Visual Languages

Definition and Applications in System Development Environments

Sherif El-Kassas
Visual Languages
Definition and Applications in System Development Environments

PROEFSCHRIJT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven op gezag van de
Rector Magnificus, prof. dr. J.H. van Lint, voor een
commissie aangewezen door het College van Dekanen
in het openbaar te verdedigen op
vrijdag 29 april 1994 om 16.00 uur

door
Sherif M. El-Kassas
geboren te Giza, Egypte

Druk: Boek- en Offsetdrukkerij Letru, Helmond, 04920-37797
El-Kassas, Sherif M.


Subject headings: visual languages / system development.
Abstract

The problem domain of this thesis is system development. It is motivated by the need to find and construct better design technologies. The use of better design technologies is viewed as imperative to deal with the increase in the size and complexity of digital systems.

This thesis takes the view that the integration of formal methods and visual languages yields a sound and powerful foundation for the definition of essential aspects of modern design technologies. Hence, the main questions addressed are concerned with: (1) the formal definition of visual languages; and (2) the utilization of the visual language model to define appropriate development environments.

In answering these questions, this thesis develops the foundation for defining design technologies that use formal visual notations and their support tools. It is asserted that such formal visual notations possess the benefits of formal approaches to system development, as well as the intuitive appeal and ease of use of graphical notation.

The question of the formal definition of visual languages is addressed in the form of developing the attribute icon-replacement grammar model, which provides a framework for the definition of the syntax and semantics of visual languages. This model is then extended to enable the definition of syntactically defined hierarchical constructs, as well as providing a framework for integrating textual and visual notations.

The question of defining development environments is addressed by identifying and addressing two essential aspects of such environments, namely: (1) editing and presentations tools, which are addressed by defining an integrated syntax-directed editing model; and (2) transformational and translation tools, which are addressed by providing attribute system based transformation, as well as pair-grammar based transformations.
I would like to thank my supervisor Professor ir. M. P. J. Stevens for giving me the opportunity to carry out this Ph.D. research and for his support throughout my stay at the Eindhoven University of Technology. I am particularly grateful for his careful reading and constructive criticism of early drafts of this thesis.

I would also like to thank all my colleagues at the Digital Information Systems Group for making my stay there a pleasant one. I would especially like to thank Rudi Bloks, Willem Rovers, Robert Huis in 't Veld, for reading and commenting on early papers and reports, and Willem Jan Withagen for his help with \LaTeX.

I would like to acknowledge the financial support of the NUFFIC and the Eindhoven University of Technology.
Contents

Abstract iii

Acknowledgement v

1 Introduction 1
   1.1 The Problem Domain: System Development 2
       1.1.1 Current Practices 4
       1.1.2 Formal Methods 7
       1.1.3 Graphical Notations 7
   1.2 The Problem: defining formal visual languages and support tools 8
       1.2.1 Potential Applications 8
   1.3 Overview of the Solution 9
       1.3.1 The Basic Language Model 9
       1.3.2 The Definition of Hierarchical Languages 12
       1.3.3 Syntax-directed editing for Visual Languages 12
       1.3.4 Pair Grammars and Syntax-Directed Translation 13
       1.3.5 The Architecture of an Integrated Development Environment 14
   1.4 Contributions 15
   1.5 Thesis Organization 16

2 Related Concepts and Other Approaches 17
   2.1 Basic concepts and terminology 17
   2.2 Criteria for considering related projects 19
   2.3 Automatic generation of syntax-directed editors 19
   2.4 Graph grammars and diagram editing 22
   2.5 Picture Layout Grammars 25
   2.6 Unification-based Grammars 26
   2.7 Summary 29
3 Attribute Icon-replacement Grammars
  3.1 Basic Concepts .................................................. 31
  3.1.1 Language model ................................................ 31
  3.1.2 Icon Grammars .................................................. 33
  3.2 Icon-replacement Grammars ...................................... 33
    3.2.1 Icon-graphs .................................................. 33
    3.2.1.1 Icons .................................................... 33
    3.2.1.2 Icon joining points ..................................... 35
    3.2.2 Labelled icon-graphs ....................................... 36
    3.2.3 Labelled icon-graphs with external joints ................. 36
    3.2.4 An order concept ............................................ 37
    3.2.5 Icon replacement ............................................ 37
      Removing icons from a graph ................................... 39
      Attaching the replacement graph ............................... 39
    3.2.6 Grammars .................................................... 40
    3.2.7 Derivation of Language Constructs .......................... 41
      3.2.7.1 Graphs defined by a grammar ......................... 41
      3.2.7.2 Languages .............................................. 41
    3.2.8 Modelling Derivations ...................................... 43
    3.2.9 Context freeness .......................................... 45
  3.3 Attribute Icon-replacement grammars ............................ 45
    3.3.1 Attributes: Basic Concepts ................................ 46
    3.3.2 The Definition of Attribute Icon-replacement Grammars .... 48
      3.3.2.1 Attribute Systems .................................... 48
        Attribute association ....................................... 48
        Attribute computation rules ................................ 48
        Synthesized vs. Inherited Attributes ....................... 49
    3.3.3 Computing Attribute Values ................................ 50
      Pre-specified traversal strategy ............................... 52
  3.4 Summary .......................................................... 53

4 Hierarchical Icon-replacement Grammar ............................. 55
  4.1 Introduction ..................................................... 55
    4.1.1 Hierarchical constructs for system development ............ 56
    4.1.2 Hierarchical constructs for the user interface ............ 56
    4.1.3 Hierarchical constructs from an implementation perspective .... 57
  4.2 Hierarchical Icon-replacement Grammar .......................... 57
    4.2.1 Hierarchical icon-graphs ................................... 58
    4.2.2 Hierarchical Grammars ..................................... 60
    4.2.3 The derivation of hierarchical graphs ....................... 61
CONTENTS

4.2.3.1 The Hierarchical Graphs Defined by a Grammar .................. 62
4.2.3.2 The Hierarchical Language Defined by a Grammar ................. 64
4.2.4 Modelling hierarchical derivations .................................. 64
4.3 Integrating Textual and Visual Constructs .............................. 66
4.4 Attribute Hierarchical Icon-replacement Grammar ....................... 66
4.4.1 The definition of attribute hierarchical icon-replacement grammar .......... 66
4.4.1.1 A hierarchy of attribute icon-replacement grammars .............. 66
4.4.1.2 Interfacing the attribute systems .............................. 68
4.5 Summary .......................................................... 69

5 Syntax-directed editing for Icon-replacement languages ................. 71
5.1 Introduction ....................................................... 72
5.2 The Edit–Parse approach ............................................. 73
5.3 The Syntax-directed editing approach .................................. 74
5.3.1 Syntax-directed editing models .................................... 75
5.3.1.1 The template model ........................................ 75
5.3.1.2 The token model .......................................... 77
5.3.1.3 Hybrid editing models .................................... 77
5.4 Syntax-directed editing for Icon-languages .............................. 78
5.4.1 Interactive derivation ............................................ 78
5.4.1.1 Partial-graph derivations ................................ 79
5.4.1.2 Combining partial-graph derivations ......................... 81
5.4.2 Editing .......................................................... 83
5.4.2.1 Editing complete structures ................................ 83
5.4.2.2 Editing partial structures ................................ 88
  Icon-graph isomorphism ........................................ 88
  Checking icon-graph isomorphism ................................ 88
  Labelled icon-graph isomorphism ............................... 90
5.4.2.3 Derivation tree editing .................................... 91
5.5 Syntax-directed browsing ............................................ 92
5.6 Syntax-directed presentation ......................................... 92
5.7 Summary .......................................................... 94

6 Grammar Pairs and Syntax-Directed Translation ......................... 95
6.1 Introduction ....................................................... 95
6.2 Pair Grammars ..................................................... 96
6.2.1 Basic Concepts ............................................... 96
6.2.2 The Pair Grammar Model ...................................... 98
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2.1 Derivations</td>
<td>101</td>
</tr>
<tr>
<td>6.2.2.2 Pair graphs</td>
<td>101</td>
</tr>
<tr>
<td>6.2.2.3 Pair derivation</td>
<td>102</td>
</tr>
<tr>
<td>Pair graphs defined by a grammar.</td>
<td>102</td>
</tr>
<tr>
<td>The pair language defined by a grammar.</td>
<td>105</td>
</tr>
<tr>
<td>6.2.3 Types of Pair Grammars</td>
<td>105</td>
</tr>
<tr>
<td>6.2.3.1 Revisable Pair Grammars</td>
<td>105</td>
</tr>
<tr>
<td>Reversing a pair grammar.</td>
<td>106</td>
</tr>
<tr>
<td>6.2.3.2 Partial Pair Grammars</td>
<td>106</td>
</tr>
<tr>
<td>6.2.3.3 Pair Grammars with Cut-off Thresholds</td>
<td>106</td>
</tr>
<tr>
<td>6.2.4 Textual language to Icon-language translation</td>
<td>107</td>
</tr>
<tr>
<td>6.3 Summary</td>
<td>112</td>
</tr>
<tr>
<td>7 An Integrated System Development Environment</td>
<td>113</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>113</td>
</tr>
<tr>
<td>7.2 Editing and Presentation Environment</td>
<td>113</td>
</tr>
<tr>
<td>7.2.1 Support for Hierarchical Languages</td>
<td>114</td>
</tr>
<tr>
<td>7.3 Pair–grammar Based Transformational Tools</td>
<td>116</td>
</tr>
<tr>
<td>7.4 The Integrated Environment</td>
<td>116</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>119</td>
</tr>
<tr>
<td>8.1 Evaluation and Overview</td>
<td>119</td>
</tr>
<tr>
<td>8.1.1 Icon languages as modelling tools</td>
<td>119</td>
</tr>
<tr>
<td>8.1.2 Icon languages as the basis for tool construction</td>
<td>121</td>
</tr>
<tr>
<td>8.1.2.1 Language-directed Editing Tools</td>
<td>122</td>
</tr>
<tr>
<td>8.1.2.2 Pair Grammars as Transformational Tools</td>
<td>122</td>
</tr>
<tr>
<td>8.1.3 Limitations of the Basic Model</td>
<td>123</td>
</tr>
<tr>
<td>8.2 Future Directions</td>
<td>124</td>
</tr>
<tr>
<td>8.2.1 Extensions</td>
<td>124</td>
</tr>
<tr>
<td>8.2.2 Future research</td>
<td>124</td>
</tr>
<tr>
<td>A Notational conventions</td>
<td>127</td>
</tr>
<tr>
<td>B MIL: a language for defining icon–replacement grammars</td>
<td>129</td>
</tr>
<tr>
<td>B.1 Introduction</td>
<td>129</td>
</tr>
<tr>
<td>B.2 Basic constructs</td>
<td>129</td>
</tr>
<tr>
<td>B.2.1 Defining icons</td>
<td>130</td>
</tr>
<tr>
<td>B.2.1.1 Basics</td>
<td>130</td>
</tr>
<tr>
<td>B.2.1.2 Generic Icons</td>
<td>130</td>
</tr>
<tr>
<td>B.2.2 Defining graph constructs</td>
<td>131</td>
</tr>
<tr>
<td>B.2.2.1 Basics</td>
<td>131</td>
</tr>
<tr>
<td>B.2.2.2 Generic Graphs</td>
<td>132</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CASE tools timeline</td>
</tr>
<tr>
<td>1.2</td>
<td>A system development model using formal methods</td>
</tr>
<tr>
<td>1.3</td>
<td>Commission, detection and removal of errors</td>
</tr>
<tr>
<td>1.4</td>
<td>Visual languages as generalizations of strings</td>
</tr>
<tr>
<td>1.5</td>
<td>Architecture of the integrated development environment</td>
</tr>
<tr>
<td>2.1</td>
<td>Examples of Pascal code visualization</td>
</tr>
<tr>
<td>2.2</td>
<td>Examples of directed graphs</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of a graph transformation rule</td>
</tr>
<tr>
<td>2.4</td>
<td>A Y production rule example</td>
</tr>
<tr>
<td>2.5</td>
<td>An example of Y rule application</td>
</tr>
<tr>
<td>2.6</td>
<td>Example of a picture defined by the attributed multiset</td>
</tr>
<tr>
<td>2.7</td>
<td>A graphical depiction of a rule with constraints</td>
</tr>
<tr>
<td>3.1</td>
<td>Visual languages seen as a generalization of string languages</td>
</tr>
<tr>
<td>3.2</td>
<td>An icon–graph example</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of icon replacement</td>
</tr>
<tr>
<td>3.4</td>
<td>An icon–replacement grammar</td>
</tr>
<tr>
<td>3.5</td>
<td>A derivation sequence example</td>
</tr>
<tr>
<td>3.6</td>
<td>A derivation tree example</td>
</tr>
<tr>
<td>3.7</td>
<td>A simplified derivation tree</td>
</tr>
<tr>
<td>3.8</td>
<td>Examples of synthesized and inherited attributes</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchical languages as generalizations of visual languages</td>
</tr>
<tr>
<td>4.2</td>
<td>A hierarchical icon–graph</td>
</tr>
<tr>
<td>4.3</td>
<td>A hierarchical icon–replacement grammar</td>
</tr>
<tr>
<td>4.4</td>
<td>A derivation of a hierarchical icon–graph</td>
</tr>
<tr>
<td>4.5</td>
<td>A hierarchical derivation tree</td>
</tr>
<tr>
<td>4.6</td>
<td>A hierarchical icon–graph combining textual and visual constructs</td>
</tr>
<tr>
<td>4.7</td>
<td>A flattened hierarchical derivation tree</td>
</tr>
<tr>
<td>Number</td>
<td>Figure Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.1</td>
<td>An environment employing the edit-parse approach</td>
</tr>
<tr>
<td>5.2</td>
<td>A visual \textit{if-then-else} template</td>
</tr>
<tr>
<td>5.3</td>
<td>The process of interactive derivation</td>
</tr>
<tr>
<td>5.4</td>
<td>A derivation tree starting at the start graph of the \textit{logic diagram} grammar.</td>
</tr>
<tr>
<td>5.5</td>
<td>A partial derivation beginning with the right hand side of a production rule.</td>
</tr>
<tr>
<td>5.6</td>
<td>Tree combination</td>
</tr>
<tr>
<td>5.7</td>
<td>Partial and complete structures</td>
</tr>
<tr>
<td>5.8</td>
<td>Integrating the layout constraint systems with the syntax-directed editing environment.</td>
</tr>
<tr>
<td>6.1</td>
<td>A pair grammar example</td>
</tr>
<tr>
<td>6.2</td>
<td>An icon-replacement grammar, the C-grammar</td>
</tr>
<tr>
<td>6.3</td>
<td>An icon-replacement grammar, the T-grammar</td>
</tr>
<tr>
<td>6.4</td>
<td>Derivation tree of a C-grammar derivation</td>
</tr>
<tr>
<td>6.5</td>
<td>Derivation tree of a T-grammar derivation</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of translation result</td>
</tr>
<tr>
<td>6.7</td>
<td>The \textit{Tree-Record} grammar</td>
</tr>
<tr>
<td>6.8</td>
<td>A derivation tree of a R-grammar structure</td>
</tr>
<tr>
<td>6.9</td>
<td>A derivation tree of a Tree-grammar structure</td>
</tr>
<tr>
<td>6.10</td>
<td>A Tree-graph example</td>
</tr>
<tr>
<td>7.1</td>
<td>The syntax-directed editing environment</td>
</tr>
<tr>
<td>7.2</td>
<td>An example of a hierarchical editing session</td>
</tr>
<tr>
<td>7.3</td>
<td>Using pair-grammars to construct different views of the same construct</td>
</tr>
<tr>
<td>7.4</td>
<td>An integrated development environment</td>
</tr>
<tr>
<td>B.1</td>
<td>An icon-replacement grammar for Actor Nets.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

With the continuing increase in the complexity and size of digital systems, the realization of such systems has become strongly influenced by the so-called design technologies[New91]. A design technology is the collection of metaphors, models, notations, languages and tools used in the development process.

As the size and complexity of systems grows it becomes more evident that such complexity can only be dealt with through the use of suitable design technologies.

Notations and languages form a fundamental part of any design technology. This is because such notations are used by system developers to express and construct models of the system being developed, and they also form the basis for constructing automated tools to be utilized in the development process.

This thesis develops a foundation for defining visual (graphical) languages and notations that can be used in system development. It presents a formal model for defining the syntax and semantics of such languages. It also discusses how this model can be used for the systematic construction
of language-based development environments and tools. Hence this thesis lays a foundation for defining essential aspects of modern design technologies.

The use of a formal model is viewed as essential, since it not only enables the production of accurate system descriptions, but it also makes formal reasoning about system properties possible. Furthermore, from a tool construction perspective, having a formal model of the language and notations used is essential for the systematic construction of tools for manipulating language constructs, such as tools for syntactic and semantic analysis, language-oriented editing, etc.

The use of graphical notations is motivated by the idea that such notations can be more expressive and easier to understand than the equivalent textual ones.

The remainder of this chapter gives an overview of this research. The next section discusses, in somewhat more detail, some of the relevant topics in system development which represent the problem domain for this thesis. Section 1.2 defines the main problem addressed in this thesis. Section 1.3 gives an outline of the solution proposed here and the main results. Finally, section 1.5 gives an outline of the remainder of the thesis.

1.1 The Problem Domain: System Development

System development paradigms, standards, methods and tools are gaining ever more attention in the system engineering community. This continuing investigation aims at finding better design technologies for dealing with the continuing growth in the size and complexity of computer based systems. It also aims at increasing the confidence in the developed systems.

This is reflected in the continuing improvement of methodologies and tools in order to deal with this increase in complexity and size. A recent survey of CASE tools [NC92] shows a timeline of the type of problems being tackled and the methodologies and tools used to solve these problems (see figure 1.1)

1Context-free grammars are a well known example of a formal model that is widely utilized to construct manipulation tools for textual programming language, such as compiler, interpreters, etc. Naturally, such tools would be much more difficult to construct in an ad-hoc manner and without the use of context-free grammars.

2The terms graphical and visual are used interchangeably throughout this chapter.
1.1 The Problem Domain: System Development

Figure 1.1: CASE tools timeline[NC92].
1.1.1 Current Practices

To gain more insight into system development, the development trajectory is often modelled as a set of activities and their relations. Each one of these activities performs a transformation on the system being developed, until, ultimately, a realizable system is delivered. Examples of such models include the "waterfall model," the "spiral model" and the "basic design cycle[Koo91]." Figure 1.2 shows an example of a simplified development model. This model includes three main phases, namely:

- the specification phase, which outputs a statement of what the desired system should do;
- the design phase, which outputs a statement of how the desired effects will be achieved; and
- the implementation phase, which outputs a working system, or a detailed description of how it would be implemented.

In order to increase the confidence in the developed system, these activities are often complemented with verification and validation activities. Verification aims at establishing that the output from each of the main development phases is consistent with its input. Validation aims at establishing that the requirements were correct to start with.

However, most of the widely practised approaches have "little formal scientific basis underlying the development paradigm" [Rat87]. These development methods (usually termed Informal Development Methods) emphasise on introducing quality into the development process via rational principles underlying systematic methods and techniques that are applied in a disciplined manner[Rat87]. However, due to the lack of sound basis, informal methods are often criticized as being ambiguous, inconsistent, and imprecise.

The ambiguous and imprecise nature of informal methods often results in incorrect or unclear system descriptions. Such inaccuracies are especially harmful when introduced in the early phases of the development cycle. Contrary to some practitioners' belief, proceeding as fast as possible to the implementation phases is usually less economical than taking the time and effort to produce well defined system specification as early as possible in the development cycle. Actually, such practices often lead to systems that do not fulfill the users' requirements and that are expensive and time consuming to repair[CHJ86]. Figure 1.3 depicts the relation between the phase in which the error is introduced and the cost of repairing such errors[CHJ86]. It shows that errors introduced earlier in the development cycle, and detected late in the cycle are the most costly to rectify.
Figure 1.2: A system development model using formal methods.
Life-cycle Phase

- Requirements Analysis
- Specification
- High-level Design
- Low-level Design
- Code
- Unit Test
- Integration
- Acceptance Test
- Use

Source of error

Cost of removal of errors (log scale)

Figure 1.3: Commission, detection and removal of errors [CHJ86].
1.1 The Problem Domain: System Development

These defects, and their effects on development, have been among the stronger motivations to seek a more formal approach to system development.

1.1.2 Formal Methods

The formal approaches to system development, termed formal methods, are based on the use of formal languages (i.e. languages with precisely defined syntax and semantics), to describe the system as it evolves in the development cycle. The use of formal languages enables the production of accurate and precise system descriptions. It also makes it possible to reason formally about the system's description. For example, to detect certain types of inconsistencies and errors, or to prove the equivalence of certain constructs\(^3\).

Although there are many advantages in using formal methods, these advantages are often outweighed by the difficulty in applying and presenting them. These difficulties are mainly due to the complexity of the mathematical notation used in formal languages. To overcome this difficulty we need languages that allows their users to express formally system concepts using intuitively appealing constructs and elements\(^4\).

1.1.3 Graphical Notations

Graphical notation seem to be suitable candidates for such an interface. This is due to the, commonly accepted, observation that the human mind is visually orientated, and that it is better optimized for multi-dimensional data [Cha87, Mye89, AB89, Rae85]. This is reflected in a large number of research and development projects that aim at creating better user environments through the use of graphical technologies. Such systems range from programming environments for the novice to sophisticated design and specification tools (e.g. see [DM90, GK86, Gog87, Göt89, Jac85, MH86, NA90, RC89, YJ88]).

In spite of their natural appeal, graphical notation are usually seen as being informal. Therefore, they are often automatically dismissed when considering formal approaches to system development. However, "strictly speaking this is not the fault of the graphic" [Göt86]. But it is actually due

---

\(^3\) A full introduction and overview of the different types and styles of formal methods is beyond the scope of this thesis. Such an introduction and overview can be found in [WL88].

\(^4\) Other research efforts, such as the work presented in [vHSV89], aim at bridging the gap between "theory and practice" by solving some of the shortcomings caused by formalism being biased towards only a few aspects of modelling.
to the absence of clear and precise syntax and semantic definition of visual languages. Moreover, the lack of precise syntax and semantic definition of visual languages also contributes to the difficulty of constructing tools that can manipulate visual language constructs.

1.2 The Problem: defining formal visual languages and support tools

From the discussion in the previous section, it can be seen that: (1) formal methods provide the necessary mathematical rigour to deal with the complexity and requirements of modern system development, but use complex notations that are difficult to apply; and (2) graphical notation can provide an intuitive and easy to use notations, but lack the formal framework needed for effective system development.

Hence, the next logical step is to provide means for integrating the benefits of both formal methods and graphical notations. Therefore, the main problems addressed in this thesis are:

- How do we provide a formal framework for defining visual languages?
- What are the essential aspects of support environments for such languages?
- How can the formal framework be used as the basis for the systematic construction of such environments?

In answering these questions, this thesis develops the foundation for defining design technologies that use *formal visual notations* and their support tools. It is asserted that such formal visual notations possess the benefits of formal approaches to system development, as well as the intuitive appeal and ease of use of graphical notation. Furthermore, the formal definition of visual languages can be utilized in the systematic construction of language-based development support environments.

1.2.1 Potential Applications

This research effort is motivated by interest in system development. Therefore, it is directed towards applications in the same field. More specifically, towards the construction of tools for defining system development environments (i.e. meta–CASE tools). In general such tools include:

- A development environment generator. Such a system would accept, as input, the definition of the syntax and semantics of a visual
1.3 Overview of the Solution

language. It will use this definition to generate an environment for creating and manipulating the language's constructs; and

• A translator system. Such a system would enable, within certain limits, the translation of visual language constructs to, the more conventional, string based (text-only) languages and vice versa. Such a development would enable the addition of graphical front ends to existing specification languages, as well as the visualization (producing a graphical representation) of specifications written in string languages.

Such tools can be used in constructing CASE (Computer Aided Systems Engineering) and CAD (Computer Aided Design) tools that possess the benefits of integrating the use of formal methods and graphical notations. Examples of such design tools might include support for the OOA (Object-Oriented Analysis) notation outlined in [HVS91], and the integrated analysis, specification, simulation and implementation system presented in [SB90].

In general, the use of such tools can be extended beyond the target application domain of system development into other application areas that use diagramming representations (e.g. chemical and biological applications).

1.3 Overview of the Solution

This section briefly outlines the basic aspects of the solution given in this thesis. It briefly outlines the basic aspects of the formal model used for defining visual languages, and how it is used as the basis for defining visual languages support environments.

