José Paulo Leal

Graphical Interaction in Logic Programming

Tese submetida para obtenção
do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Abril de 1997

To my parents

Acknowledgements

When a project unfolds for several years, it is difficult to recall everyone that somehow contributed to the final outcome, and it is particularly difficult to measure the significance of each individual contribution. Hence, I will certainly forget to acknowledge many people that contributed to this thesis but I would be ungrateful if I failed to mention those without whom it would not have been written.

I am specially grateful to my advisor, Luís Damas, for providing the early motivation of this work. To Alípio Jorge and João Pedro Santos for their collaboration in implementing and testing Edipo and to numerous users of Edipo and the Ytoolkit for their bug reports and interest in these tools.

I am also very grateful to all my colleagues at the Computer Science Department, both for the friendly and stimulating working environment and for their help in proof reading this thesis. I am particularly thankful to Vitor Santos Costa, Sabine Broda, Miguel Filgueiras and Fernando Silva. Vitor's comments where very important to make this thesis readable.

Finally, I must also thank Elsa for giving me the emotional stability I needed to complete this thesis as well as for her patience during the last months.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The fact that we can see a dragon in a cloud, a penguin in a spot of rust or a dog in a shadow has always fascinated me. If we think about it, seeing a dog when one looks at one is already fascinating, yet recognizing a dog within another object is the side of our visual perception that intrigues me. We can still see the object where the figure is represented, it is not (necessarily) a delusion, the figure is not actually there, and we are completely aware of all these facts.

I imagine that the first people that painted on cave walls (probably they drew sketches on the sand first, but no one knows) felt that something fascinating was happening; they could see the form of an animal, or of their hands, in what was still the cave's wall. In any event I do not think they realized that those were the first steps in the use of **graphical representations**, and that the descendents of those images would be, to their own descendents, almost as important as the objects themselves.

Paintings are good examples of how representations are often more valued than the subjects they represent. Take, for instance, the painting "La trahison des images" painted in 1929 by the Belgium painter René Magritte (see Figure 1.1), priceless compared to the pipe used as model. This painting is also a warning against the fallacy of images: any 6 year-old would recognize in this picture a pipe. But in his schoolmaster-like handwriting the author explains that it is not actually a pipe; rather the mere representation of a pipe, we might add (by the way, Figure 1.1 is just a representation of the actual painting).

This warning was produced when photography and cinema were not widely established, computers and television had not been invented, Internet and virtual reality were not even Science Fiction. It is only more pertinent today, in a world that is filled with pictures that look more real than reality itself.

Figure 1.1: La trahison des images

The power and the danger of graphical representations are by no means restricted to objects. Ideas too can be depicted graphically. Diagrams, graphs and plans graphically represent ideas that otherwise would be very difficult to express or understand. Graphics are used in almost every field of scientific knowledge, and even more since computers simplified the production of complex graphics.

With computers arrived the notion of **interaction** with graphic representations. Graphical users interfaces (GUI) not only allow the visualization of metaphorical objects on the screen but also the manipulation of those objects. For instance, buttons displayed on computer screens can be pushed with a pointing device such as a mouse or a light pen. Nowadays, virtually all end-user computers have a graphical interface that replaces the old text-based command interpreters of applications and operative system shells.

This approach has been used in program development to create **graphical programming environments**. In general, these environments integrate several tools, such as specialized editors, graphical debuggers, GUI builders, that use the graphical primitives and interactive graphical objects provided by windowing system toolkits. Previous research on programming environments focused on the use of graphical interfaces to replace text-based command line interpreters, and several commercial environments were developed, specially for personal computers. The level of integration of graphical tools in these environment is not yet satisfactory: including graphical primitives or graphical objects in text-based tools is frequently impossible and sharing data in graphic-based tools seldom straightforward.

Given the success of graphical users interfaces in general, and of graphical programming environments in particular, extending this interactive model has been an active area of research in the last decade. This research is based on the belief that graphics can enhance the understanding of complex ideas, such as programs and data, better than the textual representations that supported the development of programming up to its present stage. There is the conviction that an interactive environment where programs and data are manipulated graphically would simplify the task of programming, and would eventually lead to a new paradigm - **visual programming**.

Although appealing, this idea has been difficult to implement so far and visual programming languages do not seem to be replacing text-based languages. A recurrent problem in this area of research is mixing up "graphical" with "non-textual". Hence some visual programming systems try to abolish text altogether, replacing it by drawing primitives (lines, arcs, etc.), and icons (bitmaps and pixmaps). It is true that, in some cases, an icon is a good substitute for a word – a text label – to represent a symbol or an idea. Nevertheless text has some advantages: it is easy to produce using a keyboard, its meaning is less ambiguous and is simple to combine to form expressions. Last but not least, character strings are in fact graphical objects, just like icons.

The ultimate goal of visual programming, i.e. the specification of a program in two-dimensional (or higher) fashion using graphical primitives instead of the one-dimensional streams of text in conventional programming [Mye90], is probably too ambitious. Nevertheless, there are several situations in program development where the use of graphical primitives has proven to be very helpful: the browsing of large and complex data structures, the visualization of program executions for debugging, graphical interfaces specification, etc.

1.1 Motivation and Goals

There are cases where the text-based representation of data and programs is insufficient and an extended representation including other graphical elements, such as drawing primitives and icons and even graphical objects can be useful. This is a frequent case with Logic Programming (LP) and therefore it justifies the development of a new and specialized programming environment to cover its needs.

In the last decade computing environments evolved from purely text-based to incorporate graphical and even multimedia capabilities. Logic Programming (LP) environments followed this general trend. With the advent of workstations offering windowing systems, LP environments started integrating Graphical Users Interfaces (GUI). In general those GUIs were restricted to combinations of a standard widget class set providing only menu bars, dialog boxes, scrollable text windows. Only in very few cases new widget classes were designed specifically for LP applications. The exceptions include the browsing of terms and the visualization of executions, lately with emphasis on parallel LP execution. The resulting environments were an amalgam of independent tools, each with its own visualization syntax and editing features difficult to customize, and incapable of sharing formatted data.

In LP environments, most tools for displaying terms – such as term browsers – provide only a small fixed number of predefined graphical syntaxes and cannot be customized to a particular application area. This is inconvenient for fields where specific graphical syntaxes have already been created. Examples of such areas are natural language [Win83], and in particular situation semantics [BC91]. In any case, and allowing more or less customization, graphical visualization is in general limited to browsers and cannot be used in editors or other tools and programs that need user input.

A second fundamental problem with most LP environments is the limitations regarding both the sharing of formatted data among tools, and between tools and programs developed within the environment. In these environments tools are restricted to unformatted data transference in text format. Hence, a term that is displayed graphically in a term browser cannot be copied to a debugger with its original graphical layout. This implies that even in the cases where a new graphical syntax could be created for one tool, it could not be consistently used in all the environment's tools. A similar problem arises when binding input procedures to key-strokes or mouse actions, and even in its definition. The definitions attached to a tool are not shared with other components of the environment. Hence, pointing to a graphical object or pressing a key may have different semantics if the user is editing a clause in a program editor or entering a value to be processed by that same clause when the predicate is being evaluated.

I believe that an environment that results from gluing different types of tools will most certainly lack several important characteristics that a programming environment for LP should have, such as:

Support for customizable graphic syntaxes. Each LP area tends to develop its own syntax for visualization of terms. Thus, the environment should provide a simple and powerful mechanism to enhance customization of visualizations.

Interactive graphic representations. Most LP applications need more than just visualization, they require full graphic interaction; i.e. the possibility of editing graphically formatted data and converting it to their own internal data representations.

Common interaction formats. A uniform data representation enables different applications and tools to share formatted data. It is important for LP applications not just to share data, but also to share the graphical syntaxes and interaction features supported by the environment.

Text and graphic integration. LP evolved using a textual notation which is still the best for a large number of cases, and most users are familiar with it. Consequently, to keep a balance between a pure graphical editor and a textual editor, as pointed out by Petre [PP92], text-based interaction is necessary in an LP environment.

Oriented towards the LP paradigm. To include graphical interaction in all aspects of programming development in LP – program editing and execution, data browsing and editing, GUI definition, program debugging and animation, etc. – it is important to have a strong connection between graphical interaction and the programming paradigm itself.

To achieve these goals all the components of the interface must share a common set of procedures to format, display and interact with LP objects of any kind (data terms, predicate clauses, program executions, etc.). In this thesis this common set of procedures is designated as an **interaction engine**, whose role is to provide interaction services to a number of independent components of a programming environment. Hence, the two major goals of the research presented in this thesis are:

1. The definition of a **graphical interaction framework** serving as the basis for the implementation of a common interaction engine for LP environments.
2. The implementation of a **programming environment** for Prolog to test the adequacy of the proposed framework in the implementation of an integrated programming environment for a language of the LP paradigm.

1.2 Graphical Interaction in LP

The integration of graphical interaction in LP with such different purposes as program and data editing, program execution, debugging and animation, and GUI definition and management, requires a close connection between the LP paradigm and graphical interaction. A precise characterization of this connection establishes what will be designated in this thesis as a **graphical interaction framework** and will be the formal basis for the implementation of a programming environment for LP.

Following the approach proposed by Cox and Roman [CR94], the integration of graphical interaction with LP requires the solution of a fundamental problem: the identification of a correspondence between concepts of LP and graphical interactions. In particular it is necessary to map into LP the two fundamental concepts of graphical interaction: visualization and manipulation. The former relates to what is presented on screen to the user; the later relates to the user's actions using mouse and keyboard.

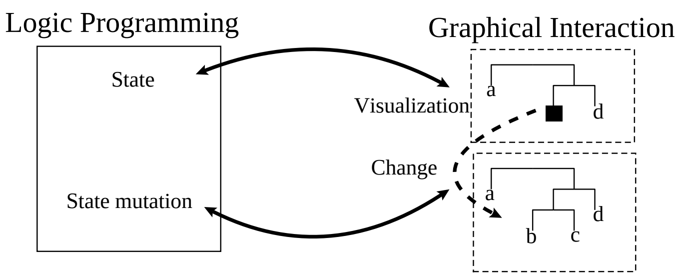

Figure 1.2: Mapping LP to graphical interaction

Mapping visualization into LP is straightforward since anything that can be regarded as a **state** can be visualized using an appropriate visualization syntax. On the other hand interaction is difficult to map into LP due to the absence of the **mutable state** concept in this framework.

The concept of variable in LP – logic variable – cannot be used to hold a changing state since it does not support destructive assignment. This problem is "solved" in almost every implementation of LP languages, in particular in Prolog compilers [DC89, C$^+$, Qui91], introducing extra-logical predicates that dynamically change the clausal database or implement an internal database.

Extending LP to integrate the notion of mutable state is a current field of research. Most proposals introduce objects in LP, that encapsulate a notion of state, either as first-class citizens [Con88] or by using concurrency where a process blocked on input is viewed as an object [ST83]. Nevertheless, these object-oriented approaches that introduce a mutable state in LP cannot be used in this work since they are not part of the basic LP framework, and hence cannot be used in the definition of an interaction framework for a language such us Prolog.

In the basic LP framework a simpler concept of "state" can also be defined using the same two approaches of the object oriented research: either as the resolution of a logic program , or as the data terms of a logic program. In the former approach the resolvent is a state and each resolution step performs a state change. These two concepts can be mapped respectively into visualization and interaction to relate LP with graphical interaction. In the latter approach data terms are identified with visualization but it does not support a mapping interaction into the LP paradigm. These two opposite approaches were used in this work to create mappings that are presented and compared in the next subsections.

1.2.1 Interactive Resolution

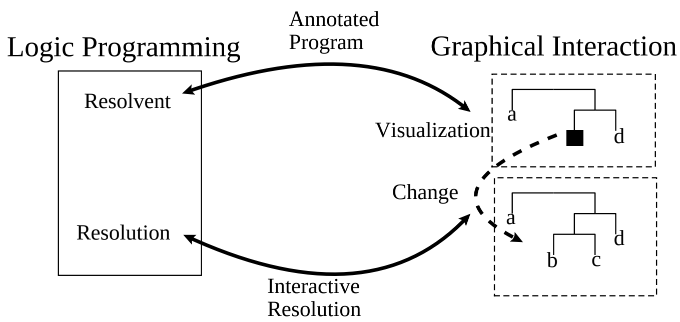

Figure 1.3: Execution based mapping

The fundamental idea of this mapping is to identify the process of solving a logic program with the process of interacting with an interface. In this mapping a resolvent (the state) is identified with the interface visualization and a resolution step (state modification) with a change in the interface, as depicted in figure 1.3. To assign a visual semantics and to interact with the execution of a logic program, the concepts of logic program and resolution have been extended [Lea94], to integrate the concepts of **annotated logic program** and of its **interactive resolution**. Clauses of logic programs are annotated to specify their visual syntax. Interactive resolution integrates interaction as part of the refutation procedure in the derivation of goals in an annotated logic program.

To exemplify how these concepts are used in the description of a graphical interaction, let us consider the following small logic program that recognizes binary trees. The example shows how the resolution of this program can be identified with the interactive process that creates the visual representation of a binary tree.

```
btree(leaf).
btree(btree(L,R)) :- btree(L),btree(R).
```

As explained, each clause is annotated with a term defining its visualization. The annotations are connected to each clause using the infix operator "\\". The string "`leaf`" is defined by the `leaf_visualization` style and the `btree_2_visualization` style defines two lines representing the branches of the binary tree.

```
btree(leaf)                              \\ leaf_visualization
btree(btree(L,R)) :- btree(L), btree(R) \\ btree_2_visualization
```

The annotated program is solved by a custom interpreter implementing an interactive resolution strategy. The interpreter creates a graphical representation of the resolvents by composing the graphical representation of each clause used in the derivation.[1] In the interactive resolution strategy the selection of a goal to be derived and the selection of a clause to expand a goal is done by the user. The interactive resolution strategy does not define how this selection should be implemented, but an actual interpreter may use buttons, pull-down menus, or any other kind of graphical object.

Three consecutive resolvents of the derivation of the goal `btree(btree(A,B))` on the previous annotated logic program are represented on figure 1.4. In this rendering of the resolvents an expansion using the first or second clause is represented, respectively, by the string "`leaf`" and by two lines with a common end-point, as defined by the annotations. The pending goals are represented as boxes.

In the first situation (left diagram) the initial goal was derived into two new goals. The user first chose to derive the goal presented in the right using the second clause of the program obtaining the central diagram, and then the user derives the goal in the left using the first clause obtaining the diagram in the right.

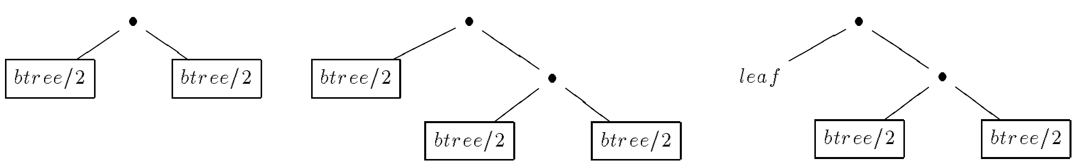

Figure 1.4: Interactive resolution of a logic program

It should be noted that the interactive resolution of annotated procedures can be combined with a standard strategy, such as the Prolog strategy, for non-annotated predicates. Visual interaction can thus be introduced in any logic program just by annotating the procedures that require user interaction.

The use of this form of graphical interaction in areas such as program execution/debugging and algorithm animation is straightforward: the interactive resolution both provides a visualization of the execution and introduces user control at the important points of the execution.

Term edition can also be performed using interactive resolution. For instance, to create a graphical editor for binary trees we can solve interactively the goal `btree(X)` in the previous annotated logic program. At the end of the execution the variable `X` must be bound to a term representing a valid binary tree. In general, it is simple to create a specialized graphical editor by just annotating a logic program that recognizes the valid terms with annotations

[1] In the prototype system the annotations are expanded at compile time into semantic rules that are used to compute an attributed tree that is the basis for visualization. At this stage the details of how the visualization is computed from the annotations are not pertinent.

that describe their visual syntax.

A disadvantage of using interactive resolution for structure editing in LP is the little freedom that the model allows. For instance, it is difficult to create a general term editor for functors without a predefined name and arity. Another inconvenient is the fact that the interaction process is regulated by a meta-program. Experiments with the prototype showed that this approach is too inefficient for user interaction. This problem could be worked around with the creation of a special compiler for annotated logic programs but that would lead to a more complex implementation of the interactive resolution strategy.

1.2.2 Term Editing

In term editing, the interaction process is identified with the edition of a set of terms. In this mapping the visualization is identified with the layout of a set of terms and changes in the interface with the edition of this set of terms.

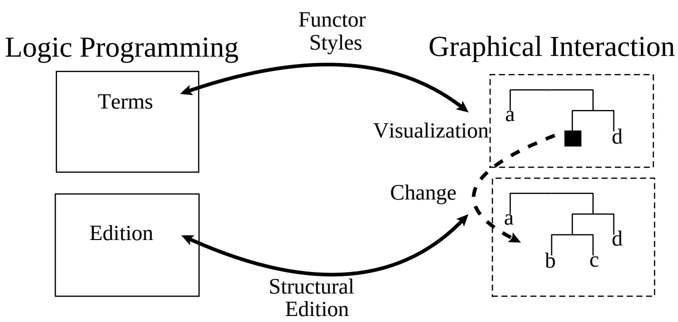

Figure 1.5: Edition based mapping

In this approach the visualization is controlled by **styles**. These have a role similar to that of annotations in interactive resolution: they describe the visualization of the individual components, in this case of functors. This mapping uses system declarations, instead of annotations, to assign visualization styles to functors. The declarations can be written in textual format using the traditional Prolog declaration syntax. A visualization similar to the `btree` example presented in the previous subsection would use the following declarations:

```
?- view leaf/0  as leaf_visualization.
?- view btree/2 as btree_2_visualization.
```

As emphasized by Figure 1.5, this mapping is not as integrated in LP as the interactive resolution since the notion of change in graphical interaction has no correspondence to a LP concept. In the interactive resolution model a modification in the interface is identified with a step in the resolution of a logic program. But if terms are selected as the basis for interface visualization then it is not possible to map the notion of interface change into a LP concept as the only term "modification" in LP framework is unification. If interface modifications were restricted to the binding of logic variables then certain modifications would be impossible to achieve. For instance, if the term `f(a,X)` represented the interface visualization then the term `f(b,C)` could not be obtained from the interaction.[2]

In spite of not providing a LP equivalent for interface modifications, the term editing model has the advantage of decoupling the semantic and syntactic representation of the interface, unlike the interactive resolution. The screen's layout is based on both a set of terms, that can be regarded as the **interface semantics**, and on an assignment of styles to functors that control the **interface syntax**. Hence, it is very simple to use different layouts for an interface. For instance, a term can switch between a conventional textual representation and a graphical representation just by assigning different styles to its functors. Similarly, it is simple to reuse a graphical syntax in a different context just by using the same assignment of style to functors. For instance, a graphical syntax developed for presenting data terms can be used in a program editor to assign the same graphical syntax for the terms when these occur in clauses.

1.2.3 Comparing Both Models

In order to choose an interaction model for a LP programming environment, the interactive resolution and the term editing models were compared both from a formal and a practical point of view. Although both these two points are important for the decision, the latter is particularly important since the main goal of this research is to actually produce an usable graphical environment for LP. This goal led to the design and implementation of prototypes for both models. The main conclusions of the test with those prototypes are summarized in the next paragraphs.

Execution versus data orientation: The interactive resolution is execution oriented while term editing is data oriented. The former is better suited to integrate graphical interaction in the execution of programs (graphical debugging, algorithm animation, etc.) and the latter is better suited to graphical data editing (specialized editors, term browsing, etc.). In any event, using meta-programming both models can be used for these two tasks. Interacting with an execution in a term oriented model can be achieved by interacting with the meta-interpreter's data, and interacting with terms

[2]Note that in the interactive resolution model this would correspond to the evaluation of the goal `f(X,Y)` in the predicate `f/2` with the clauses `f(a,X)` and `f(b,X)`.

in an execution oriented model can be achieved by using a program that recognizes a class of terms.

Input control: The interactive resolution model restricts the user input to predefined sequences of actions. The editing model does not control input, accepting any input that can be reduced to a term editing operation. Although guided input is an important feature, it is not desirable as a basic interaction methodology, since there is a significant number of cases where valid input cannot be predicted. It is preferable to introduce guided input on top of a free input model for the cases where it is relevant without losing the possibility of free input for the general case.

Formal semantics: The term editing model has deficiencies from a formal point of view since it does not provide a mapping for interaction in terms of a LP concept. On the other hand, the interactive resolution model maps both visualization and modification of the interface into the resolution of a logic program.

Efficiency: Although several optimizations can be introduced to the interactive resolution prototype, this approach relies too much on the execution of meta-programs to expect reasonable efficiency. This is a major drawback since a graphical environment must be fast enough to provide a smooth interaction with the user. An environment implemented using the term editing model is more efficient since it does not have the burden of controlling the interaction explicitly.

Flexibility: Interactive resolution ties the definition of the interactive process (the resolution of a logic program) to its visualization (the clause annotations). Term editing separates the two since the construction of visualizations is independent of the interaction process. Moreover, the visualization is defined by a set of terms and a set of styles assigned to functors, that can be seen respectively as a semantic and syntactic definition of the interface. This separation has as consequence that it is possible to independently change both representations, i.e. to change the visualization of an interface just by changing the style assignment or using the same visualization for an interface defined by different sets of terms.

Naturally each model has its own advantages and disadvantages. The interactive resolution provides a very clear mapping to the LP framework concepts but lacks efficiency when implemented in an actual environment. On the other hand, the term editing has a poorer theoretical foundation but simplifies the implementation of practical systems and enables the use of structure editing concepts [BS86, HT86]. Hence, the **term editing** model was selected as the basic interaction model of our LP programming environment. In the next chapters this interaction model and its implementation, as well as the programming environment and the packages that were developed to implement it, are discussed in detail.

1.3 Organization

The Magritte programming environment for graphical interaction in LP is based on the term editing model. It is a graphical environment for the LP language Prolog and is itself written in Prolog, and interfaces with the XWindows windowing system through the C interface library Xlib [GNS87]. This environment is structured in four main packages:

Magritte The front-end layer of the environment. It includes the graphical users interface (GUI) and a set of tools based on the proposed interaction model. This package also includes mechanisms to produce specialized editing tools and to interface with the underlying Prolog engine.

TermsIO An interaction framework that implements the term editing interaction model. This package is the core of this environment since it provides the graphical objects where terms are displayed as well as a common library of methods to modify those terms, interactively or from a program.

Ytoolkit A GUI toolkit for the creation and management of graphical objects. It includes a set of standard graphical objects – such as buttons, pull-down menus, scrollable windows, etc. – and can be used to develop specialized object classes, in particular those created specifically for TermsIO. The toolkit is used to manage Magritte's GUI and those of the applications developed within the environment.

Edipo The low-level interface with the XWindows windowing system. It provides predicates for basic interaction tasks such as window management, text/graphic drawing, and event reading.

These four packages depend on each other as represented in Figure 1.6: the packages in the bottom of the diagram provide functionality for those of the top. Hence, Edipo may be used separately just as an interface to Xlib, and is in fact used for that purpose. The Ytoolkit and Edipo packages are being distributed together as a system for the development of user interfaces in Prolog, portable across several engines [Lea90].[3] TermsIO was only used in the development of Magritte so far, but it was defined independently from Magritte and thus can be used to implement other environments based on the graphical interaction with structured data.

The organization of this thesis follows the structure of the programming environment. The next four chapters describe each package, starting with the environment's front-end, followed by the interaction framework implementing the term editing model, then by the Prolog

[3] Currently they support the Yap[DC89], SICStus [C$^+$] and Quintus [Qui91] Prolog systems.

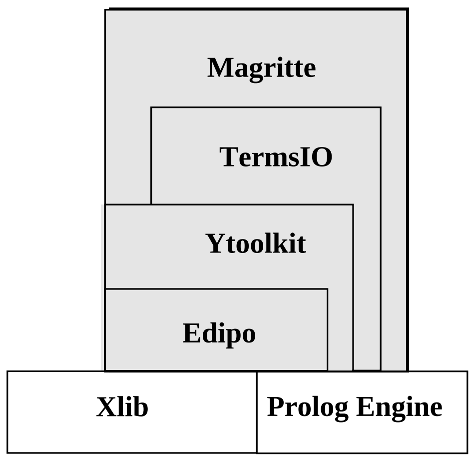

Figure 1.6: System structure

oriented GUI toolkit and finally by the low-level interface to the windowing system. Each chapter introduces the relevant research work related to its subject and defines it design goals and philosophy.

1.4 Contributions

Magritte's major contribution is the use of an interaction framework based on the concept of interaction model briefly presented in the previous section. The chosen model provides a term oriented interface definition that introduces a novel approach to the definition of interaction in LP. As a consequence, in Magritte interaction is regarded as a particular case of editing a set of terms. In this approach the definition of the interface semantics uses terms and the interface syntax associates graphical visualization rules to functors. This view of interaction simplifies the definition of interaction in such complex tasks as specialized structure editing and interaction with the execution of logic programs.

Another contribution of Magritte is the use in its front-end of concepts that have proven their usefulness in other contexts but that were never implemented in LP environments. From spreadsheets Magritte borrows the idea of integrating data, programs and executions in a single workspace, and from structure editing it borrows the idea of guided input. The contribution of this research was the adaptation of these ideas to LP. Hence, in Magritte's view of spreadsheets the matrix model was replaced by a tree model, and the structure edition does not use a context free grammar but is based on the concept of types of terms.

The interaction framework proposes a language for the graphical syntax of terms. This language associates to a class functors a set of rules that compute several visual attributes of the terms. This thesis includes a description of the programs that were implemented to "compile" this language to Prolog clauses, and to update the attribute values in response to the user's interaction. Both the language and the evaluation algorithms are based on attribute grammar concepts. The language extends the traditional semantic rules of attribute grammars with constructs for the definition in comprehension of sets of values. The evaluation algorithm of attribute grammars was also modified since the concept of types of terms replaced context free grammars as the formal definition of trees decorated with attributes.

The four packages are implemented in Prolog and executed as a single process. This simplifies the communication between all packages and their integration with the Prolog engine. The use of Prolog as the implementation language of a programming environment is also one of the novel aspects of this research work. In particular, the Ytoolkit contributes with yet another approach to object-oriented programming in Prolog.

Chapter 2

Magritte: a LP Environment

Magritte[1] is a programming environment based on a novel interaction framework specifically designed for interacting with structured data in Logic Programming (LP). This environment served the purpose of testing the the **term editing** model introduced in the last chapter. In Magritte the state of the interface is encoded by a set of terms and the layout of the workspace is the composition of the visualizations of the state components. Any modification of the layout reflects the changes in the set of terms and vice-versa. Consequently, in Magritte interaction and term editing are synonyms.

The term editing model enables the separation of the semantic and syntactic aspects of the interface. This feature is based on the fact that the meaning of an interface is independent from the details of its graphical rendering on screen. For instance, a message box with a string and a button can be rendered on screen in different shapes, sizes and colours, of the box, button and text. Nevertheless, the semantics of this interface component can be summarized by specifying that it is a message box and giving the particular string it presents. Thus, the information presented on the interface can be represented by a language that omits graphical details and concentrates on its **semantic representation**.

