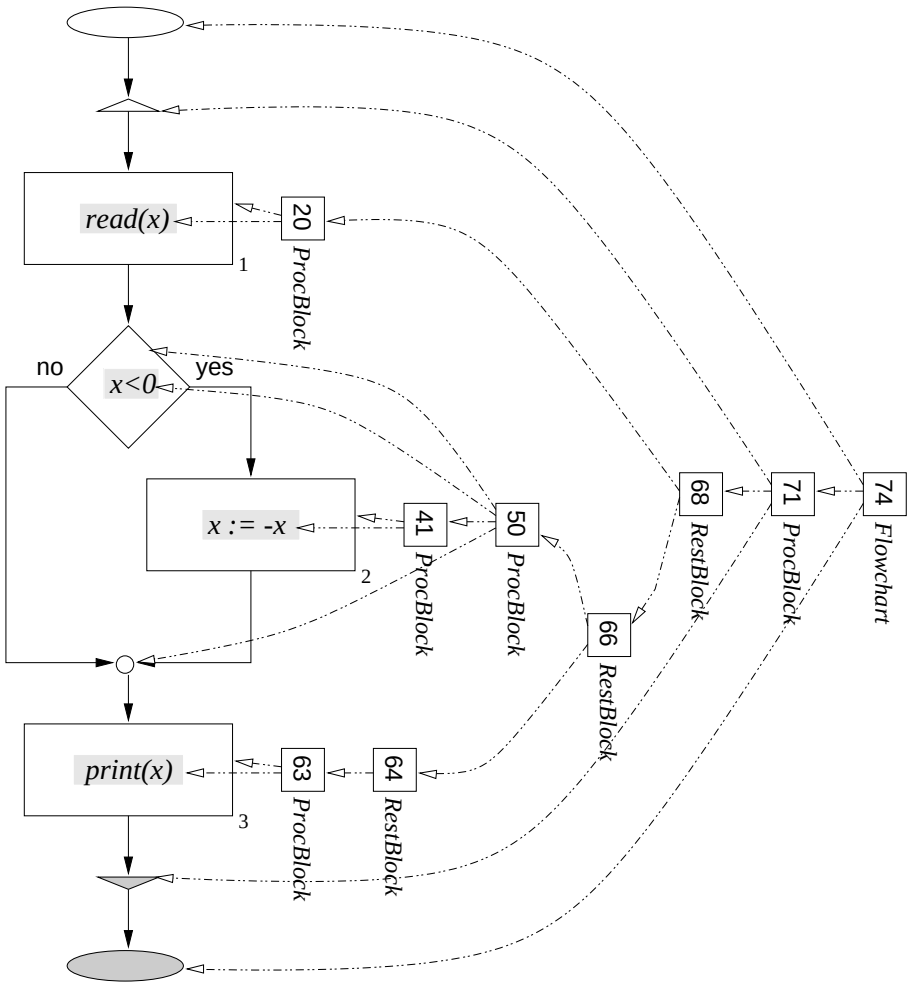


Figure 7.4: The parse tree of the input flowchart in Fig. 7.2

thus being suitable for applications where two-way transformations are common. However, here we just concentrate on one-way transformations by considering one of the grammars (G_S) as the source grammar (standing for the source trees) and the other grammar (G_T) as the target grammar (spanning the target trees). The relationship is described by associating groups of productions in G_S with groups of productions in G_T. In addition, occurrences of nonterminal and terminal symbols in G_S are group-wise associated with those in G_T.

Definition 7.1 A *relational tree transformation grammar* (RTT-grammar) is a 5-tuple $(G_S, G_T, SG_S, SG_T, SGM)$, where

1. G_S is an atomic relational grammar (the *source* grammar).
2. G_T is an atomic relational grammar (the *target* grammar).
3. SG_S is a set of *source subgrammars* where each subgrammar is a group of productions in G_S. One of the nonterminals on the left-hand side of productions is designated as the *start symbol* in each source subgrammar. Every symbol occurrence in a subgrammar must be derivable from its start symbol (i.e., each source subgrammar spans a connected region in a source parse tree). Different occurrences of the same nonterminal symbol in a subgrammar can be distinguished by specifying each of them separately with (different) productions of G_S (i.e., different occurrences of the same symbol may span different source subtrees).
4. SG_T is a set of *target subgrammars* where each subgrammar is a group of productions in G_T.
5. SGM is a set of *subgrammar mappings*. Each mapping is a 4-tuple (S_S, S_T, SOA, CG), where
   (a) S_S is a source subgrammar, $S_S \in SG_S$.
   (b) S_T is a target subgrammar, $S_T \in SG_T$.
   (c) SOA is a set of *symbol occurrence associations*, each of them mapping a grammar symbol occurrence so_S in S_S with a grammar symbol occurrence so_T in S_T. A symbol occurrence in S_T may appear at most once in an association (i.e., the transformation mappings must be unique).
   (d) CG is an optional *context guard*, that is, a Boolean expression over expander attribute and terminal symbol occurrences in S_S.

The subgrammar mappings specify the transformation from a syntactic parse tree skeleton over G_S into the corresponding tree skeleton over G_T. The relational expander linkage is induced on the target tree in the usual manner by relational constraints and attribute assignments over the atomic relational grammar G_T. Notice that the same source or target production can appear in several subgrammars

and mappings, each time with a distinct composition of symbol associations. By this, the same source tree pattern can be transformed differently in different contexts.

Since the spatial structure of the target program is usually quite different from that of the source program, it is not sensible to directly associate the expander constraints or attribute assignments in G_S with those in G_T. However, in many cases the expander information available in the source tree can be conveniently utilized as contextual information when selecting the applicable production group for the transformation. (Recall that the expander attribute instances in a valid parse tree are completely evaluated and refer to leaf terminal symbols.)

As suggested, e.g., in [Shi84], semantic information (attribute values) embedded in the source tree can be used to control its mapping into a target tree. In RTT-grammars semantic transformation conditions are expressed with context guards that shall yield *true* in order for the associated subgrammar mapping to be applicable. Another way of expressing contextual conditions is to include several productions in a source subgrammar, which in that case spans a more extensive region over the source tree than just a single one-production level. Examples of contextual conditions are given in the transformation grammar of Section 7.2.3.

The target subgrammar in a mapping may introduce new symbols that are not associated with any symbol in the source subgrammar. This makes it possible to generate additional subtrees and levels to the target tree. However, such extraneous parts must be connected regions so as to preserve the validity requirements on parse trees. Therefore each non-associated symbol occurrence in the target subgrammar must derive a unique sentence (tree cover) whose symbols (leaf nodes) are either terminals or source-associated nonterminal occurrences. The uniqueness requirement implies that no alternative or recursive productions can be given for the new target nonterminals.

The transformation from source parse trees to target parse trees is defined by the following algorithm, adapted and extended from [Lin97].

Algorithm 7.1 (TREE TRANSFORMATION VIA AN RTT-GRAMMAR)
Input. *An RTT-grammar* $(G_S, G_T, SG_S, SG_T, SGM)$*, and a parse tree* T_S *with respect to the source grammar* G_S.
Output. *A parse tree* T_T *with respect to the target grammar* G_T.

1. *Apply step 2 to all nonterminal nodes in* T_S*, in an arbitrary order. When done, go to step 3.*
2. *Let the step be applied to node* n *labeled with nonterminal symbol* A *of* G_S.
   (a) *Choose a subgrammar mapping* (S_S, S_T, SOA, CG) *in* SGM *such that the source subgrammar* S_S *has* A *as its start symbol, the tree structure spanned by* S_S *matches the subtree in* T_S *with* n *as root, and the context guard* CG *(if any) generates to* true. *If there is no such mapping, return to step 1 (in which case the whole subtree at* n *will be discarded in the transformation).*

(b) *For every production $B \rightarrow B_1 \ldots B_n$ in the target subgrammar S_T, construct a separate target subtree with the root node labeled by B and its children labeled by $B_1, \ldots, B_n$.*

(c) *For every symbol occurrence association (so_S, so_T) in SOA, associate the corresponding nodes in the parse trees. That is, introduce an association between the instance of the occurrence so_S in the source tree T_S and the instance of the occurrence so_T in the target subtree for a production $B \rightarrow B_1 \ldots B_n$ (where so_T is one of the symbols $B, B_1, \ldots, B_n$).*

(d) *For each leaf node l in a target subtree that is not associated with any source tree node, induce the target-specific syntactical cover. That is: Let the label of l be B_i. Generate the unique parse tree with B_i as root by applying the productions in S_T, and replace l with the root of the obtained tree. (As stated above, the leafs of the embedded subtree must be either final terminals or source-associated nonterminals to be expanded further.)*

3. *Apply step 4 to all root nodes of separate target subtrees created in step 2. When no more subtrees can be merged, go to step 5.*

4. *Let the step be applied to the root node m of a target subtree. Let m have the label B and a symbol association to node p in the source parse tree T_S. Find a leaf node n in another target subtree with the label B and an association to the same source node p. Merge the target subtrees at node n, i.e., replace the leaf node n with the root node m (and, consequently, with the whole subtree for m).*

5. *If the result is a connected tree and all its leaves are terminals, complete it into the target parse tree T_T with respect to grammar G_T (a) by associating the interior nodes in the tree with terminal references, obtained by evaluating the attribute instances according to the assignments in G_T; and (b) by inducing the set of expander constraints in G_T as relations over the terminal leaf nodes. Otherwise, the transformation fails.*

Notice that our tree transformation algorithm may fail, in which case the source program cannot be translated into a corresponding target program. This happens if some source subtree is not matched by any grammar mapping, or if the symbol associations are incomplete in the sense that they leave unmerged subtrees hanging on the target side. It would be possible to rule out transformation failures completely by imposing more restrictions on the formal definition of relational tree transformation grammars, but then they would become less powerful and probably too inflexible for practical applications.

Also the final layout of the target program may fail, in case the relational constraints to be solved are in conflict (for instance, requiring that symbol a is spatially both above and below of symbol b). The grammatical formalism cannot exclude such conflicts without stating severe conditions on the form of constraints

used, and therefore the checking of their satisfiability is postponed to the layout manager.

7.2.3 Example – From Flowcharts to Box Diagrams

Let us return to the language of structured flowcharts defined in Section 7.1. A flowchart visualizes the algorithmic aspects of a program by describing the flow of control within it in terms of conditional statements (*if*, *if-else*), iterative statements (*while-do*, *do-while*), and compound procedures (statement / procedure sequences). Flow of control can be described by a number of other alternative representations as well, one of the classical ones being *structured box diagrams* (also known as Nassi-Shneiderman charts).

While structured flowcharts syntactically consist of procedure blocks connected by control-flow arrows, the principle in box diagrams is to describe an algorithm as a stack of nested statement boxes with control flowing sequentially from top to bottom. Therefore a flowchart and a box diagram for the same algorithm look quite different, making a translation between them a non-trivial task.

Let us specify the translation from flowcharts to box diagrams as a relational tree transformation grammar. The source part of the translation has been given in Grammar 7.1. The target grammar is implicitly embedded in the RTT-grammar developed below. The expressions of the form

source_symbol_occurrence.target_symbol_occurrence

denote the symbol associations in the target subgrammars. The expander constraints and attribute assignments are given in the target subgrammars in terms of the target symbol occurrences. Since the constraints and assignments of the source grammar are not used in the transformation, they are not repeated in the specification. The left-hand side of the first production in a source subgrammar denotes its start symbol.

The first subgrammar mapping applies to the root of a source tree and states that whenever we find a source subtree that matches the pattern specified by the source subgrammar, we shall construct the corresponding subtree(s) as specified by the target subgrammar.