The remainder of this sections overviews the five basic aspects of the proposed solution. Subsections 1.3.1 and 1.3.2 outline the basic aspects of visual language model. Subsections 1.3.3 and 1.3.4 overview how the model can be used to construct language support tools and environments. Subsection 1.3.5 overviews how such tools can be integrated to construct integrated development environments.

1.3.1 The Basic Language Model

The basic model of visual languages presented here is based on a new form of graph grammar called attribute icon-replacement grammar. It is based on plex languages as presented in [Fed71], and hyperedges and hypergraphs as presented in [Hab89, HK86]. The idea is based on generalizing the concepts of conventional (string) languages.

Conventional language constructs are viewed as sequences of symbols, where each symbol, in a sequence, may be preceded and followed by other
Introduction

Figure 1.4: Visual languages seen as a generalization of string languages. (a) a string; (b) a string seen as a graph of connected elements; (c) a more general graph; (d) A visual language element example — an icon.

Figure 1.5: An example of a visual language construct; the schematic of an 8048 compatible microprocessor core[Ver92].
1.3 Overview of the Solution

symbols. We can then think of symbols as having two attaching points, one connecting it to its preceding member of the sequence, and the other connecting it to its following member.

To generalize this idea, we can think of symbols as having an arbitrary number of attaching points. These points can be used to connect it to an equivalent number of symbols (see figure 1.4). Such generalized symbols, or graphic objects, are referred to as icons.

In general, the basic language constructs are graphs of interconnected icons. Icons are connected by attaching one (or more) of their branches to the branches of another icon.

The interconnected icon metaphor can be used to describe a wide range of diagramming techniques. For example, the IDaSS diagram shown in figure 1.5, depicting an implementation of microprocessor[Ver92], can be seen as composed of a set of interconnected icons.

An icon-replacement grammar can be thought of as a set of rules for constructing well-formed icon-graphs. Such rules are applied to a starting graph which is defined by the grammar. The application of a rule implies replacing an icon from the starting graph with the graph construct defined in the rule. The resulting graph become the target for future application of grammar rules. This process may continue until there are no further icon that can be replaced.

The grammar model presented here is said to be context-free. This is because production rules are defined on the basis of a single icon replacement, which does not depend on, or affect, the context in which the icon appeared.

It is asserted that context-free icon grammars represent a sound framework for defining the structural aspects of visual languages. However, icon-replacement grammars alone are not well suited for defining the non-structural and semantical aspects of visual languages. Therefore, we consider the notion of attribute grammars[Knu68] as a means of attaching semantic information to visual language constructs.

Basically, an attribute icon-replacement grammar is an icon-replacement grammar with attribute evaluation functions attached to its production rules. Attributes may be viewed as a set of values describing various (non-structural) aspects of an icon-graph.

The attribute system presented here allows for the definition of both synthesized and inherited attributes, and algorithms are provided for the efficient evaluation of attribute rules.
1.3.2 The Definition of Hierarchical Languages

A hierarchical language model is presented. This model allows for the definition (syntactically defined) hierarchical constructs. The motivations for adding hierarchical constructs arise from three basic perspectives: system development, user interface, and implementation.

The hierarchical language model is based on a hierarchical model of the underlying grammar system. This new grammar model is called *hierarchical icon-replacement grammar*.

The idea is based on generalizing the concepts of icon-replacement grammars to allow the definition of hierarchical constructs. This generalization is achieved by allowing icons to effectively contain other graphs, thus arriving at the so-called hierarchical graphs. Applying this concept recursively allows for creating an arbitrary number of levels in a hierarchy.

Furthermore, the concept of hierarchical graphs is extended to allow the integration of textual and visual constructs. This done by allowing the icon to contain strings generated from some textual grammar.

It is asserted that such integration is desirable for practical system development. It is thought that, while graphical constructs are more suitable for specifying the global and high level aspects of systems, they tend to be impractical for the more detailed levels. This is reflected in many system design and specification tools that use both visual and textual constructs (for example see [Ver90, Wai89, KE, KG90, Mye89]).

Hierarchical icon-replacement grammars are also augmented with attribute systems for defining the non-structural and semantical aspects of hierarchical languages. The new attribute hierarchical icon-replacement grammar model is based on constructing hierarchies of attribute icon-replacement grammars. This model also defines methods for interfacing the different attribute systems in the hierarchy. The interface is constructed so that the evaluation of the total attribute system is simplified, and can be performed in the same manner attribute evaluations for non-hierarchical languages are performed. This effectively allows us to interface and integrate different languages into one framework that can be used for system description. Furthermore, such interfacing enables the *flattening* of hierarchical constructs, which allows for more efficient and easy processing of such constructs.

1.3.3 Syntax-directed editing for Visual Languages

Syntax-directed editing is proposed as the basis for constructing language-based development environments. Such environments typically allow developers to manipulate and transform their representations of the system
under development.

The environment outlined here is centred around an integrated model of syntax-directed editing for icon-languages. Such editing environments include editing tools for entering, editing and presenting (displaying) system representations.

This approach combines the use of templates, incremental and free form editing, and novel approaches to template substitution. This is done in order to remedy some of the well known limitation and restrictive nature of pure syntax-directed editing.

The main characteristics of the syntax-directed editing model presented in this thesis (in chapter 5) include:

- Interactive derivation;
- Partial-graph derivations;
- Combining partial-graph derivations;
- Limited free form editing;
- Editing complete and partial structures;
- Derivation tree based manipulation; and
- Syntax-directed browsing and presentation.

1.3.4 Pair Grammars and Syntax-Directed Translation

Syntax-directed translation is also concerned with applying the icon-replacement grammar model in development environment and tool construction. It is concerned with the construction of systematic methods for translation. The presented translation method may be used for icon-language-to-icon-language translation as well as string-to-icon-language and icon-language-to-string translation. The translation method is directed towards applications in construct visualization and transformation. It is based on the idea of pair-grammars originally presented in [Pra71].

Pair grammars are based on the idea of linking two grammars. These grammars are linked by pairing the production rules and non-terminals of the two grammars. By doing so, a one-to-one structural transformation system can be constructed (without having to go through attribute rules). Such a system would be quite useful in constructing visualizations (or alternative representations) of system constructs.

This system effectively allows different concrete syntax to be attached to the same abstract one.
A pair grammar can be viewed as being composed of two grammars and a function linking corresponding production in the two grammars. The language generated by such a grammar is a set of icon-graph pairs. For an unambiguous pair grammar, the language generated by such a grammar can be considered as a translation map from the *left hand language* into *right hand language*. Furthermore, if we substitute some string grammar instead of the icon-grammar, we can obtain a transformation from a string language into an icon-language and vice versa.

### 1.3.5 The Architecture of an Integrated Development Environment

An architecture of an integrated development environment is proposed. This architecture integrates the fundamental concepts of visual languages definition with those of syntax-directed editing and pair-grammars. It shows how these basic components can be used to construct development environments that support multiple notations, and which may combine the use of textual and visual notations. Figure 1.6 shows an example of such an architecture. This architecture shows a system which supports a textual language and a visual language. The graph syntax-directed editor is used to enter and manipulate graph constructs, and a pair grammar based system is used to perform the translation between the two languages.

The architecture is generic in the sense that it does not depend on certain
language constructs, but rather it is driven by the underlying grammar systems.

It is asserted that this framework, and the concepts underlying it, provide a sound basis for tools construction.

1.4 Contributions

This thesis solves the main problem mentioned earlier, namely: Defining formal visual languages and their support tools.

The main contributions and results obtained here can be broadly divided into two categories: (1) Foundational issues, which deal with the formal modelling of visual languages; and (2) Application oriented issues, which mainly deal with utilizing the model in tool construction.

The main contributions are as follows:

• Foundational issues:
  
  - Icon-replacement Grammars—the basic formal model for defining visual languages. This model enhances ideas and concepts laid down in earlier research. The attributed version of the model adds the capability of attribute definition and efficient evaluation.
  
  - Hierarchical Icon-replacement Grammars—the formal model of hierarchical visual languages. This is a new formal model for defining hierarchical languages. Extensions to this model also enable the integration of textual and visual notations within the same framework.
  
  - The interfacing and integration of different attribute systems to allow seamless semantic processing. This interface mechanism allows hierarchical constructs to be flattened and treated as non-hierarchical constructs when performing semantical analysis. This allows hierarchical icon-replacement grammars to be implemented in a convenient manner, and it also allows different languages to be used within an integrated framework.

• Application oriented issues:
  
  - The integrated syntax-directed editing model. This model integrates novel and existing ideas about syntax-directed editing into a framework that is suitable for icon-replacement grammars.
  
  - The definition of syntax-directed presentation and browsing functions. The use of language-directed editors is extended to include language directed presentation browsing.
Introduction

- The use of the pair-grammar model in construct visualization and translation schemes. The pair-grammar model is extended and defined for icon-replacement grammars and string languages to enable icon-language-to-icon-language translation as well as string-to-icon-language and icon-language-to-string translation.

- The architecture and design of an integrated development environment. A modular and generic architecture of a development environments supporting syntax-directed editing and pair-grammar based translations is proposed.

1.5 Thesis Organization

Chapter 2 provides a brief overview of related research projects that deal with the formal definition of visual languages and their use in system development.

Chapter 3 describes the basic model of visual languages. This model, called attribute icon-replacement grammar, provides the foundation for defining the syntax and semantics of visual languages.

Chapter 4 extends the model presented in chapter 3 to allow for the inclusion of hierarchical constructs as well as the integration of textual and visual notations.

Chapters 5 and 6 build on the foundation presented in previous chapters to define essential aspects of language-directed development support environments. Chapter 5 describes a model for the syntax-directed editing of visual languages. Chapter 6 discusses the use and construction of syntax-directed translation tools.

Chapter 7 briefly overviews how the basic tool components presented in chapters 5 and 6 can be integrated into a framework for constructing development environments.

Chapter 8 presents the main conclusions, and evaluates the main results of the thesis and describes suggested future extensions and research directions.

Finally, appendix A outlines some of the notations used throughout the thesis, appendix B describes the grammar description language (MIL), and appendix C outlines an alternative (undirected) basic model for the definition of visual languages.
Chapter 2

Related Concepts and Other Approaches

This chapter discusses some related concepts and research projects. These projects propose methods for defining visual languages and the use of such definitions in constructing development environments.

This chapter also outlines some of the main concepts of visual systems, and provides some common definitions of terms like visual programming and visualization.

2.1 Basic concepts and terminology

The terms visual language system and visual programming system refer, in general, to systems that provide their users with a graphical environment in which they can specify, manipulate, and display language (program) constructs. Two basic concepts of visual systems are visual languages, and visualization of language constructs. Where:

- A visual language is a language that allows its users to specify their constructs in two or more dimensions [Mye89]. Examples of such languages include: flowcharts [Tri88]; dataflow diagrams and languages [Som89, Hil92]; the UseIt systems which can be used to generate code from a binary tree like graphical representation of the system under development; the PegaSys systems, which uses formal dependency diagrams to specify dependencies among program entities; the Rehearsal World system, which is a visual programming environment devised for non-programmers and based on a theatre metaphor [AB89]; and the VISAGE system, which is a system for visualizing and visually specifying algebraic definitions of abstract data types[Wai89]; and
18 Related Concepts and Other Approaches

Function GetSymbol( Var C : Char; Var S : IdName ) : Integer;
Var
Sym : Integer; Leave : Boolean;
Begin
S := *;
Repeat
Case GetCharacter( C ) Of
' ', Begin
S := GetIdentifier( c);
GetSymbol := LookUp( S, SymbolTable[Hash(S)]);
Leave := True;
End;

' ' : Begin
End;
Until Leave;
End;

(a)

Figure 2.1: An examples of Pascal code visualization[EEK89]. (a) a Pascal function (Get Symbol); and (b) The structure tree produced by the visualization programme.

- Visualization of language constructs deals with presenting such constructs in two or more dimensions[Mye89]. This might include transforming textual presentations into graphical one, or giving alternative graphical representations to visual constructs. Examples of such systems include: the SEE systems, which attempts to make program text more readable by using multiple fonts and graphics; the Object-Oriented Diagram system which aims at illuminating the message passing structure in object-oriented systems [Mye89]; the GROPE system (Graphical Representations Of Protocols in Estell) which is a tool for graphically animating the dynamic execution of an Estell specification[NA90]; the structure diagrams used to visualize the structure of Algebraic specifications written in ASF (Algebraic specification formalism)[BHK89]; and the computer-aided documentation system which produces structural visualizations of Pascal programs (an example of such a visualization is shown in figure 2.1)[EEK89].
2.2 Criteria for considering related projects

There are many systems and research efforts that can be categorized as visual systems (e.g., see [Mye89]). However, this discussion will be restricted to systems that:

- take a more formal approach to defining the underlying language constructs; and
- are directed towards, or can be used in, applications in system development and tool construction.

These two restrictions are closely related. For example, from a system development point of view, taking a formal approach is necessary if one is to integrate the benefits of formal reasoning into visual systems. Furthermore, the precise definition of visual languages enables the systematic construction of language-directed tools. Such tools include support for the definition of language-directed environments and inter-language translation. Without a formal model it would be difficult to address such issues in a general and flexible manner.\(^1\)

The remainder of this chapter discusses some of the main systems satisfying the above criteria. Such systems include: the work presented by Arefi and others in [AHW90], Göttler in [Göt82, Göt86, Göt89], Golin in [Gol91a, Gol91b], and Wittenburg and others in [WWT91, Witt92].

2.3 Automatic generation of syntax-directed editors

Arefi and others, from the Florida International University, USA, present some research results directed towards one aspect of language-based environments, namely the automatic generation of visual syntax-directed editors [AHW90]. Their formal model of visual languages is based on a restricted form of labelled graphs, called deterministic labelled directed graphs [AHW90]. The graphs are composed of labelled nodes and edges. A graph is called deterministic if it is connected and none of its nodes are connected to more than one node using the same edge label. Figure 2.2 shows examples of deterministic and non-deterministic graphs.

The work reported in [AHW90] represents a methodology for specifying visual languages. Furthermore, the resulting language specification can be

---

\(^1\) Once more, the example of the use of context-free grammars in compiler construction provides an interesting analogy. Such tools would be much more difficult to construct in an ad-hoc manner and without the use of context-free grammars.
Related Concepts and Other Approaches

Figure 2.2: Examples of directed graphs. (a) a deterministic graph; and (b) a non-deterministic graph;

used to automatically generate a syntax-directed editor which is specifically tailored for the specified language.

Visual languages are specified by means of deterministic graph transformation systems. Such transformation systems typically specify a starting graph, which is the axiom of this grammar like structure, and a collection of graph transformation rules. Where graph transformation rules are graph rewriting rules that define how a graph (or a class of similar graphs) can be transformed into another graph. Such transformation rules are composed of two graph patterns, a left hand side and a right hand side, and define how graphs matching the left hand side can be replaced by the right hand side. All graphs used in defining transformation system must be deterministic.

The editor generator system accepts as input a visual language specification (in terms of deterministic graph transformation system), and produces as output a template based syntax-directed editor for the specified visual language.

The described work addresses some of the shortcomings of the pure template model of syntax-directed editing. Namely, the difficulty of including some common editing transformations within the framework of template editing (since, in a pure template system, the only type of syntactic replacement is that of production rule application; i.e. expansion by replacing non-terminal symbols with the right hand side of a production rule). This is because template editing is based on applying production rules to appropriate entities, and thus does not allow what might be called peer-to-peer transformations\(^2\). For example, in a pure template editor for

\(^2\)From a syntactic point of view, two constructs are said to be peers if they appear as right
a language like Pascal, it is not possible to specify a transformation that changes an IF-statement into a WHILE-statement; instead, the user must perform the tedious task of deleting the IF-statement, replace it with the WHILE-statement, and then reentering the common details.

Such problems are solved in [AHW90] by defining editing transformations in the same way that grammatical transformations are defined, i.e. in the form of graph transformation rules. For example, the transformation of an IF-statement into a WHILE-statement is described in a graph transformation rule, just as the transformation of the non-terminal statement into an assignment statement is defined.

Figure 2.3 shows an example of a transformation rule that allows an IF-statement to be replaced by a WHILE-statement. The left hand side of the rule represents an IF-statement which does not contain an ELSE-clause (this is indicated by the λ-labelled node). Edge labels are used to indicate the type of relation between nodes; for example, labels such as ch1, ch2, chn denote the first, second and nth child, respectively. Furthermore, labels such as cf, cft, cff denote control flow, control flow if true and control flow if false, respectively.

It is argued (in [AHW90]) that such an approach will solve many of the editing difficulties which arise in a pure template model of syntax-directed editing.

However, extending grammar rules to include editing functions is somewhat artificial; this is because two fundamentally different concepts are represented in such a way that they become indistinguishable. Furthermore, this will lead to cluttering the language definition with editing information.
Figure 2.4: An example of a programmable attribute graph grammar; a Y production rule example.

An alternative approach is described in chapter 5 of this thesis\(^3\). In this approach a combination of grammar rules and item replacement is used to support peer-to-peer like transformations without introducing editing information into the grammar rules.

Furthermore, deterministic labelled directed graphs (the model presented in [AHW90]) do not provide a clear way of integrating textual and graphical constructs in the language definition (except for simple node and edge labelling). Therefore, all language constructs have to be purely graphical. In many practical situations this is not desired (might even prove to be hampering) and will necessitate the display of a large amount of details.

Deterministic labelled directed graphs do not provide a clear way of defining hierarchical language constructs. Such constructs are necessary for the practical construction of large system specifications and designs.

2.4 Graph grammars and diagram editing

Göttler, from the Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, presents an ongoing research project directed towards the development of formal models of graph languages and their use in constructing CAD environments\([Göt82, Göt89]\).

The formal model, called programmable attribute graph grammars, is based on graphs with labelled edges and nodes. Graph rewriting is ba-\(^3\)This has also been presented in [EK91b]
2.4 Graph grammars and diagram editing

Figure 2.5: An example of Y rule application. (a) a graph on which the Y rule is to be applied; (b) the graph after removing the a node; (c) the graph after the application of the Y production.
Related Concepts and Other Approaches

sed on context-sensitive rules. The rules, called Y-rules\(^4\), contain three components: (1) the graph to be replaced, the rule’s left hand side; (2) the replacement graph, the rule’s right hand side; and (3) an embedding condition describing how the replacement graph is connected to the original graph, the rule’s top part.

Figure 2.4 shows an example of such a Y-rule. This Y-rule contains the three sections: the left hand side, which is the graph containing one node labelled by a; the right hand side, which is the graph composed of the two nodes labelled by e and f; and the top part which is the embedding condition. The rule is applied to the graph in figure 2.5 (a). This is done by removing the node labelled a (and surrounded by the dotted rectangle), replacing it by the right hand side graph, and then applying the embedding conditions to connect the nodes of the replacement graph to the original graph. Figure 2.5 (b) shows the graph after removing the a node and all edges adjacent to it.

To apply the embedding condition three edges have to be considered, namely those cut by the right hand side of the Y (i.e. the edges labelled k, m and n). For example, consider the edge labelled k which must go from a node labelled d to the node labelled f in the replacement graph. The embedding condition specifies that the node labelled d must be situated so that: (1) there is an edge, labelled i, from a node labelled c to the replaced node (a); and (2) there is an edge, labelled j, from the c labelled node to node d. This is shown in figure 2.5 (c) by adding an edge labelled k from d, in the original graph, to f in the replacement graph. The same procedure is applied to the edges labelled m and n resulting in the graph shown in figure 2.5 (c).

Programmable attribute graph grammars are general and flexible. However, the proposed method of graph rewriting contains inherent inefficiencies, mainly due to its context sensitive nature.

As with the work presented in [AHW90], we note here that programmable attribute graph grammars do not provide an explicit mechanism for defining hierarchical constructs and integrating textual and visual notations.

The project is also directed towards the use of programmable attribute graph grammars to develop a computer-aided method that would help in the development of syntax-directed editors for diagramming techniques.

However, the type of user interface described by Göttler does not provide for the direct manipulation of graph constructs. Instead, the user is given a list of possible global graph transformations. The user can then select the desired transformation.

\(^4\)Göttler also describes an alternative form of production rules, called X-rules. X-rules are introduced to reduce the amount of work done during rule application in order to increase the overall efficiency of rule application.
It is questionable whether this type of interface is sufficiently flexible and efficient, especially when dealing with large and complex graphs. An alternative user interface, that allows for direct manipulation, is described in chapter 5 and [EK91b].

### 2.5 Picture Layout Grammars

Golin, from the University of Illinois at Urbana-Champaign, USA, defines an extension of context-free textual grammar, called picture layout grammar, which is used to specify visual languages' syntax[Gol91a, Gol91b].

Picture layout grammars are based on a more basic model called attribute multiset grammars. Where a multiset grammar is similar to a context-free textual grammar except that the right hand side of a production is considered to be an un-ordered collection of symbols, rather than a string. The language generated by such a grammar is a set of multisets (a multiset is an un-ordered collection, i.e. a set which may have repeated elements).

Attribute multiset grammars are multiset grammars with attribute systems. This attributed version is similar to textual attribute grammars[Knu68].

Figure 2.6 shows an example of a picture defined by an attributed multiset. Elements of the multiset are graphic objects, such as rectangle, text, and arrow. Each of these elements is equipped with a set of attributes which indicate its coordinates, and in the case of text objects an additional attribute indicating the string value.

However, because multiset grammars generate un-ordered collections of symbols, attributes are used to express the various ordering relations within the generated multiset.

Picture layout grammars are attribute multiset grammars, where the symbols in such grammars are graphical elements, and the productions specify how these elements are combined to form 2-dimensional pictures. The attributes in a picture layout grammar represent the geometric information about the picture elements.

An important characteristic of this approach is the fact that it uses attributes to describe both structural information (the relation between the symbols, and what is essentially the topology of the underlying graph) and non-structural information, such as layout information. This is not completely unusual. For example, attribute systems are often used to describe the context-sensitive aspects (often called static semantics) of programming languages, to allow the rest of the structural aspects to be defined using the more efficient context-free grammars. However, in Golin's approach all the structural information is moved into the semantic domain, thus blurring the
Related Concepts and Other Approaches

Figure 2.6: An example of a picture defined by the attributed multiset: \{ rectangle[1,1,5,4], rectangle[10,1,14,4], text[2,2,4,3,"BoxA"], text[11,2,13,3,"BoxB"], arrow[5,2,10,2] \}.

line between structural and semantic information.

This is a direct consequence of the use of multiset grammars to model visual languages. This is because un–attributed multiset grammars do not provide any mechanism for expressing relations among objects in the right hand side of production rules. Hence, most of the structural information is moved into the semantic or attribute domain.

Golin also presents a development environment based on the edit–parse approach. The environment is based on two main components: (1) a graphical editor, that basically allows free editing of graphical object and produces as output a multiset representing the graphic object; and (2) a spacial parser that uses a picture layout grammar definition to processes the multiset and produce the corresponding parse tree. An alternative basis for such editing environment is presented in this thesis, which is based on attribute icon–replacement grammars and syntax–directed editing.

2.6 Unification–based Grammars and Tabular Parsing for Graphical Languages

Wittenburg and others, from Bellcore, New Jersey, USA, preset a model for specifying and parsing visual languages[WWT91, Witt92]. The model is based on extending the concatenation operation found in textual languages
to combinatory operators. Such combinatory operators allow combining language elements in a more general manner, as opposed to, for example, allowing for vertical and or horizontal concatenation. Such combinatory operators are defined in terms of relational constraints that must hold among objects.

The grammar model also uses the concept of rewrite rules to define how symbols are combined and replaced. Functional constraints are added to these rewrite rules to enable the definition combinatory operations.