The editing model uses terms to represent the meaning of an interface. For instance, the semantic representation of a message box given as example could be a term such as `message_box(Message)`. In this case the functor `message_box/1` has a special interpretation that creates a message box as shown in Figure 2.1. Nevertheless, any term can be used as part of the interface semantics and if no special graphical syntax is assigned to it the traditional textual representation is used. The terms defining the semantics of Magritte's workspace can have different interpretations. They can be interpreted as the definition of a GUI, a predicate clauses, a goal, a refutation tree, etc. The interpretation is given by Magritte's

[1] Magritte was named after the surrealist Belgium painter that warned against the fallacy of images.

`message_box(Message)`

Figure 2.1: Syntax and semantics of an interface component

tools or by the applications developed within the environment that handle these terms.

The **interface syntax** describes the aspects that determine how an interface is displayed on the screen, independently of its meaning. In the above example the functor `message_box/1` selects a graphical syntax consisting of a gray rectangle with its first argument displayed in the center and an "OK" button in the bottom. The Magritte's interaction engine (described in the next chapter) computes a complete layout for the interface by composing the individual layouts of each functor.

Magritte borrows successful ideas from other interface paradigms such as spreadsheets and structural editing. The influence of these paradigms in the design of Magritte is discussed in the next section. This chapter introduces also the environment's users interface and the interaction engine described in detail in the next chapter. The remainder of the chapter discusses how Magritte takes advantage from this interaction framework to implement the editing features and the interaction with the underlying Prolog engine.

2.1 Design Philosophy

The design of Magritte was influenced by two successful interface paradigms: spreadsheets and structural editing. In this section we discuss the important aspects of each paradigm, their limitations from a LP point of view and some related work.

2.1.1 Spreadsheet Interface

Spreadsheets are extremely successful environments that enable users to enter data and "programs"[2] and format data using a simple and **unified framework** [Hud94]. Magritte "borrows" from spreadsheets the concept of integrating data, programs, and GUI into a single, unified interface.

Spreadsheets are perhaps the only successful visual programming environment since the "visual" versions of programming languages such as Basic or C provide little more than a GUI builder. Although a "visual flavoured" dialect of one of these languages is a "macro language" for an important commercial spreadsheet, it should be noted that it is not fully integrated in the spreadsheets interaction framework, i.e. it does not use the same type of interface that is used to edit data and expressions.[3] This approach is justifiable since an imperative program and its data have a completely different structure. Having different structures for data and programs is not a difficulty for LP since in this paradigm both programs and data are sets of terms.

Merging spreadsheets and LP was first tried in Perplex [SB89] – a Prolog-like language with constraints – that uses a **matrix interface** for its system environment. A Perplex cell does not have an associated formula to compute its value. Instead, there is a list of constraints, presented side by side with the array of cells, that inter-relates several cell values and is used to update their values when a change occurs. These constraints use logic variables that are identified with cells and referenced using its row and column position as in ordinary spreadsheets. For instance, if there is a constraint stating that `A1+2=A2+5` and the number `4` is entered in `A1` then the value of `A2` is changed to `1` to satisfy the constraint. Later, if the value of `A2` is changed the value of `A1` is also changed to satisfy the constraint. These value changes can trigger further changes to satisfy other constraints on the list. The constraint evaluator handles multiple solutions that are not presented on the matrix but on a separate window. Like the traditional spreadsheets Perplex is targeted to numeric applications and provides only textual visualization of terms.

In spite of what this experiment might suggest, the standard interaction framework of a spreadsheet – a bi-dimensional array of cells containing values or expressions in textual form – is not suited for a LP environment. Trees rather than arrays are the typical data program structure in LP. As a consequence LP data (terms) are structured, unlike spreadsheet data (numbers and strings) and require a more structured form of interaction. Spreadsheets lack the power to manipulate structural information, a strong point of the other paradigm that influenced Magrittes's design philosophy – structural edition.

[2] Expressions can be seen as a sort of programming language and, in that sense, spreadsheets are a visual programming environment.

[3] The early interfaces used in fact a matrix to edit macros but were abandoned.

2.1.2 Structural Edition

The idea of creating a highly interactive environment to enhance the development of programs is not new. In the beginning of the eighties, Tim Teitelbaum and Thomas Reps developed the Cornell Program Synthesizer which they described in their seminal paper [TR81]. This work was followed by other similar environments such as the PSG System [BS86].

These systems are usually called **language environments**, since they create interactive programming environments from a formal language definition, usually the language's context-free grammar. In opposition to a traditional text-based environment, where compilers use a grammar to parse a string and produce an abstract parse tree of the program, in a language environment the grammar is used directly in the edition of the program to interactively build the abstract parse tree; consequently, this process is usually called **unparsing**.

This form of editing prevents the user from writing syntactically incorrect statements and can be used both in textual and visual languages. In visual languages the editors make use of non-textual elements, like icons and graphic primitives, to enhance the understanding of the program. Language systems were used mostly to generate environments for procedural [TR81, BS86] and functional languages [Hag92, Car83]. The languages of these programming paradigms have large grammars that benefit from using an unparser. But LP languages have only one type of construct – terms – common to program clauses and data, that have a rather simple syntax, at least if we exclude Prolog operators. In any event, even the complete Prolog syntax is not so complex as the syntax of any procedural or functional language. Hence, a "classic" language environment, where structural edition is based on the language's grammar is of little use for the LP paradigm.

Although language environments are not applicable to LP languages, structural edition can be used in term edition based on **types** of terms rather than on their syntax. The concept of type is used to characterize the classes of terms of the arguments of a predicate for which it succeeds. Using type definitions, a LP environment can build a structural editor that helps the end user to insert only valid terms as arguments of a predicate.

2.2 User Interface

The users interface of Magritte is built over a workspace where a set of terms, shared by several applications, is subject to user interaction. The terms are presented using both textual and graphical representations and can be manipulated using a set of tools that allows both textual and structural modifications. These visualization and manipulation tools are

Menu bar

common to all applications developed within the environment.

When Magritte is started its top-level window looks like Figure 2.2. It consists of a single window with a multi-layer workspace where a set of terms can be inserted, edited, formatted, browsed, evaluated, spied, etc. These operations are implemented by Prolog predicates using the **TermsIO** common interaction engine to be described in the next chapter. This subsection describes the components and their functionalities from the user's standpoint.

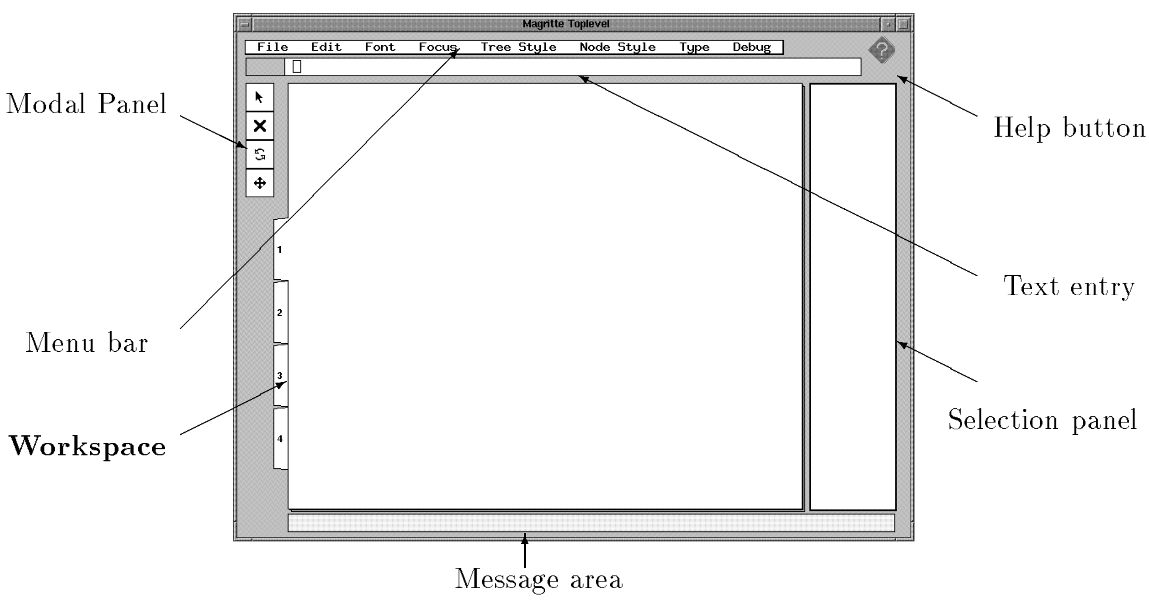

Figure 2.2: Magritte's top-level window

Workspace is the layout area in the interface, located in the center of the interface. It is a stack of several layers, each one an instance of the TermsIO widget class. The user can select the layer using the numbered labels on the left and then edit the terms on this sheet.

Modal panel aggregates several iconic buttons. These buttons switch between the editing modes which are, from top to bottom, focus selection, tree collapsing, tree unification and sheet dragging. The editing modes change the meaning of mouse clicks in the workspace.

Selection panel is the area of the interface used by Magritte to present the graphical representations of the valid inputs for the current insertion point.

Message area is used by the system to display error messages, hints and report messages.

Menu bar is composed of several menu buttons. Each menu lists several related environment commands such as file operations, edit commands, font selection or structural editing commands.

Text entry is used to provide an alternative textual editing on selected terms.

The interface was built using widget classes of the Ytoolkit described in chapter 4. The basic widgets (menu bar, dialog boxes, buttons, etc.) are specializations of the standard widget set. The main components of Magritte, the workspace layers and the selection panels, that require graphical interaction with structured data, are implemented using a special widget class provided by the interaction engine, TermsIO, discussed in the next section.

2.2.1 Workspace and Selection Panel

The workspace and the selection panel inherit the visualization features provided by the interaction engine's widget class, but each implements a different type of interaction.

The visualization procedures provided by the interaction engine display a set of terms – the interface semantics – using a set of rules – the interface syntax. These rules, known as **styles** can be modified interactively, using menu commands, or from Prolog predicates, using the TermsIO applications interface (API). These features are used by Magritte to implement a default visualization for terms, similar to the traditional textual representation, but using colours to highlight operators and variables and automatically indenting clauses. Using menu commands the user can select other pre-defined tree-like visualizations syntaxes.

Although visualization is similar in both the workspace and the selection panel, the interaction is completely different in these two widget classes since they have completely different purposes. In the workspace the user can actually edit the terms displayed on the window while in the selection panel the user is limited to the selection of one of the displayed terms. The selection can be made either using the mouse or the numeric keypad where each number selects the term displayed in the corresponding order from the top of the window.

In the workspace the interaction is directed to a special part of the interface called the **focus**. Unlike in other structural editors (cf. [Bar91]) the concept of focus in TermsIO is very clear: at any given moment each TermsIO widget instance has a single complete term that is holding the focus identified by its top functor. The term under focus is represented on screen with a different background.

Any term in the workspace can hold the focus but there is a special kind of terms, represented graphically as small, grey rectangles, and called **slots** which are the natural candidates to insertion points. Slots act as anonymous variables in the traditional textual representation, when they are interpreted as terms, but their role is to serve as place-holders for terms not yet inserted. In fact, slots have also an important role in the implementation of guided input, as described in the next section, since they are the places where terms can be expanded. For this reason Magritte simplifies insertion on slots. When a new term is inserted in a slot the focus is automatically moved to the leftmost slot of the inserted term. There are also two key bindings, the `Tab` and `Shift Tab` keys, to move the focus respectively to the following and the previous slot.

2.2.2 Interaction Modes

Navigation in the workspace is not limited to moving between slots. The user can change the focus to any term presented in the interface, either relatively – using the cursor keys to move to parent, siblings or descendants – or randomly – using the mouse to select the new focus. This form of focusing using the mouse is available when the **select mode** is set. This mode is the default and can be reset using the first button on the modal panel.

Frequently, the size of the layout displayed on the screen is too large and difficult to understand. In this case the user (and the Prolog programs using the API) can collapse the term that is holding the focus thus reducing the complexity of the figure displayed on the screen. This navigation mode is set using the second button on the modal panel, the **collapse mode**. When this mode is active a term clicked with the mouse is collapsed to a button whose label is the term's main functor name and arity. When this button is pressed the term reverts to its original format. The expansion of a term can be restricted to a certain depth by setting an horizon with menu commands. For instance, if the user sets a one level horizon then the term is expanded but its descendants remain collapsed. If the user sets a two level horizon then its descendants are expanded but their descendents remain collapsed.

The third modal button sets the mouse to the **unification mode**. When this mode is active the user can unify the term holding the focus with any term in the interface selected by pointing the mouse. Magritte uses a TermsIO predicate that implements an extension of Robinson's unification algorithm, without occur-check, by interpreting slots as unbound variables.

There are situations where the displayed layout is too large to fit in the workspace window. In this case the user can drag the layout using the mouse in **dragging mode**. This mode is set using the fourth and last modal button. The icon used to represent each mode is also used as the mouse icon in the workspace to emphasize the current mode.

2.2.3 Command Menus

The menu bar is composed of several menus, the first of which, the **File** menu, is similar to those existing in most graphical applications. It aggregates the usual file operations, such as loading and saving Prolog files, creating new empty files and (re)compiling files. All operations on files use the active layer. The **Edit** and **Font** menus include the commands to modify the workspace structure and to select fonts.

The **Focus** menu aggregates commands related to the navigation in the interface. It includes commands for: changing focus, collapsing/expanding terms and window positioning. Some of these commands are also available using the mouse in the appropriate mode. Other commands, in particular those for setting the expansion horizon, are only available through this menu.

The **Tree Style** and **Node Style** commands modify the graphical syntax. As already mentioned, styles are sets of rules used by the interaction engine to compute the layout of each term represented in the interface. The Tree Style menu assigns styles that control the visualization of terms under focus and the Node Style menu assigns styles only to the main functor of the focus.

The **Type** command is used to assign a type to the current focus and presents all the currently declared types. Magritte uses type declarations as a basis for the implementation of guided editing as explained in the next section. If a slot has a declared type then when the slot receives the focus the valid alternatives are displayed in the selection panel.

2.3 Term Editing

Editing large and complex data terms is an usual problem in Logic Programming. Textual representations are sometimes insufficient to perceive terms and several visual syntaxes have been created in areas such as natural language [Win83], in particular in situation semantics [BC91]. However, these are mostly used in written documents and are hard to use in the implementation of applications. By focusing directly on the interaction with terms Magritte simplifies the creation of specialized graphical editors for terms and eases their integration into existing applications.

Magritte supports both guided input and non-guided input. Guided input provides a simple way of inserting frequently used expressions in the edition of programs, large and complex data, GUI definitions, etc. Non-guided input is used both in the edition of terms inserted using the guided input, and as a general mechanism of term insertion.

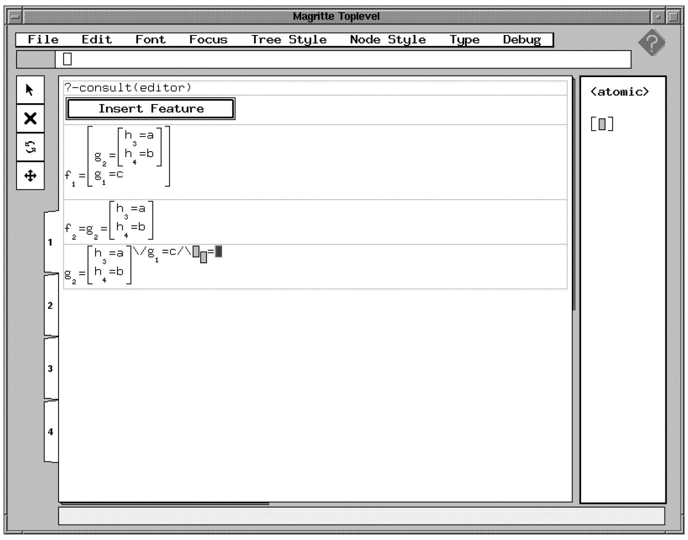

Figure 2.3: A Natural Language editor

To allow guided input, the editor presents the user with the graphical representation of several terms that can be inserted at the current focus. The options are presented in the selection panel (the rectangle shown on the right in Figure 2.3) and depend on the node that is currently holding the focus.

Language development environments that use language directed editors [TR81, BS86] promote the use of guided input. Nevertheless, guided input for program editing has been subject to criticism since the users do not like the lack of freedom it imposes. To cope with this problem, in Magritte guided input is not mandatory, and Magritte supports the "traditional" textual form of entering terms, by binding keys to structure and navigation commands.

2.3.1 Guided Input

Guided input provides a simple and error-free way of inserting frequently used expressions in programs, complex data, parameterization of interfaces, etc. In program edition, guided input can be used to insert "clichés" [Wat94] in programs, that is, sequences or structures of code that are frequently used to make a program.

In a "classic" language environment such as the Cornell Synthesizer (cf. [TR81]) structural editors for a language are derived from the language's context-free grammar. In a LP environment, the basic syntax is too simple to be a problem, but the edition of large and complex data terms is an important topic. This fact suggests that structural edition in a

LP environment should be derived from **types** rather than grammars.

Magritte follows the definition of types of terms using unary logic programs proposed by Yardeni [E.Y91]. The predicate `p/1` defines a type p, and the query goal `?- p(X)` associates X to this type. For instance, the logic program on Figure 2.4 defines the type binary tree and the query `?- btree(X)` declares the argument X as a term with that type. The reader should note that this is a recursive type definition; the second clause defines `b(Left,Right)` as a binary tree where `Left` and `Right` are also terms of the same type because variables appear as goals in the right hand side of the clause.

```
btree(leaf).
btree(b(Left,Right)) :- btree(Left), btree(Right).
```

Figure 2.4: Type definition

Magritte uses a special module of the Prolog clausal database to record type definitions, the type program. A type is assigned to a term (in general a slot) by associating it with a goal to be evaluated in the type program. Structural editing is activated when a typed slot is under focus. Magritte then uses the selection panel to present the graphical representation of the terms that can be inserted at the current focus.

The structural editor works in two stages. First, when a typed slot is focused, the argument of all clauses (in the type program) matching the type goal are presented graphically in the selection panel. Second, when a term is chosen in the selection panel it is inserted in the focused slot and all its variables/slots are typed with the goals of the right-hand side of the matching clause.

Let us take as example the edition of a slot typed as a binary tree, that is with type `btree(X)`, using the type definition in Figure 2.4. Any of the two clauses can be selected since they both match the type goal. The selection panel then shows the terms `leaf` and `b(Left,Right)` displayed using the styles associated for those functors. If the user then chooses the last term in the selection panel, the term `b(Left,Right)` would be inserted in the focused slot: the node's new label would be `b` with two descendants, both slots, associated with the variables `Left` and `Right`. Those slots would be respectively typed with the goals `btree(Left)` and `btree(Right)`, which are the goals in the clause where the respective variables occur.

This form of guided input is important, specially for novice or casual users of an application, but has also drawbacks: experienced users tend to dislike this kind of environment because it lacks the freedom of textual editing. Hence, guided input is not mandatory, i.e. the user can always ignore the expansions of a term proposed by the system and create terms with no restrictions using an interaction style similar to the text interaction.

The use of types, complemented with the use of styles is the basis for the creation of structural editors in Magritte. An example of a structural editor using this approach is the style editor. This editor does not give access to all the power of the TermsIO low-level style definition language (c.f. next chapter) but allows the interactive creation of frequently used styles.

Another structural editor provided by Magritte is the GUI editor. Widget classes from the underlying GUI toolkit can be used as styles for functors as shown in Figure 2.1. With this approach, the GUI of an application can be represented as a term, as proposed out by Abreu [Abr89], and edited using guided input as described previously. Moreover, after editing this term the GUI can be immediately tested within the Magritte's workspace.

2.3.2 Textual Interaction

Text-based interaction has strengths that should be acknowledged and incorporated into any LP interaction framework. The textual representation is simple and compact (especially for small data terms or clauses) and textual editing is an easy way to perform small or local modifications, such as correcting a functor's name or adding an extra argument to a functor. It is also important to have in mind that the LP "culture" evolved using a textual notation and most users are accustomed to it. Consequently, some form of text-based interaction is necessary in a LP environment to keep a balance between a pure graphical editor and a textual editor, as pointed out by Petre [PP92].

For this reason the guided input of Magritte is complemented with a form of free input that supports the "traditional" textual form of writing terms on top of the structure editor. This has been achieved by binding structure editing commands to keys used as delimiters in the term syntax, in particular parentheses and comas.

Let us start by considering terms with a functor name which is textually "followed" by a sequence of terms (arguments) separated by commas and surrounded by parenteses. Character keys are bound to the label insertion command; the open parenthesis key is bound to the creation of a child to which becames the focus; the comma key is associated with the creation of a sibling to which becames the focus; finally, the close parenthesis is bound to moving the focus to the parent.

For instance, if the user types the characters `f ( a , b )` with the focus on a slot then each pressed key executes a slot creation and/or a focus movement that in the end results in the insertion of the term `f(a,b)` in the original slot. Table 2.1 indicates the structure editing commands executed for each key.

key	create node	change focus	append to name
`f`			"f"
`(`	descendant	descendant	
`a`			"a"
`,`	sibling	next sibling	
`b`			"b"
`)`		parent	

Table 2.1: Insertion of a term

Dealing with operators is not so straightforward since the insertion of an operator in general introduces changes in the ancestors rather than in the descendents. Moreover, operators have an associativity and precedence that influences the creation of a term. The simple cases are unary operators that can be reduced to the general case of an application term with just one argument. In the insertion of a binary operator two new nodes have to be created: one for the operator itself and another for the left hand side operator (to where focus is then transferred). For instance, if the user types the characters `a + b` with the focus on a slot then each pressed key executes a slot creation and/or a focus movement that in the end results in the insertion of the term `a+b` in the original slot. Table 2.2 indicates the structure editing commands executed for each key.

key	create node	change focus	append to name
`a`			"a"
`+`	parent descendant	parent descendant	"+"
`b`			"b"

Table 2.2: Insertion of a term

In fact the insertion of operators is more complex than the example in Table 2.2 suggests. If the sequence of keys were **c * a + b** then this sequence of operations could not have been used since the operator `*` has precedence over `+`. The translation of operator keys to sequences of editing commands takes in consideration the precedence and associativity of the inserted operator and its ancestors. The operators in the previous example can be associated with a single key but there are operators with more than one character that force Magritte to implement a delay mechanism.

2.4 Engine Interface

In this section we discuss the mechanisms used to create a spreadsheet-like workspace where programs and data are developed and tested in an integrated environment. Unlike editing, that does not depend on a particular LP language, program development and testing requires an underlying LP execution engine, or emulation through meta-programming.

Features like updating the clausal database or animating the execution of LP programs are different in nature from the abstract editing operation presented so far. Editing commands are general and may be used with any term represented in the interface, but the features that use the underlying engine require specific types of terms. For instance, it does not make sense to evaluate a term under focus unless it is a goal. This is the reason why Magritte has a second mechanism to select workspace terms for engine operations. Engine operations are performed on a term whose main functor is a **trigger** functor and that contains the focus. Each predefined trigger functor is associated with a **term transformation** – a Prolog predicate that operates on the term that triggered its execution, and replaces it by a new term.

Triggers are activated by specific user actions such us pressing the `Enter` key, changing focus, loading a file, or selecting the 'Update' menu entry. When such an event occurs Magritte searches for a trigger from the node holding the focus up to the top node. The first trigger found launches the evaluation of the associated term transformation which modifies the selected term.

Term transformations are Prolog predicates that call the TermsIO API (see chapter 3) to navigate, query and change the current state of the workspace. They use the standard system predicates to interact with the Prolog engine.

The implementation of these predicates is greatly simplified by the separation between semantic and syntactic representation provided by TermsIO. A typical term application uses the API to retrieve the current interface representation, then calls a predicate that transforms the representation (with no intervention of TermsIO), and finally updates the interface representation in the same focus point. Visualization is defined independently by a TermsIO declaration, or by a call to an API predicate to control styles. This approach allows the programmer to change the information displayed in the interface separately from controlling how the information is displayed.

Magritte uses triggers and term applications to interact with the Prolog clausal database (loading/updating predicates or executing goals), to animate LP algorithms and to execute interactively LP programs. Triggers and term applications can also be used to extend the basic engine operations available in Magritte. In the remaining of this section an example of a simple term application is presented followed by a description of the implementation of the most relevant engine operations currently available.

2.4.1 Creating a New Engine Interface

Let us assume that we were interested in implementing arithmetic evaluation on Magritte's workspace, to work as a numeric spreadsheet. We would need a special functor, for instance the functor with name := and arity 2, to trigger the evaluation of its second argument as an arithmetic expression and place the result in the first argument.

```
executable(":=", 2, Layer,State) :-

        command(focus(change(last))),    % move focus to last (second) descendant
        command(tree(get(Expr))),        % bind Expr to the LHS expression
        (check(Expr) ->                  % if Expr is a valid expression ...
                Value is Expr            %   set Value to the value of Expr
        ;                                % otherwise
                Value  = '*Error*'       %   set Value to an error message
        )
        command(focus(change(up))),      % move focus up to the assign functor
        command(focus(change(first))),   % move focus to the first descendant
        command(tree(set(Value))),       % set the current focus to Value
        command(focus(restore(State))).  % restore focus to the previous state

:- view := / 2 using style infix(":=").
```

Figure 2.5: Term transformation

To implement this feature a new clause is added to Maggrite's predicate `executable/4` (c.f. Figure 2.5) declaring the label and arity of the trigger in the first two arguments. References to the worksheet layer and focus state are also passed as parameters. In the body of this clause all the goals `command/1` are calls to the TermsIO package's API. The purpose of each goal is commented in the clause's text.

2.4.2 Clausal Database

The basic system predicates operating on the clausal database are tied to the Magritte workspace by using standard functors, summarized in table 2.3, as triggers. Since these functors are part of the Prolog language we decided not to redefine their graphical layout.

Note that this does not mean that clauses and goals can only be represented in text format: only the standard operators are represented this way in order to not confuse the novice user.

Label	**# of descendents**	**term transformation**
`:-`	2	updates the clause in the clausal database
`?-`	1	evaluates the goal in the clausal database
`:-`	1	evaluates the goal in the clausal database

Table 2.3: Clausal database triggers

The predicate functors and particularly the functors used in the arguments can have any graphic layout that can be coded using a style.

The binary (infix) operator with label ":-" triggers updates in the clausal database. After editing a clause it is enough to press the `Enter` key to update the predicate; the predicate is abolished and all clauses are reasserted to preserve their relative order. Since this trigger is also activated when files are loaded, or focus is changed to a term with a different trigger (goal evaluation trigger, for instance) the user can be sure that the program displayed on the workspace reflects the program in the engine's clausal database. With this approach there is hardly any need for an explicit `consult` of the workspace. The main disadvantage of the current implementation is that clauses are not compiled with the obvious impact on the efficiency of the executions.