Mapping 1
Source subgrammar:

Flowchart → *start ProcBlock stop* *(1)*
ProcBlock → *begin RestBlock end* *(2)*

Target subgrammar:

Flowchart.BlockDiagram → *RestBlock.Block*
BlockDiagram.top = Block.top

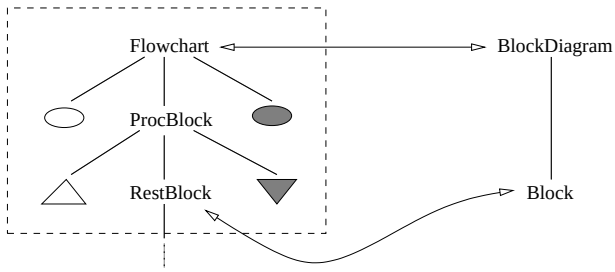


Figure 7.5: Top-level transformation.

According to this mapping, the (root) node of the target tree with label *BlockDiagram* is associated with the root node *Flowchart* in the source tree. The target root has one child, with label *Block*, associated with the nearest *RestBlock* descendant of the source root, as shown in Figure 7.5 (the matched source tree region is enclosed within a dashed rectangle, and symbol associations are denoted by curved dual arrows). Notice the flexibility of transformation provided by having two source productions in the mapping: the *ProcBlock* level in the source subtree can be completely discarded in the target subtree. Note also that the terminal flowchart symbols (start, stop, begin, end) are thrown out, since they do not appear in box diagrams.

The following mapping specifies how the transformation continues at the next level in the source tree.

Mapping 2
Source subgrammar:

$RestBlock_1 \rightarrow ProcBlock\ RestBlock_2$ *(3)*

Target subgrammar:

$RestBlock_1.Block_1 \rightarrow ProcBlock.Stat\ RestBlock_2.Block_2$
onTop(Stat.frame,$Block_2$*.top)*
equalWidth(Stat.frame,$Block_2$*.top)*
$Block_1.top = Stat.frame$
$Block_1.bottom = Block_2.bottom$

This subgrammar mapping specifies how structured blocks in box diagrams logically match the recursive pattern of procedure sequences in flowcharts. The expander relation *onTop* in the target grammar states that successive statement blocks are placed in a stack, with the first statement on top. The predicate *equalWidth* serves as an instruction for the layout manager to draw the statement blocks (boxes) equally wide in the target diagram.

The following mapping closes the recursion and specifies how the last statement in a procedure sequence shall be moved into a box diagram. Notice a similar disambiguation constraint as used in production (4) of Grammar 7.1.

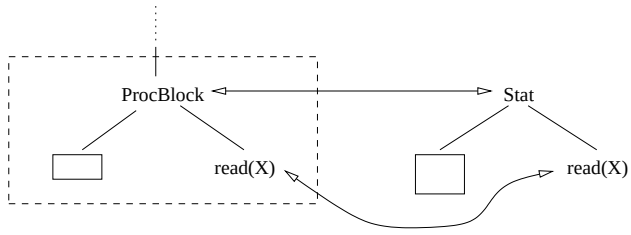


Figure 7.6: Bottom-level transformation.

Mapping 3
Source subgrammar:
RestBlock → *ProcBlock* *(4)*

Target subgrammar:
RestBlock.Block → *ProcBlock.Stat*
not exists $x \in \{rect\}$*: onTop(Stat.frame,*x*)*
Block.top = Stat.frame
Block.bottom = Stat.frame

The next subgrammar mapping transforms a single procedure of a flowchart into a statement block in a box diagram:

Mapping 4
Source subgrammar:
ProcBlock → *rect text* *(5)*

Target subgrammar:
ProcBlock.Stat → *rect text.text*
inside(text,rect)
Stat.frame = rect

The visual shape of a single procedure in a flowchart and a single block in a box diagram is the same, a rectangle containing algorithmic text. The text is the same in both diagrams, as specified by the symbol association *text.text*: in an RTT-grammar the association between terminal symbols implicitly also copies the contents of the source terminal to the target terminal. Notice that we must not copy the source rectangle in the same way, since the rectangles enclosing the text may be of different size in the diagrams. The mapping results in the tree match illustrated in Figure 7.6.

The RTT-grammar mappings given so far specify how the upper region of a parse tree for a flowchart shall be piece-wise transformed into that for the corresponding box diagram. Figure 7.7 shows how these mappings are merged by our tree transformation algorithm into a connected target tree pattern. The source program

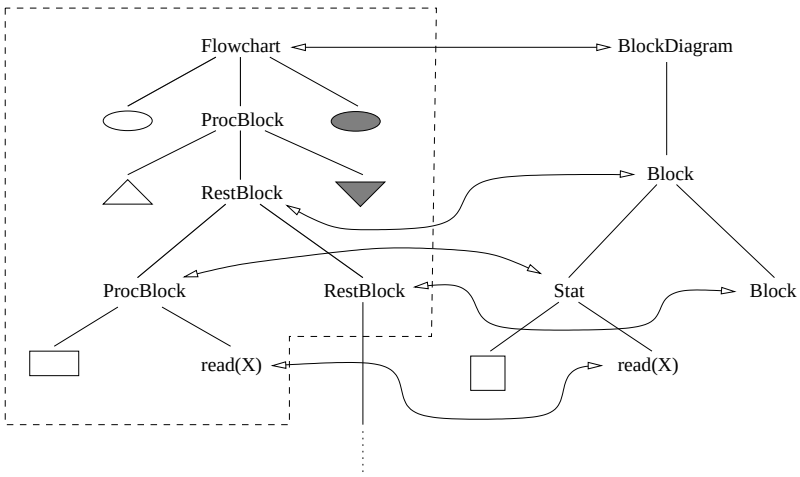


Figure 7.7: Merged transformation.

underlying the transformation has been shown in Figure 7.2 with its complete parse tree in Figure 7.4.

The next mapping specifies the transformation of an *if*-statement in a flowchart. In case the operational part embedded in the *yes*-branch is a single procedure, there will be a single statement block for it in the box diagram, whereas a compound procedure (a sequence of procedures) will be transformed into a stack of successive blocks.

This principle calls for the use of context information in the RTT-grammar. In this mapping the selective transformation is stated as a context guard that yields *true* if the operational part is a single procedure, that is, something else than a compound procedure.

Mapping 5
Source subgrammar:

$ProcBlock_1 \rightarrow$ *choice text* $ProcBlock_2$ *joint* *(6)*

Context guard:

$ProcBlock_2.in \neq$ *begin*

Target subgrammar:

$ProcBlock_1.Stat_1 \rightarrow$ *rect Condition Empty* $ProcBlock_2.Stat_2$
inside(Condition.frame,rect)
inside(Empty.frame,rect)
$inside(Stat_2.frame,rect)$
onTop(Condition.frame,Empty.frame)
$onTop(Condition.frame,Stat_2.frame)$

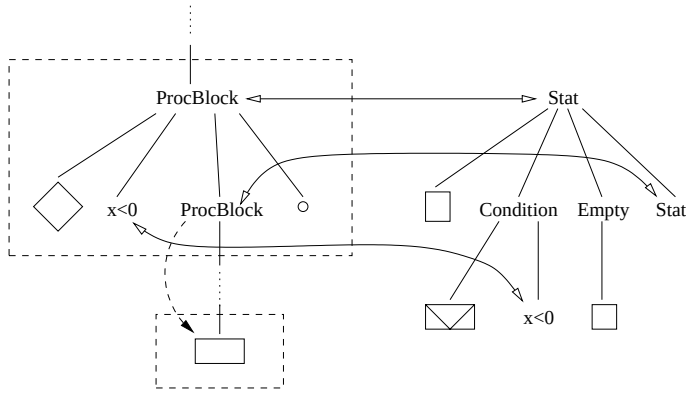


Figure 7.8: Transformation of simple *if*-statements.

toTheLeft(Empty.frame,Stat$_2$.frame)
equalWidth(Empty.frame,Stat$_2$.frame)
Stat$_1$.frame = rect

Condition $\rightarrow$ *choiceRect text.text*
inside(text,choiceRect)
Condition.frame = choiceRect

Empty $\rightarrow$ *rect*
Empty.frame = rect

This mapping also demonstrates the introduction of new nonterminals *Condition* and *Empty* and a new terminal *choiceRect* in the target grammar. Figure 7.8 shows how the nonterminals generate additional complete subtrees, and how a source symbol (*text*) can be moved into a deeper level in the target tree by association. The context information applied in the guard is depicted as a dashed arrow from the associated source nonterminal, in a way extending the matched region in the source tree.

The RTT-grammar mapping 5 above is significantly more complex than the previous ones, because the structure of an *if*-statement in a box diagram is quite different from the structure of the corresponding statement in a flowchart. On the other hand, this mapping also demonstrates the transformational power of RTT-grammars. In essence, the specification induces diagram transformations illustrated in Figure 7.9.

The transformation of *if-else*-statements follows the same principles as the transformation of *if*-statements, except that the box for the *no*-branch is this time not empty but consists of a single block or a stack of blocks. For brevity, the RTT-mappings are omitted.

In box diagrams, *while-do*-statements are represented as shown in Figure 7.10 where *cond* stands for the condition and *stat* for the statement(s) to be iteratively executed. As an example of the translation of *while-do*-statements, the grammar

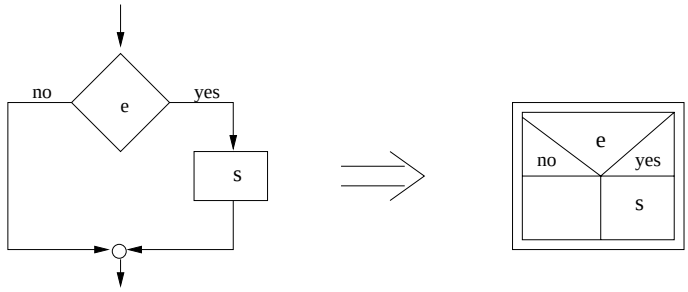


Figure 7.9: From simple *if*-flowchart to simple *if*-box.

mapping 6 below specifies the case when iteration entails a simple procedure in the source flowchart. The compound case is specified accordingly, but the details are omitted here.

Mapping 6
Source subgrammar:
$ProcBlock_1 \rightarrow$ *joint choice text* $ProcBlock_2$ *(8)*

Context guard:
$ProcBlock_2.in \neq$ *begin*

Target subgrammar:
$ProcBlock_1.Stat_1 \rightarrow$ *rect text.text* $ProcBlock_2.Stat_2$
inside(text,rect)
lowerRight($Stat_2$*.frame,rect)*
above(text,$Stat_2$*.frame)*
$Stat_1$*.frame = rect*

Do-while-statements (*repeat-until*) in box diagrams are similar to *while-do*-statements, except that the spatial position of the condition and the iterative statement is swapped. The grammatical specification of *do-while*-transformations follows the principle of *while-do*-transformations specified in RTT-mapping 6 and is therefore omitted.

Finally, the translation of compound procedures and the special case of a flowchart with a single procedure block must be specified. These translations are specified according to the principles in grammar mappings 1–6 and are therefore omitted.

When applying this RTT-grammar and the tree transformation algorithm to the transformation of the parse tree given in Figure 7.4, the target tree in Figure 7.11 is obtained. The leaf boxes in the tree are connected by arrows reflecting the relations specified in the target grammar of the transformation, with those with filled head standing for *onTop*, those with open head standing for *equalWidth*, and

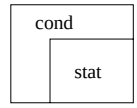


Figure 7.10: *While-do* in box diagrams.