Figure 2.7 shows a graphical depiction of a rule for constructing formulas. The figure describes a formula as being composed of two formulas separated by a horizontal line. Arrows in the figure describe various relations and constraints among the different formula components. For example, the horizontal line (H-Line) must be wider-than both formula objects. The complete rule description is given by:

\[
D_1 \ D_2 \ D_3 \rightarrow M
\]

\[
\langle \ D_1 \ syntax \rangle = \text{VertInfixDivide}
\]

\[
\langle \ D_2 \ syntax \rangle = \text{Formula}
\]

\[
\langle \ D_3 \ syntax \rangle = \text{Formula}
\]

\[
\langle \ M \ syntax \rangle = \text{Formula}
\]

\[
\langle \ M \ semantics \rangle = (\text{divide} \ (\langle \ D_2 \ semantics \rangle \ \langle \ D_3 \ semantics \rangle))
\]

\[
\langle \ M \ cover \rangle = (\text{compose} \ (\langle \ D_1 \ cover \rangle \ \langle \ D_2 \ cover \rangle \ \langle \ D_3 \ cover \rangle))
\]

\[
\text{above} \ ((\langle \ D_1 \ cover \rangle \ \langle \ D_2 \ cover \rangle))
\]

\[
\text{below} \ ((\langle \ D_1 \ cover \rangle \ \langle \ D_3 \ cover \rangle))
\]

\[
\text{widerThan} \ ((\langle \ D_1 \ cover \rangle \ \langle \ D_2 \ cover \rangle))
\]

\[
\text{widerThan} \ ((\langle \ D_1 \ cover \rangle \ \langle \ D_3 \ cover \rangle))
\]

The first line defines the language elements involved in the rule. The rule is reversed from the usual convention in order to indicate that it is used in reduction rather than generation. That is, to indicate that the elements \( D_1 \) through \( D_3 \) are to be reduced to \( M \) when recognized. This is followed by the syntactic and semantic equational constraints which are followed by the relational predicates (above, below and widerThan).

The parsing algorithm presented in [Wit92] does not assume that the language is deterministic or unambiguous. The parser simply computes all allowable derivations for a given input. The parser uses a tabular data-driven parsing algorithm. The algorithm, however, is of unknown time complexity [WWT91].

The material presented in [Wit92] also outlines the use of this model in pen-based computing systems.
Figure 2.7: A graphical depiction of a rule with constraints.
2.7 Summary

This chapter briefly over viewed some of the relevant research project. These projects are directed towards goals related to the definition of formal models of visual languages and the use of these models to construct language–based development tools.

The common theme is the recognition that the underlying model of visual language constructs is some sort of directed–graph. Where such graphs express the graphical and logical relations among objects.

The difference lies in where the structural information is located in the model. Some systems, such as the ones described in [Gol91b] and [WWT91], sacrifice the expression of structural information in the language syntax in order to avoid the problems of ordering imposed by traditional string grammars. In these systems, structural information is encoded in terms of attributes and geometric constraints. These systems also opt for the edit–parse model of development environments, rather than the more interactive syntax–directed editing model.

The systems presented in [AHW90] and [Göt89] opt for language models that are more closely related to the concepts of graph–grammars. These systems move more structural information into the syntax domain. This is done by basically defining topological information in the production rules, where topological information basically expresses how the different language elements are connected. However, the rest of the visual information is modelled in the semantic or attribute domain. Such visual information might include producing pretty graph layouts, or performing some extra visualization steps to translate topological relations into geometric ones.

The visual language model presented in chapter 3 of this thesis is also closely related to the concepts of graph–grammars. However, the model is extended to allow for defining hierarchical and textual constructs as an integral part of the visual language. It is asserted that adding the ability to define hierarchical constructs is essential for the support of system development applications, and that the integration of textual and visual notations can provide for a more effective development environment.

This thesis also presents a model of development environments. The model includes two basic aspects: a visual programming aspect and a visualization aspect. The visual programming aspect of this model is based on syntax–directed editing, rather than the batch like edit–parse approach. It is asserted that syntax–directed editing has some advantages over parsing, most notably the absence of the inherent inefficiencies of parsing visual languages. Furthermore, this environment model also includes a visualization component which is based on pair–grammars. This visualization components enables the translation between different visual notations, as well as
between visual and textual notations.
Chapter 3

Attribute Icon-replacement Grammars

"Such a life, with all visions limited to a Point, and all motion to a Straight Line, seemed to me inexpressibly dreary" —Edwin A. Abbott, FLATLAND.

This chapter presents a basic model for the formal definition of visual languages. This model is called Attribute Icon-replacement Grammar.

The following section briefly outlines some of the basic concepts used in constructing this model and how this model can be seen as a generalization of conventional one-dimensional string grammars. Section 3.2, presents the basic model for defining the structural and syntactic aspects of visual languages. This model is called icon-replacement grammar. Section 3.3 augments (complements) icon-replacement grammars with notion of attributes to enable the definition of the non-structural aspects of visual languages. And finally section 3.4 gives an outline of the main results obtained in this section.

3.1 Basic Concepts

3.1.1 Language model

The conceptual model of attribute icon-replacement grammar is based on plex languages as presented in [Fed71], and hyperedges and hypergraphs as
Figure 3.1: Visual languages seen as a generalization of string languages. (a) a string; (b) a string seen as a graph of connected elements; (c) a more general graph; (d) A visual language element example — an icon.
presented in [Hab89, HK86]. The idea is based on generalizing the concepts of conventional (string) languages.

Conventional language constructs are viewed as sequences of symbols, where each symbol, in a sequence, may be preceded and followed by other symbols (as shown, for example, in figure 3.1 (a)). Thus, one can think of symbols as having two attaching points, one connecting it to its preceding member of the sequence, and the other connecting it to its following member (for example, see figure 3.1 (b)).

To generalize this idea, we can think of symbols which have an arbitrary number of attaching points [Fed71]. These points can be used to connect it to an equivalent number of symbols (see, for example, figure 3.1 (c)). Such generalized symbols will be referred to as icons. It is clear that by using icons to build language constructs we arrive at more general graph constructs, as opposed to the linear, one-dimensional string constructs.

3.1.2 Icon Grammars

Icon grammars describe the rules for constructing well formed graphs of interconnected icons. Such graphs may be viewed as a generalization of the one-dimensional sequences or strings generated by conventional grammars.

3.2 Icon-replacement Grammars

In this section we present a formal model for specifying visual languages. We start by formally defining some central concepts, such as graphs and icons replacement. The mathematical model presented here is based on the one presented in [Hab89].

3.2.1 Icon-graphs

Icon-graphs play a central role in defining visual languages. They are composed of two types of elements, icons and icon joining points.

3.2.1.1 Icons

Icons are basic element of an icon-graph. Each icon is equipped with two (disjoint, finite, and linearly ordered) sets of attaching points called branches. These two sets correspond to incoming and outgoing connections made between the icon and its environment.
Figure 3.2: An icon–graph example.
3.2 Icon-replacement Grammars

3.2.1.2 Icon joining points

Icon joining points (or joints, for short) are auxiliary elements that are used to connect different icon branches together.

Based on the above definitions, we define an icon-graph as structure of (possibly) interconnected icons. Icons are connected by attaching one (or more) of their branches to the same joint. For example, in figure 3.2, icon $i_1$ is connected to icon $i_2$, because each of them has a branch connected to joint $j_z$. More formally, an icon-graph is a 4-tuple $(I, J, in, out)$, where:

- $I$, is a finite set of icons;
- $J$, is a finite set of icon joining points;
- $in : I \rightarrow \text{seq}[J]$, is a mapping, from the icon set $I$ to the set of all sequences over $J$, that defines how the incoming branches of the different icons are connected to the joints; and
- $out : I \rightarrow \text{seq}[J]$, is a mapping, from the icon set $I$ to the set of all sequences over $J$, that defines how the outgoing branches of the different icons are connected to the joints.

Furthermore, there are no unused joints, that is:

$$ J = \bigcup_{i \in I} (\text{ran in} i \cup \text{ran out} i) $$

The connections made to an icon's incoming branches is denoted by an entry in the $in$ function. Such an entry associates an icon with a sequence of joints. The order of the joints in the sequence is significant since it indicates how the branches are connected to the joints; the first joint in the sequence will be connected to the icon's incoming branch, the second joint to the second branch, and so on. Connections made to an icon's outgoing branches are indicated in a similar manner in the $out$ function.

For example, the graph shown in figure 3.2, is defined by:

$$ I = \{ i_1, i_2, i_3, i_4 \}, $$
$$ J = \{ j_1, j_2, j_3, j_4, j_5 \}, $$
$$ in = \{ (i_1, (j_1)), (i_2, (j_2)), (i_3, (j_3)), (i_4, (j_4)) \}, $$
$$ out = \{ (i_1, (j_2, j_3)), (i_2, (j_4)), (i_3, (j_4)), (i_4, (j_5)) \} $$
3.2.2 Labelled icon-graphs

According to the above definition of icon-graphs, the icons appearing in a given graph are given by a set of icons \( I \). Therefore, from basic set properties, there are no repeated element with in the set \( I \). This uniqueness of elements property is convenient when modelling other aspects of icon-graphs, such as the \( \text{in} \) and \( \text{out} \) functions. However, it is also often desirable to give different icons similar appearance and meaning. For example, in figure 3.2 although icons \( i_2, i_3 \) and \( i_4 \) are different we would like them to have similar appearance and meaning (all three icons represent statement blocks in this example).

This can be achieved through the use of labels, where labels are names that are associated with icons in a given graph. Hence, we may have different icons labelled with similar labels, and, thus, having the same appearance and meaning.

An icon-graph with labelled icons is referred to as a labelled icon-graph. A labelled icon-graph over a finite set of labels \( \text{LABEL} \), is modelled as a 5-tuple \((I, J, \text{in}, \text{out}, \text{label})\), where:

- \((I, J, \text{in}, \text{out})\): is an icon graph; and
- \(\text{label} : I \rightarrow \text{LABEL}\), is a mapping, assigning labels (names) to icons in the graph.

In the sequel we will assume that all icon-graphs we discuss are labelled using a fixed set of labels \( \text{LABEL} \). Furthermore, we will refer to the set of all such graphs as \( LG \).

3.2.3 Labelled icon-graphs with external joints

It is often desirable to connect (or embed) icon-graphs to other icon-graphs. To this end, the concept of external joints is proposed as a mechanism for specifying how a graph may be connected to other graphs.

A labelled icon-graphs with external joints can be modeled as a 7-tuple \((I, J, \text{in}, \text{out}, \text{label}, g_{in}, g_{out})\), where:

- \((I, J, \text{in}, \text{out}, \text{label})\): is a labelled icon-graph; and
- \(g_{in}\) and \(g_{out}\) are sequences of joints that define how the graph may be connected to other icon-graphs. Where \(g_{in}\) defines the joints where
incoming branches may be connected, and $g_{in}$ defines where the outgoing branches may be connected; the joints present in $g_{in}$ and $g_{out}$ must be in $J$, i.e.:

$$\text{ran } g_{in} \subseteq J \land \text{ran } g_{in} \subseteq J$$

We will refer to the set of all labelled icon-graphs with external joints, as $LGE$. Furthermore, we refer to the union of $LG$ and $LGE$ as the set of all labelled graphs ($ALG$).

### 3.2.4 An order concept

An icon's incoming and outgoing branches define what we will refer to as, an icon's order. The order of any icon is defined as the pair $(n, m)$, where $n$ and $m$ are the number of incoming, and outgoing branches, respectively. For an icon-graph $G = (I_G, J_G, in_G, out_G)$, the number of incoming branches of any given icon $i \in I_G$ is given by the length (cardinality) of its entry in $in_G$. This is given by: $\# in_G i$. Similarly, the number of outgoing branches is given by: $\# out_G i$.

Hence, the order of all icons in an icon-graph $G = (I_G, J_G, in_G, out_G)$, is given by:

$$\text{order}_G : I_G \rightarrow \mathbb{N} \times \mathbb{N}$$

$$\text{order}_G \triangleq \lambda i : I_G \bullet (\# in_G i, \# out_G i)$$

Furthermore, we define an order concept, analogous to the one defined for icons, for labelled icon-graphs with external joints. It is defined as the pair $(n, m)$, where $n$ and $m$ denote the number of incoming and outgoing joints in $g_{in}$ and $g_{out}$ respectively. More formally:

$$\text{order} : LGE \rightarrow \mathbb{N} \times \mathbb{N}$$

$$\text{order} \triangleq \lambda g : LGE \bullet (\# g_{in}, \# g_{out})$$

### 3.2.5 Icon replacement

Icon replacement is the act of removing an icon from a graph, and then connecting a substitute graph in its place. The activities involved are:

1Functions are defined as follows: first the function's signature is shown (i.e. the function's domain and range), and then its definition is given as a typed lambda expression.
Attribute Icon-replacement Grammars

Figure 3.3: An example of icon replacement.
3.2 Icon-replacement Grammars

- Removing an icon: which is achieved by disconnecting its branches from the corresponding joints and removing it from the graph; and

- Attaching the replacement graph: which is done by connecting the replacement graph’s input and output branches to the corresponding input and output joint of the removed icon.

Furthermore, the icon and its replacement graph must have the same order, i.e. for an icon \( i \) and a replacement graph \( R \): \( \text{order}_G i = \text{order}_R \).

Figure 3.3, shows an example of icon replacement. Where the icon \( i_b \), in the original graph, is replaced by a graph with \( g_{\text{in}} = (j_1) \) and \( g_{\text{out}} = (j_3) \).

We now give a more formal definition of icon replacement, based on two functions \( \text{Remove} \), and \( \text{Join} \), which correspond to the above mentioned steps of icon replacement. That is, \( \text{Remove} \) defines how an icon is removed from a graph, and \( \text{Join} \) defines how the replacement graph is attached in its place.

Removing icons from a graph In order to remove an icon from a labelled graph, the icon must be deleted from the graph’s icon set \( (I_G) \), as well as from the graphs input and output mappings \( (\text{in}_G \text{ and } \text{out}_G) \) and labelling function \( (\text{label}_G) \). This is modelled by the function \( \text{Remove} \) which is defined as follows:

\[
\text{Remove} : \text{ALG} \times I \rightarrow \text{LG}
\]

\[
\text{Remove} \triangleq \lambda G : \text{ALG}; i : I \mid i \in I_G \ast
\]

\[
( I_G \setminus \{i\}, J_G, \{i\} \lessdot \text{in}_G, \{i\} \lessdot \text{out}_G, \{i\} \lessdot \text{label}_G )
\]

Attaching the replacement graph The function \( \text{Join} \) defines how a replacement graph can be attached in place of a removed icon. \( \text{Join} \) takes two graphs as input: the original graph from which the icon was removed and the replacement graph to be embedded in the first graph. \( \text{Join} \) also takes two joints sequences as input, which correspond to the input and output joints of the removed icon. The two joint sequences are used as attaching points for the replacement graph. Hence, branches connected to input joints of the replacement graph are attached to the icon’s input joint sequence. Similarly, branches connected to output joints of the replacement graph are attached to the icon’s output joint sequence. The result returned by \( \text{Join} \) is the labelled icon graph resulting from merging the two graphs.
Attrlbute kon-replacement Grammars

Join : ALG × LGE × seq[J] × seq[J] → LG
Join ≜ λ G : ALG; R : LGE; Jin, Jout : seq[J] •
( I_G ∪ I_R, J_G ∪ J_R \ (ran g_{inR} ∪ ran g_{outR}),
in_G ∪ Jrep(I_R, in_R, g_{inR}, Jin),
out_G ∪ Jrep(I_R, out_R, g_{outR}, Jout),
label_G ∪ label_R )

Where the auxiliary function, \( Jrep \), replaces the external joints of the replacement graph with the corresponding input and output joints of the removed icon, respectively. \( Jrep \) is defined by:

\[
Jrep : P I × (I → seq[J]) × seq[J] × seq[J] → (I → seq[J])
Jrep ≜ λ I_R : P I; io_R : I → seq[J]; g_{io_R} : seq[J]; io_i : seq[J] •
\{ x : I_R \mid ran io_Rx ∩ ran g_{io_R} = ∅ • x ↦ io_Rx \} \cup
\{ x : I_R \mid ran io_Rx ∩ ran g_{io_R} ≠ ∅ •
x ↦ ((io_Rx ⇒ ran g_{io_R})∪
\{ n : 1 .. \#g_{io_R}; m : 1 .. \#io_Rx \mid
(io_Rx)m = g_{io_R}n • m ↦ io_i n \})\}
\]

We can now define the icon replacement functions as:

\[
Replace : ALG × I × LGE → LG
Replace ≜ λ G : ALG; i : I; R : LGE •
Join(Remove(G, i), R, in_G i, out_G i)
\]

Clearly, the \( Replace \) function will only yield sound results if the two icon sets \( I_G \) and \( I_R \) are disjoint (the same holds for the two joint sets, \( J_G \) and \( J_R \)). However, this assumption does not represent a serious restriction on the model. Because, although all icons must be unique, different icons can be given similar syntactic and semantic meaning through the use of labels.

3.2.6 Grammars

Icon–replacement grammars are means of defining icon–graph languages. An icon–replacement grammar can be seen as a set of rules, that can be used to generate all graphs belonging to a certain language. An icon–replacement grammar is defined by a 4–tuple \(( V, N, S, P)\), where:

- \( V ⊆ LABEL \), is the vocabulary set, i.e. the (finite) set of all labels used in the language;
- \( N ⊆ V \), is the set of non–terminal vocabulary;
• \( S : ALG_V \), is the starting graph\(^2\); and

• \( P : N \leftrightarrow ALG_V \), is a finite set of production rules.

For example consider the grammar shown in figure 3.4, which may be used to generate simple logic diagrams.

### 3.2.7 Derivation of Language Constructs

Applying a production rule \( p = (l, R) \), to a graph \( G \), means finding an icon in \( G \) which is labelled with \( l \), and then replacing it with \( R \). The icon (to be replaced) and \( R \) must have the same order. This is denoted by the predicate \( \text{Applicable} \), which is defined as:

\[
\text{Applicable}(p, G) \triangleq \exists i : I_G \bullet \text{label}_G i = l \land \text{order}_G i = \text{order}_R
\]

The graph resulting from applying \( p \) to icon \( i \) in \( G \), is said to be derived from \( G \). A derivation is denoted by: \( G \Rightarrow_{(p, i)} G' \). And the resulting graph is given by:

\( G' = \text{Replace}(G, i, R) \).

Furthermore, a sequence of derivations: \( G \Rightarrow_{(p_1, i_1)} G^1 \Rightarrow_{(p_2, i_2)} G^2 \cdots \Rightarrow_{(p_n, i_n)} G^n \), is denoted by: \( G \Rightarrow_d G^n \), where \( d = ((p_1, i_1), (p_2, i_2), \cdots (p_n, i_n)) \), or alternatively \( G \Rightarrow^* G^n \) if we are not interested in the exact derivation sequence.

Figure 3.5 shows a derivation sequence example which is based on the grammar shown in figure 3.4.

#### 3.2.7.1 Graphs defined by a grammar

The set of all graphs defined by an icon grammar \( \Gamma \), is the set of all graphs generated by it. This set is defined by:

\[
\text{Graphs}(\Gamma) \triangleq \{ G : ALG_V \mid \exists d : \text{seq}[P] \bullet S_\Gamma \Rightarrow_d G \}
\]

#### 3.2.7.2 Languages

The language defined by an icon grammar \( \Gamma \) is a subset of \( \text{Graphs}(\Gamma) \) that is restricted to graphs that do not contain non-terminal elements (i.e. icons with non-terminal labels). The language is denoted by \( \text{Lang}(\Gamma) \), and defined as:

\[
\text{Lang}(\Gamma) \triangleq \{ G : \text{Graphs}(\Gamma) \mid \text{ran} \text{label}_G \cap N_\Gamma = \emptyset \}
\]

\(^2\)\( ALG_V \) is a restriction of \( ALG \) to all labelled icon–graphs with labels that are in \( V \), i.e. 

\( ALG_V = \{ G : ALG \mid \forall i \in I_G \bullet \text{label}_G i \in V \} \)
Logic = (V, N, S, P)

V = { gate, and, or, not, connector}

N= { gate }

S = gate

P = {
    (gate, gate, gate, gate),
    (gate, gate, gate, gate),
    (gate, gate, gate, gate),
    (gate, gate, gate, gate),
    (gate, gate, gate, gate)}

Figure 3.4: A simple icon-replacement grammar.
3.2 Icon-replacement Grammars

3.2.8 Modelling Derivations

Derivation trees have been used to describe derivations of string and graph languages (e.g. see [Knu68, Hab89]). In this subsection we also define a tree structure that may be used to describe icon-graph derivations, and evaluate attributes of attribute icon-replacement languages.

The root of a derivation tree always corresponds to the starting graph of the grammar. Each node in a derivation tree is labelled with a pair, \((p, i)\), where \(p\) is a production rule and \(i\) is the icon to be replaced when \(p\) is applied.

The production, \(p\), is applied to the graph of its ancestor node. Where the graph of a node is defined as the replacement graph appearing in the production rule (i.e. the production's right hand side).

Figure 3.6 shows the derivation tree of the derivation in 3.5.

Derivation trees will be modelled with the relation:

\[
Tree : Node \times \text{seq}[Tree]
\]

where

\[
Node : P \times I
\]

We also define a function, \(Result\), that traverses a derivation tree, and reproduces the corresponding graph. For an icon grammar \(\Gamma\), \(Result\) is
Figure 3.6: A derivation tree example.
3.3 Attribute Icon-replacement grammars

defined as:

\[
\text{Result} : \text{Tree} \rightarrow \text{Graphs}(\Gamma)
\]
\[
\text{Result}(((p, i), s) : \text{Tree}) = 
\begin{cases} 
  s \neq \langle \rangle & \text{Replace}'(\text{RHS } p, \text{Result}' s) \\
  s = \langle \rangle & \text{RHS } p 
\end{cases}
\]

\[
\text{Replace}' : \text{ALG} \times \text{seq}[\text{LGE} \times I] \rightarrow \text{ALG}
\]
\[
\text{Replace}'(G : \text{ALG}; s : \text{seq}[\text{LGE} \times I]) = 
\begin{cases} 
  s \neq \langle \rangle & \text{Replace}'(\text{Replace}(G, s 1), \text{tail } s) \\
  s = \langle \rangle & G 
\end{cases}
\]

\[
\text{Result}' : \text{seq}[\text{Tree}] \rightarrow \text{seq}[\text{LGE} \times I]
\]
\[
\text{Result}' \triangleq \lambda st : \text{seq}[\text{Tree}] \cdot 
\{ n : 1 \ldots \#st; ((p, i), s) : \text{Tree} \mid ((p, i), s) = st n \cdot 
  n \mapsto (\text{Result } st n, i) \}
\]

Notes:

- To preserve the uniformity of derivation trees, we introduce the label \(((b_0, S), i_0)\), which is used to label the root node. Where \(b_0\) and \(i_0\) are dummy elements, and \(S\) is the grammar's starting graph.

- A simplified drawing of the derivation tree is often used instead of the more detailed version shown in figure 3.6. Figure 3.7 shows such a simplified tree.

3.2.9 Context freeness

The icon grammars and languages defined above are called context-free. Because production rules are defined on the basis of a single icon replacement, which does not depend on, or affect, the context in which the icon appears.

3.3 Attribute Icon-replacement grammars

Context-free icon grammars represent a sound framework for defining the structural aspects of visual languages. However, icon-replacement grammars alone are not well suited for defining other aspects of visual languages.
Figure 3.7: An example of a simplified derivation tree.

Such aspects include display considerations, such as graph layout information and the composing an icon's image, as well as the semantical aspects of the language.

Therefore, we consider the notion of attribute grammars [Knu68] as a means of attaching display and semantic information to visual language constructs.

An attribute icon-replacement grammar is an icon-replacement grammar with attribute evaluation functions attached to its production rules. Attributes may be viewed as a set of values describing various (non-structural) aspects of an icon-graph. Their evaluation depends on the derivation process followed to construct a given graph.

3.3.1 Attributes: Basic Concepts

Attributes can be partitioned into two classes, synthesized and inherited [Knu68], where synthesized attributes are based on the attributes of the descendant of a node in a derivation tree, and may be directly associated with the terminal entries in a graph. An example of a synthesized attribute might be the shape or image associated with an icon's label.
Inherited attributes, on the other hand, are based on the attributes of ancestor nodes. An example of such an attribute might be the position of an icon on the display, which is dependent on the position of its ancestor.

Figure 3.8 shows an example of how attribute computation rules may be attached to productions, and how they can be used to compute display and layout information. The Shape attribute is given as an example of a synthesized attribute that is passed bottom up in the derivation tree, and position attributes, such as Xcoor and Ycoor are given as examples of inherited attributes.
3.3.2 The Definition of Attribute Icon-replacement Grammars

3.3.2.1 Attribute Systems

To incorporate the notion of attributes within the icon-replacement grammar model we associate an attribute system with each icon-replacement grammar. Where, an attribute system has two components:

- An attribute association function, Attr, which associates a finite attribute set with each label in the grammar; and
- A finite set of attribute computation rules (also called semantic rules) which may be used to calculate the value of a given attribute.