It should be noted that the clause whose body is the goal `true` cannot be simplified, i.e. in our notation `length([],0)` is a general term and only `length([],0) :- true` is a clause. Since program clauses, goals, and other terms can coexist in the workspace it cannot be assumed that a term is a clause, as is done by the standard predicate `consult/1`.

The usual Prolog unary functors with labels "`:-`" and "`?-`" trigger a term transformation that evaluates the argument as a goal using the current clausal database and `call/1`. If the call succeeds then the argument is unified with the solution; multiple solutions can be retrieved by repeating the action that triggered the term transformation. Success or failure of the goal is displayed in the message area.

Edition and evaluation of Prolog clauses benefit from the guided input available in Magritte. When focus is on a "top-level" node the selection panel presents the trigger functors in Table 2.3. The guided editor can then be used to enter Prolog goals.

2.4.3 Algorithm Animation

In order to understand the behaviour of a program it is sometimes necessary to visualize its execution. This technique is usually known as program animation or **algorithm animation** and has been proven to be quite useful both for education and research in the design and analysis of algorithms.

Several systems have been developed with the sole purpose of animating algorithms. The first systems appeared during the early 1980's [BS84] and just used monochrome displays. Other important systems developed since then, integrated new possibilities that meanwhile became available on workstations, such as colour graphics, sound and 3D graphics.

These systems operate on a client program, usually called the **algorithm**, and a set of declarations that control the visualization of the client program. The algorithm is edited to include calls to visualization routines and then executed in a special environment where the visualizations are presented to the user.

The animation of a Prolog algorithm in Magritte is quite simple. First the algorithm (i.e. a Prolog program implementing the algorithm) is edited to add the system predicate `show/3`. This predicate receives as arguments a term to be visualized, a style to visualize that term and a delay (in seconds). Then the algorithm's toplevel goal is evaluated and the successive "pictures" of the execution are displayed in a newly created frame.

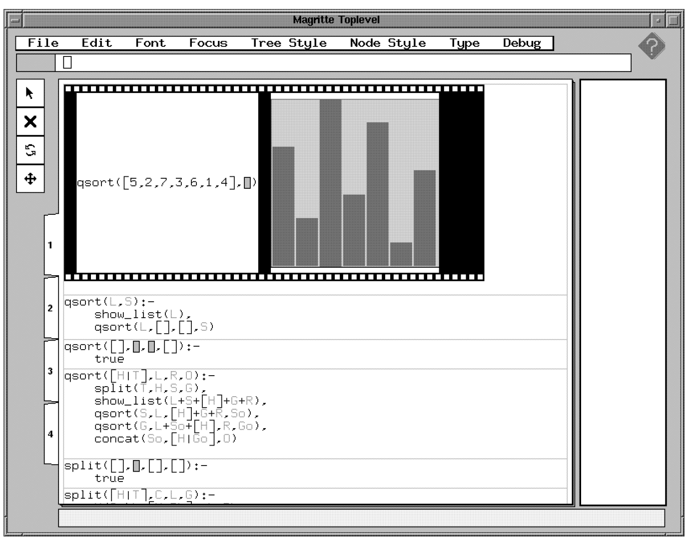

Figure 2.6: Animation of the quick-sort algorithm

To illustrate how algorithm animation is used, Figure 2.6 presents a screen shot with a canonical example in the field: a sorting algorithm. The elements to be sorted are shown as a row of sticks, where the height of each stick is proportional to the value of the corresponding element in the list. The workspace includes both the animation frame and the quick sort program. The program clauses were edited to include two extra parameters, the lists on the left and right of the list being sorted, and the goal `show_list/1` that takes as input a concatenation of all the lists.[4] The `show_list/1` predicate is implemented using the `show/3` Magritte system's predicate and uses a style whose definition is presented as an example in the next chapter.

2.4.4 Graphical Interaction with Logic Programs

The visualization of programs in LP is being used for almost ten years [EB88, DC86] as a pedagogical tool for novice programmers, and for implementing graphical tracers/debuggers for more experienced programmers. More recently, there has been also some work in the visualization of parallel LP executions [KP93, LD87, GCH92] with the purpose of monitoring concurrent LP processes. Although interesting, the latter kind of visualization is outside the scope of Magritte. The remainder of this chapter presents tools to visualize and interact with the sequential executions of logic programs.

The decoupling of interface semantics and syntax simplifies the graphical interaction with logic programs execution in Magritte. To implement these features Magritte combines **meta programming** with triggers and TermsIO to produce an interactive execution of a Prolog goal. The role of triggers and TermsIO does not differ much from what was presented previously. The new aspect is the use of specialized interpreters and meta-programs to perform term transformation in Magritte's workspace.

Let us take as an example the vanilla interpreter (see Figure 2.7) that solves a LP goal using the Prolog strategy. In a conjunction, goals are selected from left to right. Clauses whose head matches the leftmost goal are expanded (using `clause/2`) and evaluated recursively.

```
eval(true)  :- true.
eval((A,B)) :- eval(A), eval(B).
eval(Goal)  :-
        clause(Goal,Body),
        eval(Body).
```

Figure 2.7: Vanilla interpreter

A simple way of displaying the execution of a logic program is to produce a **proof tree** representing successive goal reductions in obtaining the solution of a goal. In the proof tree the root node represents the initial goal and its descendants correspond to goals obtained by goal reduction during the computation. The specialized interpreter `proof/2` presented in Figure 2.8 returns a term defining the proof tree provided it succeeds in solving the goal

[4] The infix operator "+" in this context acts as a list concatenator.

given as first argument.

```
proof(true,[true]).
proof((A,B),P) :-
        proof(A,PA),
        proof(B,PB),
        append(PA,PB,P).
proof(Goal,[node(Goal,Tree)]) :-
        clause(Goal,Body),
        proof(Body,Tree).
```

Figure 2.8: Interpreter for proof trees

The visualization of the terms instantiated to the second argument of `proof/2` is regulated by the styles presented in the screen shot in Figure 2.9 that presents also a resolution tree. The first declaration associates the functor `node/2` with a simple visualization style that packs vertically and centered the visualizations of the arguments. The most important thing to note in this style is the fact that it declares the styles of the arguments as `text` and `tree`, respectively the goal (first argument) and its expansions (second argument). The former is a pre-defined style of TermsIO and the latter is presented in the same figure. The style `tree` is defined in the second declaration and is used to draw straight lines between the top and each of its descendants (other trees). The style is prepared to handle an indefinite number of descendents.

Although in the literature proof trees are often used for graphical representation of executions, they are seldom used by execution visualization tools. They only give a brief account of the solution steps performed during the computation, and present neither the goals that have been tried, nor the clauses that could been used to find an alternative solution.

To cope with those problems, most execution visualization tools use a data structure based on AND/OR trees. The work of Eisenstadt and Brayshaw [EB88] is particularly important in the definition of Augmented And/OR Trees, also known as AORTA trees, that decorate the basic AND/OR trees with other information collected during the execution. Both the basic AND/OR trees and AORTA trees can be implemented in Magritte using the same approach described for proof trees: a meta-program evaluates the goal and collects a tree describing the evaluation; a set of styles assigns a graphical syntax to the generated tree. The approach to program visualization described above can be extended to incorporate other features that we expect to develop in a near future:

Visualization of intermediate steps Merging program animation, as described in the previous section, with the visualization of executions is a simple approach to display

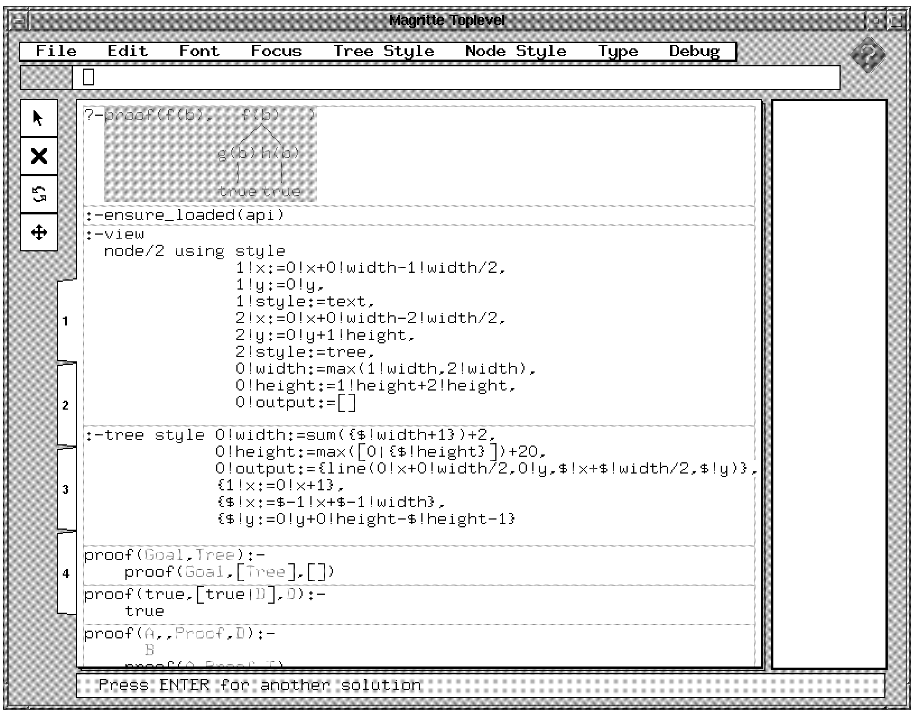

Figure 2.9: Generation of a simple proof tree

graphically the intermediate resolution steps.

Visualization of constraints The visualization of each resolution step can also be extended to include other information used in the resolution. This is particularly important in constraint logic programming where pending constraints influence the resolution.

Interaction with resolution Magritte uses a special meta-program that evaluates a logic program defining a type to implement the selection of terms in guided input. A similar approach can be used in the visualization of executions to select goals and clauses used in the resolution. This feature is similar to the interactive resolution described in sub-section 1.2.1

Chapter 3

TermsIO: an Interaction Engine

This chapter describes **TermsIO**, an implementation of the framework designed to enhance the Logic Programming (LP) interaction model by providing graphical interaction with terms. This package serves as an **interaction engine** for tools and applications in a LP environment, enabling the visualization and edition of the terms subject to user interaction.

The emphasis of TermsIO is on interaction with terms. It thus simplifies the creation of graphical LP tools that operate on terms – such as editors, debuggers and execution tracers – and also of LP applications with non-standard interaction needs. This is especially the case for applications processing large and complex data, such as Natural Language Processing. The use of terms as the basis for interaction makes it possible to use graphical interaction with meta-programs and thus implement application tools such as debuggers and algorithm animators. Other examples of visual meta-programming include meta-interpreters that allow graphical interaction with the resolution of logic programs, for instance, to control the selection of goals and clauses or to visualize the unification (especially in Constraint LP). Furthermore, the term-based approach allows the integration of data used by different tools since it provides an unified view of all the terms involved in the interaction. This enables the implementation of spreadsheet-like environments, such as **Magritte**, previously described.

The chapter first introduces the design philosophy of TermsIO. The following section describes the system's internal state, in particular the attribute tree that models the displayed terms. Section 3.3 presents the languages that define the graphical syntax of attribute trees. The algorithms for attribute evaluation using these languages are presented in Section 3.4 and Section 3.5 discusses the main implementation issues of the interaction engine. The final section presents the commands that form the systems applications interface (API).

3.1 Design Philosophy

The TermsIO library provides a layer of abstraction between LP applications and the windowing system. Applications communicate with TermsIO via the **client-server** model, where TermsIO assumes the role of server. This abstract model of communication requires a clear specification of the protocol used by client and server. It should be noted that the client-server model does not require the execution as separate processes of TermsIO and its clients, which in fact does not happen.

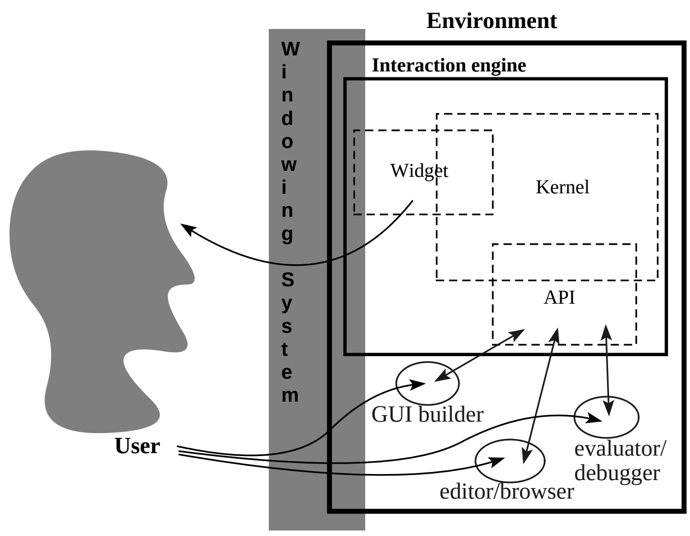

Figure 3.1: Integration of TermsIO in a LP environment

The system was designed to accommodate the needs of two types of clients: LP environments that aggregate several cooperating tools and applications, as well as tools and applications that have to present LP data for user interaction. From the perspective of both types of clients TermsIO is divided into three basic components, as shown in Figure 3.1.

Applications Interface (API) used by applications and tools to modify both data and its visualization. These commands are used to implement editing tools, as specialized data and program editors, and other tools requiring special visualizations such as algorithm animators and debuggers.

Visualization Class of the widgets where the current interaction state is displayed. This class – implemented using the Ytoolkit (cf. chapter 4) – was designed to be a meta class for interaction widgets of an LP environment. Specialized classes are then responsible for binding user events (such as keyboard and mouse events) to application predicates.

For instance, in Magritte, the specializations are the workspace layer class and the selection panel layer class.[1]

Kernel responsible for maintaining and manipulating the interaction data. These data consists of a **display tree** – an attribute tree that serves as an internal representation of the terms under interaction – and of **styles** – semantic rules that control the (re)evaluation of the attribute tree. The kernel implements the procedures that modify and maintain the consistency of these data structures.

The whole interaction engine is a **state machine**: the current state, kept in the kernel, is displayed graphically in a widget of the visualization class and is changed using API commands. The structure of this abstract machine is illustrated in Figure 3.2. The conceptual modules are represented by thick boxes with the control flow represented by solid arrows. Dotted arrows represent data flow to and from the two databases that store interaction data, represented by thin boxes.

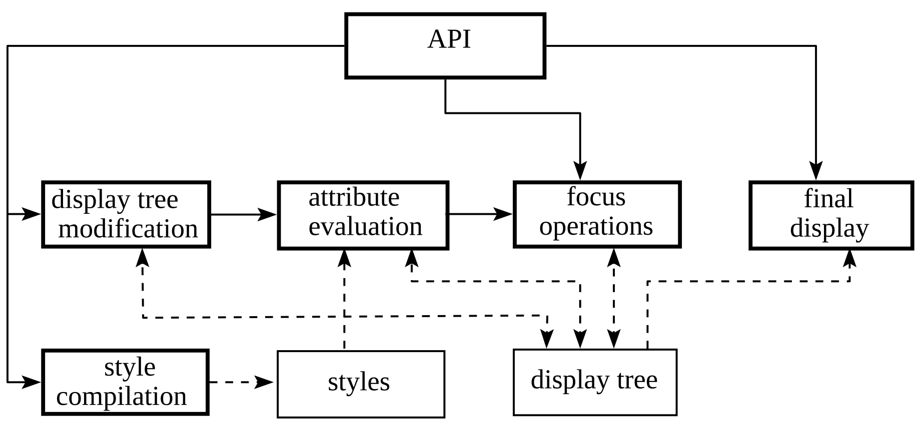

Figure 3.2: Abstract machine of TermsIO

These modules and databases are organized into three conceptual layers: the top layer is composed solely of the command interpretation module that implements API commands and controls all other modules; the middle layer corresponds to a **pipeline** of operations triggered by a modification on the displayed tree; the bottom layer includes the modules that manage interaction data.

With the exception of compilation commands for styles, all TermsIO commands trigger a sequence of operations in the pipeline. These commands typically start with a modification in the display tree that forces the reevaluation of the attributes, modifies the focus properties

[1]A widget from a sub-class of TermsIO is meant to be shared by different tools and implement a spreadsheet-like environment, rather than being the interface of a single tool.

and terminates with a display update. In practice, redisplaying the screen is a costly operation, and it is explicitly controlled by the client application. Note that focus can be modified without changing the display tree while refocusing does not require attribute reevaluation.

The TermsIO abstract machine has a procedural approach to the manipulation of terms. The alternative is to use a declarative description of the screen's layout. This alternative would be highly desirable in a logic programming environment since a graphical interface with a declarative definition would be easier to develop and maintain, but would be also less efficient and less controllable. With the procedural approach the programmer has finer control of the presentation of terms on screen and the interaction is faster and smoother. The kernel of TermsIO can be seen as the non-declarative layer that bridges Prolog applications to the widgets providing graphic interaction with terms. These widgets take advantage both from the declarative definitions of visualization and manipulation provided by TermsIO, as well as from its efficiency.

3.2 Internal State

This subsection introduces the **display tree**, an internal TermsIO data type that defines the set of terms under interaction, and explains how this data structure relates with terms in the Prolog engine.

Since terms in logic programming are basically trees, this structure is the first natural candidate to represent them. Nevertheless, the use of logic variables may result in more complex structures. These structures are similar to directed acyclic graphs (DAGs)[2] which could, in principle, be selected to represent terms graphically.

There is a good number of algorithms to layout general undirected graphs [KK89, FR91] and hierarchies [STT81]. They have been used in several graph editing systems, such as EDGE [pT90] and daVinci [FW94]; a comprehensive list of graph drawing algorithms was compiled by Cohem and others [CdBTT92]. Unfortunately, these algorithms are very complex, leave little control over the visualization process to the user, and are not efficient enough to update the graphical output in order to provide a good interaction.

It should also be noted that terms are not exactly DAGs, although most Prolog engines represent them internally as such. In fact the parent-child edges are directed but the edges introduced by logic variables are undirected (taken here as not circular). This fact suggests that these two components of the data structure should be separated.

[2]Cyclic graphs are ruled out by the occur check in the unification procedure.

3.2.1 Display Tree

TermsIO interaction data is modeled by two distinct layers: a **structural** layer where all data is related by ordered parent-child arcs; and a **relational** layer where data is related by unordered links. In Figure 3.3 nodes are represented by circles linked by thick arrows representing the structural layer, and by thin lines representing the relational layer. The display tree in Figure 3.3 represents the term `f(a,g(X,Y),X)`, where the bounded variable X introduces a relational undirected edge between the two nodes labeled with `X` while the other directed edges connect nodes to the nodes of their arguments.

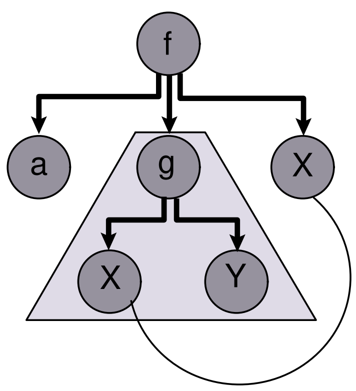

Figure 3.3: TermsIO data model

The API commands always operate on part of the display tree designated as the **focus**. Unlike other structural editors (cf. Barfield [Bar91]) the concept of focus in TermsIO is very simple since the focus is a complete sub-tree. Multiple foci are not allowed and selecting a node implies the selection of all its descendants. Hence, the focus can be used either to identify a sub-tree (the node and all its descendants) or a single node (the sub-tree's top node). In Figure 3.3 the gray trapezium is the current focus thus selecting the sub-tree associated with `g(X,Y)` and the top node labeled `g`.

In summary, a display tree has an underlying tree structure. Each tree node has a potential for several **ordered** children and a leaf can be connected with **links** to other leaves. Each node is labeled with a string of characters.

3.2.2 Trees and Terms

The sub-tree holding the focus can be easily mapped into a Prolog term (or vice versa) using the relationship summarized in table 3.1. In fact, there is a correspondence between node labels and functor names, and between the number of descendants of a node and the arity of a functor. Consequently, the combination of label and number of descendants can be identified with a functor.

Display Tree	**Terms**
Label	Name
Number of children	Arity
Slot	Anonymous Variable
Link	Bound Variable

Table 3.1: Relationship between linked trees and terms

Tree leaves are mapped into Prolog atoms, numbers and variables depending on the morphology of their labels. Inner nodes are mapped into application terms where the functor is created using the label and the number of descendants. Each argument is recursively identified with the terms corresponding to each descendant, associating arguments and descendants in the same order.

As they are the obvious candidates to be insertion points, leaves with empty label have a special role in editing operations and are designated as **slots**. Nevertheless, all nodes, and not just slots or leaves, can be modified using the API commands introduced in the last section of this chapter.

Slots are nodes that have not yet been filled with data. Links between slots result from unification. They are mapped to bound logic variables when they connect slots within the scope of the subtree that is unified to a term. Otherwise (when they connect nodes of two subtrees that will be unified with two different terms) they should be seen more like references in a conventional spreadsheet.

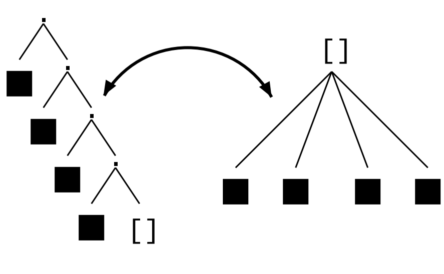

Figure 3.4: Mapping between lists and trees

Lists are a special case in the mapping between terms and display trees. TermsIO considers two different cases of lists: closed lists and open lists. In both cases the lists are bijectively mapped into a one-level tree where all list elements are mapped into siblings, maintaining their relative order. Closed list are lists whose "last" term is nil (`[]`) and in open lists the "last" term may have any other value. Trees representing closed and open lists are distinguished by their labels: the string `"[]"` in the first case and the string `"[|]"` in the second. The special treatment of list has two advantages. First, from the point of view of term visualization, this mapping simplifies the visualization of lists that are usually represented through a special notation in textual format. Second, from the point of view of graphical editing, this mapping simplifies the implementation of sequences with an indefinite number of elements that share the same visualization style and inherit common properties.

3.3 Styles

Styles are the cornerstone of TermsIO since they control the evaluation and incremental reevaluation of the attributes display tree. A **style** is a set of rules that determines how to visualize a class of "functors" – i.e. a combination of label and number of descendents – and of its descendents.

To understand how styles can be combined to create a complete visualization let us start by analyzing the textual visualization of LP terms. Let us consider only ground terms since logic variables do not differ significantly from atomic values in their textual representation. Ground terms can be recursively defined as a functor with name N and arity n followed by n other terms. A functor with arity 0 is called atomic.

Atomic terms are represented textually just by their name, a string of characters. Application terms are represented by the functor's name (also a string of characters) followed by the representation of its arguments. The arguments representation is surrounded by parentheses and separated by commas.

The visual representation of terms provided by TermsIO is similar to the recursive definition of the textual representation: 1) a functor can be represented using any 2D graphic primitives and not just text; 2) functor and arguments can be placed using 2-dimensional coordinates and not just concatenated as in the textual representation.

3.3.1 Attributes

Associated to each tree node there is a **set of attributes** that stores the values used to display terms. Table 3.2 lists all attributes with a special meaning in TermsIO. Styles may define new attributes for their own purpose and benefit from the TermsIO attribute evaluation mechanism to maintain their values updated.

name		meaning
`x`	`y`	position
`width`	`height`	size
`layout`		layout
`label`	`n_desc`	functor
`value`		term representing the sub-tree
`style`		style that governs the node

Table 3.2: Predefined node attributes

The node's standard attribute set includes position **(x,y)**, size **(width,height)**– the node's **bounding box** – and the graphical primitives that display the node – the node's **layout** – . The node's label and number of descendents are also recorded as attributes. The bounding boxes computed by styles are used by TermsIO to display the focus. The attributes in Table 3.2 are needed by TermsIO and must be defined for every node. Hence every style must include rules to define their value.

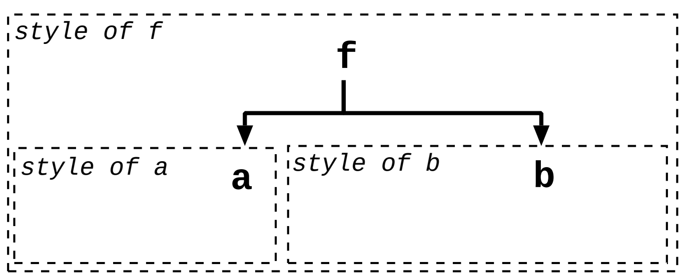

Figure 3.5: Visualization styles and bounding boxes

Since applications can define different styles to compute attributes of the same display tree there are some basic rules that each individual style must follow in order not to interfere with the rest of the visualization:

- the bounding boxes of descendents must be contained in the bounding box of the parent.
- the bounding boxes of siblings must not overlap.
- the layout of a tree must be contained within its bounding box.

Styles are dynamically assigned to nodes during interaction. Each time that the node's label or the number of descendents is changed, its style is checked and, if changed, the node's layout is changed accordingly. To assign a style S to a node, TermsIO uses the following criteria:

1. If the attribute `style` is defined then its value S is the style of the term.
2. Otherwise, if there is a rule associating the functor (defined by attributes `label` and `n_desc`) to a style S, then S is the style of the term.
3. Finally, the node's ancestors are looked up for a default style P that is used as the style S. The top node of the display tree is created with a default style that guarantees that a node has always a default style.

It should be noted that visualizations controlled by styles are the graphical representations of the structural layer (i.e. nodes and parent-child links) and that TermsIO does not draw a graphical representation of the relational structure. Several experiments were made in order to represent graphically the links between leaves using polygonal lines. In these experiments the layout of the relational layer was computed from the node's bounding boxes but without using styles. Unfortunately, in most cases the visualizations were too difficult to understand due to the large number of crossing lines. Since the results were not encouraging this visualization of the relational layer was switched off from the interaction engine. Nevertheless, there are situations where the visualization of links may help the user in finding dependencies. For instance, the user may click in a variable to find all the variables it is connected to. This kind of visualization can be easily implemented on Magritte.

3.3.2 Style Definition

The definition of styles in TermsIO can be made using two layers of languages:

Low level definitions, where styles specify the values of attributes with a full set of arithmetic and relational operators, a conditional operator and aggregate operators (max, min and sum). Sets can be defined extensionally or intensionally. The latter enables the creation of styles for terms with an undefined number of descendents.

High level definitions, where styles are defined by a set of terms that is rewritten into low level definitions at loading time. The system provides a basic set of rewrite rules, but the application programmer can define his/her own rules.

Any of these two methods can be used when submitting a style definition to TermsIO since these are processed as summarized in Figure 3.6. The Prolog terms defining styles are

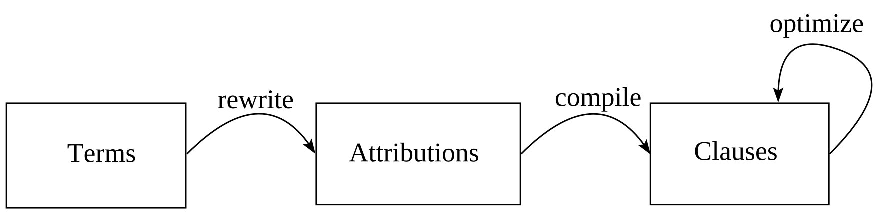

Figure 3.6: Style processing

rewritten to sets of attributions, then compiled to Prolog clauses which are asserted in the clausal database, after being optimized. This process is described in detail in section 3.5.