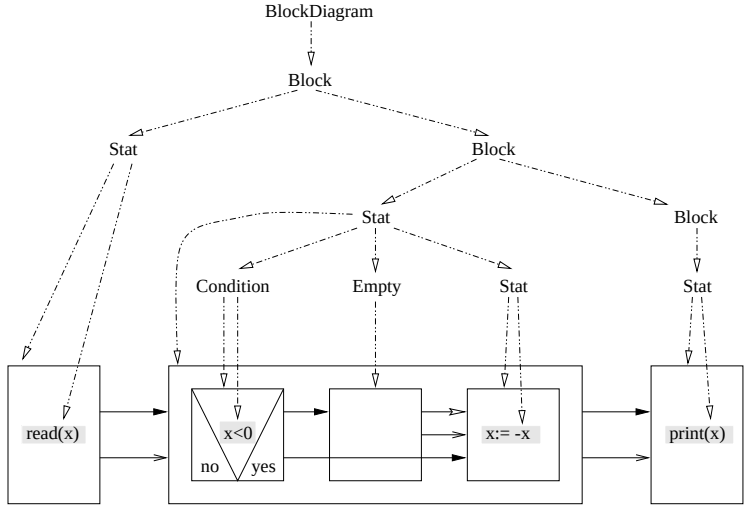


Figure 7.11: Target parse tree for a box diagram.

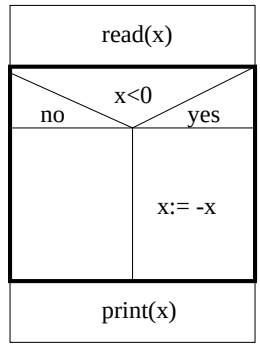


Figure 7.12: Target box diagram.

that with hollow head standing for *toTheLeft*. The relation *inside* over leaf nodes is depicted by spatial enclosure.

This parse tree can then be coded into an internal representation that stores the terminal symbols and their relations (including the logical containment information implicit in the tree). Finally a layout manager processes the internal representation, interprets the spatial relations, and produces the target program. The resulting box diagram, corresponding to the source flowchart in Figure 7.2, might look as that shown in Figure 7.12.

7.3 Integration to VILPERT

This work was originally done for atomic relational grammars and published in [PT98]. We have not implemented the method yet.

The transformation method described above needs small extensions to cover also the new features of extended atomic relational grammars: iterative and optional right-hand side symbols, predicates, and ordering expressions (disambiguation constraints were already used in the original method). However, fundamental changes to the method are not necessary in order to support the syntax-directed translation of visual languages specified by EARG grammars.

VILPERT provides good facilities for the integration of a translation tool based on our method into the framework. The translator can be handled simply as a semantic processor; the parser produces an explicit parse tree for the translation and the input relations can be easily made accessible to the translator. Furthermore, VILPERT maintains a mapping between the terminal symbols (input objects) processed by the parser and the graphical objects in the editor. This makes it possible to access the graphical attributes (location, size, etc.) of the original input objects when creating a layout for a translated diagram that consists of generated objects. Of course, implementing the translation algorithm is still a non-trivial task.

Chapter 8

Related Work

In this chapter, we present the work closely related to ours and point out the differences to our work. We start in Section 8.1 by describing several representative approaches to the specification and implementation of diagrammatic visual languages; we look at grammar-based, object-oriented, and other approaches. Next, in Section 8.2 we discuss error handling in visual language parsing. Finally, we discuss source-to-source translation in Section 8.3.

8.1 Specification and Implementation of Visual Languages

In the literature, several different approaches have been presented to visual language specification and recognition (parsing). See [MMW98] for an extensive survey. In the following, we look at systems that cover the specification as well as the implementation of visual languages. Most of the following systems employ a grammatical model for specifying the syntax and (in some cases) the semantics of a visual language. The grammatical models have associated parsing algorithms for analyzing visual programs. There are also approaches that rely on object-oriented frameworks for constructing implementations of visual languages and approaches where a visual programming environment is created automatically from a metamodel specified by the user.

8.1.1 Grammar-based Approaches

VLCC

Costagliola & al. describe in [CTOL95] Visual Language Compiler-Compiler (VLCC) that is a graphical system for the automatic generation of visual programming environments. VLCC assists a designer to implement a visual language by

providing support for the specification of the syntax, the semantics, and the graphical objects of the language.

VLCC uses *positional grammars* (PG) [CLOT98] to automatically generate visual programming environments. Languages specified by positional grammars belong to the same class in the CCMG hierarchy of visual languages [MM98a] as atomic relational grammars. This means that the two grammar formalisms have roughly the same expressive power.

Positional grammars are an extension of context-free grammars [AU71]. Thus, most results from LR parsing can be extended to PGs. The VLCC system deals with textual languages as special visual languages so that languages mixing textual and graphical elements can be specified.

Like conventional grammars for string languages, PGs have nonterminals, terminals, a start symbol, and a set of productions. In addition to these, PGs also have a set of relation symbols that denote binary relations between grammars symbols. Each symbol has syntactic attributes that are used by relational expressions in grammar productions to specify how (nonterminal) symbols are composed.

The productions of PGs have the form

$$A \rightarrow x_1 R_1 x_2 \ldots R_{m-1} x_m, \Delta$$

where each R_i is a compound relation that can refer to more than one binary relation between the next right-hand-symbol x_{i+1} and some previous symbol in $x_0 \ldots x_i$. The attribute inference rule Δ defines how the syntactic attributes of A depend on the attributes of x_i where $0 \leq i \leq m$.

The parsing algorithm for PGs is an extension of the LR parsing method for string languages. The main difference to string language parsing is that the relations between symbols are used to decide which input symbol should be scanned next. The complexity of parsing is the same as for conventional LR parsing if the lookup (called spatial query) for the next token to scan is quick.

Figure 8.1 shows the structure of the VLCC system. The system has a graphical editor for specifying the grammars visually. The parameterized grammar editor (PGE) allows the user to create the productions of the grammar and all the production elements: terminal and nonterminal symbols, (graphical) relations between the symbols, and syntactic attribute inference rules. It is also possible to associate semantic attributes with grammar symbols and give semantic actions for evaluating the semantic attributes. The attribute inference rules and the semantic actions are given in the C language since VLCC employs YACC [ASU86] to produce the final compiler. It is possible to store symbols, relations, and attribute inference rules in libraries. The PGE can be configured to support different classes of visual languages.

PGE translates the visual grammar into YACC input format and includes the necessary information for producing the editor for the language. The visual programming environment generator (VPEG) uses the input from PGE to generate the C

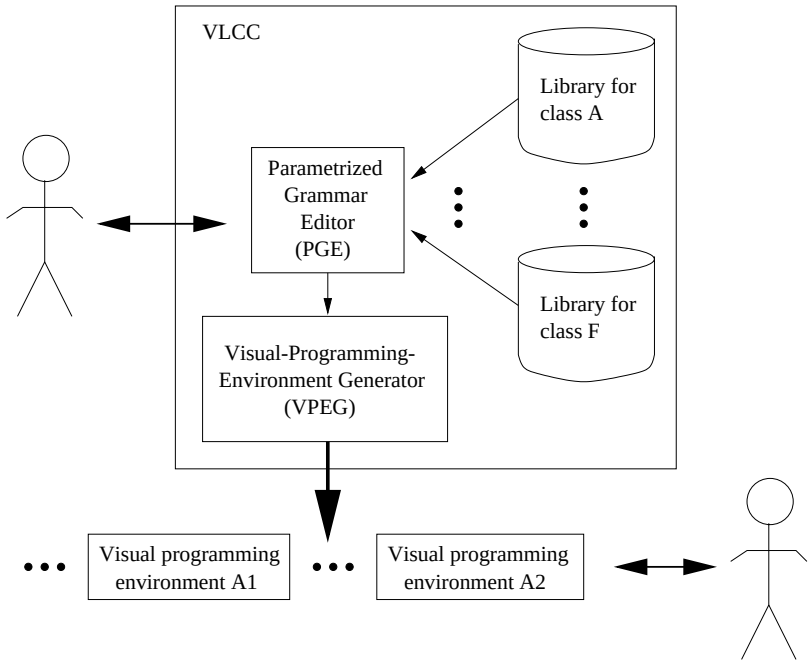


Figure 8.1: The VLCC system (from [CTOL95]).

compiler program and to customize a predefined editor template with the symbols and relations specified by PGE. With the editor generated by VPEG, the user can enter a visual program and execute it according to the semantics specified in the YACC grammar.

The visual languages supported by VLCC are limited to context-free iconic languages and context-free plex-like languages where the nodes of a (graph-like) diagram have a fixed number of connecting points for attaching connectors between nodes. In [CLOT97a] the authors discuss the possibility to incorporate special syntactic models for different classes of visual languages (iconic, plex, box-and-graph) into the VLCC system. In [CDLO94] the authors discuss extensions to the PG model in order to give better support for flow-graph languages.

Penguins

The Penguins system by Chok and Marriott [CM98] [CM95] supports the development of intelligent diagram editors. The intelligent diagram is a metaphor for diagramming where the underlying graphical editor parses the diagram as it is being constructed, peforming error correction and collecting geometric constraints about the relationships between diagram components. A constraint solver uses the geometric constraints to maintain the diagram's semantics during diagram manipulation.

Penguins automates the construction of graphical editors that support the intelligent diagram concept. The system follows the compiler–compiler approach to the generation of the diagramming editor. The generated editor supports the creation, manipulation, and interpretation of diagrams in the particular visual language whose high-level specification is provided (by the programmer) in a specification language based on constraint multiset grammars (CMG).

CMGs [Mar94] are a high-level declarative language. They belong to attributed multiset grammars and use constraints to specify topological, geometric, and semantic relations between subdiagrams or tokens in a diagram. This means that there is no explicit representation of spatial relations. The expressive power of CMGs is reflected in the fact the CCMG language hierarchy [MM98a] is based on a restricted class of full CMGs.

For example, the following CMG production is from a grammar for finite state automata:

```
TR:transition ::= A:arrow, T:text
   where exists R:state, S:state where
   T.midpoint close_to A.midpoint,
   R.radius = distance(A.startpoint, R.midpoint),
   S.radius = distance(A.endpoint, S.midpoint)
   and TR.from=R.name, TR.to=S.name, TR.label=T.string.
```

The production defines a transition to consist of an arrow object and a text object that is near to the midpoint of the arrow. Furthermore, the startpoint and endpoint of the arrow are constrained to reside on the perimeter of a state object. In the production above, *midpoint*, *startpoint*, *endpoint*, and *radius* are geometric attributes whereas *from*, *name*, *string* and *label* are semantic attributes.

The recognition algorithm for full CMGs has exponential complexity. However, cycle-free, stratified, and deterministic CMGs [CM95] have polynomial time complexity. These restricted CMGs seem to be more expressive than atomic relational grammars because they can epress context sensitive constraints in grammar productions.

Figure 8.2 shows the overall structure of the Penguins system. In Penguins, the parser generator *VisualGen* generates from a CMG specification an incremental diagram parser (spatial parser) which is incorporated into the standard diagramming environment *VisualEdit*. The diagramming editor provides standard graphic primitives (lines, circles, text, arrows). In order to provide support for free-hand drawing with a pen (as an alternative input method), the system provides also a tokenizer that can map input gestures to the graphic primitives. A constraint solver is used by the editor to provide the constraint solving mechanism needed in geometric error correction and diagram manipulation (error handling will be discussed in more detail in Section 8.2). It is also possible to extend certain modules to cater for application specific computation.