Attribute association. For a given icon-replacement grammar \( \Gamma = (V, N, S, P) \), the attribute association function, \( Attr \), is defined as:

\[
Attr : V \rightarrow P \text{ASET}
\]

Where \( \text{ASET} \) is the finite set of all attributes used in the grammar. Furthermore, we restrict \( Attr \) such that an attribute is associated with exactly one label. That is:

\[
Attr x \cap Attr y \neq \emptyset \Rightarrow x = y
\]

Hence, the attributes associated with label \( v \in V \) are given by \( Attr v \). Furthermore, we will write \( v.a \) to denote an attribute, \( a \), which belongs to \( Attr v \).

Attribute computation rules. Attribute computation rules are functions which are associated with grammar production rules. These functions can be used to compute the attribute values of the labels involved in this particular grammar production rule. To include this notation in icon-replacement grammar we define a function, called \( ACR \) (short for attribute computation rules), which associates a finite set of attribute computation rules with each grammar production rule.

More formally, for a given icon-replacement grammar \( \Gamma = (V, N, S, P) \), \( ACR \) is defined by:

\[
ACR : P \rightarrow (A_p \rightarrow \text{AttributeFunctionSet})
\]

Where \( A_p \) is the set of all attributes associated with all labels that occur in production \( p \). Hence, for a production \( p = (l, G) \), \( A_p \) is given by:

\[
\bigcup_{v \in \{s\}_{l \text{ran label}G}} Attr v
\]
Furthermore, *AttributeFunctionSet* is a the set of functions used to compute attribute values. Where each function $f$ in the *AttributeFunctionSet* evaluates the corresponding attribute in terms of other attributes occurring in the same production $^3$.

**Synthesized vs. Inherited Attributes.** For a given production $p = (l, G)$ we define the set of all attributes computed in this production ($p$) by:

$$AC_p = \{ a : ASET \mid a \in \text{dom} ACR p \}$$

An attribute $v.a$ is called synthesized if $v$ occurs in the left hand side of some production $p$ and $v.a$ is computed in this production $p$. More formally, we define the predicate $\text{isSyn}$ as:

$$\text{isSyn}(v.a) \triangleq \exists p : P \rightarrow p = (v, G) \land v.a \in AC_p$$

Similarly, an attribute $v.a$ is called inherited if $v$ occurs in the right hand side of some production $p$ and $v.a$ is computed in this production $p$. Hence we define a predicate $\text{isInh}$ as:

$$\text{isInh}(v.a) \triangleq \exists (l, G) : P \rightarrow v \in \text{ran} label_G \land v.a \in AC_p$$

Furthermore, for each label $v$ we define two sets: $\text{Syn}_v$ and $\text{Inh}_v$. Where $\text{Syn}_v$ is the set of all synthesized attributes associated with $v$, and $\text{Inh}_v$ is the set of all inherited attributes associated with $v$. More formally:

$$\text{Syn}_v = \{ v.a \mid \text{isSyn}(v.a) \}$$

$$\text{Inh}_v = \{ v.a \mid \text{isInh}(v.a) \}$$

Consider the icon-replacement grammar shown in figure 3.4. As an example we will add an attribute system to this icon-replacement grammar to be able to compute the semantics of a diagram generated by the grammar. Hence, we must define the two components of the attribute system:

- The attribute association function, $Attr$, which is given by:

$$Attr = \{
\begin{align*}
(gate, \{\text{gate.func}, \text{gate.pos}\}), \\
(and, \{\text{and.pos}\}), \\
(or, \{\text{or.pos}\}), \\
(not, \{\text{not.pos}\}), \\
(connector, \{\text{connector.pos}\})
\end{align*}
\}$$

$^3$To distinguish between labels that appear more than once in a production we add an icon subscripts to such labels. That is, for $(i, v) \in label_G$ we will write $v_i.a$ instead of $v.a$ to avoid such confusion.
The attribute computation rules, $ACR$, which is given by\textsuperscript{4}:

$$ACR = \{$$

\begin{itemize}
  \item $(p_1, \{ (\text{gate}.\text{func}, \lambda x, y \cdot x \land y),
                  (\text{and}.\text{pos}, \text{gate}.\text{pos}) \})$,
  \item $(p_2, \{ (\text{gate}.\text{func}, \lambda x, y \cdot x \lor y),
                  (\text{or}.\text{pos}, \text{gate}.\text{pos}) \})$,
  \item $(p_3, \{ (\text{gate}.\text{func}, \lambda x, y \cdot y \land \neg x),
                  (\text{and}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_3.1}),
                  (\text{not}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_3.2}) \})$,
  \item $(p_4, \{ (\text{gate}.\text{func}, \lambda x, y \cdot x \land \neg y),
                  (\text{not}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_4.1}),
                  (\text{and}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_4.2}) \})$,
  \item $(p_5, \{ (\text{gate}.\text{func}, \lambda x, y \cdot
                  \text{gate}_{i_1}.\text{func}(\text{gate}_{i_2}.\text{func}(x, y),
                  \text{gate}_{i_3}.\text{func}(x, y)),
                  (\text{gate}_{i_1}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_5.1}),
                  (\text{gate}_{i_2}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_5.2}),
                  (\text{gate}_{i_3}.\text{pos}, \text{gate}.\text{pos} + \text{const}_{p_5.3}) \})$,
\end{itemize}

It follows from the above that:

$$\text{Syn}_{\text{gate}} = \{ \text{func} \},$$
$$\text{Inh}_{\text{gate}} = \{ \text{gate}.\text{pos} \},$$
$$\text{Inh}_{\text{and}} = \{ \text{and}.\text{pos} \},$$
$$\text{Inh}_{\text{or}} = \{ \text{or}.\text{pos} \},$$
$$\text{Inh}_{\text{not}} = \{ \text{not}.\text{pos} \},$$
$$\text{Syn}_{\text{and}} = \text{Syn}_{\text{or}} = \text{Syn}_{\text{not}} = \emptyset$$

### 3.3.3 Computing Attribute Values

Computing attribute values is the act of evaluating the semantic rules associated with each attribute (present in a given derivation tree). This entails

\textsuperscript{4}The labels $p_1$ through $p_5$ are used here to denote the five production rules of the grammar shown in figure 3.4, respectively.
traversing the derivation tree and evaluating the semantic rules found at each node.

For example, evaluating the attribute rules associated with the derivation tree shown in figure 3.7 would yield the following:

1. From applying $p_5$ we get:

   \[
   \begin{align*}
   gate_{i_1}.pos &= gate_{i_0}.pos + const_{p_5,1} \\
   gate_{i_2}.pos &= gate_{i_0}.pos + const_{p_5,2} \\
   gate_{i_3}.pos &= gate_{i_0}.pos + const_{p_5,3} \\
   gate_{i_4}.func &= \lambda x, y \cdot gate_{i_0}.func(x, y), gate_{i_2}.func(x, y) \\
   
   \end{align*}
   \]

2. From applying $p_4$ we get:

   \[
   \begin{align*}
   not.pos &= gate_{i_1}.pos + const_{p_4,1} \\
   &= gate_{i_0}.pos + const_{p_5,1} + const_{p_4,1} \quad \text{from (1)} \\
   and.pos &= gate_{i_1}.pos + const_{p_4,2} \\
   &= gate_{i_0}.pos + const_{p_5,1} + const_{p_4,2} \quad \text{from (1)} \\
   gate_{i_5}.func &= \lambda x, y \cdot x \land \neg y \\
   
   \end{align*}
   \]

3. From applying $p_3$ we get:

   \[
   \begin{align*}
   and.pos &= gate_{i_2}.pos + const_{p_3,1} \\
   &= gate_{i_0}.pos + const_{p_5,2} + const_{p_3,1} \quad \text{from (1)} \\
   not.pos &= gate_{i_2}.pos + const_{p_3,2} \\
   &= gate_{i_0}.pos + const_{p_5,2} + const_{p_3,2} \quad \text{from (1)} \\
   gate_{i_6}.func &= \lambda x, y \cdot y \land \neg x \\
   
   \end{align*}
   \]

4. From applying $p_2$ we get:

   \[
   \begin{align*}
   or.pos &= gate_{i_3}.pos = gate_{i_0}.pos + const_{p_5,3} \quad \text{from (1)} \\
   gate_{i_7}.func &= \lambda x, y \cdot x \lor y \\
   
   \end{align*}
   \]

Finally, by substituting from 2, 3 and 4 into 1 we get:

\[
\begin{align*}
\text{gate}_{i_0}.func &= \lambda x, y \cdot (\lambda a, b \cdot a \lor b) \\
  &\text{((\lambda u, v \cdot u \land \neg v)(x, y)),} \\
  &\text{(\lambda u, v \cdot v \land \neg u)(x, y))} \\
  &= \lambda x, y \cdot (x \land \neg y) \lor (y \land \neg x)
\end{align*}
\]

Hence, we are able to evaluate all the semantic rules and arrive at the corresponding attribute values.
Clearly, such an evaluation is not always possible. For example, if circular dependencies exist between the semantic rules [Knu68]. Furthermore, the order of rule evaluation and the number of required passes over the derivation tree also depends on the dependencies between rules. Therefore, in this treatment of attribute icon-replacement grammar, we will restrict the types of allowable rule dependencies to enable simple and efficient rule evaluation. These restrictions will enable attribute evaluation in a pre-specified, one pass traversal of the derivation tree.

Pre-specified traversal strategy To enable (and guarantee) such a one pass traversal we say that for any production \( p = (l_0, G) \in P \) that occurs in a derivation tree the following conditions must hold:

- The inherited attributes of \( l_0 \) must be computable and may not depend on any of the attributes of the right hand side symbols;
- The inherited attributes of the right hand side symbols may only depend on \( \text{Inh}_{l_0} \);
- The synthesized attributes of a given right hand side symbol may depend on its inherited attributes, the inherited attributes of the left hand side symbol, as well as the synthesized attributes of its descendant nodes in the tree; and
- The synthesized attributes of the \( l_0 \) may depend on any of the above items.

More formally, for a production \( p = (l_0, G) \in P \) where the right hand side symbol set, \( RS \), is given by:

\[
RS = \{ i \mid (i, l) \in \text{label}_G \}
\]

The semantic rules dependencies must be such that it is always possible to compute the rules associated with \( p \) in any of the following sequence:

\[
\{(\text{Inh}_{l_0}, s, \text{Syn}_{l_0}) \mid s \in \text{seq}_{\#RS}[A]\}
\]

where

\[
A = \{(\text{Inh}_{l_i}, \text{Syn}_{l_i}) \mid l_i \in RS\}
\]

\[
\text{seq}_n[X] \triangleq \{ v : \text{seq}(X) \mid \#v = n \land \#\text{ran} v = n \}
\]

Given the above constraints, it is possible to evaluate all attributes using a one-pass top-down traversal of the derivation tree. The traversal strategy

\[5\text{See, for example, [DJL88, WG84] for a more complete treatment and taxonomy of the different types of attribute grammars.}\]
partitions the attributes rules into two set. With the first set being the set of inherited attributes and the second being that of synthesized attributes. When a node (in the derivation tree) is visited, its inherited attributes (i.e. the inherited attributes associated with it) are evaluated; then its descendant nodes are visited, and finally, its synthesized attributes are computed.

The following algorithm outlines the procedure:

\[ \text{Eval((node, treeSeq) : Tree)} \]

1. compute the inherited attribute at node
2. if treeSeq \( \neq \emptyset \) then
   \[ \forall t \in \text{treeSeq} \]
   \[ \text{Eval}(t) \]
3. compute the synthesized attribute at node

### 3.4 Summary

This chapter presented attribute icon-replacement grammars as the basis of defining visual languages. Related research and development projects have been described in [AHW90], [Göt82], [Göt89], and [ELS86]. However, they are based on a different conceptual and formal model. It is the view of this author, that icon-replacement grammars provide a simpler and more intuitively appealing model for visual language definition.

Furthermore, it is worth noting that the way in which icon-graphs are used to model systems is different from most graph-grammar based approaches. Since in most graph-grammar based approaches, systems are modelled using two basic graph elements: nodes and edge.

With icon-replacement grammars and icon-graphs icons are the only type of basic element (note that joints are only used to denote connections and are not used to model system entities). Hence, icons are used to model all system entities, whether they represent objects or relations among objects. It is asserted that such uniformity is desirable for effective modelling, and that it is more in the spirit of modern approaches to system development, such as the various object-oriented approaches.

Icon-replacement Grammars are augmented with attributes to enable the definition of the non-structural aspects of icon-languages. The attribute systems addressed here allow for the use of both inherited and synthesized attributes. Furthermore, the types of dependencies among attributes are restricted to allow for efficient pre-specified traversal strategy.
Chapter 4

Hierarchical Icon-replacement Grammar

"out of flatland"

This Chapter presents some extensions to the basic model presented in chapter 3. The main extensions are based on the introduction of a hierarchical model of icon-replacement grammar. This model, called hierarchical icon-replacement grammar, is then extended to enable the integration of textual and visual notations.

A model for defining the semantics and non-structural aspects of hierarchical languages is defined. This model enables the integration of different attribute icon-replacement grammars to construct an attribute hierarchical icon-replacement grammar. It provides mechanisms to interface the different attribute systems.

Section 4.1 introduces the main motivations for adding hierarchical language constructs. Section 4.2 presents the hierarchical icon-replacement grammar model. Section 4.3 describes how this model can be used to integrate textual and visual notations. Finally, section 4.4 discusses attribute hierarchical icon-replacement grammar.

4.1 Introduction

The motivations for adding (syntactically defined) hierarchical constructs arise from three basic perspectives: (1) from a system development perspective, which is concerned with the application and use of visual languages as
development tools; (2) from a user interface perspective, which is concerned with the ease of use and effectiveness of visual representation from a user interface point of view; and (3) from an implementation perspective, which is concerned with the difficulties in implementing and displaying large visual constructs.

### 4.1.1 Hierarchical constructs for system development

In this thesis visual languages are proposed as formal specification languages. That is, languages that can be used in system development. Therefore, they must satisfy certain criteria to establish their usefulness in this application domain.

One of the main functions of a specification language is to provide means for managing the size and complexity of the problem being solved[EK90]. It is asserted that providing support for modelling hierarchical abstractions is an effective method for managing this complexity[MMP88, KM88]. Such hierarchical abstractions include:

- support for composition and decomposition, such as, for example, the functional decomposition scheme found in structured design methods;
- support for classification and exemplification, such as the class–object relation in class–based systems; and
- generalization and specialization, such as the inheritance relations found among classes in an object–oriented class hierarchy.

Therefore, the icon–replacement grammar model is extended with the ability to define such hierarchical constructs.

In addition to the above, dividing a language into a hierarchy of levels, where each level supports a different set of abstraction, allows the problem to be dealt with at the appropriate level with the most appropriate set of abstractions. This effectively allows the integration of different languages into a single framework.

### 4.1.2 Hierarchical constructs for the user interface

From a user interface perspective, adding hierarchical abstractions is essential for dealing with large and complex graphs. Since visual constructs consisting of a single large icon–graph are cumbersome to deal with. Furthermore, as the size of the graph increases, a point is reaches where the advantages of visual representation are lost in the clutter of details[RW91].
The problem is essentially a user interface one, since it is basically bound by the human field of vision rather than, for example, the display size.

Hence, it is essential to provide support for hierarchical abstractions to enable the user to deal with large and complex graphs, and to retain the benefits of visual representation.

### 4.1.3 Hierarchical constructs from an implementation perspective

From an implementation perspective, hierarchical constructs can help solve the difficulties that arise from display limitations and the size, appearance and density of visual constructs [Mye89]. This is different from the user interface considerations, since the main concern is with how the implementation may represent or display a given construct on the limited display resources, rather than how acceptable or useful the displayed structure may be from the user's perspective.

It is asserted, in [Mye89], that almost all visual representations are physically larger than the textual representations they replace. Hence, better ways are needed to manage the display and presentation of this increase in the size of display data.

Hierarchical constructs provide means for dealing with these presentation problems.

Furthermore, we propose that the integration of textual and visual constructs is desirable for practical system development. It is thought that, while graphical constructs are more suitable for specifying the global and high level aspects of systems, they tend to be impractical for the more detailed levels. This is reflected in many system design and specification tools that use both visual and textual constructs (for example see [Ver90, Wai89, KE, KG90, Mye89]).

### 4.2 Hierarchical Icon–replacement Grammar

In this section we introduce, and formally define, the notion of hierarchical icon–replacement grammars.

The idea is based on generalizing the concepts icon–replacement grammars to allow the definition of hierarchical constructs. This generalization is achieved by allowing icons to be labelled with other graphs, rather than simple labels, thus arriving at the so-called hierarchical graphs (see figure
Hierarchical icon-replacement Grammar

Figure 4.1: Hierarchical visual languages seen as a generalization of visual languages. (a) a general graph (visual) language construct; (b) a hierarchical graph.

Furthermore, applying this concept recursively allows for creating an arbitrary number of levels in a hierarchy.

4.2.1 Hierarchical icon–graphs

In this section icon–graphs are generalized into the so-called hierarchical graphs. As mentioned earlier, this is done by allowing icons to contain (or be labelled by) other icon–graphs. The resulting constructs are called hierarchical icon–graphs (see figure 4.1 (b)).

More formally, a hierarchical icon–graph is defined as the pair $(G, \text{GraphAssignment})$. Where:

- $G$, is an icon–graph, called the base graph; and

- $\text{GraphAssignment} : I_G \rightarrow \text{GlobalGraphSet}$, is a partial function that defines the hierarchical component by assigning graphs to icons. $\text{GraphAssignment}$ is modelled as a partial function to express the property that not all icons in the base graph (i.e., $i \in I_G$) must have hierarchical components. $\text{GlobalGraphSet}$ is the set of all allowable labelling graphs\(^1\).

\(^1\)For the purpose of this definition $\text{GlobalGraphSet}$ may contain any well defined icon–graph.
Figure 4.2: An example of a hierarchical icon–graph.
Hierarchical kon-replacement Grammar

For example, the graph shown in figure 4.2 is defined by:

\[(G_B, \{(i_1, G_H)\})\]

where

\[G_B = (\]

\{i_1, i_2, i_3\},
\{j_1, j_2, j_3, j_4\},
\{(i_1, <j_1>), (i_2, <j_2>), (i_3, <j_3>), (i_4, <j_4>), (i_5, <j_5>), (i_6, <j_6>)\}
\)

\[G_H = (\]

\{i_6, i_7, i_8\},
\{j_8\},
\{(i_6, <>), (i_7, <j_8>), (i_8, <j_5>), (i_9, <j_6>)\},
\{(i_6, <j_8>, j_8), (i_7, <j_6>), (i_8, <j_7>), (i_9, <>), (i_10, <>)\}
\)

4.2.2 Hierarchical Grammars

In order to define hierarchical grammars, the basic model is extended by adding an extra component that allows the generation of hierarchical icon-graphs. This is achieved by associating an icon-replacement grammar with each terminal label. This means that the terminal label may be replaced by any of the graphs generated by that grammar (i.e. graphs that belong to \(\text{Lang(grammar)}\)). This allows icons to effectively contain other graphs, and thus, enables the creation of hierarchical icon-graphs.

Furthermore, we will assume in the sequel that all the grammars used in defining hierarchical graphs belong to the finite grammar set \(GS\).

More formally, a hierarchical icon-replacement grammar is defined by a 5-tuple \((V, N, S, P, H)\), where:

- \((V, N, S, P)\): is an icon-replacement grammar, which we will call the base grammar; and

- \(H: V \setminus N \leftrightarrow GS\): is a relation that associates terminal labels with items belonging to the grammar set \(GS\).

This definition restricts the set of graphs that can be used to define the hierarchical component of a graph. This is true because the relation \(H\) effectively defines the set of graphs that may be associated with a given icon. Hence, the \(\text{GraphAssignment}\) component of a hierarchical graph is now given by:

\(\text{GraphAssignment} : I_G \rightarrow \text{GlobalGraphSet}(I_G)\)
4.2 Hierarchical Icon-replacement Grammar

\[ \text{flowchart} = \{ V, N, S, P, H \} \]

\[ V = \{ \text{block, condition, statement} \} \]

\[ N = \{ \text{block} \} \]

\[ S = \{ \text{block} \} \]

\[ P = \{ (\text{block, block}), (\text{block, statement}) \} \]

\[ H = \{ (\text{condition, condition-grammar}), (\text{statement, statement-grammar}), (\text{statement, flowchart}) \} \]

Figure 4.3: A simple hierarchical icon-replacement grammar.

Where \( \text{GlobalGraphSet}(I_G) \) is the set of all possible labelling graphs for this particular base graph. It is given by:

\[
\text{GlobalGraphSet}(I_G) \triangleq \bigcup_{i \in I_G} \text{graphset}(i)
\]

where:

\[
\text{graphset}(i) \triangleq \{ x : H \mid \text{label } i = \text{LHS } x \cdot \text{Lang(RHS } x) \}
\]

For example, see the grammar defined in figure 4.3. This grammar can be used to generate simple hierarchical flowcharts.

4.2.3 The derivation of hierarchical graphs

The derivation of hierarchical graphs can be divided into two parts:

- Deriving the base graph. This involves applying productions from the base grammar as described in section 3.2.7.
• Deriving the hierarchical components. This involves:

1. Selecting a suitable replacement grammar for each icon; and
2. Replacing the icon’s label with one of the graphs belonging to the language generated by that grammar.

Figure 4.4 shows an example of such a derivation. This derivation is composed of two steps. The first step uses the base grammar to derive the base graph. The second step uses the hierarchical component to derive the hierarchical component. This is shown by replacing the condition label in the base graph by the binary expression tree shown in the figure.

More formally, we say that a hierarchical graph, \( HG = (G, \text{GraphAssignment}) \) is derived from a given hierarchical grammar, \( \Gamma = (V, N, S, P, H) \), if:

• the base graph of \( HG \) (i.e. \( G \)) is derived from \( S \). That is, \( G \in \text{Graphs}(V, N, S, P) \); and
• The hierarchical component is derived from a legal grammar. That is, the following predicate must hold:

\[
\forall i : I_G \mid i \in \text{dom} \ \text{GraphAssignment} \bullet \\
(\exists \gamma \in \{ (\text{label}_G i, x) \in H \bullet x \} \bullet \\
S, \gamma \Rightarrow^* \text{GraphAssignment} i)
\]

4.2.3.1 The Hierarchical Graphs Defined by a Grammar

The set of all hierarchical graphs defined by a hierarchical icon grammar \( \Gamma \), is the set of all graphs generated by it. This set is defined by:

\[
\text{Graphs}(\Gamma) \triangleq \{ \\
G : \text{Graphs}( (V_T, N_T, S_T, P_T) ); \\
G\text{Assign} : I_G \rightarrow \text{GlobalGraphSet}(I_G) \\
\mid \forall (i, g) : G\text{Assign} \bullet g \in \text{graphset}(i) \\
\bullet (G, G\text{Assign})
\}
\]

\(^2\)Applying this step may involve repeating the whole procedure if the \( H \) relation contains hierarchical grammars. It is this recursion that allows the construction of hierarchies of arbitrary depth.
4.2 Hierarchical Icon-replacement Grammar

Figure 4.4: An example derivation of a hierarchical icon-graph. With condition-grammar taken as the grammar of binary expression trees.
4.2.3.2 The Hierarchical Language Defined by a Grammar

The language defined by a hierarchical icon grammar $\Gamma$ is a subset of $Graphs(\Gamma)$ that is restricted to base graphs that do not contain non-terminal elements (i.e. icons with non-terminal labels). The language is denoted by $Lang(\Gamma)$, and defined as:

$$
\text{Lang}(\Gamma) \triangleq \{
G : \text{Lang}( (V_\Gamma, N_\Gamma, S_\Gamma, P_\Gamma ));
GAssign : I_G \rightarrow \text{GlobalGraphSet}(I_G)
\mid \forall (i, g) : GAssign \bullet g \in \text{graphset}(i)
\bullet (G, GAssign)
\}
$$

4.2.4 Modelling hierarchical derivations

Subsection 3.2.8 introduces derivation trees as tool for modelling derivations and evaluating attribute rules. In this subsection this basic model of derivation trees is extended to capture the details of hierarchical derivations.

A hierarchical derivation tree can be conceived as a three dimensional tree over a set of planes, where each plane represents a node in the tree. Furthermore, each of these plane contains a (flat) derivation tree.

The root plane of a given hierarchical derivation tree contains the derivation tree of the base graph. Each one of the other planes will contain the derivation details of one of its hierarchical components.

Figure 4.5 shows an example of a hierarchical derivation tree.