Both style definition methods are important to TermsIO. Using attribution sets is convenient since it is a simple and powerful way of defining attribute values, the main purpose of styles. The cost is large and complex definitions. Rewriting terms into attributions results in more compact definitions that are easier to understand.

3.3.2.1 Style attributions

Style definition in the TermsIO low-level language are essentially sequences of attributions, where the left-hand side designates an attribute to be set and the right-hand side the expression that computes its value. The Prolog predicate `style/1` in Figure 3.7 recognizes valid styles defined in the low-level style definition language.

In low-level definitions attributes are terms recognized by the predicate `attribute/1` in Figure 3.7: the node is referenced by a relative position (a non-negative integer), the name is a Prolog atom, and both are joined by an exclamation mark ("`!`") used as an infix operator. The node to which the definition is associated is referenced as the node zero, and a positive integer references its descendents in the standard order. For instance, if a style definition S is given to node n then `0!x` in S refers to the horizontal coordinate of node n and `1!width` to the width of its first descendent.

Attribute expressions, recognized by predicate `expression/1` in Figure 3.7, can be either constants, references to the value of other attributes (of the same node or of one of its children), functions of attribute expressions, or set expressions. Set expressions are intensional definitions of lists of values that allow the definition of styles for nodes with an undefined number of children. The evaluation of set expressions returns a list of elements by replacing the constant `$` with the reference of each descendent of the base node. For instance, the expression `max({$!height})` computes the maximum height of all children of a node, where `{$!height}` is the list of all heights.

```
style((A,B))                :- style(A), style(B).
style( Attr := Exp )        :- attribute(Attr,basic), expression(Exp,basic).
style({Attr := Exp})        :- attribute(Attr,set),   expression(Exp,set).


expression(C,_)             :- constant(C).
expression(A,Type)          :- attribute(A,Type).

expression((A->B;C),Type)   :- expression(A,Type),
                               expression(B,Type),
                               expression(C,Type).

expression(A+B,Type)        :- expression(A,Type), expression(B,Type).
expression(A-B,Type)        :- expression(A,Type), expression(B,Type).
expression(A*B,Type)        :- expression(A,Type), expression(B,Type).
expression(A/B,Type)        :- expression(A,Type), expression(B,Type).
expression(A>>B,Type)       :- expression(A,Type), expression(B,Type).
expression(A<<B,Type)       :- expression(A,Type), expression(B,Type).

expression(max(L),Type)     :- sequence(L,Type).
expression(min(L),Type)     :- sequence(L,Type).
expression(sum(L),Type)     :- sequence(L,Type).

sequence([],_).
sequence([E|T],Type)        :- expression(E,Type), sequence(T,Type).
sequence({E},_)             :- expression(E,set).


constant(C)                 :- atomic(C).


attribute(Index!Name,Type) :- index(Index,Type), name(Name).


name(Name)                  :- atom(Name).


index(N,_)                  :- integer(N), N>=0.
index($,set).
```

Figure 3.7: Style language

```
:- bar_graph style
        0!width := 200,
        0!height:= 200,
        0!bars  := 0!n_desc,
        0!colw  := (0!width-3)/0!bars,
        0!max   := max([1|{$!value}]),
        0!output:= [
                        rectangle(0!x,0!y,0!width,0!height),
                        {frectangle($!x,$!y,0!colw-2,$!value/0!max*0!height)},
                        fail
                    ],

        {$!x    := 0!x+($-1)*0!colw+2},
        {$!y    := 0!y+0!height-$!value/0!max*0!height}.
```

Figure 3.8: Bar graph style

The style that displays the bar graph in the algorithm animation example in the previous chapter illustrates the use of style language constructs. Its code is shown in Figure 3.8. Please recall that TermsIO uses a special representation for lists, where all elements of a list are children of the same node. The style is assigned to the list node and assumes that all descendents are integer values; it computes the layout of the bar graph using the values of its children and the predefined attribute `value`. The style computes the layout of the whole bar graph and does not require the descendents (the list elements) to layout themselves. Since that is the normal layout strategy of TermsIO the attribute `output` of this style must end with `fail`. The rule is that when the goal that generates a node's layout fails, TermsIO does not continue to layout the descendants.

3.3.2.2 Rewrite rules

Using the sequences of attributions the interface programmer is able to control the position and appearance of every single element that represents a LP term. Unfortunately this freedom and power has the following drawbacks:

Size: The expressions that compute attributes frequently refer many other attributes thus resulting in large style definitions. For instance, to compute the vertical position of a node, it may be necessary to write an expression that relates the position of sibling nodes, the node's height and the parent's node height. This kind of definition may not be simple to understand for a person that is starting to use TermsIO, especially when the definitions deal with an arbitrary number of descendents and have intensional set definitions.

Reusability: The expressions that define the attributes values tend to follow general patterns. For instance, all styles stacking vertically and centering their descendents, have basically the same position and size attributes. An even more general definition for position and size attributes can be obtained considering alignment as a parameter (e.g.: center, left, right). Unfortunately, defining styles with sequences of attributions prevents us from reusing the definitions.

To solve these problems the interaction engine uses **rewrite rules**. TermsIO accepts both simple rules, specifying a pattern in the original term and its replacement, and conditional rules, where the pattern is replaced only if the condition succeeds. The rules are represented as terms with the following syntax, where `LHS` and `RHS` are terms and `GOAL` a Prolog Goal.

```
LHS => RHS.
LHS => RHS <- GOAL.
```

The semantics of these rules is very simple. If the term in the left hand side of a rewrite rules matches a sub-term in the style definition, then it is replaced by the term in the right hand side. It should be noted that rule matching is unification; thus the variables occurring both in the left and the right hand side are unified before replacement.

The `GOAL` in conditional rules can be used as a **guard**, testing the applicability of the rule. It can be used to evaluate other predicates that implement special purpose transformations, for instance, to force arithmetic evaluation. It can also be used to trigger side effects at style compilation time, like loading external resources. A typical application is loading bitmaps files for styles that use them as part of their layouts.

Rewrite rules can be used to replace long expressions that are commonly repeated with a few parameters changed. Figure 3.9 presents a simple set of rewrite rules that expand the definitions `arrow(X1,Y1,X2,Y2)` and `double_arrow(X1,Y1,X2,Y2)` into long sequences of lines and polygons and that can be used in several styles, providing a simple way of introducing a new graphic "primitive". As mentioned before, TermsIO provides a basic set of rewrite rules, including those in Figure 3.9, and an application can also define new rules.

3.4 Attribute Evaluation

The graphical layout of a display tree is determined by the attributes values of its nodes. These values are computed by two attribute evaluators. The **global evaluator** is responsible for the complete evaluation of all attributes of a given subtree while the **incremental**

```
arrow(X1,Y1,X2,Y2) =>
        [
            line(X1,Y1,X2,Y2),
            tip(X2,Y2,X2-X1,Y2-Y1)
        ].

double_arrow(X1,Y1,X2,Y2) =>
       [
            line(X1,Y1,X2,Y2),
            tip(X1,Y1,X1-X2,Y1-Y2),
            tip(X2,Y2,X2-X1,Y2-Y1)
       ].

tip(X,Y,DX,DY) => fpoly([ (X,Y),
       (X - integer(5*(DX+DY)/norm(DX,DY)), Y-integer(5*(DY-DX)/norm(DX,DY))),
       (X - integer(5*(DX-DY)/norm(DX,DY)), Y-integer(5*(DY+DX)/norm(DX,DY))),
       (X,Y) ]).

norm(X,Y) => sqrt(X*X+Y*Y).
```

Figure 3.9: Rewrite rules for arrows

evaluator is responsible for updating attributes whose values change as the result of user interaction.

The two evaluators work together to maintain the consistency of the attribute tree. For instance, when a subtree is inserted in the focus point, the attribute values of the new subtree are computed using the global evaluator. The incremental evaluator is then used on the rest of the tree to update the values that may have changed as result of the global evaluation.

The attribute values are computed using styles, the sets of attributions introduced in section 3.3. Styles and the way they are related with nodes may change during interaction. Nevertheless, the set of styles and the association of nodes to styles is fixed for the evaluation process.

The order in which attributes are computed is of extreme importance since attributes may depend on any other attribute. The expression that computes an attribute (the right hand side of a style attribution) usually refers to other attributes values, which in turn may depend on other attribute values, creating complex dependencies. Thus, finding an efficient attribute evaluation order is the central problem of attribute (re)evaluation.

Given an attribute tree, we can build a direct graph whose nodes correspond to each of the attributes in the tree and whose edges are the dependencies between them. This graph is usually called **dependency graph** [Alb91b, AsU86]. It is trivial to notice that an attribute can only be (re)computed if all attributes having an edge pointing to it have already been (re)computed.

A straightforward approach to evaluate an attribute tree is to first build its dependency graph, and then perform a topological sort of the graph to determine an evaluation order. This procedure is used by the so called **dynamic** evaluators, but can be rather inefficient since the dependency graph can be implicitly used without the burden of maintaining a data structure. Moreover, in the case of incremental attribute evaluation, most parts of the dependency graph are not necessary if the attributes they refer do not have to be recomputed.

Some incremental attribute evaluation algorithms [RTD83, Hud91] use this approach with a twist: the topological ordering is derived dynamically using another **global model graph** to direct its actions. Unlike the case with dependency graphs, the size and structure of global model graphs changes dynamically as evaluation proceeds. In spite of the gain in efficiency, maintaining this data structure requires a lot of space since the evaluator has to record control information associated with each node and edge in the dependency graph.

An alternative approach is to determine in advance the attribute evaluation order based on the properties of attribute evaluation rules [Alb91c]. The evaluators that use this approach are called **static** evaluators. This approach is used by both attribute evaluators in TermsIO.

Most static attribute evaluators, and those of TermsIO in particular, use **multi-pass algorithms**. As the name suggests, the evaluation procedure performs several passes through the tree and in each node computes only some of the attributes. A general description of multi-pass algorithms is made in subsection 3.4.1.

What attributes are to evaluate in each pass, and the order in which to evaluate them, must be determined in advance using static analysis over the attribute evaluation rules. In most cases where attribute evaluation is used, these rules are embedded in a context-free grammar, forming an **Attribute Grammar** (AG) [Alb91c]. Applying the usual static analysis of AG [Alb91c, MW91] to TermsIO is not straightforward since that kind of static analysis is based on concepts difficult to map into the TermsIO framework, as shown in subsection 3.4.2. The adjustments to the AG static analysis in order to use TermsIO styles are discussed in subsection 3.4.3. The actual algorithms used by TermsIO evaluators, namely the global and the incremental evaluators, are presented in subsections 3.4.5 and 3.4.7 respectively.

3.4.1 Multi-pass Algorithms

We start by introducing a class of static evaluation methods, the multi-pass classs, that is the basis for both incremental and global algorithms implemented by TermsIO. In this class of methods the (re)evaluation of an attribute tree is performed in n **passes**. Each pass traverses the attribute tree from top-to-down. In the first pass the attributes that do not depend on others are initialized. In each subsequent pass the evaluator computes only those attributes that depend exclusively on attributes whose values have already been computed or updated. The set $\mathcal{A}$ of attributes to be evaluated must be partitioned into disjoint sets $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \cdots \cup \mathcal{A}_n$ and each subset associated with a pass. All attribute names in $\mathcal{A}_i$ depend only on attributes in $\mathcal{A}_j$ with $0 \leq j < i \leq n$.

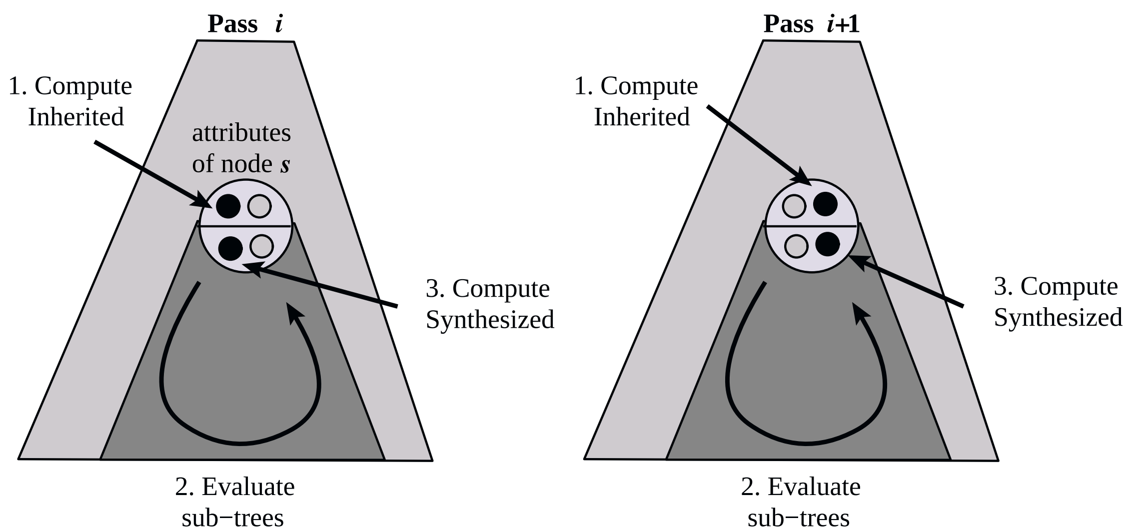

Figure 3.10: Multi-pass evaluation strategy

Figure 3.10 shows the different stages of an attribute evaluation in two consecutive passes. In this figure the tree node s is represented by a large circle, with four attributes represented by smaller circles, and the sub-tree rooted in this node is represented by a triangle with a darker background. Given a sub-tree rooted in node s, the evaluation of its attributes in $\mathcal{A}_i$ (represented with black foreground for node s) during pass i is divided in three parts:

1. The attributes of s in $\mathcal{A}_i$ depending on the attributes of its parent or sibling nodes are evaluated; these are usually called **inherited attributes**.

2. The evaluation of pass i is performed recursively in all sub-trees of s.

3. The attributes of s in $\mathcal{A}_i$ depending on the attributes of descendent nodes are computed; these attributes are usually known as **synthesized attributes**.

Hence, the attributes in $\mathcal{A}_i$ are further divided in two disjoint sets, the synthesized attributes $\mathcal{A}_i^s$ and the inherited $\mathcal{A}_i^i$, such that $\mathcal{A}_i = \mathcal{A}_i^i \cup \mathcal{A}_i^s$. Moreover, in any given pass i, both the attributes in $\mathcal{A}_i^i$ and in $\mathcal{A}_i^s$ must be evaluated in an order that guarantees that they are evaluated after the attributes in the set they depend upon.

A multi-pass algorithm uses an attribute evaluation order determined beforehand. In other words, the partition of attributes in passes, their division in inherited and synthesized, and the evaluation order within the same pass, is determined from the rules that compute the attributes before the evaluation of a given attribute tree.

3.4.2 AG Static Analysis

In the AG framework, the attribute evaluation order is determined by statically analyzing the attribute evaluation rules. A static analysis for an AG evaluator is described by Albas [Alb91c], and is the basis for the static analysis in TermsIO. Although similar, the two frameworks have important differences that prevent the use of exactly the same procedure. In particular, the concept of grammar symbol does not exist in the TermsIO.

In the AG framework, a context-free grammar assigns a tree structure to a sentence and each node represents a grammar symbol recognized as a substring of the sentence. One can view nodes as records holding information in fields whose names correspond to attribute names. Attribute grammars use different **types** of "records" depending on the symbol of each node. Nodes of a given symbol hold instances of the symbol's attributes. In fact, all attribute instances of the same symbol are computed exactly in the same way and have a similar set of dependencies. This is the reason why the static analysis of AG is based on the concept of symbol's attribute.

Although TermsIO display trees are similar to annotated parse trees they do not result from parsing a string of terminal symbols using a context-free grammar. Trees in TermsIO are edited without an underlying context-free grammar. In consequence, nodes do not have a type inherent to the language specification, that would determine the evaluation of its attributes. Nevertheless, the concept of "type of node" can be defined in TermsIO based on the styles that define the nodes attribute values.

The attribute names and evaluation rules associated with a node are determined by two styles: the **inherited style** – assigned on node creation and defined by its parent node – and the **synthesized style** – assigned on node expansion to introduce the descendents nodes. Figure 3.11 shows a tree with four nodes (represented by the larger circles) and the attributes computed by two styles that determine the attributes of the central node (with a darker background). It should be noted that a style computes attributes of two nodes and a node needs two styles to compute all its attributes. In summary, a node type in TermsIO is the combination of inherited and synthesized styles plus the position among its siblings.

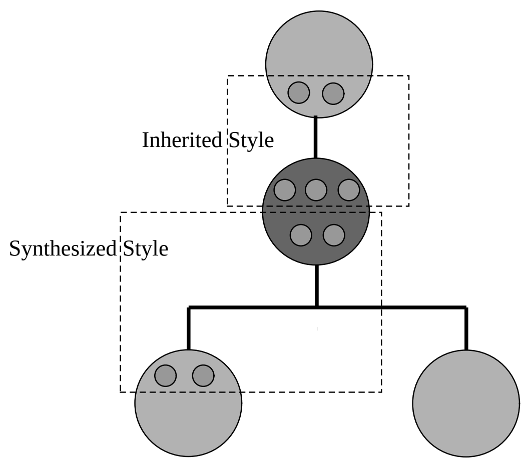

Figure 3.11: Inherited and synthesized styles

The node's position among its siblings is relevant since an (inherited) style S may define the inherited attributes of each children with different dependencies.

The concept of type of node in TermsIO can be used to define a mapping into the AG static analysis, based on the relationships between symbols attributes. This approach requires the precise identification of attributes that is presented in the remainder of this section.

Given the set of styles $\mathcal{S} = \{S_1, \cdots, S_n\}$, and denoting by $\texttt{\#}(S)$ the number of children of a style S, we can define the set of types $\mathcal{T}(\mathcal{S}) = \{(S_i, p, S_j) : S_i, S_j \in \mathcal{S} \wedge 0 \leq p < \texttt{\#}(S_i)\}$. The **attribute names** of the "symbol" $T = (S_i, p, S_j)$, denoted as $\mathcal{N}_T$, are the attribute names defined either by S_i or by S_j. In other words, the name `n` is an attribute name of T iff it occurs in the left-hand side of an attribution in S_i or S_j. The set of attributes of the style set $\mathcal{S}$ is defined as $\mathcal{A}(\mathcal{S}) = \{\texttt{n of } T : T \in T(\mathcal{S}) \wedge \texttt{n} \in \mathcal{N}_T\}$.

A relationship between attributes of $\mathcal{A}$ is defined using the dependencies in style attributions. The attribute `a of` U **precedes** the attribute `b of` T, and is denoted as `b of` $T \leftarrow$ `a of` U with $T = (I^T, k, S^T)$ and $U = (I^U, k', S^U)$, iff there is a common style S in both T and U with an attribution that creates a dependency. In detail, there is a style K such that $\exists(K \in \mathcal{S}) : (K = I^T \vee K = S^T) \wedge (K = I^U \vee K = S^U)$ and containing an attribution of the form n`!a := f(`m`!b)`. The set of all precedences defines a relation in $\mathcal{A}(\mathcal{S})$ that is denoted by $\mathcal{P}(\mathcal{S})$.

The set of attributes $\mathcal{A}_\mathcal{S}$ and the precedence relation $\mathcal{P}_\mathcal{S}$ define a **precedence graph** for the set of styles $\mathcal{S}$, equivalent to the precedence graph defined for symbol attributes in the AG framework. A straightforward approach for finding an attribute evaluation order for the attributes in $\mathcal{S}$ would be using the static analysis of AG on this precedence graph.

3.4.3 Simplifying Precedence Graphs

The precedence graph is, in general, too large to be of any practical use. In fact, the number of types of nodes in $\mathcal{T}(\mathcal{S})$ is of the order of n^2 where n is the cardinal of the set of styles. An environment using this approach would therefore not scale up.

In order to be able to use the standard tools of the AG static analysis we developed a **simplified** version of the precedence graph. This graph is based on the observation that only some sets of styles compute consistently all attribute values. Moreover, these sets of styles introduce constraints excluding certain combinations of inherited and synthesized styles. For instance, a style that uses but does not define the value of an attribute cannot be combined with a style that also does not define that attribute.

The concept of well-defined specification also exists in the AG framework. A well-defined AG [Alb91c] assigns a single value to every attribute associated with a tree node. Similarly, a set of styles is well-defined if it assigns a single value to every name of the display tree. A set of styles $\mathcal{S} = \{S_1, \cdots, S_n\}$ is **well-defined** iff:

1. $\forall$ ($\texttt{n of } T$) $\in \mathcal{A}(\mathcal{S})$, `n` has a *single* definition in T
2. $\forall$ ($\texttt{m of } U \leftarrow \texttt{n of } T$) $\in \mathcal{P}(\mathcal{S})$, `n` has a definition in T

A well-defined style set has several properties that support a simplified version of the precedence graph, and enable us to base the static analysis on attribute names and precedence between them, rather than on attributes of node types. These properties simplify the precedence relationship by avoiding the need to consider node types in TermsIO static analysis.

To illustrate these properties let us take as example the set with two styles in Figure 3.12. The style `label`, associated with leaves (atomic terms), simply displays the node's label within the bounding box. The style `over` places the bounding box of the first descendent over the bounding box of the second, and draws a horizontal line between them.

As in the AG framework, attributes can be divided into two sets according to their relationship to the node that defines them. Synthesized attributes are computed using attributions with a left-hand-side of the form `0!`*name* in the synthesized style; inherited attributes are computed by the attributions of the form n`!`*name*, where $n > 0$ is the position of the node among its siblings, belonging to the inherited style.

An important property of well-defined style sets is that all attributes with the same name are either inherited or synthesized. In a well-defined set of styles $\mathcal{S} = \{S_1, \cdots, S_n\}$ an

```
:- label style
        0!width := str_width(0!label),
        0!height:= str_height(0!label),
        0!output:= writestring(0!x,0!y+0!height,0!label).

:- over style
        1!x := 0!x + 1,
        2!x := 0!x + 1,
        1!y := 0!y,
        2!y := 1!y + 1!height + 2,
        0!width := max(1!width,2!width) + 2,
        0!height := 1!height + 2!height + 2,
        0!output := line(0!x,2!y-1,0!x+0!height,2!y-1)
```

Figure 3.12: A set of two styles

attribute with name `n` in two different styles S_i and S_j is either inherited or synthesized in both. Otherwise, if `n` where inherited in S_i and synthesized in S_j then the node with type $T = (S_i, k, S_j)$ would have two different definitions of the same attribute. As a corollary, an attribute with name n has the same relationship with the node – inherited or synthesized – for every style in $\mathcal{S}$.

In the previous example the attributes `width`, `height` and `output` are consistently defined as synthesized in both styles. The same thing happens with the attributes `x` and `y` that are consistently defined as inherited. The reader should notice that the first style does not define inherited attributes for its children since it is meant for leaves only.

In the TermsIO framework, attribute precedences are divided in internal and external precedences. Internal precedences result from references to attributes that are defined by the same style. They are of the form `n of` $T \leftarrow$ `m of` U where either:

1. $T = (S', i, S)$ and $U = (S'', j, S)$, with $S, S', S'' \in \mathcal{S}$, or
2. $T = (S, i, S')$ and $U = (S, i, S'')$ with $S, S', S'' \in \mathcal{S}$.

External precedences are precedences that are not internal, i.e. of the form `n of` $T \leftarrow$ `m of` U where either:

1. $T = (S', i, S)$ and $U = (S, j, S'')$, or
2. $T = (S, i, S'')$ and $U = (S', j, S)$.

The attributes whose names occur in the right-hand side of an external dependency are

external attributes. In a well-defined set of styles $\mathcal{S} = \{S_1, \cdots, S_n\}$ an external attribute must be defined in every node type $T \in \mathcal{T}_{\mathcal{S}}$. Consider an external dependency `n of` $T \leftarrow$ `m of` U generated by an attribution of S. If `m` is inherited and $U = (S, j, S'')$ then `m` is defined in S'', otherwise the dependency is internal. Similarly, if `m` in synthesized and $U = (S'', j, S)$ then `m` is also defined in S'', otherwise the dependency is internal. In both cases the attribute `m` must be defined in every $S'' \in \mathcal{S}$.

In the previous example all nodes use the attributes `x`, `y`, `width`, `height` and `output` that are referred in external precedences. The attribute `label` has internal precedences (and is not defined in any style) thus can be ignored by static analysis. The reader should also notice that this set of styles does not define the attributes `x` and `y` for root nodes. All the inherited attributes of root nodes are initialized by the attribute evaluation procedure as it happens in the AG framework.

In summary, all node types generated by a well-defined set of styles have the same set of external attributes, and each attribute name is either inherited or synthesized for all node types. These two facts point out that node types play a minor role in the precedence of attributes. Thus, for the purpose of static analysis, it can be assumed that there is a single type of node and we can consider only the dependencies between attribute names.

The set of attribute names for a given set of styles $\mathcal{S} = \{S_1, \cdots, S_n\}$ is the union of the attribute names defined in each style. A style defines a name iff it occurs in the left-hand side of an attribution, hence an attribution n`!a :=` e defines name `a`. The simplified precedence relation connects two attribute names occurring in different sides of an attribution of $S_i \in \mathcal{S}$. In detail, an attribution n`!a := f(`m`!b)` creates a precedence denoted by `a` $\leftarrow$ `b` and read as `b` precedes `a`. For instance, the attribution `0!a := 1!b + 2!b` defines the name `a` and a precedence between the name `b` and the name `b`.

Static analysis of a given set of styles S is based on its **simplified precedence graph**. This graph, denoted as $SP(\mathcal{S})$ is a tuple $(\mathcal{A}, \mathcal{E}, c)$ of attribute names $\mathcal{A}$, a set of directed edges $\mathcal{E} = \{(s,t) : s, t \in \mathcal{A}\}$ (the precedences between attributes names), and a colour function $c : \mathcal{E} \rightarrow \{ < \leq \}$. The arcs are coloured with the symbol "$\leq$" when the attribute name can be evaluated in the same pass as the attribute name in the target and "$<$" when it must be evaluated in a subsequent pass. The set of external attribute names for the two styles in Figure 3.12 and their set of precedences creates the simplified precedence graph of Figure 3.13.

The precedence graph $SP(\mathcal{S})$ of the set of styles $\mathcal{S}$ is the basis for computing the order of attributes evaluation using the standard AG methods [Alb91a].