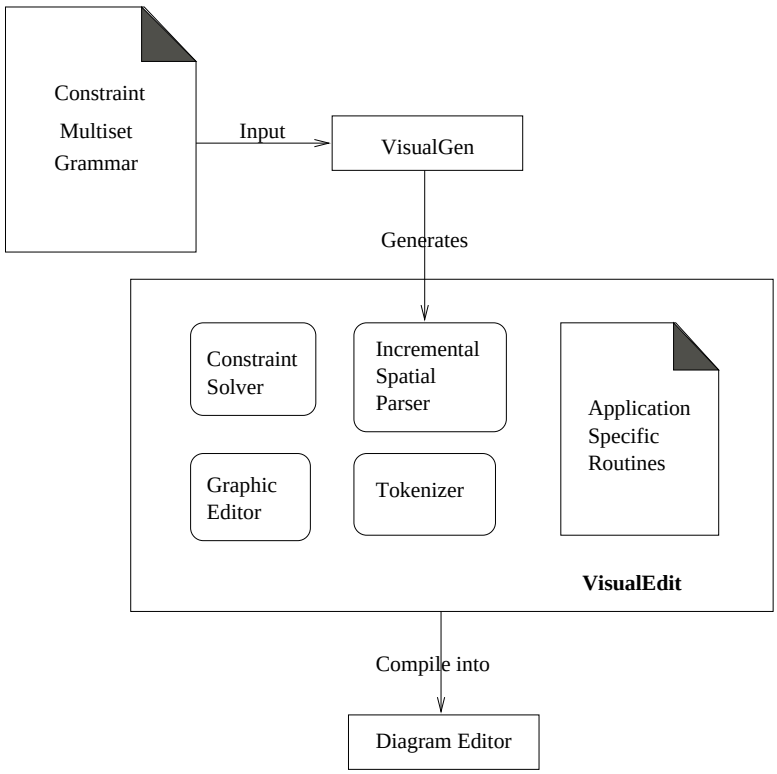


Figure 8.2: An overview of the Penguins system (from [CM98]).

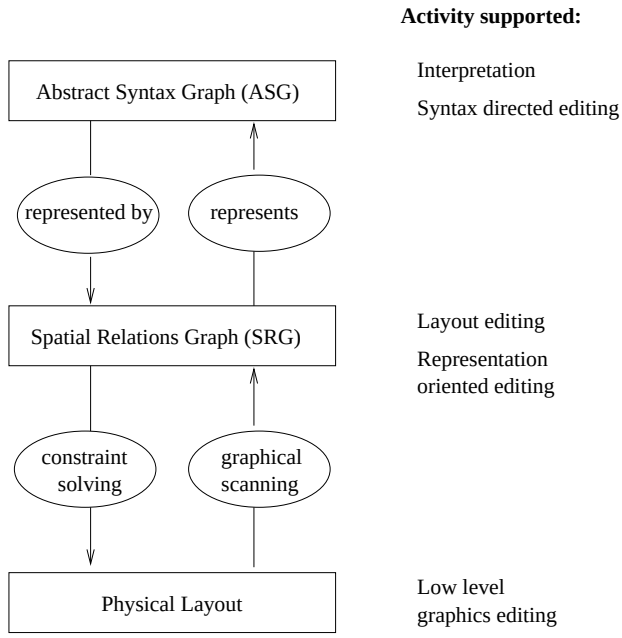


Figure 8.3: The three level representation of visual programs (from [RS96]).

In addition to error correction, constraints can be used also to help the layout of visual programs. Constraint based diagram beautification in Penguins is discussed in [CMP99].

Graph Grammar Approaches

PROGRES

Rekers and Schürr present in [RS96] the infrastructure for a graph-based visual environment generator. The central concept of their design is a three level representation of visual programs (diagrams) shown in Figure 8.3.

The lowest level of the model is the graphical representation of a diagram consisting of graphic primitives (lines, circles, characters etc.) with properties like size and location. Graphical scanning produces a spatial relations graph (SRG) that describes the structure of the diagram in terms of higher level spatial relationships that hold between primitive objects (touches, contains, is-a-label-of etc.). The third level provides the most abstract representation that describes the visual program in terms of the concepts of the language. That is, the edges and nodes of the abstract syntax graph (ASG) gives a graph presentation of the meaning of the visual program.

The authors suggest to use graph grammars for describing the structure of visual

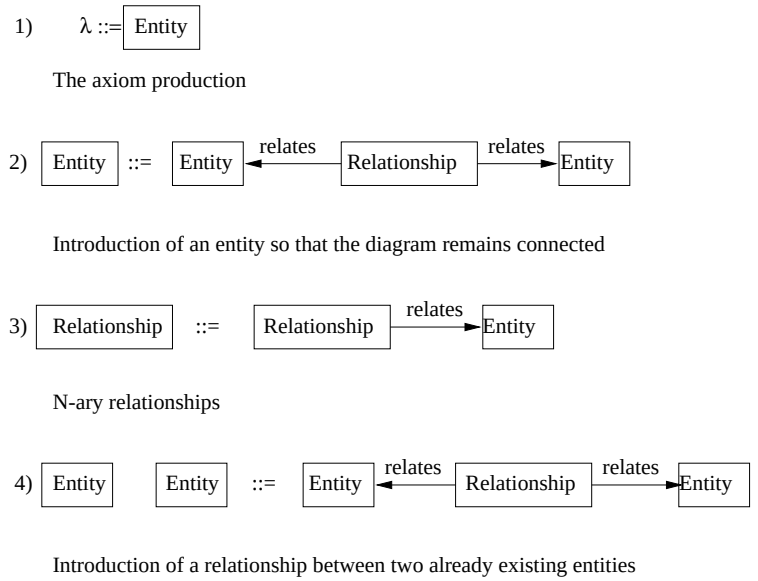


Figure 8.4: A part of the graph grammar for the ASG of E-R diagrams (from [RS97]).

sentences and for describing all kinds of operations (graph transformations) on the sentences. In [BS99] Blostein and Schürr discuss in more detail the issues involved in (visual) programming with graph transformations. The strong point of graph grammars is that they can have context-sensitive grammar productions. In unrestricted context-sensitive productions the left hand side is a subgraph instead of just a single nonterminal. When the production is applied, the left hand side graph is replaced with the graph on the right hand side. In order to support syntax-directed graph transformations, also deletions can be modelled with productions where the right hand side contains less graph elements than the left hand side. Figure 8.4 shows a graph grammar fragment of the ASG for Entity-Relationship (ER) diagrams.

The complexity of graph parsing arises from the context-sensitivity of productions and the need to perform graph matching. Another technical issue is the embedding of the right hand side production elements into the context (surrounding graph) of its application. In [RS97] the authors discuss the class of layered graph grammars that allow for context sensitive productions but restrict the right hand side of a production to be lexicographically smaller than the left hand side. The lexicographic order of graphs is based on the decomposition of node and edge labels into a set of layers. Further, the right hand side graphs must be connected and they must add new graph elements when applied. The authors also present a parsing algorithm for this class of graph grammars. The authors claim that layered graph grammars are expressive enough and that the related parsing algorithm is efficient enough to be practical.

The PROGRES tool [SWZ95] employs the graph grammar approach described above to deliver a graph grammar engineering environment. PROGRES is a visual language and a tool where users can edit and execute graph grammar productions. The idea of the PROGRES language is to support the design of graph structures and the implementation of graph manipulation tools. The PROGRES tool provides a standard editor environment. It is not clear, whether the editor can be customized to support application specific graphics.

DiaGen

Various versions of the DiaGen system for generating editors for diagramming tools are decribed by Minas and Viehataedt in [MV95], by Minas in [Min97], and by Hoffmann and Minas in [HM00]. In their approach, diagrams are internally represented by hypergraphs. A visual language, which they call a diagram class, is specified by a hypergraph language and a mapping from hypergraphs to their visual representation as diagrams. The hypergraph language is specified by a context-free hypergraph grammar. Special hypergraph transformations can be specified to cater for context-sensitive properties of diagrams.

The nodes and edges of hypergraphs have attributes and the productions of a hypergraph grammar are adorned with constraints on the attributes. The constraints direct the layout of a diagram derived by applying the production. A constraint solver is employed to provide automatic layout of diagrams where the user can adjust layout.

A diagramming tool derived with DiaGen maintains the hypergraph presentation of a diagram during editing. The specification of the diagram language can be augmented with transformation rules which make it possible to provide syntax-directed manipulation of diagrams. The transformation specifications define editing actions that transform a diagram from one valid state to another.

In the version of DiaGen described in [MV95], the use of graph transformations was the only way to provide support for non-syntax-directed editing actions. As noted in [Min97] the transformation rules could make up 90% of the grammar specification and the rules could still miss some frequently used transformations. As a solution to this problem, the version of the system described in [Min97] has been extended to support free-order drawing tool behavior by a employing a hypergraph parser that can distinguish between correct and incorrect parts of diagrams. Now, specifications of complex diagram transformations can be omitted. Some basic transformations can still be included for convenience.

Figure 8.5 shows the productions of a hypergraph grammar for Nassi-Schneiderman diagrams. Each edge of a hypergraph has a type (label) and a number of connection points that determine how many nodes the edges *visit*. The nodes stand for points (in the plane) and the hyperedges represent diagram elements. The nodes that a hyperedge visit determine the position of the diagram element represented by the hyperedge.

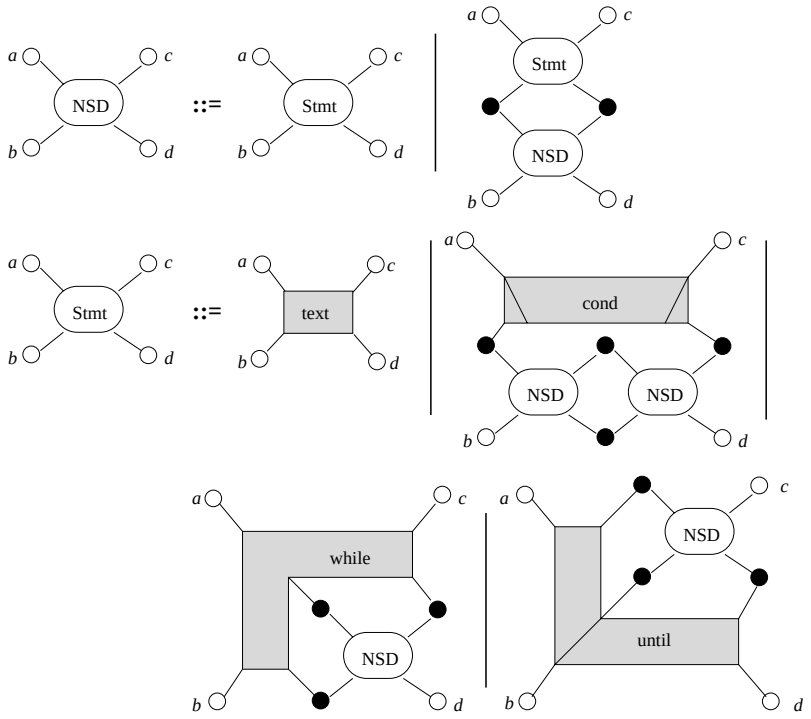


Figure 8.5: A hypergraph grammar for Nassi-Schneiderman diagrams (from [Min97]).