To formalize this further, a hierarchical derivation tree is modeled by a pair: $(\text{BaseTree}, \text{SubTrees})$, where:

- $\text{BaseTree} : \text{Tree}$, is the base derivation tree (i.e. the derivation tree of the base graph) representing the root plane in the hierarchy; and

- $\text{SubTree} : (\text{Node} \times I) \rightarrow \text{Tree}$, define the sub–planes of the tree. It does so by associating icons within certain nodes of the base tree with the derivation trees of the hierarchical components.

Hence, hierarchical derivation trees can be modelled as:

$$
\text{HDT} : \text{Tree} \times ((\text{Node} \times I) \rightarrow \text{Tree})
$$

For example, the derivation tree shown in figure 4.5 is given by:

$$
\text{HT} = (T_1, \{ ((\text{node}_2, i_2), T_2), ((\text{node}_3, i_3), T_3) \})
$$
Figure 4.5: A hierarchical derivation tree example.
4.3 Integrating Textual and Visual Constructs

Textual constructs are added by a further generalization of hierarchical icon-replacement grammars. This is done by extending the grammar set $G_S$ to allow the inclusion of textual grammars as well as icon-grammars.

This effectively allows icons to contain (or be labelled by) strings that are generated by the replacement grammar. This goes beyond simple icon labelling and enables the definition of hybrid languages. Where, the higher level and more abstract constructs can be expressed visually and the more detailed and concrete constructs can be expressed in textual form.

Figure 4.6 shows a simple example of a hierarchical graph which contains textual constructs.

4.4 Attribute Hierarchical Icon-replacement Grammar

As with icon-replacement grammars, it can be stated that hierarchical icon-replacement grammars represent a sound framework for defining the structural aspects of visual languages. It can also be stated that hierarchical icon-replacement grammars alone are not well suited for defining the non-structural aspects of hierarchical visual languages. It is, therefore, necessary to augment hierarchical icon-replacement grammars with mechanisms for defining these "non-structural" aspects. Once more, attribute systems offer a suitable solution.

4.4.1 The definition of attribute hierarchical icon-replacement grammar

To incorporate the notion of attributes in hierarchical icon-replacement grammars the definition will be modified in two ways: firstly, attribute hierarchical icon-replacement grammars will be defined as hierarchies of attribute icon-replacement grammars (as opposed to hierarchies of icon-replacement grammars); and secondly, an interfacing mechanism is defined between the different attribute systems in the hierarchy.

4.4.1.1 A hierarchy of attribute icon-replacement grammars

In section 4.2.2 hierarchical icon-replacement grammars are defined as a hierarchy of icon-replacement grammars. Where a hierarchical icon-replacement grammar is defined as having a base grammar component
Figure 4.6: An example of a hierarchical icon–graph combining textual and visual constructs.
Hierarchical Icon-replacement Grammar

(which is an icon-replacement grammar) and a relation, $H$, that defines the hierarchical component of the grammar. In order to include the notion of attributes, an attribute hierarchical icon-replacement grammar is also defined as having corresponding components. That is, a base grammar and a hierarchy definition relation. However, in order to include the notion of attributes, the base grammar as well as the hierarchical components are taken to be attribute icon-replacement grammars.

More formally, an attribute hierarchical icon-replacement grammar is defined as a 7-tuple $(V, N, S, P, Attr, ACR, H)$, where:

- $(V, N, S, P)$ is the base icon-replacement grammar;
- $(Attr, ACR)$ is the attribute system associated with the base grammar;
- $H : V \setminus N \leftrightarrow AGS$, is a relation that associates terminal labels with items belonging to the grammar set $AGS$. Where $AGS$ is the set of all attribute grammar used in defining attribute hierarchical icon-replacement grammars.

4.4.1.2 Interfacing the attribute systems

Interfacing the attribute systems is essential for integrating the different levels in the hierarchy. For example, to compute the semantics of the hierarchical flow chart shown in figure 4.6, it must be possible to pass some information from the deeper level in the hierarchy to the basic level. Furthermore, to compute the layout and positioning information of the inner layer some information must be passed down from the basic level.

Clearly, a link already exists between the various grammatical levels; which is the hierarchical component of the grammar. For example, the hierarchical component of the grammar shown in figure 4.3 associates the terminal element $condition$ with another grammar called $conditionGrammar$.

The hierarchical derivation tree model—our tool for computing attribute values—makes this interface more explicit. This is because it relates a given symbol, within one of the nodes of the base tree, with the root of the derivation tree of a hierarchical component.

For example, the derivation tree of figure 4.5 relates icon $i_2$, which is within node $n_2$, to the derivation tree $T_2$.

This relation between the various grammatical levels will be used to provide the interface between the attribute systems. This is achieved by linking the attributes of the terminal element of the base grammar (denoted by $t_b$) with the attributes of the start symbol of its hierarchical component (denoted by $S_h$). Furthermore, the attributes are linked so that information can be transferred bottom up, from the hierarchical components to the
base components, as well as top down, from the base components to the hierarchical ones.

The interface is established by two associations, namely:

1. associating the intrinsic\(^3\) attributes of \(t_b\) with the synthesized attribute of \(S_b\) (providing bottom up information flow); and

2. associating the intrinsic attributes of \(S_b\) with the inherited attribute of \(t_b\) (providing top down information flow).

This simply means that we may write computation rules to overwrite the values of the intrinsic attributes.

From the point of view of attribute evaluation, this association combines \(t_b\) and \(S_b\) into one symbol, and, thus, effectively flattens the derivation tree.

Figure 4.7 shows an example of such a flattened derivation tree; it corresponds to the hierarchical derivation tree shown in figure 4.5.

This flattening of derivation trees allows us to evaluate the attribute rules in the same manner described in section 3.3.3.

### 4.5 Summary

This chapter proposes hierarchical icon-replacement grammars as the basis for defining hierarchical visual languages that allow the integration of textual and visual constructs.

Related research and development projects have been described in [KG90, RW91]. However, the issues are addressed from the point of view of defining a certain visual language and its environment rather than defining a formal model for describing visual languages in general.

This chapter also defines an attributed version of the hierarchical model. This attributed version allows for the definition of the non-structural aspects of hierarchical languages, as well as enabling the integration of different languages (and their corresponding attribute systems) into a single framework.

---

\(^3\)Intrinsic or initialized attributes are special attributes that do not have attribute computation rules, but instead have an initial value associated with them.
Figure 4.7: A flattened hierarchical derivation tree.
Chapter 5
Syntax-directed editing for Icon-replacement languages

"A software tool that is used to aid in producing and modifying source programs under development." [LS86]
5.1 Introduction

In Chapters 3 and 4 a model for defining Icon-replacement languages was presented.

This chapter addresses a complementary problem. That is, *How can this model be used to define tools and environments for system development?*

More specifically, it addresses the problem of using such a definition to construct language-based environments. These environments would typically allow developers to manipulate and transform their representations of the system under development. Ultimately, such a development environment would be composed of a complete developer workbench that would include editing tools for entering, editing and presenting (displaying) system representations, as well as tools for transforming and checking the validity of these designs. An example of such an environment might be an entity relationship diagram editor that would allow a developer to enter and manipulate such diagrams and would also be capable of transforming such diagrams into alternative representations such as predicate logic. An alternative example might be a logic diagram editor, which would allow the developer to enter logic diagrams and then transform them into $\lambda$-expressions denoting the corresponding mathematical functions.

This chapter proposes a specification of such an environment. Most aspects of this environment are syntax-driven and are, hence, based on the underlying syntax definition of the language. Activities such as creating and editing language constructs are directly based on, and driven by, the underlying language syntax. While other activities, such as transformations and layout computation are based on the language's attribute system, and hence, are indirectly dependent on the syntax of the language.

However, before going into the characteristics of the solution proposed here, it is beneficial to reiterate the goals of this environment and to briefly examine the different alternatives to realizing these goals.

Such a development environment has two main functions:

1. To act as a medium for entering and documenting system descriptions at various designs and specification phases; and

2. To provide a framework for possible transformations and validity checks on the system description. This is usually accomplished by recognizing (parsing) the concrete language constructs and transforming them into some abstract form (abstract syntax) which is more amenable to transformations and semantic analysis.

In the sequel, different alternatives to constructing such an environment are discussed. Two alternatives are examined, namely the traditional batch
5.2 The Edit–Parse approach

A visual language based development environment modelled according to the edit–parse model would typically have two components:

1. A generic graph–editor. This is a tool for entering and editing language constructs. It typically possesses no knowledge of the language being edited and, hence, merely performs as a drawing editor; with the possibility of including some generic form of graph processing, such as producing pretty graph layouts. Examples of such graph–editors include the graph editor EDGE [Pau90] and variants of the editors constructed using the Unidraw toolbox (such as idraw drawing editor)[VL, Vli90].

2. A graph–parser. This is typically a tool (programme) that reads the data captured by the graph–editor, checks its validity, and, ultimately, performs some sort of transformation on this data. Such transformations usually aim at producing the derivation tree of the given graph or some such transformation. Examples of graph parsers and parsing algorithms have been described in [WWT91, Gol91a].

Figure 5.1 shows an example of a system diagram of an environment constructed using the edit–parse model.

The edit–parse approach is generally considered useful because of its flexibility and because of the clear separation of tasks. Its flexibility, from a design point of view, mainly stems from the nonrestrictive nature of generic graph–editors. This allows system developers to enter and change intermediate designs that may not be strictly correct or complete according to the design language; this is often useful as part of the design (thinking) process.
Although the separation of tasks provides a desirable framework to conceptuallyize and understand such an environment; it poses some fundamental difficulties when processing the output graph. This is due to the intrinsic difficulty of graph parsing (recognition).

In general, the graph parsing or recognition problem is attempting to verify if a given graph \( G \) can be generated from a grammar \( \Gamma \). In other words, it is attempting to prove that \( G \in \text{Lang}(\Gamma) \). To prove this is equivalent to asking the question: are any of the graphs in \( \text{Lang}(\Gamma) \) isomorphic to \( G \). More formally:

\[
\exists x : \text{Lang}(\Gamma) \bullet \text{isomorphic}(G, x)
\]

The problem of graph–isomorphism is, in general, a difficult one. It is interesting from the theoretical complexity point of view, because of its unknown complexity status. Since graph–isomorphism is not known to be in \( P \) and yet it possesses properties that seem to make it unlikely to be \( \text{NP–complete} \). Hence, it is thought to be of intermediate complexity [Hof82]. From a practical point of view, however, graph–isomorphism remains too complex for efficient processing. This makes such parsers inherently inefficient and awkward to use. Naturally, heuristics and special cases can be found to make graph parsing more efficient and acceptable to use. Examples of such implementations include the RL system [Wit92] and visual language parser presented in [Gol91b, Gol91a].

Other disadvantages of using the edit–parse approach include: (1) the lack of immediate feedback from the parser, since the user must wait until the end of the editing session before such feedback is available; and (2) the inherent drawback of using a generic (language unaware) editor is that many of the possible visual abstractions and layout constraints are language dependent rather than generic, and, hence, cannot be supported in a language unaware editor.

5.3 The Syntax–directed editing approach

The limitation and difficulties in applying the edit–parse approach have led to investigating the more interactive syntax–directed editing approach.

In an syntax–directed editing environment the user manipulates \emph{language–constructs} which are based on the syntax of the language. To compare it with the edit–parse approach one can think of syntax–directed editing as synchronized editing and parsing. That is, whenever a change is made to the graph being edited it is reflected in the state of the parsing process and its output (e.g. a derivation tree). It also seeks to take advantage of the user
knowledge of the language being used. Thus reducing the amount of effort that has to be spent on recognizing graph (sub) components.

5.3.1 Syntax-directed editing models

Before examining the editing model proposed for icon-languages we briefly examine the different approaches taken to syntax-directed editing. Most of these approaches have been applied to textual and visual languages.

5.3.1.1 The template model

The template model is based on the ideas of templates and placeholders. Templates are language constructs which are derived from the grammar's production rules. Templates contain placeholders or holes that are to be filled by other templates or by some final text or graph.

For example, in a textual language a template for an if-statement might look like:

```plaintext
if condition-place-holder
  then statement-place-holder
else statement-place-holder
```

Figure 5.2 shows a similar template example for an icon-language.

Placeholders indicate the class of components that might replace them. For instance, in the above example, the `statement-place-holder` may be replaced by valid statement template such as a `while-statement` template.

In a template-based editor, the starting point for creating a new program or document is the grammar's start symbol. Typically, the user is prompted with a template representing the grammar's start symbol; the user may then proceed by replacing placeholders with the appropriate templates. This activity may continue until the placeholders can no longer be expanded and the only allowable constructs are terminal symbols.

This activity (expanding placeholders) is often guided by menus (or some similar selection metaphor). This ensures that the constructs being entered are always syntactically correct and eliminates the need for remembering details of the concrete language syntax.

Template-based editors are useful for creating new programmes (documents). This is because template expansion is relatively easy and allows for fast creation of correct language structures. However, editing and changing existing constructs can be somewhat awkward, especially if replacing one construct with another causes all information belonging to the original construct to be deleted. This is often remedied by providing a sort of cut-and-paste mechanism [Log88]. Another solution proposed in [AHW90] is
Figure 5.2: An example of a visual if-then-else template.
to enrich the language's production rules with special rules that can be used in such substitutions.

5.3.1.2 The token model

The token model of syntax-directed editing is based on the idea of re-synchronizing the parse state with editing changes that are made in the document. This is done by continually checking the constructs being entered. Incremental parsers are often used to avoid large delays resulting from re-parsing the complete program (document) each time a change is made. The main problem with this approach is the difficulty in constructing fully incremental parsers due to the amount of administrative work required to keep track of the parse status [Log88]. Language specific token based editors have also been constructed that take advantage of particular language features to avoid full implementations of incremental parsing [EK85].

Apart from Syntax checking functions, token based editors can also provide other useful functions such as token and keyword completion. Furthermore, the token model can be combined with templates to provide template extension based on token identification.

5.3.1.3 Hybrid editing models

Hybrid models seek to combine features from the template, token and generic editing models. A common example of this is the combination of free editing with template based editing [Log88]. In such hybrid editors a focus is used to delimit an area in which free editing can be performed. Another common combination is having two separate modes: a template mode, and a free editing mode. The user may switch from template editing to free editing and perform typical generic editing tasks. When the editor is switched back to the template mode a parser is used to check the document and update the parse tree.

Another example is combining token and template editing. This can be done to provide template extension based on token identification.

In general, hybrid editors seek to find a successful compromise and to combine the best of the different editing models. There are a number of examples of hybrid editors for textual languages (see [Log88] for an overview and comparison of such editors), however, such editors are less common for visual languages.
5.4 Syntax-directed editing for Icon-replacement languages

The following sections describe the scheme introduced in this thesis for the syntax-directed editing of Icon-languages. This approach combines the use of templates, incremental and free form editing, and novel approaches to template substitution.

5.4.1 Interactive derivation

Creating graphs is one of the fundamental functions of an editor. The editor is seen as a derivation engine that starts with an axiom (i.e. the starting graph of the grammar), and attempts to derive a final graph (i.e. a graph without non-terminal elements). When the derivation arrives at a point where a choice has to be made, the user is asked to select one of the possible options.

Each time a production rule is selected for application, the derivation tree is updated. So that it always reflects the current state of the derivation. Attributes are also computed upon rule selection, and based on the derivation tree.

In general the user can influence the derivation process in two ways: (1) pointing to a non-terminal symbol (i.e. an Icon labelled with a non-terminal) to be replaced; and (2) selecting the production rule to be applied on it.

For a given Icon-replacement grammar, \( \Gamma \), and a graph, \( G \in Graphs(\Gamma) \), the derivation process can be modelled as follows:

1. the selection of a non-terminal (or hole/place-holder) to be replaced is given by the function:

   \[ P \text{ \texttt{\textmd{oint}}} : P \rightarrow IS \]

   The \texttt{Point} function actually models the user's selection of the non-terminal Icon to be replaced.

2. once an Icon has been selected, the system constructs a set of possible derivation alternatives. These derivation alternatives are referred to as the menu set. The construction of the menu set is given by:

   \[ Menu : IS \rightarrow P \rightarrow IS \]

   \[ Menu \triangleq \lambda i : IS \cdot \]

   \[ \{ (l, R) : P \mid label i = l \land OrderG i = Order R \} \]
3. the user's second action is to select a production rule to be applied. It is given by:

\[
\text{Select} : P \rightarrow P_1
\]

4. after the selection is made, the resulting graph is given by:

\[
\text{Replace}(G, \text{Point } I_0, \text{RHS Select Menu Point } I_0)
\]

This process is referred to as interactive graph derivation. It is especially useful for creating new graphs, because it allows fast graph creation with relatively little effort on the user's part (since it does not require the user to specify all the graph details), it also guarantees syntactic correctness. Interactive graph derivation is similar to the template model of syntax-directed editing.

Figure 5.3 shows an example of interactive graph derivation. Figure 5.3 (a) illustrates the starting point of the derivation process, where the user is prompted by the starting graph of the logic diagram grammar, and the Rule Menu shows a list of graphs that can replace the selected icon (enclosed in the dotted rectangle). Figure 5.3 (b) shows the result of applying one of the rules. Figure 5.3 (c) shows a further derivation step.

5.4.1.1 Partial-graph derivations

Although graph derivations are considered useful for creating new graphs, it has the restrictive that the generated graph must be correct and complete according to the grammar definition. The graph must be complete in the sense that it must be derived from the grammar's start symbol and, of course, must be syntactically correct. In a final design document or programme this must clearly be the case, however, in intermediate stages it is often desired to produce partially correct graphs. That is, graphs that are not generated from the start symbol but from some arbitrary production rule selected by the user. This is thought to add a certain degree of flexibility to the derivation process and allows the user to experiment with substructures.

The process of partial derivation is similar to interactive derivation except for the fact that it starts from the right hand side of an arbitrary derivation rule.

The derivation tree generated to represent such a partial derivation is called a partial derivation tree. Unlike complete derivation trees, the root of such a partial tree is the right hand label of the production rule used to start this particular derivation.

\(^1\)An example from the world of programming would be to start by writing a correct statement and then building the rest of the programme's structure. Of course, the statement alone may not be valid as a complete programme or language structure.
Figure 5.3: An example of the process of interactive derivation.
5.4 Syntax-directed editing for Icon-languages

5.4.1.2 Combining partial-graph derivations

To make partial derivations useful in the final goal of creating a complete graph, it is necessary to be able to combine partial derivations to construct a complete graph.

To combine partial derivations we directly manipulate language structures in the form of their derivation trees. Derivation trees may be combined to produce other partial derivations, until, ultimately, a complete derivation tree is formed. In general, each combination step involves two derivation trees: a root tree and a leaf tree. The two trees are combined in such a way that the root of the leaf tree is linked to one of the leaf nodes of the root tree. The trees must be linked through a valid production rule. That is, the production rule used to construct the root of the leaf tree must be applicable to the linking leaf node of the root tree.

Figures 5.4, 5.5 and 5.6 show an example of partial derivation tree combination. The tree in Figure 5.4 shows a derivation starting at the start graph of the logic diagram grammar. Figure 5.5 shows a partial derivation that begins with the right hand side of a production rule. Finally, figure 5.6 shows one of the ways in which the trees can be combined to produce a valid derivation tree.

In general, the process of tree combination must be user guided. This is due to the fact that there may not be a unique way in which two trees may be combined. This is due to the fact that any given production rule may be applicable to more than one leaf node in the root tree. Hence, the root production of the leaf tree must be applicable to one of the leaf nodes in the root tree.

Before formally defining a function for combining trees it is useful to define some auxiliary functions. The first function, \texttt{leafs}, accepts a derivation tree as an argument and returns a set nodes which are the leaf nodes of this tree. More formally:

\[
\text{leafs} : \text{Tree} \rightarrow \mathcal{P} \text{ Node}
\]

\[
\text{leafs}( (n, s) : \text{Tree} ) = \begin{cases} 
    \varnothing & \{ n \} \\
    s \neq \varnothing & \bigcup \text{leafs} \mathcal{t} \\
    t \in s & 
\end{cases}
\]

Furthermore, not all leaf nodes can be used as the connecting point. This is because the root production may not be applicable to each node. It is therefore possible to select a subset of the leave node to which the production may be applied. If this subset is empty then that indicated that these two trees cannot be joined.

A function, \texttt{Applicable}, is defined. \texttt{Applicable} accepts a set of nodes and a production rule, and return a the subset of nodes to which the production
is applicable. Applicable is given by:

\[
\text{Applicable} : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}
\]

\[
\text{Applicable} \triangleq \lambda N : \mathbb{P}; (l, G) : \mathbb{P} \cdot
\]

\[
\{ ((l, R), i) : N \mid
\exists x : I \cdot
\text{label}_R x = l \land \text{order}_R x = \text{order} G \}
\]

From the function \text{Applicable} two cases that do not need user input can be deduced:

1. The case where the application of \text{Applicable} returns an empty set, and hence the two trees cannot be joined; and

2. The case where there is a unique joining point and hence the joining can be done without user intervention. This case may arise when there is exactly one node in the set return by \text{Applicable}, and that node contain exactly one icon to which the production in question may be applied.

The condition for the second case is given by the predicate \text{Unique} which is defined as:

\[
\text{Unique}(\text{nodeSet} : \mathbb{P}; (l, G) : \mathbb{P}) \triangleq \#\text{nodeSet} = 1 \land
\]

\[
\exists_1 x : I \cdot
\text{label}_R x = l \land \text{order}_R x = \text{order} G
\]

In the general case, however, user input is necessary to produce an unique tree combination. The user must select an icon within one of the nodes in the set given by \text{Applicable}. The user’s actions are modeled by the node selection function \text{iconSelect}. It is given by:

\[
\text{iconSelect} : \mathbb{P} \rightarrow (\mathbb{Node} \times \mathbb{I})
\]

Once a unique icon is selected the two derivation trees can be combined. This is done by inserting the leaf tree as one of the descendants of the selected node. This is done by redefining the root node of the leaf tree to reflect the fact that it represents an application of a production rule to the selected icon. The modified node is then added to the descendant node list. The final act of combining the two trees is defined by the function \text{Combine}. It accepts three arguments: the first argument is the root tree \(T_r\), the second is the leaf tree \(T_l\), and the third is the selected node and icon pair at which the leaf tree is to be inserted.
5.4 Syntax-directed editing for Icon-languages

*Combine* is defined by:

\[ \text{Combine} : \text{Tree} \times \text{Tree} \times (\text{Node} \times I) \rightarrow \text{Tree} \]

\[ \text{Combine} (T_r, T_l) = \text{Replace}(T_r, \text{Subtree}(T_r, n), \text{Attach}(\text{subtree}(T_r, n), i, T_l)) \]

*Combine* makes use of three auxiliary function: *Subtree*, *Replace*, and *Attach*. The three functions are defined as follows:

- The function *Subtree* accepts two arguments: a tree and a node within that tree. It returns the subtree rooted at this node.

- The function *Attach* accepts three arguments: a host tree, an icon which is present in the host tree's root node, and the leaf tree to be attached at that icon. *Attach* is defined by:

\[ \text{Attach} : \text{Tree} \times I \times \text{Tree} \rightarrow \text{Tree} \]

\[ \text{Attach}((n, s) : \text{Tree}; i : I_n; ((p, i_0), s_1) : \text{Tree}) = \]

\[ (n, s \bigcup ((p, i), s_1)) \]

- *Replace* accepts three arguments: a tree, a subtree with that tree, and a replacement of that subtree. Both the subtree and its substitution tree must have the same root node.

5.4.2 Editing

Interactive derivation alone is not a sufficient basis for syntax-directed editing. Since creating graphs is only part of the editing process. The user will often need to modify, and change language constructs. Therefore, it is important to consider methods of graph modification.

Because of the structure based nature of the editing process presented here, two types of graph modifications are considered: those effecting partial structures and those effecting complete structures. Where a complete structure is defined as the complete (sub) graph produced by applying a production rule (i.e. the graph corresponding to the *RHS* of a production), and a partial structure is only a part of the produced structure. Figure 5.7 shows examples of complete and partial structures.

5.4.2.1 Editing complete structures

Complete structure editing operations manipulate such a structure as a whole. Such operations can be divided into two broad categories: undoing or cancelling previous derivation steps, and structure substitution.
Figure 5.4: A derivation tree starting at the start graph of the logic diagram grammar.
Figure 5.5: A partial derivation beginning with the right hand side of a production rule.
Figure 5.6: Tree combination.
Figure 5.7: Examples of partial and complete structures.