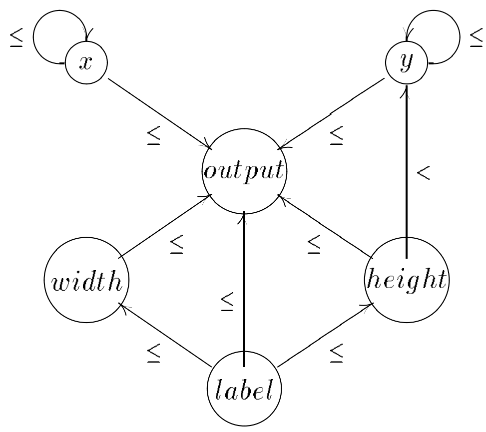

Figure 3.13: Example of a simplified precedence graph

3.4.4 Computing Evaluation Order

As explained in subsection 3.4.1, the attribute evaluation order of the multi-pass algorithm requires the assigment of pass numbers, types (inherited or synthesized) and orders to each attribute. In the case of TermsIO this information is associated to attribute names and is obtained by statically analysing a given set of styles.

The primary goal of static analysis in TermsIO is the definition of a pass function $p : \mathcal{A} \rightarrow \mathbb{N}_0^+$ that maps attribute names into pass numbers, using the simplified precedence graph for a set of styles. To define this function we start by introducing the concept of **gap** between two attributes. Given the attributes a and b the function $g : \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{Z}$ defines the gap between the attributes, i.e., the minimum number of passes between their evaluations that ensures that b is computed after all its dependent attributes. The reader should note that, by definition, $g(a, b) = p(b) - p(a)$ hence $g(a, b) = -g(b, a)$ for all $a, b \in \mathcal{A}$. In practice, we need only to define the gap function for its non-negative values in order to determine the pass function.

The edges in the simplified precedence graph are the basis for computing the gap function. If the inherited attribute b depends on the synthesized attribute a, then $g(a, b) \geq 1$; otherwise, if b is not inherited or a is not synthesized, then $g(a, b) \geq 0$. It should be noticed that the existence of a edge labeled $\geq$ between two attribute names does not guarantee that the gap between them is zero. For instance, there is an edge labeled $\geq$ between `height` and `output` in the example of Figure 3.13 but their gap is 1. This is due to the dependency of `output` from `y` that has to be computed after `height` and before `output`. Hence, the gap can be regarded as a cost associated with following a path in the simplified precedence graph.

Given an simplified precedence graph $SP(\mathcal{S})$ a **path** $[(s_1, t_1), \cdots, (s_n, t_n)]$ is a sequence of $n > 0$ edges where each edge (s_i, t_i) has it source s_i equal to the previous edge target t_{i-1}, i.e. $s_i = t_{i-1}$. For instance, the graph in Figure 3.13 contains several paths connecting `label` and `output` such as `[(label,width), (width,output]` and `[(label,height), (height,y), (y,output)]`. The set of all paths in the simplified precedence graph $\mathcal{G}$ is denoted as $\mathcal{P}_{\mathcal{G}}$ and the set of paths connecting the nodes a and b in the $\mathcal{G}$ is denoted by $\mathcal{P}_{\mathcal{G}}^{a,b}$.

The cost of following an edge is defined according to its colour. The cost of following a path is the sum of the cost of following each edge. The function $c : \mathcal{P}_{\mathcal{G}} \rightarrow \mathbb{N}_0^+$ assigns a cost to each path in the simplified precedence graph, and has the following recursive definition.

$$\begin{aligned} c([(a,b)]) &= \begin{cases} 1 & \text{if } a \text{ is synthesized and } b \text{ is inherited} \\ 0 & \text{otherwise} \end{cases} \\ c([(s_1,t_1),\cdots,(s_n,t_n)]) &= \sum_{i=0}^{n} c([(s_i,t_i)]) \end{aligned}$$

The cost of a path is thus the number of passes between the evaluation of two attribute names considering precedences between attributes represented by the edges in the path. Consequently, if attribute name b depends on attribute name a then the gap between them is defined as the maximum cost for all paths connecting the two nodes.

$$g(a,b) = max(\{c(p) : p \in \mathcal{P}_{\mathcal{G}}^{a,b}\})$$

An algorithm to determine the gap function for its non-negative values is given by the predicate `gap/1` in Figure 3.14. This predicate creates a list of terms `g(A,B,D)`, where `A` and `B` are attribute names and `D` is the gap between them, using the predicates `depend/2` and `type/1` that record the dependencies and type (inherited or synthesized) respectively.

The algorithm starts by collecting the cost of following edges (given by predicate `edge/3`), according to the type of each attribute name (given by predicate `type/2`). Afterwards, the transitive closure of the `Initial` set of cost is computed. Finally, the gap function for each pair of attribute names is defined as the maximum cost of all paths connecting the two nodes.

Given the algorithm for gap, the pass number of an attribute name can be easily computed using a recursive definition. If no attribute names precede a given attribute then it can be computed at pass number zero. Otherwise, since $g(a,b) = p(b) - p(a)$, the pass number can be computed recursively when a non-negative gap exists.

```
gap(Gap) :-
        findall(g(A,B,AB),edge(A,B,AB),Initial),
        closure(Initial,Cost),
        maximum(Cost,Gap).

edge(A,B,D) :-
        depend(A,B), \+ A=B,
        (type(A,synthesized), type(B,inherited) -> D=1; D=0).

closure(Dist,Close) :- (close(Dist,New) -> closure(New,Close); Close=Dist).

close(Dist,[D|Dist]) :-
        in(g(A,B,AB),Dist,_),
        in(g(B,C,BC),Dist,_),
        AC is AB+BC, D=g(A,C,AC),
        \+ in(D,Dist,_).

maximum(Dist,Max) :- (max(Dist,New) -> maximum(New,Max); Max=Dist ).

max(Dist,Rest) :-
        in(g(A,B,AB),Dist,Rest),
        in(g(A,B,BA),Rest,_),
        AB < BA.
```

Figure 3.14: Algorithm of the gap function

$$p(x) = \begin{cases} p(y) + g(y,x) & \text{if } \exists y \in \mathcal{A} \ : \ g(y,x) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The predicate `pass_number/3` in Figure 3.15 instantiates the pass number `Pass` for a given attribute name `Attr` using a list of gaps `Gaps` computed by the predicate `gap/1`. It starts by selecting a (non-negative) gap to an attribute name that `Preceds` the given attribute. If it exists then the pass number for the attribute `Precedes` is recursively computed using the `Rest` of the gap function (to avoid infinite recursion), and added to the selected gap. Otherwise, if no non-negative gap exists, the pass number of the attribute is defined as zero.

The information on attribute names collected by TermsIO static analysis is condensed in the predicate `attribute/4` that relates the attribute name, pass number, type and order. Hence, the goal `?- attribute(Name,Pass,Type,Order)` instantiates the correspondent information for each attribute name defined in the set of styles.

```
pass_number(Attr,Gaps,Pass) :-
        (in(g(Precedes,Attr,D),Gaps,Rest) ->
            pass_number(Precedes,Rest,P),
            Pass is P + D
        ;
            Pass is 0
        ).
```

Figure 3.15: Algorithm of the pass function

3.4.5 Global Attribute Evaluation

The global attribute evaluation in TermsIO computes **all** the attribute values of a given subtree, following the definition in a given set of styles. As explained, the implemented algorithm is **static** [Alb91c] since it consumes an attribute evaluation order defined beforehand using the static analysis algorithms of the previous subsection.

```
global_evaluation(Node) :- passes(N), evaluate_pass(Node,N,0).

evaluate_pass(Node,N,I) :- I>N.
evaluate_pass(Node,N,I) :- I=<N,
        evaluate_tree(Node,Pass),
        J is I+1,
        evaluate_pass(Node,N,J).

evaluate_tree(Node,Pass) :-
        compute_node_attributes(Node,Pass,inherited),
        evaluate_descendants(Node,Pass),
        compute_node_attributes(Node,Pass,synthesized).

evaluate_descendants(Node,Pass) :-
        child(Node,Child),
        evaluate_tree(Child,Pass),
        fail.
evaluate_descendants(Node,Pass).
```

Figure 3.16: Multi-pass attribute evaluation algorithm

The tree-walk algorithm in Figure 3.16 follows the general pattern of multi-pass class described in subsection 3.4.1. It evaluates the attributes of the subtree rooted in `Node` in `N` top-down passes, whith `N` determined by the predicate `passes/1` resulting from static evaluation. For a given `Pass`, the subtree rooted in `Node` is evaluated in tree parts: first, the algorithm computes the inherited attributes of `Node` in `Pass`; then, it recursively computes the attributes of the dependents `Node` in the same `Pass`; finally, it computes the synthesized attributes of `Node` in `Pass`. The recursive evaluation of descendants is based on the predicate `child/2` that, by backtracking, instantiates its second argument with a child of the node given as first argument.

The predicate `compute_node_attributes/3` in Figure 3.17 implements the TermsIO algorithm for evaluation of attributes of `Node` in a given `Pass`. The generation of the attribute `Name`, given `Pass` and `Type` (inherited or synthesized) is made by predicate `attribute/4`. Then, the goal `compute(Name,Node,Value)` computes the `Value` of the attribute `Name` of `Node` that is recorded in the attribute database using the predicate `set_attr_value/1`

The predicates `attribute/4` and `compute/3` result from the "compilation" of a given set of styles. The `attribute/4` predicate, generated by the static analysis, by backtracking instantiates the attribute names in a correct evaluation order, for a pass number and type. The generation of predicate `compute/3` from a set of styles is discussed in Section 3.5.

```
compute_node_attributes(Node,Pass,Type) :-
        attribute(Name,Pass,Type,_Order),
        compute(Name,Node,Value),
        set_attr_value(Node,Name,Value),
        fail.
compute_node_attributes(Node,Pass,Type).
```

Figure 3.17: Computing node attributes

The algorithm just presented assumes that all attributes are computed in left to right passes. It can easily be extended to include right-to-left passes provided the necessary adjustments in the predicate `attribute/4` are generated by static analysis. The current implementation of TermsIO allows only left-to-right passes since in practice right-to-left precedences are seldom needed.

3.4.6 Updating the Display Tree

The role of the incremental attribute evaluator is to update the attribute values after a modification of the display tree. There is no need to use the global attribute evaluator to maintain the consistency of the attribute tree since most modifications do not actually require a complete recomputation. In fact, if a small change in an attribute of a subtree (for instance, adding a character to a functor), required a complete reevaluation of the display tree then an environment based on this approach would be infeasible. Fortunately, in most cases, the modified subtree is very small when compared with the complete display tree and has limited influence on the rest of the attributes. Thus, the recomputation of attributes not directly affected by a modification can be performed with a different strategy to avoid the complete reevaluation of all attributes.

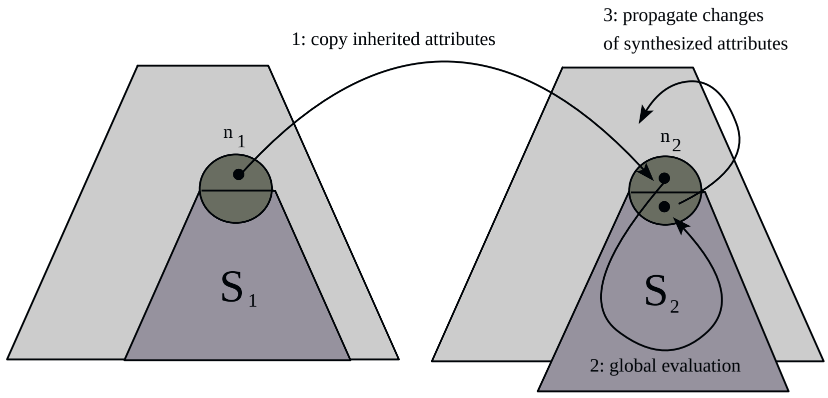

Figure 3.18: Updating display tree after a modification

A modification of the display tree is a set of **elementary modifications**. Each elementary modification is a replacement of a sub-tree S_1, rooted on node n_1 by another sub-tree S_2, rooted on node n_2, after which the attribute values are updated as sketched in Figure 3.18. After the replacement of S_1 by S_2 the attributes are updated in three steps:

1. The inherited attribute values of n_1 are copied to n_2 (if $n_2 \neq n_1$).
2. The attributes of S_2 are computed using the global attribute evaluation algorithm.
3. The changes in the synthesized attributes of n_2 are propagated to the rest of the display tree using an incremental attribute evaluator.

Most of the attributes of the display tree need not be evaluated during step 3. An attribute needs to be reevaluated only if the value of an attribute it depends on was both reevaluated **and** changed its value. In fact, it is frequent that attribute reevaluation does not change the attribute's value since they are computed by conditional expressions or by expressions using the maximum and the minimum of a set of other values.

A modification of the display tree is frequently a composition of several elementary transformations with non-overlapping trees replaced in sequence. In this case, TermsIO does not repeat this procedure after each elementary transformation, as that would lead to several reevaluations of some attributes. The process would be time consuming and contrary to the idea of incremental reevaluation. Instead, the procedure is applied to the sub-tree rooted on the **least common ancestor** of all elementary transformations.

A common ancestor of a set of nodes is defined as a node from which there is a path of parent-child arcs to all nodes in the set. A common ancestor of a set is the least common ancestor iff is a descendant of all other common ancestors. It is trivial to notice that the least common ancestor is unique.

3.4.7 Incremental Attribute Evaluation

The TermsIO static evaluator is a rigid evaluator in the sense that it follows a predetermined evaluation order derived from static analysis. The incremental methods based on the multi-pass algorithm [Alb91b] use also static analysis information to predetermine an evaluation order and dependencies between attributes to avoid visiting the subtrees that where not influenced by modified attributes. This kind of algorithm implies a complete visit to the attribute tree, and thus is not appropriate for modifications in small subtrees influencing few dependent attributes.

The TermsIO incremental attribute evaluator uses a flexible algorithm that dynamically determines the attributes needing reevaluation and the evaluation order. The evaluator uses a data-driven algorithm [Hud91] that maintains the set of attributes needing reevaluation in the topological order of the dependency graph.

```
incremental_evaluation(Attributes) :-
        sort_attributes(Attributes,SortedAttributes),
        reevaluate(SortedAttributes).

reevaluate([]).
reevaluate([Node!Name|Rest]) :-
        get_attr_value(Node,Name,Old),
        compute(Name,Node,New),
        (New == Old ->
            Dependent=[]
        ;
            set_attr_value(Node,Name,New),
            dependent_attributes(Node,Name,Dependent)
        ),
        merge(Dependent,Rest,Attributes),
        reevaluate(Attributes).

dependent_attributes(Node,Name,Dependent) :-
        setof(DNode!DName,
              depend(Name,Node,DName,DNode),
              Set), !,
        sort_attributes(Set,Dependent).
dependent_attributes(Node,Name,[]).
```

Figure 3.19: Attribute reevaluation algorithm

The incremental evaluation algorithm, presented in Figure 3.19, processes a list of synthesized `Attributes` of the modified tree's root node, whose value has changed. The attributes are sorted by the topological order of the dependency graph and passed to the predicate `reevaluate/1`, and are maintained as an ordered set of attributes over the recursions.

The predicate `reevaluate/1` processes the ordered attribute set until this set is empty. In each iteration the first element `Node!Name` from the attribute set (consequently an element that does not depend on other attributes in the set), is selected. The attribute `Name` of `Node` is computed by the predicate `compute/3`, derived from the set of styles and whose definiton is discussed in the next section. If the computed value of the attribute differs from its previous value, them the new value is recorded in the attribute database and the set of attributes with direct dependencies from the changed attribute is determined. The `Dependent` attributes are merged with the `Rest` of the attribute set before the next iteration of `reevaluate/1`.

The set of dependent attributes is collected in the predicate `dependent_attributes/3` using the predicate `depend/4`. This predicate generates the arcs of the dependency graph that is not maintained explicitly as a data structure. The predicate `depend/4` is "compiled" from the set of styles, together with the predicate `compute/3`.

```
before(NodeA!NameA,NodeB!NameB) :-
                NameA == NameB,
                attribute(NameA,_,synthesized,_),
                ancestor(NodeB,NodeA).
before(NodeA!NameA,NodeB!NameB) :-
                NameA == NameB,
                attribute(NameA,_,inherited,_),
                ancestor(NodeA,NodeB).
before(NodeA!NameA,NodeB!NameB) :-
                NameA != NameB,
                precedes(NameA,NameB).
```

Figure 3.20: Attribute name precedence

The set of attributes used in the reevaluation algorithm are maintained in a list which is sorted using the topological order of the dependency graph. This requires sorting in the topological order the initial list of attributes, and the set of dependent attributes, using the predicate `sort_attributes/2`. Moreover, the set of dependent attributes and the set of attributes are merged, using the predicate `merge/3` that maintains the same order. These two predicates use an attribute order relationship, defined by `before/2` in Figure 3.20, based on the attribute information collected by static analysis in the predicate `attribute/4` and in the relationship of the nodes in the display tree.

For attributes with the same name the order is defined using the `ancestor/2` relation that succeeds when the first argument is an ancestor of the second. Attributes with the same name can be related by attributions such as `1!x := 0!x`. In this case the attribute of the ancestors must be computed before that of the descendants, since the attribute `x` is inherited. If the attribute were synthesized, as in `0!width := max({$!width})`, then the attribute of the ancestors should be computed after those of the descendants.

```
precedes(NameA,NameB) :-
                attribute(NameA,PassA,_,_),
                attribute(NameB,PassB,_,_),
                PassA < PassB.
precedes(NameA,NameB) :-
                attribute(NameA,Pass,inherited,  _),
                attribute(NameB,Pass,synthesized,_).
precedes(NameA,NameB) :-
                attribute(NameA,Pass,Type,OrderA),
                attribute(NameB,Pass,Type,OrderB),
                OrderA < OrderB.
```

Figure 3.21: Attribute order

For different attribute names there is always a precedence between them induced by the predicate `attribute/4`, as defined by predicate `precedes/2` in Figure 3.21. If the attribute names are associated with diferent passes, then the attribute with a lower pass number must be evaluated first. If the two attribute names are computed in the same pass, then an inherited attribute is always computed before a synthesized one. Attributes with the same type and pass number are compared in their order.

3.5 Processing Styles

This section presents the TermsIO module that converts style definitions into the attribute evaluation predicates used by both the global and incremental attribute evaluators. It describes how styles are preprocessed, how clauses are generated from them, and finally how these clauses are optimized before being asserted in the clausal database.

3.5.1 Preprocessing

The style definitions that are loaded into TermsIO are defined by sets of terms that have to be normalized in sequences of attributions. This process is performed by a **rewrite system** [DJ90] that reduces general terms representing a style to a normal form.

The system implements a simple but efficient strategy. It traverses the term from top-down, left to right, in a single pass; at each node it applies all the matching rules. This strategy assures both termination and confluency provided that the set of rules has these properties [O'D85, HO80]. However, the rewrite system does not check the set of rules for the sake of efficiency.

As the reader has probably guessed by now, the term rewriter is completely written in Prolog Not only the rewriter itself is written in Prolog but the rewrite rules are also converted to Prolog clauses when they are loaded into the system by using Prolog's built-in term expansion mechanism [C^+, DC89, Qui91]. This approach benefits from both Prolog's support for unification and clause indexing in predicate evaluation.

As explained in section 3.2, the TermsIO rewrite rules are terms of the form `LHS => RHS`, where `LHS` and `RHS` are general Prolog terms. TermsIO supports also conditional rules of the form `LHS => RHS <- GOAL`, where `GOAL` is a valid Prolog goal. Each rule is converted into two Prolog clauses of the predicates `rewrite_first/2` and `rewrite_next/2`. Both clauses rewrite terms matching the rules left-hand side `LHS`, passed as first argument, and instantiate the second argument with the rewritten term. The `rewrite_first/2` clause rewrites recursively the sub-terms matching the rule's variables before the substitution, and hence is evaluated only in the first attempt to normalize a term. The `rewrite_next/2` clause assumes that the sub-terms matching the rule's variables are already in normal form after the `rewrite_first/2` clause is applied.

```
X - X => 0                      rewrite_first(X-X,0) :- true.
                                rewrite_next( X-X,0) :- true.

X + 0 => X                      rewrite_first(X+0,Y) :- rewrite(X,Y).
                                rewrite_next( X+0,X) :- true.
```

Figure 3.22: Rewrite clauses

Figure 3.22 presents the clauses generated from rule `X - X => 0` and `X + 0 => X`. Let us consider the clauses derived for the first rule. The first argument of both clauses match terms of the form `X - X`. Since the sub-terms matching `X` are not used in the reduced term the clauses `rewrite_first/2` and `rewrite_next/2` are similar. Now, let us consider the clauses derived for the rule `X + 0 => X`. Again, the first argument of both clauses matches terms of the form `X + 0`, but the `rewrite_first` clause invokes the rewriter to normalize the sub-term matching `X` and returns the normal form; the `rewrite_next/2` clause assumes that the term matching `X` is already in normal form thus unifies the normal form with `X`.

These two predicates are used by the rewrite engine presented in figure 3.23. The predicate `rewrite/2` starts by trying the `rewrite_first` predicate derived from the rewrite rules. If it fails then there is no rule to reduce the term, hence the rewriter decomposes `T` and rewrites each of its arguments, composing a new term `W` with the same functor and the rewritten arguments.

```
rewrite(T,R) :- rewrite_first(T,R), !.
rewrite(T,R) :-
        T=..[F|A], rewrite_args(A,B), W=..[F|B],
        reduce(W,R).

rewrite_args([],[]).
rewrite_args([H|T],[R|S]) :- rewrite(H,R), rewrite_args(T,S).

reduce(T,R) :- rewrite_next(T,R), !.
reduce(T,T).
```

Figure 3.23: Rewrite engine

After normalizing the arguments of `T`, a rule may exist that reduces the transformed term `W`. For instance, the term `a+(a-a)` is not rewritten by rule `X + 0 => 0` in `rewrite_first/1`, but after normalizing the second operand, using the rule `X - X => 0`, the rule is applicable. At this point any sub-term that may match the rule's variables is already normalized. Thus, the predicate `reduce/2` is applied to reduce the term using the `rewrite_next/2` predicate and returns the original terms if this goal fails. The use of two rewrite clauses for each rewrite rule make the rewrite engine more efficient, since it avoids a lot of unnecessary reductions.

```
rewrite_first(norm(X,Y),Z):-
        rewrite(X,A), reduce(A*A,C),
        rewrite(Y,B), reduce(B*B,D),
                      reduce(C+D,E),
                      reduce(sqrt(E),Z).

rewrite_next(norm(X,Y),Z):-
                      reduce(X*X,C),
                      reduce(Y*Y,D),
                      reduce(C+D,E)),
                      reduce(sqrt(E),Z).
```

Figure 3.24: Rewrite clauses for `norm(X,Y) => sqrt(X*X+Y*Y)`

In general the rules used by TermsIO are more complex than those presented in Figure 3.22. Figure 3.24 shows the `rewrite_first/2` and `rewrite_next/2` clauses for the rule `norm(X,Y)=>sqrt(X*X+Y*Y)`. The right-hand side of this rule is a term with several functors that may be reduced by another rewrite rule in the same set. The two clauses are similar, differing only in the fact that the sub-terms matching `X` and `Y` are rewritten for `rewrite_first/2`. In both clauses, the sub-terms of every functor are reduced with predicate `reduce/2`.

```
first_clause(T,R,Call) :- var(T), !, Call=rewrite(T,R).
first_clause(T,R,Call) :-
        T=..[F|A],
        first_args(A,B,Call,reduce(NewT,R)),
        NewT=..[F|B].

first_args([],_,[],Call,Call).
first_args([H|T],[R|S],Call,Diff) :-
        first_clause(H,R,H2R),
        first_args(T,S,Call,(H2R,Diff)).
```

Figure 3.25: Algorithm for converting rules into clauses (I)

The algorithms for converting the clauses `rewrite_first/2` and `rewrite_next/2` are presented in Figure 3.25 and Figure 3.26 respectively. The predicate `first_clause/3` receives the left-hand side and right-hand side of the rule to convert in the first two arguments, and instantiates the third with body of the clause. The `first_clause/3` predicate traverses the left-hand side `T` and generates a `Call` to `rewrite/2` for every variable. Non-variable terms are decomposed, and their arguments are converted recursively. The sequence of goals that convert each argument, terminated by a reduction of the composed term, is the goal of the rewriting clause.

```
next_clause(T,T,true) :- var(T), !.
next_clause(T,R,Call) :-
        T=..[F|A],
        next_args(A,B,Call,reduce(NewT,R)),
        NewT=..[F|B].

next_args([],_,[],Call,Call).
next_args([H|T],[R|S],Call,Diff) :-
        next_clause(H,R,H2R),
        next_args(T,S,Call,(H2R,Diff)).
```

Figure 3.26: Algorithm for converting rules into clauses (II)

The algorithm for converting the `rewrite_next/2` clause presented in Figure 3.26 is similar to that of `rewrite_first/3`. The major difference is in the first clause, for variables, that does not generate a goal to rewrite the term.

The conditional rules of the form `T => R <- Goal` are processed in a manner similar to that of simple rules. The `Goal` is simply appended to the right-hand side of the converted clauses `rewrite_first/2` and `rewrite_next/2` clauses.

3.5.2 Generation

The styles submitted to TermsIO are normalized into sequences of attributions defining the expressions that compute the attributes of the display tree. The styles attributions are then used in the attribute evaluators of TermsIO. First, they are statically analyzed to determine the attribute names evaluation order, as presented in the previous section. Then, they are used to generate the predicates that actually compute values and dependencies in the attribute evaluation procedures. The attribute evaluators of TermsIO use two predicates, called **style predicates**, generated from a set of style attributions:

`compute(Node,Name,Value)` computes the `Value` of attribute `Name` of `Node`.

`depend(Node,Name,DNode,DName)` by backtracking returns the pair `DNode` end `DName` identifying the attributes that depend attribute `Name` of `Node`.

Every style loaded into TermsIO contributes with clauses to each style predicate. In the case of `compute/3`, each clause corresponds to the computation of an attribute name of a node type. In the case of `depend/4`, each clause generates the dependent attributes for a given attribute name and node type.

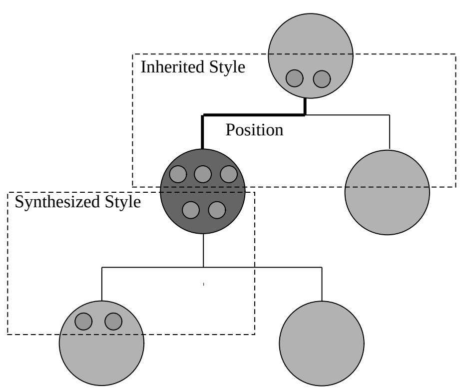

Figure 3.27: The type of a node

The `node's type` has an important role in the selection of the relevant clause in a style predicate. Since a style defines both the node's synthesized attributes and the inherited attributes of its descendents, it is not possible to assign a single style to each node. As we have seen in subsection 3.4.2, in order to compute all the node attributes it is necessary to know their inherited style, their synthesized style and their relative position among the siblings. This information is encoded in a term of the form `type(Inherited,Position,Synthesized)` and is recorded in the node's attribute database.

```
compute(Node,Name,Value) :-
        get_attr_value(Node,type,Type),
        compute(Name,Type,Node,Value).

depend(Node,Name,DNode,DName) :-
        get_attr_value(Node,type,Type),
        depend(Node,Name,Type,DNode,DName).
```

Figure 3.28: Style predicates

The style predicates, defined in Figure 3.28, interface with the predicates that are actually generated from styles. They retrieve the node's type from the attribute database, indexed by the node's identification, using the goal `get_attr_value(Node,Name,Value)`, and use it to select the relevant generated clauses.