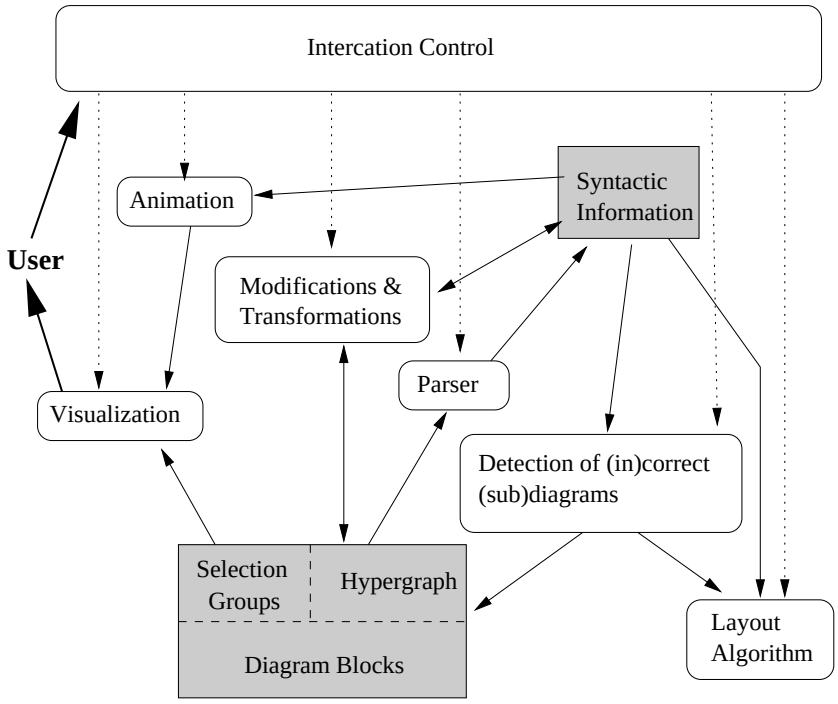


Figure 8.6: The DiaGen system (from [Min97]).

In Figure 8.5 nonterminal hyperedges are depicted as ovals and terminal hyperedges as gray polygons. The nodes are labelled to show how the hypergraph on the right hand side of a production is embedded into the graph on the left hand side. The productions are context-free since the left hand side of every production consists of a single nonterminal and the right hand sides does not contain any other nodes except those already present in the left hand side graph and the nodes added by the production.

With the hypergraph grammar of a diagram class (a visual language), the hypergraph parser employed by DiaGen constructs a representation of the syntax of the language. Using this information, the parser searches from an input diagram for maximal subgraphs that are syntactically correct and creates syntax trees for the subgraphs. The parser can work also incrementally by building the syntax trees while the user is editing the input diagram. Figure 8.6 shows the overall view of DiaGen.

The implementation of a diagram type (a visual language) and the corresponding editor in DiaGen requires four different specifications [Dia00]:

1. the visual appearance of the diagram, i.e. the visible diagram components and the spatial relations between them that are important,

2. the logical diagram structure, which is described by hypergraph transformation rules and a hypergraph grammar,

3. constraints on the diagram layout that help to maintain this structure, and

4. syntax-directed editing operations (similar to macros) that provide a way to implement frequently needed complex manipulations of the diagram.

The hypergraph parser used in DiaGen is based on the CYK algorithm [You67] for parsing context-free string grammars. The complexity of the hypergraph parser is not given, but some concrete time figures are given for parsing context-free input (the complexity of the CYK algorithm is $O(n^3)$). In [Min97] Minas describes an extension to the hypergraph grammar model that allows for restricted use of context-sensitive elements in the right hand sides of productions. This makes it possible to model general graph structures. The impact of this extension to the parsing algorithm is not reported.

In [HM00] Hoffman and Minas discuss recent extensions to the DiaGen approach by elaborating on the relationship of the visual representation of diagram syntax and semantics. Following the three-layer approach described above in the case of PROGRES, they separate between an explicit description of the graphical syntax of a diagram and the abstract syntax in terms of semantic constructs of the language.

In the model, scanning creates a spatial relationship graph that captures the lexical structure of diagrams. As before, the edges of the SRG represent diagram components and the nodes represent the attachment areas that provide the connections between components. However, connected attachment areas are now represented by spatial relationship edges that can denote any (semantically) important spatial relation. Previously, there was no explicit notion of spatial relationships like inclusion or touching.

The scanning process also entails a reduction phase where subgraphs representing graphical relationships are reduced to more simple terminal edges that denote a syntactic relationship between diagram components. Then, the hypergraph parser processes the reduced graph (called a hypergraph model, HGM) and produces an abstract syntax graph (ASG) that gives an even higher level representation of the diagram.

The authors claim that context-free and restricted context-sensitive hypergraph grammars are suitable for modeling any kind of diagrammatic visual language. The examples they provide include strucuted flowcharts, Nassi-Schneiderman diagrams, Petri-nets, message sequence charts etc.

Comparison with VILPERT

It is difficult to draw definitive conclusions about the relative expressive power of different grammatical models. However, the notion of iterative and optional

right-hand side elements that we have introduced to atomic relational grammars are significant extensions to the original ARG formalism [Wit96]. These features make it possible to write concise grammars that reflect naturally the graphical structure of diagramming languages. Of course, the context-free nature of EARGs limit what kind of rules can be expressed in the syntactic specification of the language. However, the remote references that we have introduced make it possible to express constraints on the immediate lexical context of nonterminal instances.

The problem with graph grammar -based approaches is that all the syntactic constructs of a visual language must be expressed in terms of graphs and graph transformations. That is, all syntactic relations between the symbols of a language must be reduced to edges between the symbols. This can lead to unnatural representations for non-graph like properties of visual languages. We feel that relational grammars provide a more flexible and more general grammatical model for specifying the syntax of visual languages.

The central goal of our work was to make the implementations of visual languages open for modifications, extensions, and reuse. Therefore, we chose the object-oriented framework-based approach for VILPERT. The other approaches described above do not provide this kind of openness. Also, our work demonstrates that with careful design, it is possible to retain the benefits of the meta-compiler approach; that is, automatic checking of grammar specifications and the automatic generation of parsers.

The main novel contribution of our work is the handling of syntax errors that makes the edit-compile style of user interaction feasible. We will discuss this in more depth below in Section 8.2.

8.1.2 Object-Oriented Language Engineering

Visual Languages

There exists a few object-oriented frameworks for the implementation of graphical editors [Jin90, VL90, Bra95]. See Section 6.2 for an overview of the HotDraw frameworks.

In these frameworks, the syntax and semantics of a visual language are defined operationally. That is, the graphical objects that are manipulated by the editor have also semantic attributes and operations. The semantic attributes are used to store user data that define part of the the meaning of a drawing. Typically, the frameworks employ some form of constraints over graphical and semantic attributes of objects for specifying either layout rules or rules about the values of the semantic attributes. In many cases, complex rules must be coded by hand as checks that are executed whenever a drawing is modified. So, the editors derived from such frameworks are syntax-directed in the sense that a drawing must always obey the rules of the language.

The Vampire system by McIntyre [McI95] employs a framework for developing visual programming languages based on transformation rules on iconic graphical

objects. Again, there is no notion of grammar which is noted as a deficiency by the author.

Our work extends the general JHotDraw editor framework with a rigorous specification technique for the syntax of visual languages based on extended atomic relational grammars. This means that most of the rules of the language can be expressed with declarative and concise expressions as a grammar instead of hand-coded methods that are part of the implementation of the editor. The grammar-based approach also enforces the practice of separating the graphical objects (the drawing) from the semantic objects (the meaning). Furthermore, the generic technique of handling syntax errors that is part of the parser helps in automatically creating error messages based on the grammar of the language. Also, the editors derived from *Vilpert* need not be syntax-directed.

We have also extended JHotDraw with hierarchical composite figures that facilitate the creation of nested figure containers (e.g. UML packages). See the discussion of the *Draw* package in Section 6.4 for more details.

Textual Languages

When compared to the object-oriented system TaLE for developing processors for textual languages by Järnvall *et al.* [JKN95] and to the approach for extensible language processors based on meta-language and delegating compiler objects (DCO) by Bosch [Bos96], our framework does not concentrate on modelling language-independent concepts as separate classes. Instead, our approach is closer to the meta-language-based approaches for generating language processors from a grammar specification.

However, the *Relap* framework of VILPERT provides the flexibility needed for incremental language development even if at a more coarse-grained level than TaLE or DCO. One reason for this is that in the case of EARGs, there are more elements in the grammar productions than in textual languages and, hence, more interrelationships and restrictions within and between productions. So, in our framework, reuse is confined rather within a language family than between languages of different ancestry.

8.1.3 Meta-Modeling Approach

MetaEdit

MetaEdit+ [Met01] is a tool for creating CASE tools that comprise a domain specific visual language. To specify a language, the user specifies (with the method workbench toolset) the concepts of the domain, the rules for using and composing the concepts, the graphical notation that corresponds to the concepts, and a set of generators (specified in a scripting language) that transform models into some external format (code, documentation, data dictionary, etc.). The goal is to create

a complete CASE tool tailored for a specific domain and a specific development process. So, the scope of the MetaEdit approach is broader than in VILPERT and therefore, we concentrate here only on those aspects of the approach that are related to our work.

The heart of the approach is the creation of a metamodel that specifies the domain specific languages. The elements of the GOPRR metamodeling language are graph, object, property, relationship, and role. A model is a graph consisting of objects with properties and relationhips to other objects. Objects may have roles in the relationships that they participate in. The method workbench of MetaEdit+ provides tools for specifying all these elements of a domain specific language. The system then derives from the specification a set of (syntactic) well-formedness rules that the specifier can tailor (choose which rules to include in the final method). The specifier can also create more complex (e.g. semantic) checks on the models using the scripting language provided by the tool. The specification of the generators for the target language is also based on writing processors in the scripting language for the models created with the target language.

MetaEdit+ can support only those kind of languages that can be expressed using the GOPRR metamodeling language. Basically, this means graphs of objects. For example, the current version of the tool cannot support UML message sequence charts because of the limitations in the underlying metamodeling language that does not distinguish any kind of physical ordering of the relations of objects. Our approach is more general because with VILPERT the language designer can specify also other kind of languages. Furthermore, the resulting editor of a visual language need not be syntax-directed and dialog-based as with MetaEdit+. Of course, MetaEdit+ provides strong support for the creation of the kind of languages that can be specified with it.

8.2 Error Handling in Visual Languages

According to the survey by Marriott & al. [MMW98, pp. 64-67], error handling in visual language parsing is a mostly unexplored area. However, it is an essential part of edit-compile style of visual programming that facilitates free-order editing of visual programs.

Our work concentrates on error handling in off-line parsing of visual languages specified by atomic relational grammars (relational laguages). Off-line parsing implies an edit-compile style of visual programming where the syntax of a visual program is checked after editing the program. We are not aware of any other work on error handling in parsing relational languages. On the other hand, error handling strategies have been developed for visual languages specified by other formalisms.

In the following, we first survey the work on error handling in incremental on-line approaches to visual language parsing. Then, we look at off-line parsing techniques.

Incremental On-line Parsing

The importance of free-order editing in diagramming tools has been recognized by several authors. Most of the free-order editing approaches presented in the literature apply incremental *on-line* recognition (parsing) of visual programs (diagrams). That is, the syntactic and semantic validity of a visual program is constantly checked while the user is editing the program with a graphical tool. This makes it possible to give the user immediate feedback about the validity of a visual program during editing.

In [Ser95], Serrano presents an approach where the semantics of a diagramming notation is defined by constraints over the visual objects constituting the diagrams. A diagram is defined to be in a valid, inconsistent, or wrong state depending on which constraints are satisfied. Constraint satisfaction is continuously tested when the user is editing a diagram. By allowing a diagram to be in an inconsistent state during editing, a degree of editing freedom is provided. The freedom is not total because editing actions that would lead to a wrong state are not allowed.

Minas and Viehstaedt [MV95, Min97] suggest incremental on-line parsing of visual languages with a possibility to perform error correction. In the DiaGen framework for implementing visual languages, the syntax of a visual language is specified by a context-free hypergraph grammar. An incremental hypergraph parser is used to analyze visual programs and the parser has the ability to identify correct and incorrect subgraphs. Incorrect subgraphs can then be highlighted by an editor to provide feedback to the user. However, the parser is only able to indicate a part of the input that is incorrect but cannot provide any feedback about what is wrong in the incorrect input. In contrast, in our approach, the parser is able to produce error messages based on the grammar of the visual language in order to provide more useful feedback.