Cancelling a derivation step (which produced the complete structure in question) is also referred to as complete structure deletion. It is viewed as the reverse of the derivation operation. That is, attempting to reproduce the non-terminal whose expansion introduced the deleted construct.

This is implemented by maintaining (backwards) pointers from the graph constructs to the derivation tree. The effect of such an operation on the displayed graph would be the substitution of the complete structure with the non-terminal element, i.e., just as it was before applying the production.

Complete structure substitution or replacement is the act of replacing such a structure with another. This can be broken down into two steps: (1) cancelling the derivation step which produced the complete structure; and (2) the derivation process re-performed on the restored non-terminal element to yield the replacement structure.

Complete structure editing as presented here provides a solution to one of the basic problems of template based syntax-directed editing, which is the problem of replacing one language construct with another without losing common information. The solution proposed here can be seen as a compromise between the low level approach of providing additional cut-and-paste windows cited in [Log88], and the approach presented in [AHW90] which adds artificial production to the language in order to enable easy transformations (however, distorting the language in the process). Complete structure editing provides a generic and language independent model for structure substitution: any complete structure, \( S_1 \), can be changed into any other structure, \( S_2 \), provided that \( S_2 \) can be derived from the symbol \( S_1 \) was derived from.
5.4.2.2 Editing partial structures

Recovering after a partial structure modification involves checking the correctness of the resulting graph, and, if it is correct, reconstructing the derivation tree to reflect the new state of the derivation. Checking the correctness of the resulting graph can be done as follows:

1. identifying the smallest complete structure that includes the partial structure, which implies identifying the production rule and the replaced icon (i) that produced the structure;

2. constructing the set of all possible replacements of that structure, which is given by Menu i; and

3. finally checking if the user edited version belongs to that set.

The third step implies checking where there is a graph in Menu i that is isomorphic to the one entered by the user2.

Icon-graph isomorphism. Two icon-graphs are said to be isomorphic if there exists a connection preserving mapping between the icons and joints of both graphs. This means that two icon-graphs, $G_1 = (I_1, J_1, in_1, out_1)$ and $G_2 = (I_2, J_2, in_2, out_2)$, are said to be isomorphic if there exits two mapping (i.e. bijective functions) $f_I : I_1 \rightarrow I_2$ and $f_J : J_1 \rightarrow J_2$ such that the following predicate holds:

$$\forall i : I_1 \bullet$$
$$\forall n : 1 \ldots \#(in_1 i) \bullet (in_1 i) n = f_I \; in_2 \; f_I i$$
$$\wedge$$
$$\forall n : 1 \ldots \#(out_1 i) \bullet (in_1 i) n = f_J \; out_2 \; f_J i$$

This means that the mapping $f_I$ and $f_J$ are such that they preserve the graph configuration.

Checking icon-graph isomorphism. Following an outline of the algorithm for detecting the isomorphism of two icon-graphs is given. The algorithm is influenced by ideas used in the group-theoretic approach to graph isomorphism[HoF82].

The algorithm attempts to construct an icon-mapping between the two graphs. This is basically done by considering different permutations of the

---

2It is clear that if such an isomorphism exists, it will be unique. This is implied by the icon-replacement grammar definition, since, by definition the production set may not include duplicate productions.
icon set $I_2$, where each permutation is taken to indicate a mapping from $I_1$ to $I_2$. For each of these permutations, an attempt is made to construct a joint mapping from $J_1$ to $J_2$. If such a joint mapping can be constructed then the two graphs are isomorphic.

Clearly, a simple precondition for testing isomorphism would be that both graphs have the same number of icons and joints, that is:

$$\#I_1 = \#I_2 \land \#J_1 = \#J_2$$

Furthermore, it is clear that considering all icon permutations might not be feasible. Since, considering icons having different orders will not yield a correct mapping.

Hence, the algorithm can be divided into three main sections: (1) dividing the icon sets into subsets of equal order; (2) constructing the permutations (and associated mapping) of each of the subsets of $I_2$; and (2) for each of these permutations attempt to construct the joint mapping.

**Dividing the icon sets.** Dividing the icon sets into subsets of equal order yields a number of disjoint icon sets. For the given graphs ($G_1$ and $G_2$) we say that the icon sets can be divided into $m$ subsets such that:

$$I_1 = \bigcup_{k:1..m} I_{1,k}$$
$$I_2 = \bigcup_{k:1..m} I_{2,k}$$

The sets must be disjoint and contain icons of equal order, hence:

$$\forall k:1..m \bullet \forall i,j : I_{1,k} \bullet$$
$$\text{order } i = \text{order } j \land$$
$$\forall k, l : 1..m \bullet$$
$$k \neq l \Rightarrow I_{1,k} \cap I_{1,l} = \emptyset$$

The same must also hold for $I_2$.

Furthermore, before proceeding to the next step we must ensure that the corresponding subsets of $I_1$ and $I_2$ have an equal number of elements. That is:

$$\forall k : 1..m \bullet \#I_{1,k} = \#I_{2,k}$$

In general, the construction of the subsets can be constructed in a number of steps equal to the number of icons in a graph. That is: $\# I_1$. 
Syntax-directed editing for Icon-replacement languages

Constructing the icon map. The construction of a mapping between corresponding icon subsets is done by considering possible permutations of one of the icon subsets. Hence, for each corresponding pair \( I_{1,k} \) and \( I_{2,k} \) we can construct \((#I_{2,k})!\) different permutations, and hence mapping from \( I_{1,k} \) to \( I_{2,k} \).

For \( m \) icon subsets the number of possible mapping to be considered is given by:

\[
\prod_{k=1}^{m} (#I_{2,k})!
\]

The simplest (most efficient) case would occur if each of the icon subsets contained exactly one icon. In this case there would be exactly one possible mapping from \( I_1 \) to \( I_2 \). The most complex case would occur if all icons were of the same order, and, hence, it would not be possible to subdivide the icon sets. In this case the number of possible mapping to consider would be: \((#I_1)!\).

Constructing the joint mapping. For each of the icon mapping (denoted by \( f_I : I_1 \rightarrow I_2 \)) constructed in the pervious step an attempt is made to construct the joint mapping. The algorithm will halt when either a valid joint mapping is constructed, or when all icon mapping are tested unsuccessfully.

To proceed with constructing the joint mapping, a relation, \( R_{I_1,I_2} \), between \( J_1 \) and \( J_2 \) is constructed. This relation is constructed by associating joints connected to icons in \( I_1 \) to joints connected to the related icon in \( I_2 \) (given by the \( f_I \) mapping).

More formally, \( R_{I_1,I_2} \) is given by:

\[
R_{I_1,I_2} \triangleq \bigcup_{i \in I_1} \{ x : 1 .. #(in_1 i) \bullet ((in_1 i) x, (in_2 (f_I i)) x) \} \cup \{ x : 1 .. #(out_1 i) \bullet ((out_1 i) x, (out_2 (f_I i)) x) \}
\]

if \( R_{I_1,I_2} \) is a bijective function from \( J_1 \) to \( J_2 \), then a joint mapping, \( f_J \), has been found and is equal to \( R_{I_1,I_2} \).

Labelled icon-graph isomorphism Two labelled icon graphs \( G_1 = (I_1, J_1, in_1, out_1, label_1) \), and \( G_2 = (I_2, J_2, in_2, out_2, label_2) \) are said to be isomorphic if their underlying unlabelled graphs are isomorphic and the isomorphism preserves labelling. That is, the following predicate must hold:

\[
\forall i : I_1 \bullet label_1 i = label_2 f_I i
\]
The algorithm for checking labelled icon-graph isomorphism is similar to the one presented for unlabelled graphs, except that the icon sets are divided into subsets of equal order and equal label values.

Notes:

- Although the algorithm presented above is not efficient for worst case analysis, it remains useful for typical graphs and when applied within the application domain described above. Hence, a good language (grammar) design heuristic would be to limit the size of graphs appearing in the right hand side of production rules.

- There exists a number of more efficient graph isomorphism algorithms for restricted forms of graphs (see, for example, [Hof82, Mil83, Luk80]). However, since we are dealing with potentially small graphs the decision has been made not to restrict the class of allowable graphs in the grammar.

- Several improvements of the current algorithm are possible. For example:
  - The proposed icon subsets could be further divided into subsets based on icon branch valance. Where an icon branch valance is the number of branches (belonging to other icons) that are connected to this particular branch. This would eliminate the need to consider a number of useless permutations.
  - Eliminating permutations by examining the icon connection patterns within the graph.

5.4.2.3 Derivation tree editing

The editing of derivation trees can be viewed as an alternative method of graph modification. In addition to allowing the user to manipulate the icon-graph being created, derivation tree editing allows for modifying the derivation tree, or, alternatively put, to modify the derivation itself.\(^3\)

In general, a derivation can be modified in three ways:

\(^3\)It is worth noting that some of the effects of derivation tree based editing can be achieved using the editing interface described in section 5.4.2. However, it is asserted that the availability of such alternative means of accomplishing the editing tasks is essential to increase the usability of the system. Since at many points it is not the question of what is possible, but rather that of how convenient or intuitive is the means of accomplishing it.
• sub-derivation deletion, where the application of a given production (or sequence of productions) is eliminated from the derivation process (undoing a production sequence);

• sub-derivation copying or insertion, where a sub-production sequence is repeated or inserted at a given point; and

• rule substitution or replacement, where an alternative production rule is applied to a given node.

5.5 Syntax-directed browsing

Apart from creating and editing graphs, editors are also used as tools for viewing and browsing documents. Functions such as zooming and creation of multiple views, are examples of such useful utilities. Such functions are usually thought of as language independent. However, language dependent constructs, such as derivation trees, can provide an attractive basis for such functions. For example, zooming functions, such as displaying a complete structure as a single icon, can be defined on the basis of the structure of the derivation tree. For example, a complete structure can be displayed as the non-terminal from which it was derived. Furthermore, it is possible to create multiple views by assigning different display windows to subtree of the original derivation tree.

5.6 Syntax-directed presentation

Graph presentation problems are those problems dealing with automatically producing nice graph layouts.\(^4\)

Such automated layouts, although difficult to produce, are quite important especially when dealing with large and complex graphs, or when dealing with graphs generated from other systems.

In general layout problems can be divided into two classes:

1. Generic, language independent, problems. Such problems include:
   layout computation problems (e.g. minimum branch crossing), and other presentation and editing related problems such as layout stability; and

\(^4\)Computing optimal graph layouts is a well known NP-complete problem.
2. Language dependent problems. Such special layout constraints that are desirable in a certain language. For example, in a logic diagramming language it is desirable to keep all input connection on one side of the display and all output connection on the opposite side.

Generic layout problems are thought of as lower in level and are common to all icon-languages. Therefore, such problems are handled by a generic graph layout engine.

An interesting solution to generic layout construction has been proposed in [Pau90]. This solution is based on combining layout algorithm with user defined layout constraints, where layout constraints are linear relations among icon coordinates. The generic layout algorithm is combined with user constraints by translating the layout produced by the algorithm into a set constructs. A constraint manager is used to reconcile the constraints from the user and the layout algorithm, and conflicts are eliminated by deactivating constraints with low priority (user supplied constraints have higher priority by default).

This approach can be incorporated with in the syntax-directed editing environment by adding layout constraints that are specified in the grammar in the form of attributes (see, for example, figure 3.8). Figure 5.8 shows how the scheme presented in [Pau90] can be integrated in the syntax-directed editing environment.
5.7 Summary

This chapter proposes syntax-directed editing as the basis for constructing support environments for icon-languages. The syntax-directed editing model presented here employs: interactive derivation of partial and complete structure; structure substitution through re-derivations; free from editing; derivation tree based editing; and syntax-directed presentation.
6.1 Introduction

In Chapters 3 and 4 a model for defining *Icon-replacement languages* was presented. The goal of this model is to provide a framework for defining the syntax and semantics of such languages. That is, constructing formal definitions of the so-called *Icon-replacement languages*. 
Chapter 5 used the infrastructure put down in previous chapters to address certain aspects of the complementary problem of defining system development environment and tools. The issues addressed in chapter 5 mainly concentrated on syntax-directed editing and presentation of language constructs.

This chapter is also concerned with applying the icon-replacement grammar model in development environment and tool construction. However, it is concerned with a different aspect of such environments, namely the construction of systematic methods for translation. The translation methods presented here may be used for icon-language-to-icon-language translation as well as string-to-icon-language and icon-language-to-string translation. The translation method is directed towards applications in construct visualization and transformation. It is based on the idea of pair-grammars originally presented in [Pra71].

Unlike traditional syntax-directed translation methods, which are based on attribute systems that operate on derivation trees, pair grammars are based on the idea of paring two grammars. These grammars are paired by linking the production rules and non-terminals of the two grammars. By doing so, a one-to-one structural transformation system can be constructed (without having to go through attribute rules). Such a system would be quite useful in constructing visualizations of system constructs.

Although it is quite possible to construct an attribute system that transforms one language into another, in the case of language with equivalent syntactic structure it is somewhat artificial (and unnecessary) to resort to attributes. This is because attributes are basically means for attaching semantics to language structures, and have traditionally been used to close some semantic gap between two languages. Furthermore, adding such semantic rules to the attribute system would clutter it with unnecessary information.

What is essentially needed is a system that would effectively allow different concrete syntax to the same abstract one. In other words, we need to be able to produce different concrete representation of what is essentially the same language. Pair grammars can be effectively used for such systems.

6.2 Pair Grammars

6.2.1 Basic Concepts

The idea of pair grammars is based on the concept of having two grammars which contain equivalent (related) production rules (and therefore constructs). These equivalent production rules are linked together to indicate their correspondence. Furthermore, non-terminals appearing within
Figure 6.1: A pair grammar example.
these production rules are linked together.

A pair grammar can be viewed as being composed of two grammars and a function linking corresponding production in the two grammars. The language generated by such a grammar is a set of icon–graph pairs. Therefore, for an unambiguous pair grammar, the language generated by such a grammar can be considered as a translation map from the left hand language into right hand language. Furthermore, if we substitute some string grammar instead of the icon–grammar, we can obtain a transformation from a string language into an icon–language and vice versa. For example, figure 6.1 shows a pair grammar linking part of a Pascal–like grammar to a flowchart grammar. Such a grammar can be used to construct visualizations of Pascal programs in terms of of structured flowcharts.

In the remainder of this section pair grammars and their properties are examined and a formal model is presented. This model is then used to discuss and illustrate the utility of pair grammars in visualization.

6.2.2 The Pair Grammar Model

A pair grammar is typically composed of two icon–replacement grammars and a mapping (paring) function. The mapping function determines how the production rules of the two grammars are linked; it also determines how the non–terminal symbols (in the linked rules) are linked.

More formally, a pair grammar is a 3–tuple \((\Gamma_l, \Gamma_r, \mathcal{P})\), where:

- \(\Gamma_l, \Gamma_r\) are icon–replacement grammars representing the left hand side and right hand side grammars, respectively; and

- \(\mathcal{P}\) is the pairing or linking function, and is given by:

\[
\mathcal{P} : P_{\Gamma_l} \rightarrow (P_{\Gamma_r} \times \mathcal{H})
\]

Where \(\mathcal{H}\) defines how non–terminals within paired production are linked. It is given by:

\[
\mathcal{H} = Pn_{\Gamma_l} \rightarrow Pn_{\Gamma_r}
\]

Where \(Pn\) are the production non–terminals\(^1\). And the set \(Pn\) is given

\(^1\)Since a non–terminal label may occur more than once in a given production, it is important to distinguish between such multiple occurrences to enable an unambiguous (unique) paring of non–terminal labels. Hence, the use of \(Pn\) instead of simply using \(N\).
6.2 Pair Grammars

\[ C = ( V_c, N_c, S_c, P_c ) \]

\[ V_c = \{ C, x \} \]

\[ N_c = \{ C \} \]

\[ S_c = \]

\[ \]

\[ P_c = \{ p_1, p_2 \} \]

Where:

\[ p_1 = ( C, \]

\[ x \]

\[ ) \]

\[ i_1 \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]
$T = (V_t, N_t, S_t, P_t)$

$V_t = \{ T, t \}$

$N_t = \{ T \}$

$S_t = \text{\begin{tikzpicture}
\node[circle, draw] (T) at (0,0) {T};
\end{tikzpicture}}$

$P_t = \{ p_3, p_4 \}$

Where:

$p_3 = (T, \text{\begin{tikzpicture}
\node[circle, draw] (t) at (0,0) {t};
\end{tikzpicture}})$

$p_4 = (T, \text{\begin{tikzpicture}
\node[circle, draw] (t) at (0,0) {t};
\node[circle, draw] (T) at (-1,-1) {T};
\node[circle, draw] (T) at (1,-1) {T};
\node[coordinate] (ia) at (0,1) {ia};
\node[coordinate] (ib) at (-1,0) {ib};
\node[coordinate] (ic) at (1,0) {ic};
\end{tikzpicture}})$

Figure 6.3: An icon-replacement grammar, the T-grammar.
is defined as follows:

\[ \mathcal{P}(\sigma, \tau) = \{ \]
\[ (p_1, (p_3, \{ \}),)
\[ (p_2, (p_4, \{ (C_1, T_1), (C_2, T_1c) \})) \]
\]

6.2.2.1 Derivations

The above definition of pair grammars provides a framework for discussing derivations done according to some pair grammar.

An essential aspect of pair grammars is the correspondence between derivations performed using the left hand side grammar and those performed using the right hand side grammar. That is, every derivation step performed using the left hand side grammar identifies a corresponding derivation step to be performed using the right hand side grammar.

Hence, for a pair grammar \(PG = (\Gamma_l, \Gamma_r, \mathcal{P})\), we can think of pair derivations as two derivation processes being performed in parallel. These two processes are referred to as the left hand side derivation (using \(\Gamma_l\)) and the right hand side derivation (using \(\Gamma_r\)). Both derivation start at the corresponding grammar’s start symbol, and proceed by applying production rules to the appropriate non-terminals. Furthermore, the left hand side derivation imposes restrictions on the right hand side derivations. That is, each left hand side derivation step identifies a unique right hand side derivation step.

For example, given a right hand side derivation \(S_{r1} \Rightarrow (p', i') G'\), then the left hand side derivation is given by: \(S_{l1} \Rightarrow (p, i) G\). Where \(p'\) is given by: \(LHS \mathcal{P} p\). Furthermore, the non-terminals of the two resulting graphs (\(G\) and \(G'\)) are related through the function given by: \(RHS \mathcal{P} p\).

6.2.2.2 Pair graphs

From the above discussion on derivations we identify the concept of pair graphs.

Pair graphs are those structures generated from each parallel derivation step. In general, a pair graph is given by: \((G_l, G_r, h)\). Where:

- \(G_l\): is an icon-graph derived from the left hand side grammar, i.e. \(G_l \in Graphs(\Gamma_l)\);

- \(G_r\): is an icon-graph derived from the right hand side grammar, i.e. \(G_r \in Graphs(\Gamma_r)\); and

\[\text{provided that the left hand side production is represented in the domain of } \mathcal{P}\]
6.2.2.3 Pair derivation

We can think of pair derivation steps as derivations being applied on pair graphs. Each such derivation step is actually composed of two derivation steps: one corresponding to the derivation step from the left hand side grammar, and the other corresponding to the derivation step from the right hand side grammar. In general, a pair derivation step is denoted by: 

\[(G_l, G_r, h) \Rightarrow ((p_l, i_l), (p_r, i_r)) \quad (G'_l, G'_r, h').\]

Where:

- \(G'_l\) is derived from \(G_l\) and is given by: \(G_l \Rightarrow (p_l, i_l) \Rightarrow G'_l\);
- \(G'_r\) is derived from \(G_r\) and is given by: \(G_r \Rightarrow (p_r, i_r) \Rightarrow G'_r\). Where:
  - \(p_r\) is given by: \(RHS \ P p_l\) and
  - \(i_r\) is the non-terminal icon to be replaced, it is defined by the pairing function \(h\) and is given by: \(RHS h \ (\text{label}G_l, i_l, i_l)\)
- \(h'\) is the paring function of the resulting pair graph, and it is given by: \(LHS \ P p_l\)

Note:
To preserve the uniformity of the pair derivations we say that a pair grammar, \(PG = (\Gamma_l, \Gamma_r, \mathcal{P})\), has a pair starting symbol which is given by: \((S_{\Gamma_l}, S_{\Gamma_r}, h_0)\)

Where:

- \(S_{\Gamma_l}\) is the start graph of the left hand side grammar;
- \(S_{\Gamma_r}\) is the start graph of the right hand side grammar; and
- \(h_0\) is a dummy (trivial) mapping linking the two non-terminals in the two start graphs.

Figure 6.4 shows a derivation of a graph from the C-grammar. Figure 6.5 shows the corresponding (or pair) derivation generated from the T-grammar; it is called its pair derivation because each time a production from the C-grammar is applied the corresponding production from the T-grammar is applied. Finally, figure 6.6 shows the corresponding graphs resulting from both derivations.

Pair graphs defined by a grammar. From the above definitions, the set of pair graphs defined by a pair grammar, \(PG = (\Gamma_l, \Gamma_r, \mathcal{P})\), is given by:

\[
\text{Graphs}(PG) \triangleq \{ \quad (G_l, G_r, h) \quad | \quad \exists \ d : \text{seq}[P_l \times P_r] \\
\quad \cdot (S_{\Gamma_l}, S_{\Gamma_r}, h_0) \Rightarrow_d \ (G_l, G_r, h) \quad \}
\]
Figure 6.4: A derivation tree of a derivation from the C-grammar.
Figure 6.5: A derivation tree of a derivation from the T-grammar.
6.2 Pair Grammars

The pair language defined by a grammar. The Pair language defined by a pair grammar is the set of all terminal pair graphs that can be generated from that grammar. For a pair grammar, \( PG = (\Gamma_l, \Gamma_r, \mathcal{P}) \), is given by:

\[
\text{Lang}(PG) \triangleq \{ (G_l, G_r, h) : \text{Graphs}(PG) \mid \text{ran } label_{G_l} \cap N_{\Gamma_l} = \emptyset \land \text{ran } label_{G_r} \cap N_{\Gamma_r} = \emptyset \}
\]

6.2.3 Types of Pair Grammars

This subsection discusses some interesting types of pair grammars. These types allow for different types of mappings between language and have some useful applications in system (design) visualization.

6.2.3.1 Revisable Pair Grammars

Revisable pair grammars are those grammars that constitute a one-to-one mapping between the right and left hand side languages. Such grammars are interesting since they constitute a two way translation map between the two languages. One obvious application of such a mapping is the ability to use (or switch to) any of the two languages whenever convenient without any loss of structural information.

\[\text{For example, the development environment proposed in chapter 7 makes extensive use of such a scheme to enable its users to alternate between textual and visual representation of their designs.}\]
A pair grammar is said to be revisable if each left hand productions is mapped onto a unique right hand side production, and all left and right hand productions are represented (i.e. mapped onto \( P_r \)).

More formally, a pair grammar \( PG = (\Gamma_l, \Gamma_r, P) \), is revisable iff the set given by: \( \{ (p_l, (p_r, h)) : P \cdot (p_l, p_r) \} \) is a total bijective function from \( P_l \), to \( P_r \).

For example the pair grammar \( P_{CT} = (C, T, P_{(C,T)}) \) (which relates the C-grammar, shown in figure 6.2, and the T-grammar, shown in figure 6.3) is an example of a revisable pair grammar. This can be easily seen by constructing the set relating the two production sets. This set represents a bijective function relating \( P_c \) to \( P_t \); it is given by: \( \{ (p_1, p_3), (p_2, p_4) \} \).

Reversing a pair grammar. For a given revisable pair grammar, \( PG = (\Gamma_l, \Gamma_r, P) \), the reverse pair grammar, denoted by \( PG^{-1} \), is defined by:

\[
PG^{-1} \triangleq (\Gamma_r, \Gamma_l, Rev(P))
\]

where

\[
Rev(P) \triangleq \{ (p_l, (p_r, h)) : P \cdot (p_l, p_r, h^{-1}) \}
\]

For example, \( P_{CT}^{\text{-1}} \) is given by: \( (T, C, P_{(C,T)}^{\text{-1}}) \). Where \( P_{(C,T)}^{\text{-1}} = P_{(T,C)} \) is given by:

\[
P_{(T,C)} = \{
(p_3, \{ (p_1, \{ \}) \}),
(p_4, \{ (p_2, \{ (T_h, C_i), (T_{i_c}, C_{h}) \}) \})
\}
\]

6.2.3.2 Partial Pair Grammars

The second class of pair grammars is that of Partial pair grammars. Partial pair grammars are partial mappings from the right hand side language to the right hand side language.