Let us start by analyzing the clauses of `compute/4` generated from styles. As explained, the predicate is deterministic and uses the head arguments `Name` and `Type` to select a unique clause for every evaluation. The body of the clause uses the argument `Node` to access the attribute's database (via `get_attr_value/3`) and computes the attribute's `Value`. As explained in section 3.3, the database constrains several data indexed on `Node`, namely the attributes themselves, the node's `type`, the `parent` node and each of the node's sons indexed by their position in `son(`*i*`)`.

```
1!i := 0!a              compute(i,type(foo,1,_),Node,Value) :-
                                get_attr_value(Node,parent,P),
                                get_attr_value(P,a,Value).

0!s := 1!b + 2!c        compute(s,type(_,_,foo),Node,Value) :-
                                get_attr_value(Node,son(1),S1),
                                get_attr_value(S1,b,B),
                                get_attr_value(Node,son(2),S2),
                                get_attr_value(S2,c,C),
                                Value is B+C.
```

Figure 3.29: Clause for inherited and synthesized attributes

Figure 3.29 exemplifies how the definition of the inherited attribute `i` and the synthesized attribute `s` of the style `foo` are converted to Prolog clauses. A clause of `compute/4` is generated for each style attribution. The clause derives its head and body respectively from the left and right hand side of the attribution. The name of attribute determines the head's first argument, and the style's name (`foo` in the example) the second. The name of the style is placed in the first or third argument of the `type/3` structure according to the kind of attribute (inherited or synthesized). When the attribute is inherited the second argument of the `type/3` structure is complemented with the node's position among its siblings.

3.5.3 Optimizations

The bodies of clauses, generated automatically according to the ideas described in the previous section, may be inefficient due to redundancies in calls to the internal database and in the evaluation of predicates. TermsIO addresses this problem by using several standard optimization techniques [AsU86], used by intermediate code generators for imperative languages. Fortunately, the optimizations are required only in the bodies of the generated clauses, where no branching occurs, thus simplifying the flow analysis for TermsIO. Moreover, there is a resemblance in the concept of variable in both LP and intermediate code that simplifies the implementation of the standard algorithms.

The concept of variable used in the generation of intermediate code is similar to the concept of logic variable: each variable has at most a single value during execution and variables are not "reused" (there is no destructive assignment). Since TermsIO is implemented in Prolog, it uses logic variables to simplify the replacement of redundant variables using unification. This idea is not exactly new. In his seminal paper "Logic Programming and Compiler Writing" David Warren [War80] presented this idea, taking advantage from logic variables to write a one pass code generator in Prolog.

The style predicates are deterministic when called with an attribute name and a type, since these arguments select a single of the generated clauses. Hence, the optimizations are only required in the body of the generated clauses. The body of a `compute/4` clause, for instance is a term recognized by predicate `goal/1` in Figure 3.30. It is a sequence of goals with set aggregations (`findall/3`) and conditionals; in the base case goals are conditions (recognized by `test/1`) and attributions. A simple attribution "assigns" a value to a logic variable. In a second form a logic variable may be instantiated as the result of evaluating a goal.

The optimizations performed by the TermsIO compiler consist of the following transformations:

Elimination of redundant evaluations: if two goals `G1` and `G2` occur in the body of a computation clause, so that `attribution(G1,V1,E1)` and `attribution(G2,V2,E2)` have `E1==E2` (i.e. `E1` and `E2` are strictly equal) then `V1` and `V2` can be unified (`V1=V2`))

```
goal((A,B)) :- goal(A), goal(B).
goal((V -> T ; F )) :- test(V), goal(T), goal(F).
goal(findall(Term,Goal,List)) :- goal(Goal),
goal(Goal) :- attribution(Goal,Var,Expr).

attribution(Var is Expr,Var,Expr).
attribution(Term,Var,Expr) :-
        Term=..[Functor|Args],
        append(Args_1,[Var],Args),
        Expr=..[Functor|Args_1].
```

Figure 3.30: Optimization goal

and the second goal occurring in the body can be eliminated. Most of the optimizations performed by TermsIO are of this nature.

Execution of ground evaluations: if an expression of an assignment does not contain other variables, then the variable can be instantiated with its value at compile time and the corresponding goal deleted from the body. Only in a few cases the expressions depend on graphic context values (for instance, font sizes of ground strings) that may be reevaluated in result of global changes (for instance, changing de default font). This kind of expressions is not evaluated at compile time. Instead they are replaced with calls to a caching mechanism that reduces the cost of these operations without avoiding a possible re-computation. For instance, if the default font changes and text sizes have to be recomputed then the cache is reset.

Elimination of unused evaluations: if the variable of an attribution is not subsequently used it is removed from the goal. This straightforward optimization has little impact when compared with the previous two, since it is rare that an unused expression is generated.

The optimizations described above were tested with the TermsIO default set of styles. This set contains 21 styles that generate 481 clauses with a non-empty body. The number of goals in the optimized clauses is reduces by 21.60% relatively to the original clauses. The optimizations are particularly expressive in large clauses resulting from complex expressions in styles. In these cases the goals tend to have more common references to attributes that can be eliminated.

The Table 3.3 summarizes the optimization gains in the default style set. The results were grouped by the number of goals per clause. For each group are listed the number of clauses, the total number of goals, the total number of goals after the optimization and the percentage of gain in number of goals. The gains for clauses with less than 5 goals are not significant.

n =goals per clause	# clauses	original	optimimized	% gain
$n >= 15$	7	211	94	55.45%
$10 <= n < 15$	23	277	203	26.71%
$5 <= n < 10$	60	397	313	21.16%
$0 <= n < 5$	391	629	577	8.27%

Table 3.3: Optimization

The gains increase with the number of goals per clause. The maximum gain was obtained in a clause with 42 goals with a 74% reduction in the number of goals. Although small in number, the large clauses are frequently executed since they are generated from the node's default styles. This kind of style contains complex set expressions since has to be prepared to handle nodes with an indefinite number or children.

3.6 Applications Interface

The applications interface (API) provides the client applications with a set of **commands** to manipulate the system's internal state. The interface consists of a single predicate that is called with a single argument – the command to be executed on the internal state. The predicate can be seen as a command interpreter receiving the abstract parse tree of a complex command line.

```
command(control(Command)) :- control(Command).
command(focus(Command))   :- focus(Command).
command(tree(Command))    :- tree(Command).
command(node(Command))    :- node(Command).
command(attr(Command))    :- attr(Command).
```

Figure 3.31: TermsIO commands

The command terms available to application programmers are the terms accepted by the predicate `command/1` defined in Figure 3.31. They include several types of commands, to control TermsIO, to change the focus, to manipulate sub-trees of the display tree, edit nodes of the display tree and use attributes. These types of commands are described in detail in the following sub-sections.

3.6.1 Controlling TermsIO

The control commands are mostly used in the implementation of the TermsIO widget class. As explained in the the chapter's introduction, this widget class is the TermsIO interface to the windowing system. In other words, the display tree is drawn in the windows of these widgets. The valid control commands are recognized by the predicate `control/1` in Figure 3.32.

```
control(start(Window))        :- window(Window,[]).
control(reset(Style,Window)) :- window(Window,[]), atom(Style).
control(focus(Window))        :- window(Window,[]).
control(restyle(Window))      :- window(Window,[]).
control(flush(Window))        :- window(Window,[]).
control(current(Window))      :- window(Window,[]).

control(A:=V) :- atom(A)
control(A=V) :- atom(A).
control(clear(A)) :- atom(A).
control(toggle(A)).
```

Figure 3.32: Control commands

The commands in this set perform control tasks on a widget's window that uses TermsIO. In the following paragraphs `Window` is the window identifier recognized by Edipo (cf. chapter 5).

`start(Window)` Initializes a window to start using TermsIO. This predicate must be executed only once for each window.

`control(reset(Format,Window))` Removes all trees from `Window` and sets the nodes default style to `Style`. This predicate must also be executed when a window starts using TermsIO and may be reexecuted to reset the window to its initial state.

`control(focus(Window))` Inform TermsIO that `Window` received input focus[3] from the windowing system.

`control(restyle(Window))` Recomputes all attributes. This command is typically used when the windows default font changes and all nodes have to be repositioned and layouts redrawn.

[3]The windowing system focus informs the window that it will receive all keys typed by the user until another focus is set. This focus is not related with the TermsIO focus.

`control(flush(Window))` Flushes pending changes to `Window`. The commands that modify the display tree do not automatically change a window's content to avoid the flickering caused by several updates performed in a row. The window's image is updated after the execution of this command.

`control(current(Window))` Succeeds if `Window` is currently declared in TermsIO. If `Window` is uninstantiated then by backtracking returns each window declared in TermsIO.

The four commands in the bottom of Figure 3.32 are used for global control of the TermsIO system, for setting, querying, clearing and toggling (between true and false) the values of global attributes. In the current implementation these tools are used for managing global attributes of the collapse/expand feature described in Subsection 3.6.4.

`minimize`: records a boolean, that indicates if nodes are collapsed on focus.

`horizon`: records a positive integer, the number of levels that are expanded.

3.6.2 Focus Operations

Most API commands operate on the subtree currently under focus. A new focus can be set either at an arbitrary location (specified using the mouse, or other pointing device) or at a relative location based on the current focus.

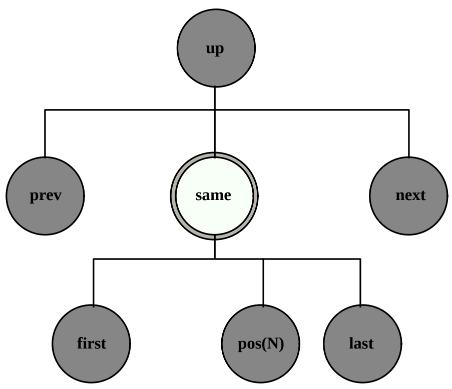

Figure 3.33: Nodes referenced relatively to current focus

The focus root node is the center of a set of nodes that can be referenced relatively to it, such as its parent, the previous and following siblings, and the first and the last of its children. This relationship is illustrated in Figure 3.33. The API focus commands are recognized by the predicate `focus/1` in Figure 3.34.

```
focus(save).
focus(restore).
focus(save(State)).
focus(restore(State)).

focus(change(Position)) :- position(Position).
```

Figure 3.34: Focus commands

Not all focus commands actually change the focus. The commands recognized by the first four clauses of the predicate `focus/1` are used to save and restore the current focus position. The `focus(save)` and `focus(restore)` options keep the recorded focus in a last-in first-out stack. The `focus(save(Focus)` binds `Focus` with information about the current focus that can be used by `focus(restore(Focus)` to restore the previous focus. These commands are useful in situations where it is necessary to keep several foci.

```
position(same).
position(up).
position(first).
position(next).
position(prev).
position(last).
position(pos(N)) :- integer(N).

position(Window+(X,Y)) :- atom(Window), integer(X), integer(Y).
position((A,B)) :- position(A), position(B).   % path
```

Figure 3.35: Positions

The commands recognized by the last clause of `focus/1` change the current focus to a position determined by its argument. The predicate `position/1` in Figure 3.35 recognizes the valid position terms: the relations described in Figure 3.33, and the random selection of a node using a coordinate pair. A position can also be referenced as a **path**, i.e. a sequence of relative positions from the current focus point.

3.6.3 Tree Manipulation

The tree manipulation commands operate on the sub-tree currently holding the focus. The valid tree commands are given by the predicate `tree/1` whose definition is presented in Figure 3.36.

```
tree(unify(Term)).
tree(unify_tree(Position)) :- position(Position).
tree(get(Term)).
tree(set(Term)).

tree(collapse).
tree(expand).
tree(collapse_expand).
tree(full_expand).
tree(style(Style)).
```

Figure 3.36: Tree manipulation commands

The commands recognized by the first four clauses exchange data between Prolog and TermsIO. To convert between Prolog terms and sub-trees of the display tree, TermsIO uses the relationship between them discussed in the previous section and summarized in Table 3.1.

The first two commands in this set are extensions of the unification algorithm due to Robinson [Rob65] without implementing the occur-check. The first, `unify(Term)`, unifies the sub-tree under focus with `Term`. The second, `unify_tree(Position)`, is similar to the previous command but unifies the sub-tree under focus with another sub-tree referenced by `Position` (the predicate `position/1` in defined in Figure 3.35). The executions of the unification commands fail if the term/tree or the trees are not unifiable. On a successful unification the nodes corresponding to variables are replaced with trees corresponding to the matching sub-terms.

The arguments `set(Term)` and `get(Term)` of the `tree` command transfer data respectively to and from TermsIO without unifying trees and terms. If necessary, `set(Term)` replaces sub-trees in the focus (even sub-trees not corresponding to variables) to make equivalent `Term` and focus. Conversely, `get(Term)` unifies `Term` with the focus without replacing sub-trees in the focus.

The tree manipulation commands also include commands to control the visualization. The `collapse` option reduces the sub-tree under focus to a simple label, and the `expand` option reverts it to the previous state. The `collapse_expand` option toggles between the two states. The `full_expand` option can be used to expand completely a collapsed sub-tree that contains collapsed descendents.

The final command in this set is used to assign a default style to all tree nodes. This style is used as the default style for all the nodes of the tree.

3.6.4 Node Manipulation

This set of commands operates only on the top nodes of the focused sub-tree. They deal with node editing and navigation, style update, and label editing.

```
% node editing and navigation
node(create(Position) :- position(Position).
node(reset).
node(delete).

% style update
node(style(Style)) :- atom(Style).
node(default).

%label editing
node(label(Label)) :- string(Label).    % set label
node(chars(Chars)) :- string(Chars).    % add chars to label
node(backspace).                        % remove last char in label

string([]).
string([Char|Tail]) :- integer(Char), string(Tail).
```

Figure 3.37: Node manipulation commands

The first commands are the node editing commands. The `create(Position)` command uses the references recognized by `position/1` to create a new node (a slot) in a given position to the current focus. The `reset` option transforms the current node into a slot by removing all its descendents and clearing its label. The `delete` option removes the node and all its descendents.

The two following commands change the node's style without affecting the styles of its descendents. The option `style(Style)` of the `node` command sets `Style` as the node style. The `default` option clears this setting. If no style is explicitly assigned to a node using a `node(style(Style))`, then TermsIO uses a style declared with `tree(style(Style))` to locate the rules that compute the attribute values of a node. In this case TermsIO looks up for a declaration in the ancestors of the current node, from the current node to the top node of the display tree, using the first style found. As mentioned before, the top node is created with a tree style that guarantees that a node has always a default style.

The last commands of this set manipulate the node's label. These commands differ from the attribute manipulation commands described in the next subsection because after modifying the label attribute they trigger the attribute reevaluation process.

3.6.5 Attribute Database

Each node has a private attribute database that is implemented using Prolog's internal database, using the node identifier as indexing key. The attribute commands supported by TermsIO are recognized by the predicate `attr/1` in Figure 3.38.

```
attr(Attr=Value)  :- atom(Attr).
attr(Attr:=Value) :- atom(Attr).
attr(clear(Attr)) :- atom(Attr).
```

Figure 3.38: Attribute commands

The API commands for interaction with the attribute databases are intended for those rare situations where an application needs to set, query or clear an attribute. This situation may arise when applications use the TermsIO attribute evaluator to compute and update private attributes. In this situation the programmer must extend the relevant styles with rules to compute the attribute values.

Chapter 4

Ytoolkit: a Graphical Object Toolkit

The Ytoolkit is a graphical package for Prolog applications completely developed in Prolog. The system consists of a graphical interface toolkit and a set of interaction objects to build Graphical User Interfaces (GUI). The Ytoolkit is neither a binding to a windowing system nor a binding to a graphical toolkit. It actually implements its own set of graphical objects, with their own look and feel, and a set of tools to create and manage the interface.

Like most graphical toolkits, the Ytoolkit structures the interface using an Object Oriented (OO) approach. The object oriented framework is implemented in Prolog using the standard module system to encapsulate and reuse the definitions of graphical objects. The use of Prolog as the implementation language simplifies the creation of new widget classes with different functionalities, and not only mere specializations of a set of predefined classes.

This chapter is organized as follows. The following section presents other graphical packages for Prolog that can be related to the Ytoolkit. Section 4.2 introduces the Ytoolkit's design philosophy. The predicates for managing interfaces and the default class set are introduced in Section 4.3. Finally, in section 4.4, are explained the workings of the graphical objects and the tools to extend the standard widget class set.

4.1 Related Work

Several Prolog systems have been released with interfaces to specific graphical systems, such as X-Windows, MacOS or Windows95. ProXl [LJ89] is an example of an X-Windows binding using the Xlib. ALPES XProlog [Abr89] interfaces to the standard X11 toolkit via the WDL (Widget Description Language) to allow the creation of new classes of widgets using basic objects. On BIM Prolog, the Carmen tool [Cal90] interactively generates user interfaces, described as BIM Prolog specific code, for either X-Windows or Suntools. An interesting alternative are the systems where the application and the interface are very loosely connected. An example of this approach is the SICStus Prolog GM [C+], where the interaction is supported by a separate process that communicates with the application through sockets. Nevertheless, this system is also not extensible since it is basically an interface to Interviews.

As far as we know, these systems do not use Prolog as the implementation language. They make use of Prolog to describe an interface that, in some cases, is "compiled" to a more conventional form. The Garnet Toolkit [Mye89] interfaces X-Windows to Lisp programs. The toolkit uses Lisp as the implementation language, and has the same concern for extensibility.

4.2 Design Philosophy

The ultimate goal of the Ytoolkit is to simplify the creation of a non-standard GUI for Prolog programs. It is assumed that the creation of a non-standard GUI implies the extension of the basic widget set with classes. In these cases, it is important to have control of every aspect of the widgets **look and feel** , that is, of their layout on screen and reactions to user actions, but without the details on the windowing system interface libraries. Hence, the Ytoolkit has two major design goals:

Extensibility Prolog is a language particularly suited for testing new approaches to complex programming problems. Frequently, these prototypes require user interfaces with more sophisticated widgets than the buttons or scrollbars provided by the standard widget set. Hence, the design and integration of new classes of widgets must be supported by the system.

Simplicity Most toolkits use an object oriented approach to structure the interface. This approach simplifies the definition and management of a GUI built using a standard widget set. In general, the definition of a new widget class is more complex and requires a deep knowledge of the workings of the toolkit and of the underlying windowing system. Hence, the graphical toolkit's internal structure must be simple enough to encourage the development of new widget classes.

These design goals were met using Prolog as the implementation language with an OO approach to the interface structure. The use of Prolog as the implementation language simplifies the integration of user interfaces with applications, the extension of the basic widget set with new classes, and the portability of applications across Prolog engines as well as windowing systems. The interface with the underlying windowing system is implemented in a separate package - Edipo - that is described in the next chapter.

Like most graphical toolkits, the Ytoolkit promotes an OO style of programming. The OO approach is particularly suited for structuring a graphical interface package since widgets are natural candidates to objects. Moreover, classes can be used to group graphical objects with a similar look and feel.

The Ytoolkit does not aim at being a full fledged OOP system. It implements the OOP functionalities needed for structuring interfaces. In particular, the Ytoolkit borrows from the OOP paradigm the following concepts:

Encapsulation From the point of view of the programmer, a widget class is a collection of procedures for managing a set of similar widgets. In this view, each widget is a collection of data defining the details of a class instance. Hence, the Ytoolkit defines widget classes using code encapsulation, and widgets using data encapsulation.

Inheritance In some cases the differences between similar widgets are too many to be encoded in the widget's state, but are too little to justify the development of a completely new class. By using inheritance the programmer is able to reuse an existing class and thus avoid implementing it from scratch. Inheritance can also be used for abstracting a set of functionalities used by more than one class. For instance, radio buttons and check buttons share the basic functionality of a standard button.

Messages In this OOP model, objects (widgets) communicate by sending messages. This can be seen as invoking procedures in the object receiving the message. Using this approach, a class defines a set of message handlers that is used by the Ytoolkit for managing widgets during their life cycle. Moreover, message passing is a convenient mechanism for forwarding windowing system events (cf. Chapter 5) to the widgets.

The development of the Ytoolkit was made along the following major lines:

- The creation of a set of basic tools to manipulate graphical objects (widgets) allowing the casual interface programmer to improve the quality of an application.
- The creation of a flexible mechanism to handle events and manage the interaction between the widgets and the user.
- The specification of an object oriented formalism to design new classes of widgets and the development of tools to help in their implementation.

4.3 Interface Programming

The Ytoolkit object structure is the basis of a library of tools and of a set of standard classes that trivializes the task of programming a GUI for Prolog applications. This section presents an example of interface programming using the Ytoolkit and describes its interface and interaction management tools.

4.3.1 An example

Figure 4.1 presents the "Hello world" program. The example is similar to the well know example of the C programming language, and X tutorials [MA88]. It exemplifies how a simple GUI can be created using the Ytoolkit.

```
:- use_module(library(ytoolkit)).

go :-
        new_widget(_,[],top),
        new_widget(button,[
                parent=top,
                x=100, y=50,
                text=hello,
                callback=great(top)
        ],_),
        new_widget(button,[
                parent=top,
                x=100, y=100,
                text=quit,
                callback=exit_loop
        ],_),
        loop.

great(Top) :- draw(Top,writestring(20,20,"hello world")).
```

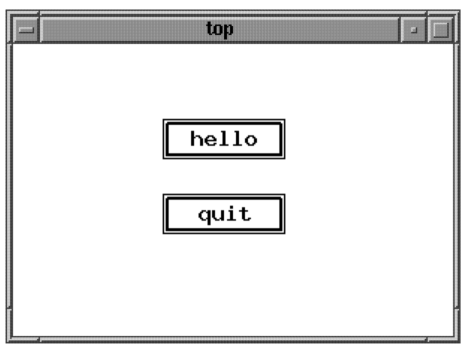

Figure 4.1: The "hello world" example

This application uses an interface with two buttons that allows the user to choose between two different alternatives. The code of this example is organized in several parts:

Declarations A program using the Ytoolkit starts by declaring the libraries containing the imported predicates. In general, the only declaration needed is the library module `ytoolkit`, as in the first line of Figure 4.1. Other libraries, namely of the widget classes used by the interface (in this case `button`) are automatically loaded when needed.

Interface management The predicate `go/0` is responsible for creating the interface and handling the interaction. This predicate starts by creating three widgets (a toplevel window containing two buttons) using the predicate `new_widget/3`, whose arguments are:

- **class** The first argument declares the class that controls the widget's graphical layout and reaction to user manipulation. The toplevel widget (the first call to `new_widget/3`) has no class and the two other widgets belong to class button.
- **attributes** The second argument is a set of attribute-value pairs that initialize the widget's state variables. The attribute `parent` implements the interface structure. The attributes `x` and `y` control the widget's position. The attribute `callback` identifies the predicate that is evaluated in response to the user's manipulation.
- **name** The third argument identifies the widget. In this example the toplevel widget (the first call to `new_widget/3`) is called `top` and this name is used to manage the interface structure.

Event processing The goal `loop/0`, the last goal of predicate `go/0`, is responsible for handling the interaction with the user. The toolkit then receives all the events from the windowing system interface and dispatches them to the appropriated widgets. This predicate processes events continuously until the goal `exit_loop` is executed from a callback.

Callbacks definitions: The callbacks are actions executed in response to an user interaction such as pressing a button or selecting a menu entry. Ytoolkit **callbacks** are Prolog goals recorded as a widget's attribute `callback`. By default, callbacks are evaluated in the Prolog module where the widget was created[1]. Consequently, the predicate `great/1`, used as callback by the first button, is also defined in this program. The callback of the second button, the goal `exit_loop`, is the Ytoolkit predicate for terminating the event dispatcher `loop/0`.

After consulting/compiling this program, the evaluation of `go/0` creates a new window named `top` in the workstation screen, controlled by the window manager, with two buttons managed by the Ytoolkit. The application waits indefinitely for the button `quit` to be pressed. Only then will the user be able to type new goals in the Prolog engine toplevel window.

[1] It is possible to override this behaviour by explicitly specifying the class/module where the goal should be executed.

4.3.2 Interface Management

An interface built with the Ytoolkit can be seen as a collection of graphical objects of different classes. Depending on its class, each object has a set of attributes that complement its layout and behaviour. The Ytoolkit thus views the interface as a **database** of graphical objects, suggesting that the predicates necessary to maintain the interface are the four basic operations on a set of data: creation, query, update, and destruction.

- `new_widget(Class?,Attr_List?,Name?)`
  creates a new widget of `Class` with attributes in `Attr_List` and identified by `Name`.
- `current_widget(Class?,Attr_List?,Name?)`
  queries the widgets database and, by backtracking, unifies with all widgets named `Name` from `Class` whose attribute-value pairs match those in `Attr_List`.
- `change_widget(Class?,Attr_List?,Name?)`
  changes the `Class` and attributes in `Attr_List` of the widget identified by `Name`.
- `kill_widget(Class?,Attr_List?,Name?)`
  destroys the widget identified by `Name` and all its sub-widgets. Class and the attributes declared in `Attr_List` can be used to constrain the success of this predicate.

All four predicates have similar argument types: the first argument specifies the class, the second a list of attributes and the third the widget's name.

On widget creation, the specified class definition is searched and dynamically loaded if necessary. The Ytoolkit provides a library of common widgets classes, such as buttons, scrollbars and editable fields. The programmer can implement new classes using Ytoolkit tools, as explained in the next section.

The third and last argument of the widget manipulation predicates identifies the widget. These are uniquely identified by a "name", that must be an atomic value (i.e. atom or number). On creation, if this argument is not bound when `new_widget/3` is evaluated, then a new unused name is automatically generated.

The second argument to all interface manipulation predicates is a list of pairs of attributes and values, each one of the form `Attr=Value`. The order of attributes is immaterial and its is not required that all attributes relevant to a class are specified. In this case the Ytoolkit uses default values for the missing attribute pairs. The Ytoolkit distinguishes different types of attributes:

Windowing system attributes The Ytoolkit supports all window and graphic context attributes provided by Edipo (cf. Chapter 5). These attributes can be used to modify the properties of the widget's window and are automatically passed to the windowing system via the Edipo predicates.

Configuration attributes These attributes record the position, size and border width of the widget's window. Although these attributes are in fact windowing system attributes, they receive a special treatment from the Ytoolkit. The geometry attributes can be set using either numeric values or numeric expressions that may involve other configuration attributes (including those from other widgets). In this last case the Ytoolkit computes their initial value by dereferencing the attribute names and evaluating arithmetic expressions. As explained before, the Ytoolkit then updates the attribute's value each time a dependent attribute is changed, either due to modifications in other widgets or to actions of the window manager.

Class specific attributes Almost all the predefined widget classes have their own specific attributes. For instance, a button has a `text` attribute for its label. These attributes are recorded in the widgets database and the class module can define special action to take place when the values of these attributes are changed or queried.

Most Ytoolkit's GUI use widgets from the standard class set. These classes can be used to assemble an interface by using widget manipulation predicates, or to create new classes by specialization, as described in the next section. The classes of the Ytoolkit standard class set are described in Appendix A.