The most advanced error handling technique in incremental on-line recognition of visual languages has been presented by Chok and Marriot [CM95, CM98]. As part of the Penguins system, they have developed an error correction technique for an incremental parser of constraint multiset grammars. In a parser generated for a visual language by Penguins, the error handling mechanism of the parser automatically corrects geometric errors in the input that the user is editing. The error correction mechanism is based on the concept of the geometric distance between sentences. A sentence is a set of tokens which have geometric and semantic attributes. The distance between two sentences can be computed by considering each mapping between the two token sets and summing the distances between the tokens (the difference of the values of their geometric attributes) of each pair of the mapping. The smallest sum is the geometric distance between the sentences. By computing the geometrically closest sentence that belongs to the language, an incorrect sentence can be automatically corrected by changing attribute values of the tokens. The error correction mechanims uses heuristics to quickly find sentences that are reasonably close to the incorrect input to meet the preformance requirements of on-line parsing. The error correction seems to be limited to the geometric attributes of graphical tokens.

It is not feasible to try to adapt the incremental techniques described above to off-line parsing of relational languages, since the specification formalisms, the representation of the input to the parser, and parsing algorithms are different. Also, the scope and the limitations of the proposed error handling techniques are not clear. The most obvious difference between our approach and the other approaches is static parsing and the attempt to find as many errors in the input as possible.

Off-line Parsing

Wittenburg's parsing algorithm for atomic relational languages is an extension of Earley's general parser for context-free string languages. Others have also extended string language parsing and grammars to visual languages but have not considered error handling mechanims. For instance, Costagliola & al. [CLOT97b] describe the VLCC system that extends LR parsing to visual languages. The *pLR* parser of VLCC halts at the first syntax error reporting a general 'parse error' and does not recover. The paper gives heuristics for solving LR parsing conflicts caused by ambiguous grammar rules and ambiguous input. The heuristics help the parser to choose between more than one possible input object to be scanned next or between the possible parse actions to be taken in the current parse configuration. These heuristics enlarge the set of parsable visual languages but seem not to help in the handling of erroneous input.

In [RS96] Rekers and Schürr discuss the possibility of off-line parsing of languages specified by graph grammars. The idea is to support free-order editing of visual programs. However, they do not address error handling issues. In [Sch97] Schürr states that some kind of error correction is possible "on the fly" when interpreting the *textual* version of the PROGRES graph language.

The general ideas of error handling in parsing textual languages (e.g. [SSS90, Chap. 9] and [WM80]) can be adapted to relational languages. The main problem in this is that the sentences of relational languages lack a predetermined linear scanning order of input. On the other hand, as our work shows, the graph-like form of the input provides new opportunities for error recovery.

8.3 Source-to-Source Translation of Visual Languages

Systematic techniques have been developed for source-to-source translation of textual languages (see [AU71], [LMW88], [Yel88]), whereas in the context of visual languages, source-to-source translation is an unexplored area and we are not aware of any other formal work in the area. This is a reflection of the immature nature of visual language processing in general: no commonly accepted specification methods or grammatical models have been developed for visual languages, which implies that the processing (parsing, analysis, translation) of visual languages is quite hard to automate with current technology.

While there is a lack of solid methods and tools for the transformation *between* two different visual languages, some kind of transformations are common *within* the same language, making it possible to automatically tune a diagram flexibly into another form in a dedicated editor. For instance, DiaGen (see p. 160 above) provides a number of diagram modifications, such as automatically transforming a *while*-loop into an *until*-loop in a Nassi-Schneiderman chart. However, DiaGen does not support transforming the chart into a totally different notation, such as a flowchart.

Source-to-source techniques can as well be applied for the transformation between other kinds of structured information. For instance, many recent document transformation systems are grammar-based and syntax-directed [CK95] [KP93] [LTV96] [MOB94]. While the idea behind syntax-directed (tree) transformation is quite universal, the formalisms and techniques originally developed for textual programming languages usually do not directly apply in other contexts but must be adapted. For instance, the inherent unambiguous and deterministic nature of textual programming languages has to be relaxed in most other areas, also in the processing of visual languages due to the lack of a unique “order” of symbols in visual programs.

Chapter 9

Conclusions

In this thesis, we have studied the problem of specifying and implementing visual languages. We have analyzed an existing grammatical model for specifying and parsing visual languages and presented an extended version of the grammar formalism. Extended atomic relational grammars provide better support than the original formalism for specifying graph-like visual languages. Our changes to the formalism and to the parser make parsing deterministic that enables effective handling of syntax errors.

Our work shows that it is possible to develop a practical error handling scheme for the parsing of relational languages. We have introduced the notion of parser-defined syntax errors, presented two error recovery techniques and showed how error detection and recovery can be incorporated to the parser for extended atomic relational grammars.

We have also presented a formal grammatical model for the source-to-source translation of visual languages. The model is based on the transformation between parse trees that are spanned by atomic relational grammars for the languages involved. The transformation is formally specified with a mapping between the grammars.

We have implemented extended atomic relational grammars and the error handling scheme as part of the VILPERT system that is the product of the constructive part of our research. VILPERT combines a formal grammar for the underlying language and a graphical editor framework into an object-oriented framework for implementing visual languages.

We have validated our solution by implementing three well-known visual languages that represent typical notations used in software engineering (UML structural diagrams, UML statecharts, and flowcharts) and other small experimental languages. The implementations of the visual languages show a high degree of reuse: the language (application) specific parts of the implementations comprise less than 20% of the total size of the applications.

The syntaxes of the languages have been specified by extended atomic relational grammars using the grammar framework of VILPERT and the editors for the lan-

guages have been derived from the editor framework of VILPERT. The editors allow syntax-free editing of visual programs. A visual program (diagram) is analyzed by a parser automatically produced from the grammar of the language. The language analyzer is able to form meaningful error messages about syntax errors in the input, and it can recover from syntax errors to a certain extent to continue parsing. From a correct input, the analyzer produces a parse tree that can be used by post-parse processors to perform further analysis and transformations to the program. The main benefit of our approach is a rigorous implementation methodology that does not compromise the usability of the resulting tools.

An interesting future direction is to investigate the theory and the mechanisms of developing domain specific error handlers for particular kinds of visual languages. They should try to recognize 'typical errors' and automatically perform error correction. This would require suitable abstractions to be developed for the specialization of the error detection and recovery part of the parser. The explicit parse graph and the 'openness' of the interpretive parsing framework seem to provide good opportunities for implementing more specialized techniques. Also, benchmarking input sets should be developed for comparing the relative effectiveness of different error recovery techniques.

Besides the development of more effective error recovery techniques and the development of architectural support for domain specific error handling, an important issue to be addressed in future work is the further validation of the (graphical) interaction of error handling by empirical studies. Only practice will tell the user perceived effectiveness of the techniques.

Implementing the source-to-source translation scheme as an independent part of VILPERT is an obvious extension to the framework. Also, it would be interesting to study the usage of the edit-compile style of visual specification (or programming) in a tool that maintains an editable semantic representation (model) in addition to the graphical representation (diagrams) of the specification. In this kind of tool, the user can manipulate the model directly through a model browser or indirectly by editing the graphical representation. Here, the challenge would be maintaining the consistency between the two representations while allowing the free editing of both representations.

Finally, we could develop a 'pure' metacompiler interface for specifying EARG grammars. That is, we could define a metalanguage for writing grammar specifications. Grammar specifications could then be translated into Java class specifications in a very direct manner. The resulting class could be automatically compiled into Java byte code, and the grammar checking facilities of the *Relap* part of VILPERT could be used to analyze the grammar. Already, the grammar checking operations issue error messages (as Java exceptions) that include a textual transcription of the erroneous expression. Also, the metalanguage should support the extension of grammars through inheritance like the current implementation of EARG grammars in VILPERT. The metacompiler approach would not help much in reducing the programming work when implementing a visual language because the grammars are usually short.

Bibliography

[AK94] A. L. Ambler and T. D. Kimura, editors. *Proceedings of the IEEE Symposium on Visual Languages*, St. Louis, Missouri, 1994. IEEE Computer Society Press.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, Reading (MA), USA, 1986.

[AU71] A. V. Aho and J. D. Ullman. Translations of context-free grammars. *Information and Control*, 19:439–475, 1971.

[BCLM98] P. Bottoni, M. F. Costabile, S. Levialdi, and P. Mussio. Specification of visual languages as means for interaction. In Marriott and Meyer [MM98b], chapter 13, pages 353—375.

[Bos96] J. Bosch. Tool support for language extensibility. In L. Bendix, K. Nørmark, and K. Østerbye, editors, *NWERP'96 Nordic Workshop on Programming Environment Research*, pages 3—17, Aalborg, Denmark, 1996.

[Bra95] J. M. Brant. Hotdraw. Master's thesis, University of Illinois at Urbana Champaign, 1995.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[Bro87] F. P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10—19, April 1987.

[BS99] D. Blostein and A. Schürr. Computing with graphs and graph transformations. *Software–Practice and Experience*, 29(3):197—217, 1999.

[CBL+99] S. K. Chang, M. M. Burnett, S. Levialdi, K. Marriott, J. J. Pfeiffer, and S. L. Tanimoto. The future of visual languages. In Proceedings of 1999 IEEE Symposium on Visual Languages [Pro99].

[CDLO94] G. Costagliola, A. De Lucia, and S. Orefice. Towards efficient parsing of diagrammatic languages. In T. Catarci, M. F. Costabile, S. Levialdi, and G. Santucci, editors, *Proceedings of the Workshop on Advanced Visual Interfaces, AVI'94*, pages 162—171, Bari, Italy, 1994. ACM Press.

[CK95] K. Chiba and M. Kyojima. Document transformation based on syntax-directed tree translation. *Electronic Publishing – Origination, Dissemination and Design*, 8(1):15—29, 1995.

[CLL99] P. Coad, E. Lefebyre, and J. De Luca. *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice-Hall, 1999.

[CLOT97a] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A framework of syntactic models for the implementation of visual languages. In Storms [Sto97], pages 58—65.

[CLOT97b] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12):777—799, 1997.

[CLOT98] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. Positional grammars: A formalism for lr-like parsing of visual languages. In Marriott and Meyer [MM98b], chapter 5, pages 171—191.

[CM95] S. S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In Haarslev [Haa95], pages 242—249.

[CM98] S. S. Chok and K. Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of the ACM Symposium on User Interface Software and Technology UIST'98*, pages 185—194, San Francisco, California, 1998. ACM Press.

[CMP99] S. S. Chok, K. Marriott, and T. Paton. Constraint-based diagram beautification. In Proceedings of 1999 IEEE Symposium on Visual Languages [Pro99], pages 12—19.

[CTOL95] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia. Automatic generation of visual programming environments. *Computer*, 28(3):56—66, 1995.

[DBT88] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61:175—198, 1988.

[Dia00] DiaGen. http://www2.informatik.uni-erlangen.de:80/IMMD-II/Research/Activities/DiaGen/index.html, 2000.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94—102, February 1970.

[FSJ99a] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. Application frameworks. In M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks, Object-Oriented Foundations of Framework Design*, chapter 1, pages 3—27. Wiley, 1999.

[FSJ99b] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks, Object-Oriented Foundations of Framework Design*. Wiley, 1999.

[GE96] E. Gamma and T. Eggenwailer. JHotDraw Java–framework. members.pingnet.ch/gamma/JHD-5.1.zip, 1996. Copyright by IFA Informatik and E. Gamma, 1996, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[GHLP95] K. Granö, J. Harju, T. Larikka, and J. Paakki. Object-oriented protocol design and reuse in Kannel. In *Proceedings of the 21st Euromicro Conference on Design of Hardware/Software Systems*, pages 465—472, Como, Italy, 1995. IEEE Computer Society Press.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214—230, 1993.