Such pair grammar are useful, for example, when it is desirable to produce partial visualizations of language constructs. One common example, is the visualization of the data structure definition section of a program.

6.2.3.3 Pair Grammars with Cut-off Thresholds

Such pair grammars allow all of the left hand side productions to be linked to right hand productions. However, the requirement that every mapping in \( \mathcal{H} \) be a bijection is relaxed to allow functions in \( \mathcal{H} \) to be total injections (instead of total bijections).
6.2 Pair Grammars

Such mappings are typically useful when the left hand language is to be used to define the more abstract aspects of the system under development, then when the abstract phase is completed and the design is translated into a less abstract form (the right hand language) where the missing details are to be filled in.

6.2.4 Textual language to Icon-language translation

Pair grammars can also be used to link string or textual and icon-replacement grammars. This can be done by substituting a string grammar in place of the icon-replacement grammar.

This can be used to give two concrete representations (syntaxes) to the same underlying abstract language; one of which might be a visual representation and the other a textual one. Structure visualization is an important application of such a set up.

For example, consider the following string grammar, called the R grammar, which can be used to define Pascal-like record structures:

\[
\begin{align*}
    \text{Record} &::= \text{'begin'} \text{ BasicFieldType} \text{'end'} & p_a \\
    \text{Record} &::= \text{'begin'} \text{ BasicFieldType} \text{ Record} \text{'end'} & p_b \\
    \text{BasicFieldType} &::= \text{'aField'} & p_c \\
    \text{BasicFieldType} &::= \text{'aField'} \text{ BasicFieldType} & p_d
\end{align*}
\]

The \textit{R} grammar is defined by means of four production rules (\(p_a\) to \(p_d\)), and the actual field definition is emitted (and hence the definition of \textit{aField} as a terminal symbol) since it is not of particular interest here.

One possible visualization of such \textit{Records} is to represent them as tree structures. The icon-replacement grammar shown in figure 6.7 gives a definition of such a tree grammar.

To link the string record grammar and the tree grammar we define the following pair grammar:

\[
P_{RT} = (R, T, P_{(R,T)})
\]

Where:

\[
P_{(R,T)} = \{
    (p_a, (p_1, \{(\text{BasicFieldType}, U_{\bar{a}})\})), \\
    (p_b, (p_2, \{(\text{BasicFieldType}, U_{\bar{b}}), (\text{Record}, T_{\bar{b}})\})), \\
    (p_c, (p_3, \{\})), \\
    (p_d, (p_4, \{\{(\text{BasicFieldType}, U_{\bar{d}})\}\})),
\}
\]
Grammar Pairs and Syntax-Directed Translation

\[ T = (V_t, N_t, S_t, P_t) \]

\[ V_t = \{ T, U, t \} \]

\[ N_t = \{ T, U \} \]

\[ S_t = \begin{array}{c}
T
\end{array} \]

\[ P_t = \{ p_1, p_2, p_3, p_4 \} \]

Where:

\[ p_1 = (T, ) \]

\[ p_2 = (T, u_4, u_5, u_6, u_7, u_8) \]

\[ p_3 = (T, v) \]

\[ p_4 = (T, v, u) \]

Figure 6.7: The Tree-Record grammar, used in visualizing record declarations.
For example, consider the following string derivation (using the $R$ grammar):

\[
\begin{align*}
Record \Rightarrow_p \text{'begin'} \\
& \quad BasicField Type Record \text{'end'} \\
& \Rightarrow_p \text{'begin'} \\
& \quad BasicField Type \text{'begin'} \\
& \quad BasicField Type \text{'end'} \\
& \quad \text{'end'} \\
& \Rightarrow_p \text{'begin'} \\
& \quad 'aField' \text{'begin'} \\
& \quad BasicField Type \text{'end'} \\
& \quad \text{'end'} \\
& \Rightarrow_p \text{'begin'} \\
& \quad 'aField' \text{'begin'} \\
& \quad 'aField' BasicField Type \text{'end'} \\
& \quad \text{'end'} \\
& \Rightarrow_p \text{'begin'} \\
& \quad 'aField' \text{'begin'} \\
& \quad 'aField' 'aField' \text{'end'} \\
& \quad \text{'end'}
\end{align*}
\]

This derivation can be modelled by means of the derivation tree shown in figure 6.8.

Using $P(R,T)$ the corresponding derivation for the pair $T$–graph can be constructed.

Figure 6.9 shows the corresponding derivation tree derived from the pair grammar rules, and figure 6.10 shows the corresponding record–tree visualization of the textual construct.
Figure 6.8: A derivation tree of a R-grammar structure.
Figure 6.9: A derivation tree of a Tree-grammar structure.
6.3 Summary

This chapter presented pair grammars as the basis for defining syntax-directed translation mechanisms.

A model of pair grammars for icon-replacement grammars is defined, and shown how it can be used to define language translation schemes. The pair grammar model is used to define the notion of pair derivations and pair graphs, and the definition of pair languages, which can be seen as translation maps between the left-hand-side and the right-hand-side languages composing the grammar.

Different types of pair grammars are also examined, and their applications are outlined.

Finally, the model is extended to allow the combination of string and icon-replacement grammars. This allows pair grammars to be used as an effective basis for string language visualization systems.

It is asserted that the significance of the pair grammar model is in its ability to link different concrete syntax definitions, which can be seen as alternative front ends to the same abstract language syntax. Thus, enabling the construction of translation mechanisms that exploit the inherent equivalence of the underlying abstract (syntax) languages.
Chapter 7

An Integrated System Development Environment

7.1 Introduction

Chapters 5 and 6 presented two aspects of development environments that are based on the icon-replacement grammar model. This chapter outlines the architecture of an integrated development environment that employs the basic components presented in chapters 5 and 6.

7.2 Editing and Presentation Environment

The editing environment (or what might be called the visual programming component) presented here is based on the syntax-directed editing model presented in chapter 5. This editing environment can be seen as: (1) a tool for entering, modifying and presenting the specification or design being created; (2) a syntax checking tool; and (3) a transformational tool that enables the construction of derivation trees corresponding to the entered structure (or graph).

Although the editing environment presented in chapter 5 is syntax driven, it is, however, language independent, as the various syntax-directed editing functions depend on the availability of a valid icon-replacement grammar, and not on the characteristics of a certain grammar or language. Hence, the syntax-directed editing functions can be used to edit and manipulate any language for which a valid icon-replacement grammar exists. Therefore, the syntax-directed editing environment can be seen as generic, in the sense that it can be combined with an arbitrary icon-replacement grammar.
This is one of the important aspects of the syntax-directed editing model, from a tool construction point of view, since it allows the construction of a generic language independent syntax editing engine that can be driven by grammar definitions, and thus can be used for different icon-languages.

Figure 7.1 shows the architecture of such a generic syntax-directed editing environment.

The editing environment, with its syntax-directed nature, provides a framework for syntax checking and the construction of derivation trees. Once a derivation tree is constructed it is possible to compute the attributes attached to the grammar (through the evaluation of attribute or semantic rules), and, hence, it is possible to perform various transformations on the language constructs.

7.2.1 Support for Hierarchical Languages

Syntax-directed editing for hierarchical language can be supported by providing a hierarchy of editing environments. The base or root of such a hierarchy is the syntax-directed editor of the base language. This base editor is used to derive the constructs of the base language. Once terminal labels of the base language are derived, and it is desired to construct a hierarchical component, then a new instance of the editor is invoked to derive the hierarchical component. Naturally, this new editor instance will use the grammar definition of the hierarchical component.

This scheme of hierarchical editor invocations follows closely the definition of the hierarchical language model, where each possible editor invocation corresponds to a level in the hierarchical language.

This can be seen as the syntax-directed editor being invoked recursively, with the appropriate grammar definition being passed to it as a parameter.

Figure 7.2 shows an example of such a hierarchical editing session.
Figure 7.2: An example of a hierarchical editing session.
7.3 Pair-grammar Based Transformational Tools

Pair-grammar based transformational tools can be thought of as similar to construct visualization tools. They are used to construct syntax based transformations between the left-hand-side and the right-hand-side grammars. They are similar to visualization tools because the right hand side representation of language constructs can be viewed as alternative representation (or visualizations) of the left hand side constructs. Pair-grammar transformations can also be used to translate visual notations into textual one and vice versa.

Pair-grammars are used to define visualization-like transformational tools, and (more generally) to present different views or concrete presentations to the same abstract syntax. For example, pair-grammars can be used to construct string-to-icon-language transformations. Section 6.2.4 shows an example of such a scheme. Figure 7.3 shows an example of how pair-grammars can be utilized to construct different views of the same abstract language.

7.4 The Integrated Environment

The two basic components described so far (i.e. the syntax-directed editing environment and pair-grammar based translations) can be composed to construct a development environment. Such an environment contains a visual programming component, which is the syntax-directed editing part, and a visualization component, which is the pair-grammar based part.

An example of such an environment is shown in figure 7.4. This particular example shows an environment that uses two equivalent notations: a textual one and a visual one.

The generic nature of the basic components of this architecture enables its extension in an easy and modular manner. For example, an integrated environment may contain more than one visual programming component, and likewise more than one visualization component. This is done by adding more pair-grammar maps, to add extra visualization components, and adding a new icon-grammar definition to use the syntax-directed editor with the desired notation.
Figure 7.3: Using pair-grammars to construct different views of the same abstract language.
Figure 7.4: An integrated development environment.
Chapter 8

Conclusions

The problem domain of this thesis is system development. It is motivated by the need to find and construct better design technologies. The use of better design technologies is viewed as imperative to deal with the increase in the size and complexity of digital systems.

This thesis takes the view that the integration of formal methods and visual languages yields a sound and powerful foundation for the definition of essential aspects of modern design technologies.

To achieve this goal two main topic areas are addressed: (1) the formal definition of appropriate visual language model, which entails identifying the essential aspects of such languages and determining suitable modelling techniques to capture these desired characteristics; and (2) the utilization of the visual language model to define appropriate development environment, which entails identifying the basic aspects of such development environment and showing how the language model can be utilized in its construction.

The remainder of this chapter evaluates the main results of this thesis, and discusses some extensions and future research.

8.1 Evaluation and Overview

8.1.1 Icon languages as modelling tools

The icon-language model presented in this thesis is an attributed grammar model that enables the definition of flat as well as hierarchical languages (chapters 3 and 4, respectively). This grammar model is directed towards the definition of languages that are to be used in system specification throughout the development process.

It asserted that the role of a specification (or design) language in the
development process is that of a modelling tool. As such, one might summarize the major functions of a specification language in the development process as follows:

- A tool for describing the system behaviour. This is probably the most obvious use of a specification language, that is producing a concise and clear description of the desired system behaviour.

- A tool for reasoning about the system description and its properties. That is to say, to enable the language's user to reason about characteristics of the specification document; for example, to detect certain types of inconsistencies and errors or to prove the equivalence of certain constructs.

- A means of managing the problem size and complexity by providing support for:
  
  - Concept modelling and construction. That is, the language should provide support for abstraction mechanisms such as: composition/decomposition, classification/exemplification, and generalization/specialization.
  
  - Managing the system's description. This is especially important for large and complex systems, where it is necessary to split the systems into smaller, and more manageable parts. A specification language should provide mechanisms to enable the definition of smaller system components and their interrelations.

To meet these criteria a language must possess certain characteristics; icon languages—or more accurately, attribute hierarchical icon-replacement grammars—provide the foundation and framework for defining such languages. This is due to the following characteristics of the model:

- Icon-replacement Grammars, the basic model of icon languages, provide a formal framework for defining the syntax and various structural aspects of icon languages.

- Attribute icon-replacement grammars, through the use of attributes, provide a formal framework for defining the semantical and non-structural aspects of icon languages.

- Hierarchical Icon-replacement Grammars provide support for modelling hierarchical abstractions by enabling the definition of hierarchical constructs; which in turn provide means for managing the size and
complexity of the problem at hand. In addition to the above, hierarchical icon-replacement grammars divide a language into a hierarchy of levels; where each level supports a different set of abstraction, allowing the problem to be dealt with at the appropriate level with the most appropriate set of abstractions.

This effectively allows the integration of different languages into one framework. Hierarchical Icon-replacement Grammars also allow for the integration of textual and visual constructs. It is asserted that this integration is desirable for practical system development. Furthermore, attribute hierarchical icon-replacement grammars allow for the integration of the various levels in a hierarchy into one hierarchical attribute systems. Such a hierarchical attribute system allows the exchange of semantical information between the various levels, and thus provides an effective interface mechanism.

To summarize, the icon language model meets the preciseness of description criteria by providing a formal framework for defining visual languages. The attributed grammar model provides the basis for reasoning about the syntactic and semantic characteristics of constructs defined in such languages.

Furthermore, it meets the need for modelling abstractions, and dealing with each sub-problem at the appropriate level of detail by providing structural modelling of hierarchical constructs, as well as a hierarchal attribute system that enables the interfacing, and thus integration, of the various levels.

8.1.2 Icon languages as the basis for tool construction

Automated development support tools are viewed as an essential part of modern design technologies. This thesis provides the foundation for constructing icon-language-based development tools. The icon language model is used as the basis for defining and specifying such tools.

This thesis identifies and addresses two basic aspects of integrated development environments, namely: (1) editing and presentation tools, where such tools would be used for entering, editing and presenting (displaying) system representations; and (2) transformational tools, where such tools would be used for transforming and checking the validity of particular system representations.
8.1.2.1 Language-directed Editing Tools

Language directed editing tools are proposed as the basis for supporting the creation and editing of system specifications.

The syntax-directed editing model for icon-languages (proposed in chapter 5) provides the basis for the editing and presentation aspects of the development environment, as well as the basic syntax and validity checking functions. Thus, the syntax-directed editing model can be seen as the visual programming aspect of the development environment.

It is asserted that the syntax-directed editing approach proposed here is sound, effective and has advantages over the traditional approach of separating editing and parsing (i.e. the so-called edit-parse approach). The traditional limitations of syntax-directed editing are addressed and novel functions are added to the model in order to increase its effectiveness.

The problems of presentation and syntax-directed browsing are also addressed. It is asserted that the language-directed nature of the editing environment can be used to provide better presentation and layout management. This is accomplished through two main mechanisms:

1. interfacing the editing environment to existing constraint-based layout systems, and thus, combining generic (i.e. language independent) aspects of graph layout requirements and the language dependent ones; and

2. through providing support for syntax-directed browsing of language constructs, thus illustrating how the language driven environment can be used to present different aspects of the system under development by providing syntax-based visual abstractions.

Although the syntax-directed editing model utilizes the underlying language definition, it is nevertheless independent of specific language definitions. That is, the syntax-directed editing model is generic in the sense that it can be used in combination of any language for which there exists a valid icon-replacement grammar definition. Thus, once fully implemented, the syntax-directed editing engine can be used as the basis for a editing environment generator.

8.1.2.2 Pair Grammars as Transformational Tools

Pair-grammars are proposed as the basis for syntax-directed transformation tools.

Syntax-directed transformations are viewed as the visualization component of the development environment. Such transformations are achieved
through the use of the pair-grammars as translation maps between the left-hand-side and the right-hand-side languages composing the grammar.

The pair-grammar model is also used to support visualization in the traditional sense by extending the model to allow the combination of string and icon-replacement grammars. This allows pair grammars to be used as an effective basis for string language visualization systems.

The significance of the pair grammar model, and its extensions, is in its ability to link different concrete syntax definitions, which can be seen as alternative front ends to the same abstract language syntax. Thus, enabling the construction of translation mechanisms that exploit the inherent equivalence of the underlying abstract languages.

The use of such translation mechanisms does not only provide for effective visualization, but also allows different concrete notations to be used in an interchangeable manner throughout the development process.

The model is also extended to enable the visualization or translation of language subsets. This is thought of as particularly useful as a visualization with abstraction tool. This is particularly useful when it is desired to construct visualizations of certain aspects of a construct, rather than visualizing it in full detail.

8.1.3 Limitations of the Basic Model

The basic icon-replacement grammar model presented and used in this thesis is based on the idea of viewing basic language constructs as graphs of interconnected icons. Thus, the model is effective in modelling graph based diagramming languages (which is its original goal). This is accomplished by constructing the model in a way that captures the topology of the graph construct in the syntax of the language, but leaves other appearance aspects (e.g. layout, shapes, etc.) to be defined in the attribute system.

From a pure presentation point of view, this does not pose a real problem, as the attribute system can be used to transform the topological information, provided by the grammar, into geometric and display information. For example, a certain connection between icons can be interpreted as a containment relation.

However, the limitation becomes more apparent when it is desired to apply the direct manipulation and editing interface (described in chapter 5) to the geometric representation of the language constructs. The limitation stems from the fact that the editing and manipulating interface is based on the syntax of the language, and hence cannot directly be used on the non-structural aspects of the language. To accomplish this, an extra interface layer should be constructed. This layer would serve as a reverse mapping...
from presentation to structure.

It is thought that such a mapping will make icon-replacement grammars more suitable for use to define non-diagramming notations, however such a mapping is not needed to deal with the particular problem domain this thesis is concerned with.

8.2 Future Directions

8.2.1 Extensions

This subsection briefly outlines the main extensions and completions to the existing research products.

- A complete implementation of the development environment is high on the list of future tasks. The current implementation comprises the compiler for the MIL language (described in appendix B), which generates an object-oriented representation of the defined grammar.

- Constructing a visual front-end to the icon grammar meta-language (MIL). This is a boot-strapping step, that is, the current (textual) meta-language would be used to define a visualization of itself. Such a visual meta-language would considerably ease the use of the icon grammar system.

- Undertaking a the conversion of an existing CASE/CAE environment using the icon grammar system and environment (such as OOA environment proposed in [HVS91], or aspects of the IDaSS system [Ver90]). The motivation for such an undertaking is two-fold: firstly, it would enable the comparison of the functionality and usability of the two systems; and secondly, it would be a step in the direction of formalizing these informal notations.

8.2.2 Future research

There is still much room for future research in the field of visual languages. This subsection briefly outlines some of the possible extensions of the approach proposed in this thesis.

- Theoretical study of visual languages. Further study and classification of various classes of icon-languages. This study would aim at identifying interesting classes that are, for example, more efficient to parse.
• Addressing the question of extending the icon-replacement grammar model to enable its use for non-diagramming languages. This entails addressing the issues outlined in section 8.1.3.

• Extending the meta icon language MIL to include the concepts of defining scalable icon-graphs. Scalable icon-graphs are icon graphs that contain certain sub-graphs that may be repeated in a pre specified manner. This can be viewed as a form of syntactic sugar that makes icon-replacement grammar definitions more convenient. The current definition of MIL includes limited support for scalable icon-graphs through the use of generic icons.

• Extending the research in other domains. It is interesting to investigate the applicability of icon-replacement grammars to other application areas. One such area is the use of visual languages to describe hypertext and hypermedia systems. It is asserted that icon-replacement grammars would be particularly easy to adapt because of the obvious analogy between the page/link metaphor of hypertext systems, and the icon/branch concepts of icon-replacement grammars.
Appendix A

Notational conventions

The notational conventions used throughout this thesis are loosely based on the Z notation as presented in [WL88] (a more detailed description of the Z notation can be found in [KSW88]).

The remainder of this appendix briefly outlines some of the notation used in thesis.

- \{x : S \mid P(x) \bullet t(x)\}: Is used for set specification. Where x is a bound variable which satisfies P(x), and the set has the form t(x).

- A \leftrightarrow B: Denotes the set of all possible relations between A and B (i.e. P(A \times B)).

- LHS, and RHS: are used to denote the left hand side and the right hand side if a tuple, respectively.

- \maple{xy}: Is used to denote a maplet, i.e. if two objects x \in X and y \in Y are related by a relation R \in X \leftrightarrow Y, then we write: x \maple{y}

- A \rightarrow B: Denotes the set of all total functions from A to B.

- A \rightarrow B: Denotes the set of all partial functions from A to B.

- \ran R: is used to denote the range of relation R.

- seq [X]: Denotes the set of all sequences over a set X.

- \{ \text{and } \} are used to enclose sequences, in order to reduce the formal clutter resulting from the use of the complete set notation. For example, \langle S_1, S_2, \ldots, S_n \rangle is used to denote: \{1 \maple{S_1}, 2 \maple{S_2}, \ldots, n \maple{S_n}\}.
• $\cdot$ is the infix operator used to indicate sequence concatenation, e.g. $(a, b, c) \cdot (x, y) = (a, b, c, x, y)$.

• $\textit{tail}$: is a function that returns the tail of non-empty sequences, e.g. $\textit{tail} \ (a, b, c) = (b, c)$

• $\lambda$ arguments $|$ conditions $|$ results: Is the general form for $\lambda$-expressions.

• $\exists x : X \bullet P(x)$: Is the typed existential qualification. This form is used as an abbreviation for $\exists x \bullet x \in X \land P(x)$.

• $\forall x : X \bullet P(x)$: Is the typed universal qualification. This form is used as an abbreviation for $\forall x \bullet x \in X \Rightarrow P(x)$.

• $\triangleleft$: Is used to denote domain subtraction. In general the relation $U = S \triangleleft R$, is similar to $R$ but has no elements of the set $S$ in its domain, i.e. $U = \{(a, b) \in R \mid a \not\in S\}$

• $\triangleright$: Is used to denote range subtraction. In general the relation $U = R \triangleright S$ is similar to $R$ but has no elements of the set $S$ in its range, i.e. $U = \{(a, b) \in R \mid b \not\in S\}$
Appendix B

MIL: a language for defining icon-replacement grammars

B.1 Introduction

MIL, Meta Icon Language, is a concrete notation for defining icon-replacement grammars. It enables the definition of icons, icon-graphs and grammars. MIL also adds a few extensions to the basic icon-replacement grammar model to make writing icon-grammar specifications more convenient. Such extensions include: generic icons and graphs, scalable and repeatable graphs.

This appendix describes the main structures that can be defined by the language and illustrates its use through examples. A concrete description of the syntax is also given.

B.2 Basic constructs

MIL can be used to define three main constructs: icons, icon-graphs and icon-replacement grammars.

In general, descriptions written in MIL (or MIL files, for short) may contain a number of icon, graph and grammar definitions. There is no (grammatical) restriction on the order or number in which they appear. Typically, definition files contain a number of icon and graph definitions and one grammar definition. Moreover, graph definitions may use previously defined icons and grammar definitions may use previously defined graphs.
B.2.1 Defining icons

B.2.1.1 Basics

Icons are defined by mean of an icon definition construct. An icon definition may contain four main fields:

1. a label field, which specifies the icon’s label;
2. a shape files, which specifies the shape of the icon;
3. input branches specification field, which specifies the number of input branches; and
4. output branches specification field, which specifies the number of output branches.

In general, the value supplied to the label and shape fields can be an arbitrary identifier. However, in the language’s implementation a fixed (but user defined) set of icon shapes is used.

For example, an and icon might be defined as:

```plaintext
andIcon: icon
    label: and;
    shape: andShape;
    inbranches: 2;
    outbranches: 1
end;
```

This definition defines a new icon structure, called andIcon, which is labelled by and, has a shape called andShape, has 2 input branches, and 1 output branch. In a specific implementation, the image of the andIcon is given by:

```
\[ \text{image} \]
```

B.2.1.2 Generic Icons

When creating icon definitions it is often the case that similar icons need to be defined. For example, it might be desirable to define a number of and icons that have a different number of inputs. It is, therefore, desirable to support mechanisms that enable the reuse of such icon definitions.

MIL provides for such reuse by enabling the definition of parameterized or generic icons.
A generic icon (or an icon class) is defined in terms of one or more icon parameters. An icon parameter can, in general, refer to the value of any of the four icon definition fields (i.e. label, shape, inbranches and outbranches).

For example, consider the definition of a function block icon with a parametric number of input branches. Such a definition is given by:

```
funcBlock : icon ( numInBranches )
  label: Gate;
  shape: rectangle(3, 4);
  inbranches: numInBranches;
  outbranches: 1
end;
```

The above definitions specifies a generic icon, called `funcBlock`, and the parameter `numInBranches` is used to specify the number of input branches. For example, the icon corresponding to `funcBlock[2]` is shown as:

```
   Gate
```

### B.2.2 Defining graph constructs

#### B.2.2.1 Basics

Graphs are defined by mean of a `graph definition` construct. A graph definition may contain five main fields:

1. An icons field, which defines the icon set used in defining the graph. Icon sets are specified by providing a list of icons, where lists of icons may contain icon definitions or identifiers of previously defined icons.