4.3.3 Managing Interaction

The Ytoolkit widgets created using the manipulation predicates will not react to events until the **event dispatcher** takes control of interaction. This predicate is responsible for receiving events from the windowing system and for processing them. The Ytoolkit considers several types of events and processes them as follows:

Configuration events Events related with changes in the configuration of a widget trigger changes in attributes of the widgets that depend on the changed values.

Visibility events Changes in widget visibility force the automatic redrawing of the widget's layout.

User events The remaining events, mostly those related to the user action with the keyboard and the mouse, are sent as messages to the widget's class.

The Ytoolkit provides two event dispatchers that implement this procedure. These predicates execute continuously until they are terminated by an `exit_loop` evaluated from a callback. The interface programmer may use the following predicates to control the interaction:

- `loop`
  During the execution of this predicate the events are sent as messages to the appropriated widget class. The predicate `exit_loop/0` must be called to stop event redirection. This predicate is deterministic.
- `exit_loop`
  Stops the event processing by the `loop/0` goal. When this goal is executed the `loop/0` finally succeeds. This predicate is normally executed from a callback evaluated in response to user action and it is deterministic.
- `inner_loop`
  The execution of a Prolog predicate can be synchronized with the graphical interaction by calling an event dispatcher and waiting for its termination. In other words, an event dispatcher is launched when another event dispatcher is already processing events. In this case the programmer must use the predicate `inner_loop`. The predicate `loop/0` assumes that no other event dispatcher is running and would erase information needed by other event dispatchers. Like `loop/0`, this predicate is deterministic.

When the Ytoolkit is processing events the only predicates executed are the widget's callbacks. The user is unable to evaluate goals in the toplevel. Consequently, the interface is usually built **before** executing the event dispatcher and passing the control of execution to the Ytoolkit. Afterwards, all executions are initiated by callbacks.

```
process :-
          initial_value(Initial)
          new_widget(modal_dialog,[
                  value=Initial,
                  callback=continuation(ModalDialog)
          ],ModalDialog).

continuation(ModalDialog) :-
          current_widget(_,[value=Final],ModalDialog),
          kill_widget(_,[],ModalDialog),
          process_value(Final).
```

Figure 4.2: Modal Dialog using continuation

This general procedure, the creation of the interface followed by the execution of callbacks, has an exception in the case of **modal dialogs**. Unlike the application's interface, which is visible throughout the life of the application, modal dialogs are created just for receiving input pertinent to a certain mode, and are destroyed afterwards.

The predicate `process/0` in Figure 4.2 creates a modal dialog with an initial value and a continuation goal as callback. When the user has terminated the interaction with the dialog (for instance, by pressing an "Ok" button), the goal `continuation(ModalDialog)` is executed. The name of the modal dialog is passed as argument and is used by the continuation predicate to complete the process, after destroying the dialog.

This approach is convenient when the amount of data passed to the continuation predicate is small. In some cases, it may be preferable to **synchronize** the interaction with the execution of the application's predicate.

```
process :-
        initial_value(Initial),
        new_widget(modal_dialog,[value=Initial],ModalDialog),
        inner_loop,
        current_widget(_,[value=Final],ModalDialog),
        kill_widget(_,[],ModalDialog),
        process_value(Final).
```

Figure 4.3: Modal Dialog using synchronization

The predicate `process/0` in Figure 4.3 uses the `inner_loop/0` event dispatcher to synchronize the execution with the interaction before reading data from the modal dialog. After creating the modal dialog, the goal `inner_loop` is evaluated. This goal processes events continuously and terminates when the goal `exit_loop` is executed in a callback of the modal dialog. When that happens, the goal `inner_loop` succeeds, and the execution of `process` continues.

4.4 Class Definition

As explained before, Ytoolkit classes are implemented using the Prolog module system. Each class is implemented as a module that encapsulates the set of predicates that manage widgets. These predicates, or class methods in the OO terminology, handle the messages received by the widgets of a class.

Widgets created with the Ytoolkit belong to one or more classes, and inherit from these classes default values and the methods that determine their layout and manage their interaction. Both methods and default values are encoded as Prolog predicates that define how

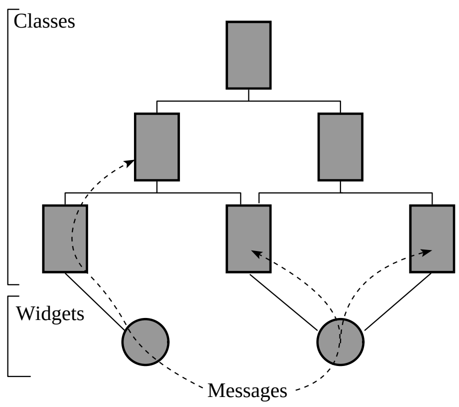

Figure 4.4: Ytoolkit objects and message passing

widgets react to the **messages** they receive during their life cycle.

Messages are the mechanism through which the Ytoolkit informs the widget of any changes in the interface. For instance, user actions that occur over the widget's window such as key pressing and mouse clicking are transformed in messages. The messages sent to a widget are translated by the Ytoolkit into the evaluation of the corresponding predicate of the class to which it belongs.

Code encapsulation and a message passing mechanism were implemented in the Ytoolkit using the visibility declarations of the Prolog module system[2]. Data encapsulation was implemented using Prolog's internal database. All predicates associated with a class are defined in a single Prolog module. Thus, a goal of the form

```
?- Module:Goal.
```

that executes `Goal` according to the predicates defined in `Module` can be seen as passing a message to an object, that is, it can be seen as:

```
?- Object:Message.
```

The Ytoolkit allows multiple inheritance: classes can inherit methods and default values (encoded as Prolog predicates) from several other classes. Inheritance is implemented using clauses such as

```
Message :- Super_Class:Message.
```

[2]When the Ytoolkit was implemented there was not a "standard" module system in Prolog. The Ytoolkit was developed using the Quintus model on which the ISO standard was defined.

that propagates `Message` to `Super_Class`

The Ytoolkit uses only a limited number of methods in the widgets life cycle. These methods are described in Subsection 4.4.1. The class methods interface the widget's state using the predicates described in Subsection 3.2.

The Ytoolkit supplies a set of tools to be used in the implementation of class methods. Subsection 4.4.3 describes the drawing tools for handling widget redrawing and Subsection 4.4.4 describes the tools for handling a set of pre-defined behaviours. This section ends with two examples of new widget classes created using these tools.

4.4.1 Class Methods

The Ytoolkit assumes that a set of predefined methods exists for every widget class. These messages are passed in key points of a widget's life cycle, and always receive an argument that identifies the class instance. This identifier can be used to access the widget's state using the predicates presented in Subsection 4.4.2, and can be compared to the **self** construct of some object oriented languages. The predicates defining messages are:

- `layout/1` is sent during widget creation to define the widget layout. This predicate is expected to use the `draw/3` predicate when calling the Edipo's graphic primitives (cf. Chapter 5) that perform the actual drawing. The predicate `draw/3` records the primitives used in the layout thus allowing the Ytoolkit to redraw widgets automatically when they are exposed or resized.
- `behavior/2` handles events generated during interaction. The arguments to this message are the widget's name and the event to be handled. Event processing can be redirected to the Ytoolkit by executing a predefined behaviour predicate from the Ytoolkit library.
- `sub_widgets/1` is also sent at widget creation time to initiate sub-widgets that implement a different layout and/or behaviour on parts of the widget. The sub-widgets are created using the `new_widget/3` predicate described above.
- `attr_change/3` is sent when the value of an attribute is changed using the predicate `change_widget/3`. For instance, when the text of a widget of class 'button' is changed it should redraw this text on the widget's window. The arguments it receives are the widget's name, the changed attribute and the new value.
- `current_attr/3` is sent when the value of an attribute is queried through the predicate `current_widget/3`. This can be useful if an attribute has to be (re)computed.

Class definition contains also attribute initializations and callback predicates. The initial value of a class attribute is recorded in a clause of the predicate `default/2`. The first attribute of this predicate is the attribute name and the second is the default value. The callback definitions are predicates evaluated in response to an user action (e.g. pressing the button). Ytoolkit **callbacks** are Prolog goals recorded as a widget's attribute (typically named `callback`). By default, callbacks are evaluated in the Prolog module where the widget was created but it is possible to override this action by explicitly specifying the class/module where it should be executed.

4.4.2 Widget's State

The interface programmers are encouraged to access the widget attributes using either the `current_widget/3` or the `change_widget/3` predicates. These predicates use the class methods `current_attr/3` and `change_attr/3`, if defined in the class module. These methods may be needed in the cases where the attribute values are computed or queried from the windowing system, or when changing the attribute should trigger a side effect.

However, there are situations where the class programmer should not use the toplevel predicates to access the attributes. For instance, to define the methods `current_attr/3` and `change_attr/3` the class programmer must access, at low-level, the attributes for which the methods are being defined, otherwise the method itself will lead to an infinite loop.

A widget identifier that is passed to the class methods and can be used to access the widgets attribute database. This is made either by the widget manipulation predicates or by using the lower-level predicates.

In the management of the widget from a class, local attributes have to be manipulated just as plain variables: setting and getting their values with nothing else implied. The predicates to manipulate a widget's attributes database at the lower level are:

- `get_attr(N,A,V)`
  Gets the attributes currently in use by a widget object. `N` stands for the widget name, `A` for the attribute name and `V` for the attribute value. This predicate is backtrackable.
- `set_attr(N,A,V)`
  Sets an attribute for a widget object. `N` stands for the widget name, `A` for the attribute name and `V` for the attribute value. This predicate is deterministic.
- `unset_attr(N,A,V)`
  Discards an attribute previously used by a widget object. `N` stands for the widget name, `A` for the attribute name and `V` for the attribute value. This predicate is deterministic.

As explained in the previous section, the attribute values are initialized at widget creation time using the predicate `default/2` presented at the class module. Each clause for this predicate must define an attribute name as the first argument, and its initial value as the second argument. The Ytoolkit manages the database keys avoiding conflicts and clearing the widget's database on widget termination.

Note that attribute values can be complex terms. With this feature attributes can be used as associative arrays to record data using atoms as indexes. For instance, the attribute callback of class button can be defined as `button(Number)` to index the callback on the button number (although the terms used as "indices" do not need to be numbers).

4.4.3 Drawing Tools

The drawing tools are used by widget classes to draw graphics on the widget's window. These tools receive a drawing predicate as argument (usually an Edipo primitive) that performs the actual drawing. The coordinates and dimensions of the Edipo's drawing primitives are numeric values. The Ytoolkit's drawing tools accept also numeric expressions. These expressions may include references to configuration attributes of the same widget or of other widgets. In this last case the expression is used as a sort of **constraint** that is reevaluated when one of the dependent attributes change. For instance, to draw a circle of radius 5 in the middle of the widget's window one can use the following goal:

```
draw(Widget,circle(width/2,height/2,5))
```

This circle is automatically redrawn when the dimensions of `Widget` are modified. The expressions for computing the arguments of drawing predicates are summarized in table 4.1. These expressions may include configuration attributes (such as `x` or `width`) either of the widget being drawn, or of other widgets. In this last case the attribute has to be preceded by the widget's name connected using the Ytoolkit tilde operator (~) (e.g. `Widget~Name`).

Type of argument	**Example**
constant values	`123`, `"hello world"`, 'Ok'
configuration attributes	`width`, `border`
other widget configuration attributes	`top~width`
expressions involving the latter	`(top~width-width)/2`

Table 4.1: Attribute expressions of drawing predicates

The set of Ytoolkit's drawing tools consists of the following predicates:

- `clear_widget(Widget)`
  Clears the widgets window and removes all information associated with redrawing and reshaping. This predicate is deterministic.
- `draw(Widget,DrawingPredicate)`
  Evaluates the Drawing predicate – usually an Edipo predicate – using `Widget` as the working window. This predicate is deterministic.
- `draw(Widget,[List_Of_Context_Attr],DrawingPredicate)`
  Like the previous predicate but specifying graphical attributes (cf. Appendix C). This predicate is deterministic.

4.4.4 Behaviour Tools

The **behaviour** is the event processing part of a widget class. Each time the window associated with a widget receives an event, the Ytoolkit redirects the event to a predicate named `behavior/2`. The first argument sent to this predicate is the name of the event received; the second is the name of the corresponding window.

The Ytoolkit defines a set of predefined behaviours, which take as arguments the event to be processed and the name of the window receiving the event. Note that before using the predefined behaviours, the module with its name must be loaded. The predefined behaviours are:

- `pressable(Event,Window)`
  To define a behaviour that is activated by mouse clicks. When a mouse button is pressed on `Window`, the predicate specified by `callback` is called. The attribute `text` is also used to specify the atom to display in the button.
- `editable(Event,Window)`
  To perform an interactive edition of an input field. The attribute `chars` is used to interchange the default string and the string typed.
- `movable(Event,Window)`
  To allow the corresponding window to be moved through mouse dragging.
- `resizable(Event,Window)`
  To support changes of the size of the corresponding window, via mouse dragging on one of its edges.

The keyboard related events (`keyPressed`, `keyReleased`) can be redirected to a particular widget; this is called **input focus**. This feature is only relevant to the behaviour `editable` but since it really affects events it can be given other uses. The predicates that control input focus are:

- `focus(Widget)`
  To focus keyboard input on `Widget`. A `leaveNotify` event is sent to the previously focused widget and an `enterNotify` event is sent to the focused widget
- `defocus`
  To turn off the focus.
- `current_focus(Widget)`
  To query the current focused `Widget`.

The creation of new behaviours can cause the need to redirect other events in a different form. In those situations the predicate `send_event(Event,Widget)` should be used to make the redirection (cf. Appendix D).

4.4.5 Class Examples

This section presents a couple of examples illustrating the use of the Ytoolkit standard class design tools. The example defines a well known class of widgets: the `button`. The second is a specialization of the standard class `field`.

4.4.5.1 Using class design tools

The widget class 'button' belongs to the standard widgets library. The Figure 4.5 presents a simplified version of that widget class. The definition of this class uses a pre-defined behaviour and drawing tools to create the widget's layout.

The file starts with several declarations. The class name (`button`) is used as the module's name and the libraries `ytoolkit` and `edipo` are imported to this module. The library `ytoolkit` is required since this class uses the drawing tool `draw/2`. The library `edipo` is required since this class uses several edipo predicates for querying the available fonts (`current_font/2`) and for computing text dimensions (`writebox/3`). This modules loads also the library `behaviors/pressable` containing the behaviour tool `pressable/2`.

The first predicate defined in this module, `default/2`, defines default values of the relevant attributes of this class. It should be noted that these values may be overriden by the

```
:- module(button,[]).

:- use_module(library(ytoolkit)).
:- use_module(library(edipo)).
:- use_module(library('behaviors/pressable')).

default(width,80).        default(height,25).
default(border,1).        default(text,text).
default(callback,true).   default(line_width,2).
default(join,bevel).      default(pad,2).
default(font,F) :-
        current_font(['9x15bold','*bold*'],F).

layout(Name) :-
        get_attr(Name,pad,S), S2 is S<<1,
        draw(Name,rectangle(S,S,width-S2,height-S2)),
        get_attr(Name,text,Text).
        writebox(Text,W,H),
        draw(Name,writeterm((width-W)/2,(height-H)/2+H,Text)).

behavior(Event,Name) :- pressable(Event,Name).

change_attr(text,Name,_) :- clear_widget(Name), layout(Name).
change_attr(pad,Name,_)  :- clear_widget(Name), layout(Name).
```

Figure 4.5: Definition of class 'button'

attributions in the manipulation predicates. Each clause defines an attribute default value. The last clause initializes the attribute `font` with the font name returned by the Edipo's predicate `current_font` which selects either the font `9x15bold` or any bold font if this one does not exist.

The method `layout/1` draws the button's text centered on the widget's window, and an inner frame using a rectangle. The position of both the rectangle and the text are constrained to the dimensions of the button's window. Consequently, this layout is prepared for any values of the widget's width and height and can be correctly redrawn on resized. The widget's layout depends also on the class attribute `pad` that controls the amount of space between the widget's border and the internal frame.

This widget has a frequent type of behaviour, consequently it uses a pre-defined tool to handle the user generated events. The method `behavior/2` simply forwards the events to the predicate `pressable/2` loaded from the library `behaviors/pressable`.

The last predicate, `change_attr/3`, is invoked when either one of the class specific attributes `text` or `pad` is changed. In either case the widget is cleared and the layout redrawn.

4.4.5.2 Specializing a widget class

The example in figure 4.6 is a specialization of the standard field class. The goal is to add a field label to the left of the input field and implement an input order among a sequence of fields, using the TAB key to focus the input in the input field. This class creates also a label to identify the field. The label is adjusted on the left side of the widget using constraints.

```
:- module(my_field,[]).

:- ensure_loaded(library(ytoolkit)).
:- use_module(library(edipo)).
:- ytoolkit:super_class(field).

default(text,field).
default(next,0).

sub_widgets(Name) :-
        get_attr(Name,text,Text),
        writebox(Text,L,_),
        Len is L+2,
        current_widget(_,[parent=Parent],Name),
        new_widget(label,[
                parent=Parent,
                x=Name~x-Len,      y=Name~y,
                width=Len,         height=Name~height,
                border=1,
                text=Text
        ],_).

behavior(enterNotify,Name) :- focus(Name).
behavior(keyPress([9],_,_,_),Name) :-       % 9 is ASCII code for Tab
        get_attr(Name,next,N),
        focus(N).
```

Figure 4.6: Specialization of the class `field`

This class inherits the methods defined in class `field`. For that purpose this module includes the declaration `super_class` with the parent class as argument. As in the previous example, this module loads the library `ytoolkit` since it uses the Ytoolkit's widget manipulation predicates `new_widget/3`, `current_widget/3`, and `change_widget/3`.

Two new attributes are introduced: `text` and `next` respectively for the field label and the name of the "next" input field. The default values for these attributes are defined in the clauses of predicate `default/2`.

The new attribute `text` is used as a label that is placed on the field's left side. The label is created using a new widget of class `label` whose size is big enough to contain the label. The size is computed using the Edipo's predicate `writebox/3`. The label is placed in the parent widget and its position is constrained to the widget's position: the vertical coordinate is the same (`y=Name~y`) and the horizontal coordinate adjusts the label in the left of the widget (`width=Len, x=Name~x-Len`). Consequently, if the widget is moved then the label is moved to maintain the same relative position.

This class extends the standard behaviour of field (text edition) and redefines the predicate when the `Tab` key is pressed. In this case, the Ytoolkit focuses the events on the widget whose name is recorded in the attribute `next`. The event can also be focused on a field of this class by moving the mouse over the field, which generated the event `enterNotify`.

Chapter 5

Edipo: a Prolog Interface to Windowing Systems

Edipo[1] is a software tool developed to provide a user-friendly interface to windowing systems such as X-Windows. It consists of a library of predicates for managing windows and graphic contexts, for drawing text and graphics on windows, and for receiving user or system events. Edipo interfaces with the windowing system's core library and provides a set of basic features that make this package more user-friendly than other existing interface libraries such as XWIP [Kim90]. The current implementation of Edipo interfaces Xlib, the C library of the X-Windows protocol, but it could be ported to other windowing systems, such as Windows or MacOS.

One problem with windowing systems operations is that they can have a very large number of parameters, most of which users want to keep as default values. Edipo is designed so that the operations that require the specification of attributes use attribute-value lists instead of fixed data structures. This approach avoids filling in a huge number of attribute slots in fixed data structures whose values are usually set using default values.

Another problem with most windowing system's functions is that they do not generate an immediate side-effect on screen. Hence, Edipo automatically executes several operations whenever they are needed to produce a graphical output. For instance, the Xlib function for creating windows do not present the window on the screen until the window is mapped, and the drawing functions have no effect until the pending outputs are explicitly flushed to the display. This kind of operation can be rather confusing for novice users so Edipo generates an immediate side-effect by automatically calling the functions that produce the output. Note that this feature does not compromise the ability to control the windowing

[1]Edipo stands for 'Enjoy Drawing In PrOlog'.

system function that produce output. Edipo also provides predicates for switching off the automatic operations and controling explicitly these windowing system features.

5.1 Windows

A **window** is a rectangular portion of the screen for outputing drawings and receiving events. Therefore, windows must be created before using other Edipo's predicates, such as drawing or event predicates. Associated with each window are **graphic contexts**. They control certain aspects of drawing, such as the width of lines or the font of text. The creation of a graphic context is not mandatory since a default graphic context always exists for any Edipo window.

5.1.1 Basic Predicates

The basic Edipo predicates are responsible for creating new windows, for changing their attributes, and for querying the existing windows and their attributes. Windowing system attributes are set and read using a list of the form `[`*`attribute=value,...`*`]` where *`value`* can either be a Prolog ground term to be passed as data, or a variable to be unified with the queried value, depending on the predicates.

- `window(W,L)`
  The window `W` becomes the current **working window**, and if it does not exist already then it is created. All subsequent graphic and text primitives will operate on this window until another one is set using this predicate. Attributes specified in the list `L` are used for updating window `W`. The list `L` must be of the form `[`*`attribute=value,...`*`]`, non specified attributes will be set to default values on window creation and to the preexisting values on window update. This predicate is deterministic. For a complete list of `window/2` attributes and their valid values refer to Appendix B.

  The attribute `parent` makes it possible to create a hierarchy of windows. A window with a parent, or **sub-window** is displayed inside the parent's window that clips all graphics drawn on the sub-window. The coordinate referential of a sub-window is the upper left corner of its parent window.

- `current_window(W,L)`
  `W` is unified with the name of an existing window, starting with the current working window, and `L` with their respective attributes. This predicate is backtrackable; thus, for instance, the goal `current_window(Id,[parent=top,width=W])` retrieves

the widths of all children of window `top`. See Appendix B, for a complete list of `current_window/2` attributes and their valid values.

5.1.2 Window Management

The predicates described in this sub-section give a higher control over windows and, in some cases, they perform changes involving the **window manager** (e.g. window stacking order). It should be noted that the window managers have priority over top-level windows management, hence the predicates described in this subsection may have different outcomes depending on a particular window manager. Note that all window management predicates are deterministic.

- `clear`
  Erases all the contents of the current working window using the background pixel or pixmap (Cf. Appendix C for colour values). Windows are automatically cleared when created using the default background pixel (white).
- `destroy`
  Destroys the current working window, or fails if none exists.
- `raise`
  Displays the current working window at the top level of the windows stack.
- `lower`
  Displays the current working window on the bottom level of the windows stack.
- `unmap`
  Makes the current working window disappear from the screen without actually destroying the window. If the working window is a top-level window then the window manager will probably create an icon to represent it. This predicate succeeds even when the current working window is already unmapped.
- `map`
  Makes the current working window reappear on the screen, if it has been previously unmapped, and does nothing if it is already mapped.
- `create_unmaped`
  Switches off the automatic mapping of windows created with `window/2`. In this case `map/0` has to be used for windows to appear on screen.
- `create_maped`
  Switches on the automatic mapping of windows created with `window/2`, which is the default procedure.

5.1.3 Graphic Contexts

A **graphic context** (GC) keeps a set of parameters that control graphic operations. A new GC is created automatically for each new window using the default values so that drawing primitives can be used immediately after creating a window. A window's default GC has the same name as the window.

Graphic contexts work exclusively on the window in which they were created. One can create GCs with the same name and different attributes but for different windows.

- `gc(N,L)`
  Creates the graphic context `N`, if it does not exist, and modifies it otherwise. In either case the GC `N` becomes the current **working graphic context**. The argument `N` must be an atomic values and the list `L` must be of the form `[`*`attribute`*`=`*`value`*`,...]`. Default values are used for all not specified attributes. Usage of non valid attributes will cause the predicate to fail. If L is `[]` then the graphic context `N` is just set to the working window without affecting the GC's attribute values. This predicate is deterministic. See Appendix C, for a complete list of `gc/2` attributes and their valid values.

- `dup_gc(N)`
  Creates a new GC with name `N` whose attributes take the values set on the current working graphic context. This predicate is deterministic.

5.2 Drawing

The Edipo's drawing primitives display simple shapes such as lines, rectangles, polygons and circles, on windows. Shapes are described using a bi-dimensional coordinate system whose origin is the window's upper left corner, with the positive x-axis pointing to the right and the positive y-axis pointing down, as is usual in windowing systems. The numeric arguments corresponding to coordinates should be positive integers. Note that Edipo always converts floats to integers before being used by Xlib functions.

As mentioned before, all drawing predicates are executed in the current working window, using the current working graphic context. It should be noted that all drawing predicates are deterministic and all coordinates must be supplied.

5.2.1 Drawing Figures

- `line(+X1,+Y1,+X2,+Y2)`
  Draws a line between the points `(X1,Y1)` and `(X2,Y2)` using the foreground colour. See Appendix C for the valid colour values of the GC attribute `foreground`. In Edipo there is no predicate specifically designed for drawing a pixel. The most efficient way to draw a pixel is drawing a line starting and ending at the same point (`line(X,Y,X,Y)`).

- `rectangle(+X,+Y,+W,+H)`
  Draws a rectangle with the upper left edge at the point `(X,Y)`, with width `W` and height `H`, using the foreground colour.

- `frectangle(+X,+Y,+W,+H)`
  Behaves like `rectangle/4` except that the rectangle is filled with the foreground colour. This primitive is influenced by the GC attributes `foreground` and `fill_style`. See Appendix C for valid attribute values.

- `crectangle(+X,+Y,+W,+H)`
  Analogous to `frectangle/4` but the rectangle drawn is filled with the background colour. See Appendix C for valid colour values for attribute `background`.

- `poly(+L)`
  Draws a polygon with edges given by list `L`, in the form `[(X1,Y1),...,(Xn,Yn)]`, where `(X1,Y1)` is the first edge, `(X2,Y2)` the second, up to the edge `n`.

- `fpoly(+L)`
  Behaves like `poly/1`, except that a line is drawn between the first and the last endpoint and all the closed areas are filled with the foreground colour. See Appendix C, for a table of `fill_style` modes.

- `circle(+X,+Y,+R)`
  Draws a circle with the center point at `(X,Y)` and with radius `R` using the foreground colour.

- `fcircle(+X,+Y,+R)`
  Draws a filled circle with the center point at `(X,Y)` and with radius `R`. The circle is filled with the foreground colour. Cf. Appendix C, `fill_style` for a table of fill modes.

- `arc(+X,+Y,+W,+H,+S,+E)`
  Draws the arc inscribed in `rectangle(X,Y,W,H)` starting at the angle `S` and ending at the angle `E`. Angles are measured in sixty-fourths of a degree. An arc of 360 degrees can be drawn starting at the angle 0 and ending at angle 23040 (23040=360*64). Behavior is undefined when one of the angles is outside the range [0-23040].

- `farc(+X,+Y,+W,+H,+S,+E)`
  Draws the `arc(+X,+Y,+W,+H,+S,+E)` and fills its interior. The arc can be closed by a chord or by a 'pie-slice', through appropriate setting of the `arc_mode` attribute. See Appendix C, for a table of fill modes for attribute `fill_style`.