[Gol91] E. J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, Dept. of Computer Science, 1991.

[Haa95] V. Haarslev, editor. *Proceedings of the 11th IEEE International Symposium on Visual Languages*, Darmstadt, Germany, 1995. IEEE Computer Society Press.

[HM00] B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. Workshop on Graph Transformation and Visual Modelling Techniques, July 15/16, Genova, Switzerland. In J. D. P. Rolim et al., editor, *ICALP Workshops 2000, Proceedings in Informatics 8*, pages 443—450. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[Jär92] T. Järvinen. Implementing a visual language. Master's thesis, (in Finnish) Comp. Sci. University of Helsinki, 1992. Report C-1992-58.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):23—35, June 1988.

[JH98] S. Jarzabek and R. Huang. The case for user-centered CASE tools. *Communications of the ACM*, 41(8):93—99, 1998.

[jho00] JHotDraw as an open source project. www.jhotdraw.org, 2000.

[Jin90] W. A. Jindrich, Jr. Foible: A framework for visual programming languages. Master's thesis, University of Illinois at Urbana Champaign, 1990.

[JKN95] E. Järnvall, K. Koskimies, and M. Niittymäki. Object-oriented language engineering with TaLE. *Object Oriented Systems*, 2(2):77—98, 1995.

[Joh92] R. E. Johnson. Documenting frameworks using patterns. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Systems, Languages, and Applications, OOPSLA'92*, pages 63—76, Vancouver, Canada, 1992. ACM Press.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127—145, 1968. Correction in *Mathematical Systems Theory* 5(1): 95—96, 1971.

[KP93] E. Kuikka and M. Penttonen. Transformation of structured documents with the use of grammar. *Electronic Publishing – Origination, Dissemination and Design*, 6(4):373—383, 1993.

[KPPM84] S. E. Keller, J. A. Perkins, T. F. Payton, and S. P. Mardinly. Tree transformation techniques and experiences. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, 1984. *ACM SIGPLAN Notices* 19(6):190-201.

[LC98] D. Lending and N. L. Chervany. The use of CASE tools. In R. Agrawal, editor, *Proceedings of the 1998 ACM SIGCPR Conference*, pages 49—58, Boston, Massachusetts, USA, 1998. ACM Press.

[Lin97] G. Lindén. *Structured document transformations*. PhD thesis, Department of Computer Science, University of Helsinki, 1997. Report A-1997-2.

[LMW88] P. Lipps, U. Möncke, and R. Wilhelm. OPTRAN – A language/system for the specification of program transformations: System overview and experiences. In D. Hammer, editor, *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *LNCS*, pages 52—65, Berlin, 1988. Springer-Verlag.

[LTV96] G. Lindén, H. Tirri, and A. I. Verkamo. ALCHEMIST: A general purpose transformation generator. *Software – Practice and Experience*, 26(6):653—675, 1996.

[Mar94] K. Marriott. Constraint multiset grammars. In Ambler and Kimura [AK94], pages 118—125.

[Mat99] M. Mattsson. Effort distribution in a six year industrial application framework project. In *Proceedings of the International Conference on Software Maintenance ICSM'99*, pages 326—333, Oxford, UK, 1999. IEEE Computer Society Press.

[Mat00] M. Mattsson. *Evolution and Composition of Object-Oriented Frameworks*. PhD thesis, University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, 2000.

[McG99] J. D. McGregor. Making diagrams useful, not archival. *Journal of Object-Oriented Programming*, pages 24—28, May 1999.

[McI95] D. W. McIntyre. Design and implemetation with Vampire. In M. M. Burnett, A. Goldberg, and T. G. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, pages 129—159. Manning Publications Co., Greenwich, 1995.

[Met01] MetaCase Consulting. Metaedit+. www.metacase.com, 2001.

[MI99] J. Maansaari and J. Iivari. The evolution of CASE usage in Finland between 1993 and 1996. *Information & Management*, 36:37—53, 1999.

[Min97] M. Minas. Diagram editing with hypergraph parser support. In Storms [Sto97], pages 226—233.

[MM98a] K. Marriott and B. Meyer. The CCMG visual language hierarchy. In *Visual Language Theory* [MM98b], chapter 4, pages 129—169.

[MM98b] K. Marriott and B. Meyer, editors. *Visual language theory*. Springer-Verlag, 1998.

[MMW98] K. Marriott, B. Meyer, and K. Wittenburg. A survey of visual language specification and recognition. In Marriott and Meyer [MM98b], chapter 2, pages 5—85.

[MOB94] S. A. Mamrak, C. S. O'Connell, and J. Barnes. *Integrated Chameleon Architecture*. Prentice Hall, 1994.

[MS99] M. Münch and A. Schürr. Leaving the visual language ghetto. In Proceedings of 1999 IEEE Symposium on Visual Languages [Pro99], pages 148—155.

[MV95] M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In Haarslev [Haa95], pages 203—210.

[Nat99] National Instruments, Inc. LabVIEW. www.ni.com/labview, 1999.

[NH98] N. H. Narayanan and R. Hübscher. Towards a human-computer interaction perspective. In Marriott and Meyer [MM98b], chapter 3, pages 87—128.

[Nic94] J. V. Nickerson. Visual programming: Limits of graphic representation. In Ambler and Kimura [AK94], pages 178—179.

[Obj99] Object Management Group. OMG Unified Modeling Language specification v. 1.3, June 1999.

[PC98] D. C. C. Poo and M. K. Chung. CASE and software maintenance parctices in Singapore. *Journal of Systems and Software*, 44:97—105, 1998.

[Pro99] *Proceedings of 1999 IEEE Symposium on Visual Languages*, Tokyo, Japan, 1999. IEEE Computer Society.

[PSTS91] L. B. Protsko, P. G. Sorenson, J. P. Tremblay, and D. A. Schaefer. Towards the automatic generation of software diagrams. *IEEE Transactions on Software Engineering*, 17(1):10—21, 1991.

[PT98] J. Paakki and A.-P. Tuovinen. Source-to-source translation of visual languages. *Nordic Journal of Computing*, 5(3):235—264, 1998.

[Ray91] D. R. Raymond. Characterizing visual languages. In L. O'Conner, editor, *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 176—182, Kobe, Japan, 1991. IEEE Computer Society Press.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RJ97] D. Roberts and R. Johnson. Evolving frameworks. In R. C. Martin, D. Riehle, F. Buschmann, and J. Vlissides, editors, *Pattern languages of program design 3*, Software Patterns Series, chapter 26. Addison-Wesley, 1997.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999.

[Rol00] RoleModel Software. Drawlets. www.rolemodelsoft.com/aboutUs/drawlets.htm, 2000.

[RS96] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In R. S. Sipple, editor, *Proceedings of 1996 IEEE Symposium on Visual Languages*, pages 148—155, Boulder, Colorado, 1996. IEEE Computer Society Press.

[RS97] 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.

[Sch97] A. Schürr. Developing graphical (software engineering) tools with PROGRES. In A. Schäfer and P. Botella, editors, *Proceedings of the 1997 (19th) International Conference on Software Engineering (ICSE'97)*, pages 618—619. IEEE Computer Society Press, 1997.

[Ser95] J. A. Serrano. The use of semantic constraints on diagram editors. In Haarslev [Haa95], pages 211—216.

[Shi84] Q. Y. Shi. Semantic-syntax-directed translation and its application to image processing. *Information Sciences*, 32(1):75—90, 1984.

[SSS90] S. Sippu and E. Soisalon-Soininen. LR(k) and LL(k) parsing. In *Parsing Theory*, volume 2. Springer-Verlag, 1990.

[Sto97] P. Storms, editor. *1997 IEEE Symposium on Visual Languages*, Isle of Capri, Italy, 1997. IEEE Computer Society.

[SWZ95] A. Schürr, A. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In A. Schäfer and P. Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 219—234. Springer Verlag, LNCS 989, 1995.

[Tuo98a] A.-P. Tuovinen. Error recovery in parsing relational languages. In K. Kelly, editor, *Proceedings of 1998 IEEE Symposium on Visual Languages*, pages 6—13, Halifax, Nova Scotia, Canada, 1998. IEEE Computer Society.

[Tuo98b] A.-P. Tuovinen. A framework for processors of visual languages. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, volume 1357 of *LNCS*, pages 119—122. Springer-Verlag, 1998.

[Tuo99] A.-P. Tuovinen. Vilpert: Visual language expert. In J. Penjam, editor, *Proceedings of the Sixth Fenno-Ugric Symposium on Software Technology FUSST'99*, Tallinn, Estonia, Aug. 19–21, 1999.

[Tuo00] A.-P. Tuovinen. Practical error handling in parsing relational languages. *Journal of Visual Languages and Computing*, 11(5):505—528, October 2000.

[TVC94] M. Tucci, G. Vitiello, and G. Costagliola. Parsing nonlinear languages. *IEEE Transactions on Software Engineering*, 20(9):720—739, September 1994.

[Vis99] Visio Inc. Visio professional. `www.visio.com`, 1999.

[VL90] J. M. Vlissides and M. A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237—268, July 1990.

[Whi97] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109—142, 1997.

[Wit92] K. Wittenburg. Earley-style parsing for relational grammars. In C. Harris, editor, *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 192—199, Seattle, Washington, 1992. IEEE Computer Society Press.

[Wit95] K. Wittenburg. Visual language parsing: If I had a hammer... In *Proceedings of the International Conference on Cooperative Multimodal Communication, Theory and Applications CMC'95, Eindhoven, Netherlands*, pages 17—33, 1995.

[Wit96] K. Wittenburg. Predictive parsing for unordered relational languages. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, volume 1 of *Text, Speech and Language Technology*, chapter 20, pages 385—407. Kluwer Academic Publishers, 1996.

[WM80] J. Welsh and M. McKeag. *Structured System Programming*. Prentice-Hall International, 1980.

[WW98] K. Wittenburg and L. Weitzman. Relational grammars: Theory and practice in a visual language interface for process modeling. In Marriott and Meyer [MM98b], chapter 6, pages 193—217.

[Yel88] D. M. Yellin. *Attribute Grammar Inversion and Source-to-Source Translation*, volume 302 of *LNCS*. Springer-Verlag, 1988.

[You67] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189—208, 1967.