2. A joint field, which defines the joint set used in defining the graph. The joint set is specified in the form of a list of integer values and joint identifiers.

3. A connections field, which specifies how the different icon branches are attached to joints.

4. An optional external joints field, which defines two lists (or sequences) corresponding to the graph's list of input and output joints, respectively.
5. An optional subgraph field, which can be used to specify some previously defined graph as a sub-component of the current graph definitions.

For example, the definition of a simple logic diagram containing an and-gate and a not-gate might be given by:

\[
\text{XandnotY : graph}
\]
\[
\text{icons: andIcon, notIcon;}
\]
\[
\text{joints: 1..4;}
\]
\[
\text{connections: notIcon.inbranch[1]=2,}
\]
\[
\text{notIcon.outbranch[1]=3,}
\]
\[
\text{andIcon.inbranch[1]=1,}
\]
\[
\text{andIcon.inbranch[2]=3,}
\]
\[
\text{andIcon.outbranch[1]=4;}
\]
\[
\text{externals: (1, 2), (4);}\]
\[
\text{end;}
\]

This definition specifies a new graph, called XandnotY. The XandnotY graph contains two (previously defined) icons: called andI con and notIcon. It also contains four joints. The definition also shows two lists of external joints, the first list, (1, 2), denotes the list of input joints, the second list, (4), denotes the list of output joints. The connections part defines how the graph is configured (i.e., how icons are connected to joints).

The XandnotY graph is shown as:

\[
\begin{array}{cc}
\text{\includegraphics[width=1cm]{and.png}} & \text{\includegraphics[width=1cm]{not.png}} \\
\end{array}
\]

B.2.2.2 Generic Graphs

When creating graph definitions it is often the case that similar graphs need to be defined. For example, it might be desirable to define two different graphs that essentially have the same connection pattern (topology), but use different icons in their definition. Hence, desirable to support mechanisms that enable the definition of such generic graphs.

MIL provides for such reuse by enabling the definition of parameterized or generic graphs.

A generic graph is defined in terms of one or more graph parameters. A graph parameter can, in general, refer to the value of any of the graph definition fields.

For example, consider the definition:
twoEntryGraph : graph( Icon anIcon )
  joints: 1..3;
  connections: anIcon.inbranch[1]=1,
               anIcon.inbranch[2]=2,
               anIcon.outbranch[1]=1;
  externals: (1, 2), (3);
end;

Where twoEntryGraph is a generic graph which uses an Icon parameter to define its exact structure. For example, the graph twoEntryGraph(andIcon) denotes:

Moreover, the graphs twoEntryGraph(orIcon) denotes:

B.2.3 Defining grammars

Grammars are defined by mean of a grammar definition construct. A grammar definition may contain four main fields:

1. The terminal set definition field, which defines the set of terminal labels (i.e labels not appearing in the left hand side of production rules) used in the grammar.

2. The non–terminal set definition field, which defines the set of non–terminal labels used in the grammar.

3. The start graph field, which defines the grammar’s axiom or starting graph.

4. The production definition field, which defines the set of productions used in the grammar. Each production associates a non–terminal label with its replacement graph.

For example, the following grammar definition:

AndOrGrammar : grammar
  terminals: { and, or };
  nonterminals: { function };
  startgraph: sGraph;
The above definition defines a simple grammar, called the AndOrGrammar. The AndOrGrammar has two terminal labels: and and or; one non-terminal label: function; a start graph identified by sGraph; and a production set containing two productions.

B.3 Example: Logic Diagrams

The following example gives the complete definition of the logic diagram grammar defined in chapter 3.

```plaintext
// File: logic.ig

// Icon classes

/---------
/ [n]---+ function +---
/       /---------

functionalBlock : icon ( numInBranches )
    label: function;
    shape: rectangle(3, 4);
    inbranches: numInBranches;
    outbranches: 1
end;

andIcon: icon
    label: and;
    shape: andShape;
    inbranches: 2;
    outbranches: 1
end;

orIcon: icon
    label: or;
    shape: orShape;
    inbranches: 2;
    outbranches: 1
```
end;

notIcon: icon
    label: not;
    shape: notShape;
    inbranches: 1;
    outbranches: 1
end;

//
// Graph definitions
//
// sGraph
//
// +---------+
// 1x--+     |
// 2x--+     |
// 3x--+ function ---x5
// 4x--+     |
//     |     |
// +---------+

sGraph : graph
    icons: functionalBlock(2);
    joints: 1..3;
    connections: functionalBlock.inbranch[1]=1,
                 functionalBlock.inbranch[2]=2,
                 functionalBlock.outbranch[1]=3;
end; // sGraph

// +------+
// 1x--+   |
//        | AND ---x4
// 2x--NOT-3x--+   |
// +------+

XandnotY : graph
    icons: andIcon, notIcon;
    joints: 1..4;
    connections: notIcon.inbranch[1]=2,
                 notIcon.outbranch[1]=3,
                 andIcon.inbranch[1]=1,
                 andIcon.inbranch[2]=3,
                 andIcon.outbranch[1]=4;
    externals: (1, 2), (4);
end;
notXandY : graph
icons: andIcon, notIcon;
joints: 1..4;
connections: notIcon.inbranch[1]=1,
notIcon.outbranch[1]=3,
andIcon.inbranch[1]=3,
andIcon.inbranch[2]=2,
andIcon.outbranch[1]=4;
externals: (1, 2), (4);
end;

XorBlockGraph : graph
icons: functionalBlock(2)[3]; // three functional blocks,
// each with 2 inputs.
joints: 1..5;
connections: functionalBlock[1].inbranch[1]=1,
functionalBlock[1].inbranch[2]=2,
functionalBlock[2].inbranch[1]=1,
functionalBlock[2].inbranch[2]=2,
functionalBlock[3].inbranch[1]=3,
functionalBlock[3].inbranch[2]=4,
functionalBlock[1].outbranch[1]=3,
functionalBlock[2].outbranch[1]=4,
functionalBlock[3].outbranch[1]=5;
externals: (1, 2), (5);
end;
B.4 Example: Actor Nets

The following example gives the definition of a subset of Actor Nets as defined in [vH94]. Figure B.1 shows the corresponding icon-replacement grammar model.

```
//
// File: actor.ig
//
// Icon classes
//
LPblock : icon ( inBranches, outBranches )
  label: LP;
  shape: rectangle(3, 4);
  inbranches: inBranches;
  outbranches: outBranches
end; //LPblock
```

```
FlatNet = (Vf, Nf, Sf, Pf)
Vf = {l, p, LP}, Nf = {LP}
Sf = LP

P =

{(LP, l p ),
 (LP, l p LP),
 (LP, l p ),
 (LP, l p LP),
 (LP, LP LP),
 (LP, )
}

Hnet = (Vf, Nf, Sf, Pf, H)
H = { (p, FlatNet) }
B.4 Example: Actor Nets

Location : icon ( inBranches, outBranches )
  label: l;
  shape: circle(4);
  inbranches: inBranches;
  outbranches: outBranches
end; //Location

Processor : icon ( inBranches, outBranches )
  label: p;
  shape: rectangle(4, 4);
  inbranches: inBranches;
  outbranches: outBranches
end; //Location

//--
//-- Graph definitions
//--

sGraph : graph
  icons: LPblock(0, 0);
end;

lpGraph : graph
  icons: Location(0, 1), Processor(1, 0);
  joints: 1;
  connections: Location.outbranch[1] = 1,
               Processor.inbranch[1] = 1;
end;

lpLPGraph : graph
  icons: Location(0, 1), Processor(1, 1), LPblock(l, 0);
  joints: 1..2;
  connections: Location.outbranch[1] = 1,
               Processor.inbranch[1] = 1,
               Processor.outbranch[1] = 2,
               LPblock.inbranch[1] =2;
end;

l2pGraph : graph
  icons: Location(l, 1), Location(l, 1), Processor(2, 0);
  joints: 1..3;
  connections: Location[l].inbranch[1] = 1,
               Location[1].outbranch[1] = 2,
               Location[2].outbranch[3] = 2,
               Processor.inbranch[1] = 2,
               Processor.inbranch[2] = 3;
  externals: (1), ();
end;
MIL: a language for defining icon-replacement grammars

12pLPGraph : graph
icons: Location(1, 1), Location(0, 1), Processor(2, 1), LPblock(1, 0);
joints: 1..4;
connections: Location[1].inbranch[1] = 1,
Location[1].outbranch[1] = 2,
Location[2].outbranch[3] = 2,
Processor.inbranch[1] = 2,
Processor.inbranch[2] = 3,
Processor.outbranch[1] = 4,
LPblock.inbranch[1] = 4;
externals: (1), ();
end;

LP3Graph : graph
icons: LPblock(0, 2), LPblock(1, 0)[2];
joints: 1..2;
connections: LPblock[1].outbranch[1]=1,
LPblock[1].outbranch[2]=2,
LPblock[2].inbranch[1]=1,
LPblock[3].inbranch[1]=2;
end;

lGraph : graph
icons: Location(1, 1);
joints: 1;
connections: Location.inbranch[1]=1;
externals: (1), ();
end;

ActorNet : grammar
terminals: {l, p};
nonterminals: { LP };
startgraph: sGraph;
productions: {
LP ::= lpGraph,
LP ::= lpLPGraph,
LP ::= 12pGraph,
LP ::= 12pLPGraph,
LP ::= LP3Graph,
LP ::= lGraph
}
end;
This section gives the detailed and concrete syntax definition of MIL. The notation used here is compatible with the YACC parser generator system.

%token ICON IDENT INTEGER END LABEL SHAPE GRAPH INBRANCHES
%token OUTBRANCHES
%token CONNECTIONS ICONS JOINTS TO GRAMMAR TERMINALS
%token NONTERMINALS
%token STARTGRAPH PRODUCTIONS BECOMES EXTERNALS
%token INBRANCH OUTBRANCH SUBGRAPHS
%token ': ' ';' ',' '.' '[' ']' '='

grammar:
  definition.list

definition.list:
  definition
  | definition.list ';' definition

definition:
  /*nothing*/
  | IDENT ':' a.definition

a.definition:
  icon.def
  | graph.def
  | grammar.def

/** Icon rules */

/*
//Example icon definition:
//
//    +-------------------+
//    |                   |
//    |  --- someLabelName ---[m]--- |
//    |    --[n]---     |
//    +-------------------+

anIcon: icon(n, m)
label: someLabelName;
shape: aShape(1, 2, 3, 4, 5);
inbranches: 2, n;
outbranches: m
MIL: a language for defining icon-replacement grammars

end;
*/

icon.def:
   ICON class.param.list icon.field.list END ;

class.param.list:
   parameters ;

icon.field.list:
   icon.field |
   icon.field.list ';' icon.field ;

icon.field:
   /*nothing*/ |
   LABEL ':' IDENT |
   SHAPE ':' IDENT shape.param.list |
   INBRANCHES ':' comma.separated.param.list |
   OUTBRANCHES ':' comma.separated.param.list ;

shape.param.list:
   parameters ;

parameters:
   /*nothing*/ |
   ' (' param.list ')' ;

param.list:
   /*nothing*/ |
   comma.separated.param.list ;

comma.separated.param.list:
   param |
   comma.separated.param.list ',' param ;

param:
   IDENT optional.index |
   INTEGER ;
B.5 BNF Definition of MIL

/** Graph rules */

/*
Example graph definition:

//
//    il    i2
//   jx        jz[a]    jy
// x--- function +--[a]---x---[a]--- function +--x
//         +--------
//

aGraph : graph( a )
    icons: il, i2;
    joints: jx, jy, jz[a];
    connections: il.inbranch[1] = jx,
                 i2.outbranch[1] = jy,
                 il.outbranch[a] = jz[a],
                 i2.inbranch[a] = jz[a];
    externals: (jx), (jy)
end
*/

graph.def:
    GRAPH class.param.list graph.field.list END
    ;

graph.field.list:
    graph.field
    | graph.field.list ';' graph.field
    ;

graph.field:
    /*nothing*/
    | ICONS ':' icon.list
    | JOINTS ':' joint.list
    | CONNECTIONS ':' connection.list
    | SUBGRAPHS ':' graph.list
    | EXTERNALS ':' parameters ',' parameters
    ;

icon.list:
    /*nothing*/
    | comma.separated.icon.list
    ;

comma.separated.icon.list:
    an.icon
    | comma.separated.icon.list ',' an.icon
    ;
an.icon:
    icon.def
    | IDENT class.param.list optional.index
    ;

graph.list:
    /*nothing*/
    | comma.separated.graph.list
    ;

comma.separated.graph.list:
    a.graph
    | comma.separated.graph.list ',', a.graph
    ;

optional.index:
    /*nothing*/
    | index
    ;

index:
    '[' INTEGER ']
    | '[' IDENT ']
    ;

joint.list:
    /*nothing*/
    | comma.separated.joint.list
    ;

comma.separated.joint.list:
    joint.range
    | comma.separated.joint.list ',', joint.range
    ;

joint.range:
    INTEGER TO INTEGER
    | a.joint
    ;

a.joint:
    INTEGER
    | IDENT optional.index
    ;

collection.list:
/*nothing*/
| comma.separated.connection.list
|

comma.separated.connection.list:
| a.connection
| comma.separated.connection.list ',' a.connection
|

a.connection:
| IDENT optional.index '.' icon.branch '=' a.joint
|

icon.branch:
| INBRANCH optional.index
| OUTBRANCH optional.index
|

/** Grammar rules */

grammar.def:
| GRAMMAR class.param.list grammar.field.list END
|

grammar.field.list:
| grammar.field
| grammar.field.list ';' grammar.field
|

grammar.field:
| /*nothing*/
| TERMINALS ':' '{' param.list '}'
| NONTERMINALS ':' '{' param.list '}'
| STARTGRAPH ':' a.graph
| PRODUCTIONS ':' production.set
|

a.graph:
| IDENT class.param.list optional.index
| graph.def
|

production.set:
| '{' production.list '}'
|

production.list:
| /*nothing*/
| comma.separated.production.list
comma.separated.production.list:
  production
  | comma.separated.production.list ',' production

production:
  IDENT BECOMES a.graph

Appendix C

Undirected Icon-replacement Grammar

This appendix presents a model for the description of undirected icon-replacement grammars.

Undirected icon-replacement grammars are similar to icon-replacement grammars except that the icon branches are not directed. It is thought that some visual languages are represented more naturally without the explicit use of directed icon branches, such languages include NS-diagrams.

C.1 Language model

The language model of undirected icon-replacement grammars is quite similar to the directed model. In fact, the same generalizations presented in chapter 3 (see figure 3.1 hold here as well.

C.2 Undirected Icon-replacement Grammars

This section briefly outlines the formal model for defining undirected icon-replacement grammars.

C.2.1 Undirected Icon-graphs

An Undirected icon-graph is a structure of (possibly) interconnected icons. Icons are connected by attaching one (or more) of their branches to the same joint.

More formally, an icon-graph is a 3-tuple \((I, J, map)\), where:

- \(I\), is a finite set of icons;
148 Undirected icon-replacement Grammar

- \( J, \) is a finite set of icon joining points; and
- \( map : I \rightarrow \text{seq}[J], \) is a mapping that defines how the branches of the different icons are connected to the joints;

C.2.2 Labelled icon-graphs

An icon-graph with labelled icons is referred to as a labelled icon-graph. A labelled icon-graph over a finite set of labels \( (LABEL), \) is modelled as a 4-tuple \((I, J, map, label), \) where:

- \((I, J, map): \) is an icon graph; and
- \( label : I \rightarrow LABEL, \) is a mapping, assigning labels (names) to icons in the graph.

In the sequel we will assume that all icon-graphs we discuss are labelled using a fixed set of labels \( LABEL. \) Furthermore, we will refer to the set of all such graphs as \( LG. \)

C.2.3 Labelled undirected icon-graphs with external joints

A labelled icon-graphs with external joints can be modelled as a tuple \((I, J, map, label, gmap), \) where:

- \((I, J, map, label): \) is a labelled icon-graph; and
- \( gmap \) are sequences of joints, that define how \( G \) may be connected to other icon-graphs.

We will refer to the set of all labelled icon-graphs with external joints, as \( LGE. \) Furthermore, we refer to the union of \( LG \) and \( LGE \) as the set of all labelled graphs \( (ALG). \)

C.2.4 An order concept

The number of icon branches define, what we will refer to as, an icon's order. The order of any icon is defined as the number \( n, \) where \( n \) is the number of icon branches. The order of all icons in an icon-graph \( G = (I_G, J_G, map), \) is given by:

\[
\text{order}_G : I_G \rightarrow \mathbb{N} \times \mathbb{N} \\
\text{order}_G \triangleq \lambda i : I_G \cdot \# \text{map } i)
\]
Furthermore, we define an order concept, analogous to the one defined for icons, for labelled icon-graphs with external joints. It is defined as the number \( n \), where \( n \) denotes the number of external joints in \( g_{map} \). More formally:

\[
\text{order} : LGE \rightarrow \mathbb{N} \times \mathbb{N} \\
\text{order} \triangleq \lambda g : LGE \bullet (#g_{map})
\]

### C.2.5 Icon replacement

Icon replacement is the act of removing an icon from a graph, and then connecting a substitute graph in its place. The icon and its replacement graph must have the same order, i.e. \( \text{order}_G i = \text{order} R \).

A more formal definition of icon replacement is given based on two functions \( \text{Remove} \), and \( \text{Join} \), which correspond to the two steps of icon replacement. That is, \( \text{Remove} \) defines how an icon is removed from a graph, and \( \text{Join} \) defines how the replacement graph is attached in its place.

\[
\text{Remove} : ALG \times I \rightarrow LG \\
\text{Remove} \triangleq \lambda G : ALG; i : I \mid i \in I_G \bullet \\
(I_G \setminus \{i\}, J_G, \{i\} \ll map_G, \{i\} \ll \text{label}_G)
\]

\[
\text{Join} : ALG \times LGE \times \text{seq}[J] \rightarrow LG \\
\text{Join} \triangleq \lambda G : ALG; R : LGE; Map : \text{seq}[J] \bullet \\
(I_G \cup I_R, J_G \cup J_R \setminus (\text{ran } g_{in_R} \cup \text{ran } g_{out_R}), \\
map_G \cup \text{Jrep}(I_R, \text{map}_R, g_{map_R}, \text{Map}), \\
\text{label}_G \cup \text{label}_R)
\]

Where the auxiliary function, \( \text{Jrep} \), replaces the external joints of the replacement graph with the corresponding joints of the removed icon. \( \text{Jrep} \) is defined by:

\[
\text{Jrep} : P I \times (I \rightarrow \text{seq}[J]) \times \text{seq}[J] \times \text{seq}[J] \rightarrow (I \rightarrow \text{seq}[J])
\]

\[
\text{Jrep} \triangleq \lambda I_R : P I; \text{io}_R : I \rightarrow \text{seq}[J]; \text{gio}_R : \text{seq}[J]; \text{io}_i : \text{seq}[J] \bullet \\
\{x : I_R \mid \text{ran } \text{io}_Rx \cap \text{ran } \text{gio}_R = \emptyset \bullet x \mapsto \text{io}_Rx\} \cup \\
\{x : I_R \mid \text{ran } \text{io}_Rx \cap \text{ran } \text{gio}_R \neq \emptyset \bullet \\
x \mapsto (((\text{io}_Rx \triangleright \text{ran } \text{gio}_R)) \cup \\
\{n : 1 \ldots \#g_{io_R}; m : 1 \ldots \#\text{io}_Rx \mid \\
(\text{io}_Rx)m = \text{gio}_R n \bullet m \mapsto \text{io}_i n\})\}
\]

We can now define the icon replacement functions as:

\[
\text{Replace} : ALG \times I \times LGE \rightarrow LG \\
\text{Replace} \triangleq \lambda G : ALG; i : I; R : LGE \bullet \text{Join}(\text{Remove}(G, i), R, \text{map}_G i)
\]
C.2.6 Grammars

An undirected icon-replacement grammar is defined by a 4-tuple \((V, N, S, P)\), where:

- \(V \subseteq LABEL\), is the vocabulary set, i.e. the (finite) set of all labels used in the language;
- \(N \subseteq V\), is the set of non-terminal vocabulary;
- \(S : ALG_V\), is the starting graph; and
- \(P : N \leftrightarrow ALG_V\), is a finite set of production rules.

C.2.7 Derivation of Language Constructs

Applying a production rule \(p = (l, R)\), to a graph \(G\), means finding an icon in \(G\) which is labelled with \(l\), and then replacing it with \(R\). The icon (to be replaced) and \(R\) must have the same order. This is denoted by the predicate Applicable, which is defined as:

\[
Applicable(p, G) \triangleq \exists i : I_G \cdot \text{label}_G i = l \land \text{order}_G i = \text{order}R
\]

The graph resulting from applying \(p\) to icon \(i\) in \(G\), is said to be derived from \(G\). A derivation is denoted by: \(G \Rightarrow_{(p, i)} G'\). And the resulting graph is given by:

\(G' = \text{Replace}(G, i, R)\).

Furthermore, a sequence of derivations: \(G \Rightarrow_{(p_1, i_1)} G_1 \Rightarrow_{(p_2, i_2)} G_2 \cdots \Rightarrow_{(p_n, i_n)} G^n\), is denoted by: \(G \Rightarrow_d G^n\), where \(d = (\langle p_1, i_1 \rangle, \langle p_2, i_2 \rangle, \cdots \langle p_n, i_n \rangle)\), or alternatively \(G \Rightarrow^* G^n\) if we are not interested in the exact derivation sequence.

C.2.7.1 Graphs defined by a grammar

The set of all graphs defined by an icon grammar \(\Gamma\), is the set of all graphs generated by it. This set is defined by:

\[
\text{Graphs}(\Gamma) \triangleq \{ G : ALG_V \mid \exists d : \text{seq}[P_\Gamma] \cdot S_\Gamma \Rightarrow_d G \}
\]

C.2.7.2 Languages

The language defined by an icon grammar \(\Gamma\) is a subset of \(\text{Graphs}(\Gamma)\) that is restricted to graphs that do not contain non-terminal elements (i.e. icons with non-terminal labels). The language is denoted by \(\text{Lang}(\Gamma)\), and defined as:

\[
\text{Lang}(\Gamma) \triangleq \{ G : \text{Graphs}(\Gamma) \mid \text{ran} \text{label}_G \cap N_\Gamma = \emptyset \}
\]
C.3 Summary

This appendix presented an alternative model for defining visual languages. This model, unlike the one presented in chapter 3, is based on an undirected icon–graph model. It is suggested that using such a model can yield a more natural presentation of some visual languages.
Bibliography


Curriculum Vitae

Sherif El-Kassas was born on 23 November 1959, in Giza, Egypt.

He received a B.Sc. in Electronic Engineering, from the Faculty of Engineering, University of Cairo, Egypt, on July 1983; the Diploma of the PHILIPS International Institute of Technological Studies, on December 1984; and the Master of Electronic Engineering (MEE), from The Netherlands Universities Foundation For International Cooperation (NUFFIC), on May 1985.

From September 1985 to January 1989 he worked as a technical staff member at The Computer Center, The American University in Cairo, Egypt. During the same period he also worked as a part time instructor in the Computer Education Program of the American University in Cairo, and as a development consultant at The Information and Decision Support Center of the Council Of Ministers, Egypt.

From April 1989 to June 1993 he worked towards his Ph.D. at the Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands.

Starting July 1993 he started his job as the manager of the Research and Development department at The Computer Center, The American University in Cairo. He also works as a consultant for the Regional Information Technology and Software Engineering Center (RITSEC).
Theses

included by the thesis

Visual Languages:
Definition and Applications in
System Development Environments

Sherif El-Kassas
1. The integration of formal methods and visual languages yields a sound foundation for the definition of modern design technologies. This thesis.

2. In system development a specification language acts as a tool for: describing system behaviour; reasoning about system description; and managing the size and complexity of the problem. This thesis.

3. System Engineering cannot be equated with the adoption of formal methods.

4. Visual languages can be seen as multi-dimensional generalizations of one-dimensional string languages.

5. Pure visual languages are not viable for system development applications.

6. An inherent drawback of using a generic editor is that many of the possible visual abstractions and layout constraints are language dependent, and, hence, cannot be supported in a language unaware editor. This thesis.

7. Often, the truth is realizable but not provable.

8. Everything should be built top-down, except for the first time. (/usr/games/fortune).

9. Visual Basic is not a visual programming language.

10. It is becoming impractical to be non-theoretical.