Appendix A

Statechart Grammar

```
package CH.ifa.draw.samples.statechart;

import relap.LanguageModel.*;
import java.io.*;
import com.objectspace.jgl.*;

public class StateChart extends RelationalGrammarImplementation {

HashSet fFixedStartAttrs;

public String terminalDeclarations () {
return "rrect text arrow initial final statePanel namePanel "+
       "itPanel pseudoPanel labelPanel";
}

public String nonTerminalDeclarations () {

return "StateChart Initial Final State StateSymbol Trans "+
     "NameCompartment StateCompartment ITCompartment Label";

}
public String startSymbolDeclaration () {

return "StateChart";

}

public String relationDeclarations () {

return "inside enters exits attached ";

}
```

```
public String attributeDeclarations () {

return "root ";

}

public HashSet fixedStartAttributes() {
if (fFixedStartAttrs == null) {
fFixedStartAttrs = new HashSet();
fFixedStartAttrs.add("root");
}
return fFixedStartAttrs;
}

public GrammarProduction StateChart_(GrammarProduction p)
throws InvalidGrammarException {

p.description("State machine");
p.rightHandSide("pseudoPanel Initial State+ Final*");
p.constraints(
    "inside(2:root,1) inside(3:root,1) inside(4:root,1)");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.StateChartRep");
return p;
}

public GrammarProduction Initial_(GrammarProduction p)
throws InvalidGrammarException {

p.description("initial (pseudo) state");
p.rightHandSide("initial");
p.predicates("1 {arrow} exits(_,1) ~{arrow}enters(_,1)");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.InitialRep");
return p;
}

public GrammarProduction Final_(GrammarProduction p)
throws InvalidGrammarException {

p.description("final (pseudo) state");
p.rightHandSide("final Trans+");
p.constraints("enters(2:root,1)");
p.predicates(
   "1 {rrect,initial} exits(2:root,_) ~{arrow}exits(_,1)");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.FinalRep");
return p;
}
```

```
public GrammarProduction State_(GrammarProduction p)
throws InvalidGrammarException {

p.description("state with zero or more incoming transitions");
p.rightHandSide("StateSymbol Trans*");
p.constraints("enters(2:root,1:root)");
p.predicates("1 {rrect,initial} exits(2:root,_)");
p.assignments("0:root = 1:root");
p.semantics("CH.ifa.draw.samples.statechart.StateRep");
return p;
}


public GrammarProduction StateSymbol_(GrammarProduction p)
throws InvalidGrammarException {

p.description("state symbol structure");
p.rightHandSide(
    "rrect NameCompartment? ITCompartment? StateCompartment*");
p.constraints(
    "inside(2:root,1) inside(3:root,1) inside(4:root,1)");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.StateSymbolRep");
return p;
}


public GrammarProduction NameCompartment_(GrammarProduction p)
throws InvalidGrammarException {

p.description("name compartment");
p.rightHandSide("namePanel text+");
p.constraints("inside(2,1)");
p.order("2 above ");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.NameCompartmentRep");
return p;
}

public GrammarProduction ITCompartment_(GrammarProduction p)
throws InvalidGrammarException {

p.description("internal transition compartment");
p.rightHandSide("itPanel text+");
p.constraints("inside(2,1)");
p.order("2 above ");
p.assignments("0:root = 1");
```

```
p.semantics("CH.ifa.draw.samples.statechart.ITCompartmentRep");
return p;
}

public GrammarProduction
    StateCompartment_empty(GrammarProduction p)
throws InvalidGrammarException {

p.description("empty state compartment");
p.rightHandSide("statePanel");
p.disambiguate("~{rrect,initial,text,final}inside(_,1)");
p.assignments("0:root = 1");
p.semantics(
    "CH.ifa.draw.samples.statechart.StateCompartmentRep");
return p;
}

public GrammarProduction
    StateCompartment_collapsedTexts(GrammarProduction p)
throws InvalidGrammarException {

p.description(
    "state compartment with collapsed text compartments");
p.rightHandSide("statePanel text+");
p.constraints("inside(2,1)");
p.disambiguate("~{rrect,initial,final}inside(_,1)");
p.order("2 above ");
p.assignments("0:root = 1");
p.semantics(
    "CH.ifa.draw.samples.statechart.StateCompartmentRep");
return p;
}

public GrammarProduction
   StateCompartment_nestedDiagram(GrammarProduction p)
throws InvalidGrammarException {

p.description(
"state compartment with nested statechart diagram");
p.rightHandSide("statePanel text* Initial? State+ Final*");
p.constraints(
"inside(2,1) inside(3:root,1) inside(4:root,1) inside(5:root,1)");
p.order("2 above ");
p.assignments("0:root = 1");
p.semantics(
"CH.ifa.draw.samples.statechart.CompositeStateCompartmentRep");
return p;
}
```

```
public GrammarProduction Trans_(GrammarProduction p)
throws InvalidGrammarException {

p.description("transition");
p.rightHandSide("arrow Label?");
p.constraints("attached(2:root,1)");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.TransitionRep");
return p;
}

public GrammarProduction Label_(GrammarProduction p)
throws InvalidGrammarException {

p.description("multi-line transition label");
p.rightHandSide("labelPanel text+");
p.constraints("inside(2,1)");
p.order("2 above");
p.assignments("0:root = 1");
p.semantics("CH.ifa.draw.samples.statechart.LabelRep");
return p;
}

}
```

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 26 (Teollisuuskatu 23)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from: Department of Computer Science, Library (A 214), P.O. Box 26, FIN-00014 University of Helsinki, FINLAND.

A-1989-1 G. Grahne: The problem of incomplete information in relational databases. 156 + 3 pp. (Ph.D. thesis).

A-1989-2 H. Tirri (ed.): Interoperability of heterogeneous information systems: final report of the COST 11^{ter} project. 110 pp.

A-1989-3 J. Tarhio & M. Tienari (eds.): Computer Science at the University of Helsinki. 57 pp.

A-1989-4 T. Alanko, J. Keskinen, P. Kutvonen, M. Mutka, & M. Tienari: The AHTO project: software technology for open distributed processing. 53 + 3 pp.

A-1989-5 N. Holsti: Script editing for recovery and reversal in textual user interfaces. 126 pp. (Ph.D. thesis).

A-1989-6 K.E.E. Raatikainen: Modelling and analysis techniques for capacity planning. 162 + 52 pp. (Ph.D. thesis).

A-1990-1 K. Pohjonen & J. Tarhio (toim./eds.): Tietojenkäsittelyopin laitoksen tutkimusraportteja 1988–89 – Research reports at the Department of Computer Science 1988–89. 27 pp.

A-1990-2 J. Kuittinen, O. Nurmi, S. Sippu & E. Soisalon-Soininen: Efficient implementation of loops in bottom-up evaluation of logic queries. 14 pp.

A-1990-3 J. Tarhio & E. Ukkonen: Approximate Boyer-Moore string matching. 27 pp.

A-1990-4 E. Ukkonen & D. Wood: Approximate string matching with suffix automata. 14 pp.

A-1990-5 T. Kerola: Qsolver – a modular environment for solving queueing network models. 15 pp.

A-1990-6 Ker-I Ko, P. Orponen, U. Schöning & O. Watanabe: Instance complexity. 24 pp.

A-1991-1 J. Paakki: Paradigms for attribute-grammar-based language implementation. 71 + 146 pp. (Ph.D. thesis).

A-1991-2 O. Nurmi & E. Soisalon-Soininen: Uncoupling updating and rebalancing in chromatic binary search trees. 12 pp.

A-1991-3 T. Elomaa & J. Kivinen: Learning decision trees from noisy examples. 15 pp.

A-1991-4 P. Kilpeläinen & H. Mannila: Ordered and unordered tree inclusion. 22 pp.

A-1991-5 A. Valmari: Compositional state space generation. 30 pp.

A-1991-6 J. Tarhio & M. Tienari (eds.): Computer Science at the University of Helsinki 1991. 66 pp.

A-1991-7 P. Jokinen, J. Tarhio & E. Ukkonen: A comparison of approximate string matching algorithms. 23 pp.

A-1992-1 J. Kivinen: Problems in computational learning theory. 27 + 64 pp. (Ph.D. thesis).

A-1992-2 K. Pohjonen & J. Tarhio (toim./eds.): Tietojenkäsittelyopin laitoksen tutkimusraportteja 1990–91 – Research reports at the Department of Computer Science 1990–91. 35 pp.

A-1992-3 Th. Eiter, P. Kilpeläinen & H. Mannila: Recognizing renamable generalized propositional Horn formulas is NP-complete. 11 pp.

A-1992-4 A. Valmari: Alleviating state explosion during verification of behavioural equivalence. 57 pp.

A-1992-5 P. Floréen: Computational complexity problems in neural associative memories. 128 + 8 pp. (Ph.D. thesis).

A-1992-6 P. Kilpeläinen: Tree matching problems with applications to structured text databases. 110 pp. (Ph.D. thesis).

A-1993-1 E. Ukkonen: On-line construction of suffix-trees. 15 pp.

A-1993-2 Alois P. Heinz: Efficient implementation of a neural net α-β-evaluator. 13 pp.

A-1994-1 J. Eloranta: Minimal transition systems with respect to divergence preserving behavioural equivalences. 162 pp. (Ph.D. thesis).

A-1994-2 K. Pohjonen (toim./ed.): Tietojenkäsittelyopin laitoksen julkaisut 1992–93 – Publications from the Department of Computer Science 1992–93. 58 s./pp.

A-1994-3 T. Kujala & M. Tienari (eds.): Computer Science at the University of Helsinki 1993. 95 pp.

A-1994-4 P. Floréen & P. Orponen: Complexity issues in discrete Hopfield networks. 54 pp.

A-1995-1 P. Myllymäki: Mapping Bayesian networks to stochastic neural networks: a foundation for hybrid Bayesian-neural systems. 93 pp. (Ph.D. thesis).

A-1996-1 R. Kaivola: Equivalences, preorders and compositional verification for linear time temporal logic and concurrent systems. 185 pp. (Ph.D. thesis).

A-1996-2 T. Elomaa: Tools and techniques for decision tree learning. 140 pp. (Ph.D. thesis).

A-1996-3 J. Tarhio & M. Tienari (eds.): Computer Science at the University of Helsinki 1996. 89 pp.

A-1996-4 H. Ahonen: Generating grammars for structured documents using grammatical inference methods. 107 pp. (Ph.D. thesis).

A-1996-5 H. Toivonen: Discovery of frequent patterns in large data collections. 116 pp. (Ph.D. thesis).

A-1997-1 H. Tirri: Plausible prediction by Bayesian inference. 158 pp. (Ph.D. thesis).

A-1997-2 G. Lindén: Structured document transformations. 122 pp. (Ph.D. thesis).

A-1997-3 M. Nykänen: Querying string databases with modal logic. 150 pp. (Ph.D. thesis).

A-1997-4 E. Sutinen, J. Tarhio, S.-P. Lahtinen, A.-P. Tuovinen, E. Rautama & V. Meisalo: Eliot – an algorithm animation environment. 49 pp.

A-1998-1 G. Lindén & M. Tienari (eds.): Computer Science at the University of Helsinki 1998. 112 pp.

A-1998-2 L. Kutvonen: Trading services in open distributed environments. 231 + 6 pp. (Ph.D. thesis).

A-1998-3 E. Sutinen: Approximate pattern matching with the q-gram family. 116 pp. (Ph.D. thesis).

A-1999-1 M. Klemettinen: A knowledge discovery methodology for telecommunication network alarm databases. 137 pp. (Ph.D. thesis).

A-1999-2 J. Puustjärvi: Transactional workflows. 104 pp. (Ph.D. thesis).

A-1999-3 G. Lindén & E. Ukkonen (eds.): Department of Computer Science: annual report 1998. 55 pp.

A-1999-4 J. Kärkkäinen: Repetition-based text indexes. 106 pp. (Ph.D. thesis).

A-2000-1 P. Moen: Attribute, event eequence, and event type similarity notions for data mining. 190+9 pp. (Ph.D. thesis).

A-2000-2 B. Heikkinen: Generalization of document structures and document assembly. 179 pp. (Ph.D. thesis).

A-2000-3 P. Kähkipuro: Performance modeling framework for CORBA based distributed systems. 151+15 pp. (Ph.D. thesis).

A-2000-4 K. Lemström: String matching techniques for music retrieval. 56+56 pp. (Ph.D.Thesis).

A-2000-5 T. Karvi: Partially defined Lotos specifications and their refinement relations. 157 pp. (Ph.D.Thesis).

A-2001-1 J. Rousu: Efficient range partitioning in classification learning. 68+74 pp. (Ph.D. thesis)

A-2001-2 M. Salmenkivi: Computational methods for intensity models. 145 pp. (Ph.D. thesis)

A-2001-3 K. Fredriksson: Rotation invariant template matching. 138 pp. (Ph.D. thesis)