5.2.2 Drawing Characters

Edipo provides a set of predicates for displaying characters in the current working window using either strings (lists of ASCII codes) or terms. This set includes also predicates for computing the bounding boxes of strings and terms to be displayed in the current working window.

- `writeterm(+X,+Y,+T)`
  Writes the term `T` at the position `(X,Y)`. This predicate is deterministic and depends on the `gc/2` attribute `font`.

- `writestring(+X,+Y,+L)`
  Writes the string `L` at the position (X, Y). This predicate is deterministic and depends on the `gc/2` attribute `font`.

- `writebox(+A,W,H)`
  Instantiates `W` and `H` with the width and the height of the term `A` respectively. This predicate is deterministic and depends on the `gc/2` attribute `font`.

- `stringbox(+L,W,H)`
  Instantiates `W` and `H` with the width and the height of the string `L` respectively. This predicate is deterministic and depends on the `gc/2` attribute `font`.

- `current_font(F)`
  Unifies `F` with the name of a font available on the X-Windows system. This predicate is backtrackable.

- `font_list(+P,F)`
  Unifies `F` with the name of a font available on the X-Windows system whose name matches the pattern `P`. This predicate is backtrackable.

5.2.3 Bitmaps

A bitmap is a simple two colour raster image. In a bitmap each pixel displays either the foreground or the background colour of the current GC. The easiest way to create a bitmap is through the X-Windows editor utility *bitmap*. Edipo's bitmap predicates are deterministic.

- `getimage(+I,+W)`
  Gets a bitmap from a file or window. The argument `I` must be an atom specifying the bitmaps name and the argument `W` can be one of the following terms:
  - `from_file(`*`file_name`*`)`
    To read the image from a X bitmap file.
  - `from_window(`*`x,y,width,height`*`)`
    To read the image from a rectangular area of a window.
- `putimage(+I,+W)`

  Writes a bitmap to a file or to a window. `I` must be an atom specifying the name of the bitmap. `W` can be one of the following terms:
  - `to_file(`*`file_name,width,height`*`)`
    To write the bitmap to a file in the X bitmap format.
  - `to_window(`*`bmx,bmy,x,y,width,height`*`)`
    To write the bitmap to a window. The arguments `bmx` and `bmy` specify the bitmap's upper left edge coordinates. The behaviour of the overlapping pixels is set by the GC attribute `function`. For a complete list of valid values for the attribute `function` see Appendix C.

5.2.4 Advanced Features

By default, all drawing predicates draw a shape on the current working window as a side effect of being executed. In other words, they are automatically **flushed** to the current window. For a program producing a large image with several drawing primitives this feature may be inconvenient. In this case, it may be preferable to display only the final drawing, not the intermediate steps, as displaying the intermediate stages slows down image construction. The predicates described below are used to control flushing and they all are deterministic.

- `dont_flush`
  Stops auto-flushing for drawing predicates. All modifications produced after the execution of this predicate are only displayed when either `do_flush/0` or `flush/0` is evaluated. Note that when auto-flushing is off the graphic predicates performance is considerably better. Therefore, `dont_flush/0` should be used whenever as possible.
- `flush`
  Flushes the pending modifications, but does not restart auto-flushing.

- `do_flush`
  Flushes the pending modifications, and restarts auto-flushing for drawing predicates.

- `color`
  Succeeds when the X-server supports colours.

5.3 Events

Events are messages sent by the windowing system (e.g. X-Windows) to notify of events such as key pressing, mouse motion, button pressing/releasing and window status modification (creation, exposition, resize, reposition, destruction, ...). See Appendix D for a complete list of the possible events. Events are read from a first-in first-out queue, in the order in which they are generated.

- `event(W,E)`
  Unifies `E` with the first event present on the event queue, and `W` with the name of the window where it occurred. The event is then removed from the queue. If the queue is empty then the execution blocks, waiting for the next event occurrence. This predicate is deterministic.

- `events(W,E)`
  Unifies `E` with the first event present on the event queue, and `W` with the name of the window where it occurred. Fails if the event queue is empty. This predicate is identical to `event/2` except that it fails if no events are present on the event queue, instead of blocking the execution. This predicate is backtrackable.

```
process_events :-
        repeat,
                event(W,E),
                window(W,[]),
        process(E).

process(buttonPress(1,X,Y))  :- circle(X,Y,3), fail.
process(keyPress("Q",_,_,_)) :- destroy.
```

Figure 5.1: Event loop

Event reading is usually performed within a repeat-fail loop that ends when a terminating goal succeeds. Figure 5.1 presents a repeat-fail loop that reads events and processes them until a particular event occurs . Note that the loop simply fails for a large number of events (all except `process(keyPress("Q",_,_,_))`). When the left mouse button is pressed within a window, a three pixel radius circle is drawn at the mouse position. When a "`Q`" key is pressed in that window, the window is destroyed, the predicate `process/1` succeeds, and the repeat-fail loop ends. In essence, the predicate `process_events/0` in Figure 5.1 is similar to the Ytoolkit predicate `loop/0` that dispatches events to the widgets.

```
event_flush :- events(_,_), fail.
event_flush.
```

Figure 5.2: Event flush

The backtrackable version of event reading is useful in situations where all pending events must be read since it does not block when no events are in the queue. The predicate `event_flush` on Figure 5.2 simply reads and discards all events in the queue.

Chapter 6

Discussion and Further Work

In the early days of computing the effort of producing programs was negligible when compared with the price of hardware. In those days programming was a time-consuming activity executed only by experts. Nowadays the situation is the reverse: computers are inexpensive and programming is a common, although not trivial, task. Given this situation, several programming environments start integrating Graphical User Interfaces (GUI) to replace command line interpreters. These environments are very helpful in tasks that require extensive knowledge of a library or API (for instance, in programming GUI). Nevertheless, they do not take full advantage of graphical interaction since they use text-based tools such as program editors and debuggers. To use the graphical capabilities of a modern workstation to enhance programming has also been the goal of an active research area in computer science. Two important approaches to this goal are language environments, on one side, and visual programming, on the other.

In general, language environments use an existing programming language and produce structure editors from the language's context-free grammar. These structure can use graphical visualization to display the program structure. Moreover, they avoid syntactic mistakes since they skip the parsing of program's text code and construct interactively the program's abstract parse tree. Unfortunately, this approach makes difficult the edition of programs, in particular if an intermediate editing stage is syntactically incorrect.

On the other hand, most visual programming languages give total editing freedom to programmers, which in some cases can actually "draw" a program using graphical primitives. In general the programmer assembles a graphical visualization of the program using icons and lines as building blocks. Nevertheless, these languages are perhaps too different from the traditional programming languages and have some difficulty in gaining popularity among programmers.

An alternative approach, explored in this thesis, is the development of a programming environment for an existing language using an interaction framework for interacting with programming objects (program code, data, executions, GUI) that profits from the graphical primitives and pointing device manipulation.

The interaction framework of traditional programming environment is based on **streams of characters**. Users produce text files, using text editors, that are read as a stream of characters by compilers and interpreters. Data IO also uses streams of characters: programs read streams of data that the operating system associates with files, network connections or keyboards; and data is also written in streams of characters that the operating system directs to files, network connections, printers or displays. The stream of characters model is very simple, is well supported by the operating system and enables the integration of independent tools.

The advent of graphical workstations made available new input/output devices for the development of applications with a friendlier GUI. After an initial period when each application had its own look and feel, there is nowadays a good level of uniformity among the GUIs of most of the windowing systems. This uniformity was achieved through the use of graphical toolkits that simplify the creation of GUIs and improve their quality with the use of a common set of graphical objects. Unfortunately, this standardization was not achieved on graphical interaction in the context of program development. Experience has shown that it is difficult to replace the stream of characters framework with a new interaction framework enabling formatted data interchange. Ideally, such a framework would be common to all kinds of objects manipulated in a programming environment, such as programs, executions, data and GUI definitions.

Defining a common interaction framework, generic for all programming environments, is perhaps an overly ambitious goal. But this approach is sensible for a restricted domain. Such is the case of LP where data and programs already share a common syntax and have a strong emphasis on structure.

In this thesis it is proposed a common interaction framework as the basis for creating interactive graphical environments for the development of LP applications. The proposed framework retains the advantages of traditional text based frameworks – the basis of programming environments in the previous decades – and introduces non-textual features to improve the perception of complex programs and data.

6.1 The Interaction Framework

The novel interaction framework proposed in this thesis uses **term edition** as a model for all types of interaction occurring in a LP environment, ranging from data and program editing to interaction with program executions and algorithm animation, and including graphical debugging and GUI definition. The choice of the term editing model relies on the fact that most of these activities deal with terms or with concepts that can be represented as terms. Any modification of the layout reflects the changes in this set of terms and vice-versa. Consequently, in Magritte interaction and term editing are synonyms.

The key advantage of the term editing model is the decoupling of the semantics and syntax of the interface. This feature simplifies both the interactive edition of terms and the implementation of applications and tools that modify the interface. Since the modification of the interface's state is independent from its visualization, both the end-user and the programmer can update the interface state without bothering to define a new visualization for the changed terms. The converse is also possible. New visualizations can be created for any part of the interface, either interactively or from program execution, without changing the information presented in the workspace.

In this decoupled view of interaction, the set of terms subject to user edition, either interactively or from programs using the interaction engine's API, can be regarded as the semantics of the interface. The interface's layout is the composition of the visualizations for each of these terms.

6.2 The Prototype

Magritte was implemented as a prototype of an environment using the term edition framework. The core of Magritte is the TermsIO package that serves as interaction engine. In other words, all tools and applications embedded in the environment use TermsIO to input and output terms.

The common interaction engine is the basis of Magritte's spreadsheet-like user interface. Spreadsheets have been very successful in numerical environments because they concentrate in a single window a collection of data that can be interrelated and manipulated through a wide range of tools. Before the popularization of spreadsheet environments each application had its own interaction mechanisms with its own windows and data was exchanged using files or clipboards.

The potential of the spreadsheet interface had never before been used in environments that manipulate structured data such as LP terms. In fact the bi-dimensional array of cells, which is the basis of the traditional spreadsheets interface, is not appropriate for structured data. To address this problem in Magritte the matrix is replaced by a hierarchy. The branches of the hierarchy can be identified with Prolog terms and the structure editing procedures of the interaction engine can be used to edit both the data terms and the interface structure.

The hierarchical structure is reflected in the TermsIO's internal data state. This state can be changed using the structure editing commands of TermsIO's API. The use of structure editing for term manipulation has two main advantages, in particular if compared with text editing:

Syntactic correctness The selection mechanism is oriented towards terms, instead of the blocks of contiguous characters used in the text-based framework. Hence, the structure editing commands operate on a complete term. Moreover, the structure editing command replace terms by other term thus assuring the syntactic correctness of the terms under interaction.

Orthogonality Structure editing can be used both for interactive editing, and for automatic editing performed by Prolog predicates. In general, commands for manipulating blocks of contiguous characters are too cumbersome for data manipulation which requires parsing text into structured data. Moreover, the structure editing commands can be decoupled from the term visualization commands, and thus they are independent from any particular graphical (or textual) syntax.

Providing a language to describe the graphical syntax of terms is a key feature in TermsIO. The use of styles to describe the visual representation of terms has several advantages that can be summarized as follows:

Expressive Styles control directly the visualization attributes of every single functor represented in the interface. Hence, any graphical interface that can be represented by a Prolog term can be implemented using styles.

Simple Styles are assigned to a functor using a Prolog-like declaration or Magritte's GUI. The interaction engine supports term rewriting on style definitions for writing higher-lever descriptions of styles, thus simplifying the definition of attributions.

Extensible The interaction engine defines default visualizations (in general the traditional textual representation), but tools and applications are able to extend the visualization of special functors using graphical primitives such as lines, circles and bitmaps.

Declarative The visualization of any type of terms is declared in high-level definitions and is independent of the predicates that actually constructed the terms.

Sharable Style definitions are loaded directly into the interaction engine and immediately available to all tools and applications developed within the environment.

6.3 Prolog as an Implementation Language

The system that served as the test-bed for the proposed interaction framework is completely written in Prolog, which is not a language commonly used in the implementation of graphical environments. In the beginning of this research, most graphical interfaces were implemented in system languages such as C. Gradually more parts of graphical interfaces were written in less efficient languages but supporting a rapid prototyping and simplifying program maintenance. Prolog is usually acknowledged as a good language for rapid prototyping, with an emphasis on the management of data structures, but is regarded as inefficient and not suited for system implementation. The experience with Magritte clearly supports the idea that Prolog is a good language for rapid prototyping, but contradicts the idea of lack of efficiency.

Edipo is the package responsible for binding Prolog to the X Windowing system through Xlib [GNS87], the library of X-Windows's C Language Interface Protocol. Edipo converts data between Prolog and C structures, maintains context information (such as working windows and graphic contexts), manages resources (such as font information) and interfaces with Xlib functions. In its first implementation Edipo was written in C and used Prolog's foreign language interface to define its predicates. This code was very difficult to maintain and was completely rewritten in Prolog, using Prolog's foreign language interface to connect directly to the Xlib functions. Only very few functions had to be defined in C, mostly to transparently associate constants defined in include files to Prolog atoms. Switching from C to Prolog simplified the **maintenance** of Edipo and had little impact on the efficiency of the package.

In the last few years it was shown that the GUI definition does not require too much computational power and can be efficiently executed by scripting languages such as `tcl` or `Perl` [Ous94, WCS96], interfacing the `tk` GUI toolkit [Ous94]. Nevertheless, these scripting languages cannot be used in the definition of new classes of widgets, which still have to be written in C. Prolog is a language designed for rapid prototyping, hence it is not surprising that it is a good language for GUI definition. Magritte showed that Prolog can be used not only to glue the interface together, but also to implement the toolkit itself, as well as the graphical objects. The Ytoolkit's structure is based on Prolog's module system. Thus the creation of a new widget class corresponds to the definition of new Prolog modules, containing a set of predicates implementing the class methods. Hence, the Ytoolkit extends the rapid prototyping of GUI definition to the implementation of new classes, allowing the definition of new and unconventional interfaces in Prolog applications.

The heart of Magritte is the interaction engine that implements the term editing interaction framework. The TermsIO package performs several computations requiring more efficiency than the GUI management. TermsIO relies on its attribute evaluators to constantly compute and update the attribute values using predicates compiled from style definitions. The algorithms used in TermsIO are written in a few lines of Prolog code, simplifying the maintenance of the package. Fast prototyping was also a beneficial Prolog feature in the development and tuning of new algorithms for attribute evaluation and compilation of styles to Prolog predicates. The attribute evaluations must be performed fast enough to update the interface after each user's event (for instance, after each key stroke), before the arrival of another event, in order to produce a smooth interaction. This level of efficiency was achieved with the Prolog engines used in the development of Magritte.

The implementation of Magritte's front-end also benefited from being written in Prolog, especially through the use of **meta-programming**. To take full advantage of the term editing framework the Magritte's front-end uses meta-programs to create specialized structure editors and to interface with the underlying Prolog engine.

6.4 Future Work

In Magritte's current implementation, a single Prolog process handles every component of the system, from the graphical interaction of Prolog terms to the management of windows. The tight coupling of the operational layer – the Prolog engine – and the interface layers – the Xlib and the graphical environment – is typical of the applications developed for windowing systems, implemented in system development languages such as C or C++. Most of these applications create and manage their interfaces using graphical toolkits such as the XToolkit [MA] for the X11 windowing system [GNS89], used as the basis for the implementation of the look and feel of Motif and OpenLook. The relationship between the application, the toolkit and the underlying windowing system can be represented by Figure 6.1.

Recently, there has been a trend to decouple the application's interface and operational layers. These two layers execute as autonomous processes that communicate via the operating system. There are several implementations of this approach, some even in the Prolog field like the SICStus Prolog's GM [C+]. These systems implement a looser communication channel between application and interface layer as depicted in Figure 6.2.

Arguably, one of the most successful packages using this approach is the `tk` [Ous94] toolkit,

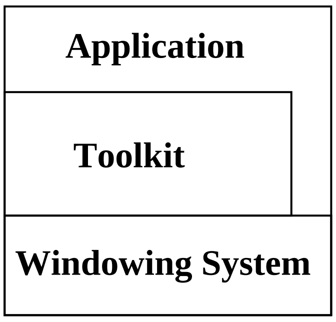

Figure 6.1: Tightly coupled interfaces

that can be used from several scripting languages such as `tcl` [Ous94] or `perl` [WCS96]. The toolkit and these scripting languages allow the programmer to implement complex applications using small building blocks, where the interface is one of these components. This kind of programming has recovered some of the original Unix philosophy of dividing large applications in smaller components glued together via scripting languages; this programming philosophy was abandoned in the programs using tightly coupled GUI that required the linkage of all components to produce a single program.

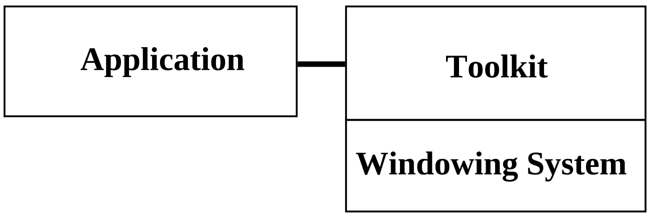

Figure 6.2: Loosely coupled interfaces

In the near future it is foreseeable that the separation of the interface and operational layers will grow further, as more applications use the Internet to distribute the interface and to communicate with remote objects. An interesting development to the work described in this thesis is the creation of an environment similar to Magritte but with a separation of interface and operational layers.

Such an environment would require a different communication with the operational layer but could be used for a broader class of languages and would not be restricted to Prolog. In fact, an environment with those characteristics could be useful to any language that uses structured terms as data and programs, which includes not only the Logic Languages and Functional Languages, but also some features of the Object Oriented languages as well.

Appendix A

Predefined Widget Classes

label Basic class to display text with alignment . This widget requires the definition of the `text` attribute, which represents the text to be displayed.

Attribute	Type	Meaning
`text`	term	the label's text
`font`	atom	(Edipo) font of text
`align`	`left`	left horizontal alignment of the label
	`right`	right horizontal alignment of the label
	`center`	central horizontal alignment of the label

button Class of widgets that display a text and respond to mouse pressing by evaluating a callback. This widget requires the definition of the `text` and `callback` attributes. The attribute `callback` can be specified with a button number as argument. The button number (1 for left, 2 for middle and 3 for right) identifies the mouse button that reacts to the user action.

Attribute	Type	Meaning
`text`	term	the text that appears as the buttons label
`font`	atom	(Edipo) font of text
`sep`	int	distance of inner rectangle from border
`line_width`	int	(Edipo) line width of the button's frame
`callback`	goal	a goal evaluated when the button is released
`callback(B)`	goal	goal evaluated only if button `B` was released

field Class with editing capabilities using an Emacs like key-binding. The attribute `chars` passes the initial string to the widget. This attribute can be read later, through the predicate `current_widget/3`, to obtain the string typed by the user. If defined, the attribute `callback` is executed when the user types `RET`.

Attribute	Type	Meaning
`chars`	list of chars	the text characters in the field
`font`	atom	(Edipo) font of text
`callback`	goal	a goal evaluated when `RET` is released

popup Special button that creates a pop-up menu with several entries, each with an associated label and callback. The attributes `text` and `menu` stand respectively, for an atom with the name of the menu and a term defining the menu options and the associated callback predicates. The value of the attribute `menu` must be a variable arity term of the form `menu(...,item(Name,Callback),...)`, where `Name` is an atom specifying the item's text and `Callback` the predicate to be evaluated when that item is selected.

Attribute	Type	Meaning
`text`	term	the menu label
`font`	atom	(Edipo) font of text
`sep`	int	labels space from the menu's left side
`itemFont`	atom	font of the menu items
`menu`	*see above*	menu items and their callbacks

scrollbar_x (scrollbar_y) General purpose horizontal (vertical) scrollbars. The position of the bar can be modified by pressing the mouse buttons. The center button places the bar at the mouse position, the left button decreases the slide position, and the right button increases the slide position.

The attribute `pos` sets the initial position of the bar, and `part` sets the size of the slide as a percentage of the size of the window. The attribute `callback(P,R,H)` is evaluated when a button is released over the scrollbar. The arguments are used to pass additional information to the predicate invoked by the scrollbar, respectively the slide's position, the ratio between slide and scrollbar and the scrollbar width (height).

Attribute	Type	Meaning
`pos`	int	position from the left of the scrollbar
`part`	int	ratio between scrollbar and slide
`callback(P,R,H)`	goal	goal executed when a button is released
		`P` is the position of the slide in the scrollbar
		`R` is the ratio between scrollbar and slide
		`H` is the width (height) of the scrollbar

scrollwindow Widgets having a general purpose window, movable by scrollbars. Graphics drawn on a scrollable window, the `back` window, are clipped by the scrollwindow's front window. The back window is movable using the scrollbars and thus the user can navigate in the data presented in the scrollwindow.

Attribute	Type	Meaning
`type`	`sx`	widget with just a horizontal scrollbar
	`sy`	widget with just a vertical scrollbar
	`sxy`	widget with both scrollbars
`back`	atomic	widget moved by the scrollbars

showtext Vertical scrollable widget that display text items

Attribute	Type	Meaning
`line(N)`	atom	text of line N
`line_font`	atom	font of text
`item_height`	int	height in pixels of each item

list Vertical scrollable widget of pressable text items. Each label is associated to a callback through an attribution of the form `item(Label)=Callback`. Each labels has a button like behaviour: when the mouse button is released the callback is evaluated.

Attribute	Type	Meaning
`item(Label)`	goal	goal evaluated when `Label` is pressed
`item_height`	int	height in pixels of each label

Appendix B

Window Attributes

The `window/2` predicate supports the following attributes. It should be noted that invalid values cause predicate failure.

- `display=Display_id`
  Specifies the X-Windows server to use to display this window. The default value is `''`, which corresponds to the machine where **YAP** is being executed. A display id is an atom of the form `Machine_id:Server_number.Display_number` (Example: `display=moon:0.0`). The predicate fails if non-atom values are supplied. Specifying non existent display servers causes execution to be aborted.

- `parent=Parent_name`
  Specifies the parent window of the window in question. Valid values are names of windows created previously. The predicate fails if names of non-existent windows are given.

- `x=X_position`
  Determines the horizontal coordinate of the window relatively to the upper left corner of its parent window, or to the upper left corner of the screen, in the case of toplevel windows. The default value is `x=20` and values must be positive numbers.

- `y=Y_position`
  Determines the vertical coordinate of the window relatively to the upper left corner of its parent window, or to the upper left corner of the screen, in the case of top-level windows. The default value is 30 and values must be positive numbers.

- `width=Window_width`
  Determines the horizontal dimension of the window. The default value is 350 and valid values are positive numbers.

- `height=Window_height`
  Determines the vertical dimension of the window. The default value is 350 and valid values are positive numbers.

- `cursor=Cursor_type`
  Sets the mouse icon to the specified type. Some of the available values for `cursor_type` are:

`X_cursor`	`circle`	`cross`	`crosshair`
`dot`	`hand2`	`heart`	`left_ptr`
`man`	`pencil`	`pirate`	`right_ptr`
`spraycan`	`tcross`	`watch`	`xterm`

  These are the standard names used in the X-Windows system, which are defined in the include file *X11/cursorfont.h*. Please consult that file for a complete list of valid names for `cursor`.

- `bkg_pixel=Colour`
  Sets the background colour of the current working window. The default colour is `white`. See Appendix E, for a list of available colours.

- `bkg_pixmap=Pattern`
  Sets the background to be *pattern*. *Pattern* can be a previously loaded image, through the `getimage` predicate, or a pattern list. A pattern list is a list of strings containing spaces and/or `#`. Spaces represent pixels of the foreground colour and `#` represent pixels of the background colour. (Example: `bkg_pixmap=["####","# ","# ","# "]` will produce a background consisting of small squares.)

- `border_pixel=Colour`
  Sets the colour of the border of the current working window. The default colour is `black`. See Appendix E for a list of available colours.

- `border=Border_width`
  Specifies the width of the border of a window. This attribute has no effect if the window has no parent. The default value is `0` (zero) and valid values are positive numbers.

Appendix C

Graphic Context Attributes

Graphic context's attributes are set to the default value when the first graphic context is created, and set to the previous working graphic context value on the subsequently created graphic contexts. The attributes supported by `gc/2` are:

- `function=`*`term`*
  This attribute allows the definition of the behaviour of pixel superpositions. The defined terms and their meaning are listed below.

Attribute name	Meaning
`clear`	*false*
`and`	`src` *and* `dst`
`andReverse`	`src` *and not* `dst`
`copy`	`src`
`andInverted`	(*not* `src`) *and* `dst`
`noop`	`dst`
`xor`	`src` *xor* `dst`
`or`	`src` *or* `dst`
`nor`	`src` *nor* `dst`
`equiv`	(*not* `src`) *xor* `dst`
`invert`	*not* `dst`
`orReverse`	`src` *or* (*not* `dst`)
`copyInverted`	*not* `src`
`orInverted`	(*not* `src`) *or* `dst`
`nand`	(*not* `src`) *or* (*not* `dst`)
`set`	*true*

  The default value is `copy`.

- `foreground=`*`colour`*
  Sets the `gc/2`'s foreground to *colour*. See Appendix E for a list of available colours.

- `background=`*`colour`*
  Sets the `gc/2`'s background to *colour*. See Appendix E for a list of available colours.

- `line_width=`*`value`*
  Sets the line width in pixels with which graphic primitives (lines, circles, rectangles, etc), will be drawn. The default value is `0` (zero) and valid values are positive numbers. Other values will cause the predicate to fail.

- `line_style=`*`term`*
  Sets the type of the line to be drawn. Possible terms are:

  - `solid`
    To obtain continuous lines.
  - `onOffDash`
    To obtain interrupted lines.
  - `doubleDash`
    To obtain mixed-lines, where the even dashes are filled differently of the odd dashes.

  The default value is `solid`.

- `cap=`*`term`*
  Determines the shape of the endpoints of a line. Possible terms are:

  - `notLast`
    The same as `butt`, except that for line width of `0` the endpoints are not drawn.
  - `butt`
    The line is square at its endpoints.
  - `round`
    The line is round at its endpoints.
  - `projecting`
    The line is square at its endpoints but a projection is drawn beyond the endpoint, for a distance equal to half the line width.

  The default value is `butt`.

- `join=`*`term`*
  Determines how corners are drawn for wide lines. Possible values for `join` are:

- `miter`
  The outer edges of two lines extend to meet an angle. `bevel` is used when angles are less than 11 degree.
- `round`
  The corner is round, centered at the join point.
- `bevel`
  `butt` endpoints are drawn, with the triangular notch filled.

The default value is `miter`.

- `fill_style=`*term*
  Determines the type of surface filling. Possible values for `fill_style` are:
  - `solid`
    Surfaces are filled with foreground.
  - `tiled`
    Surfaces are tile filled.
  - `stippled`
    Surfaces are filled with foreground masked by stipple.
  - `opaqueStippled`
    Surfaces are filled with foreground where stipple is one and background where stipple is zero.

  The default value is `solid`.

- `fill_rule=`*term*
  Defines which pixels are drawn when surface filling is performed. Possible values are:
  - `evenOdd`
    A point is drawn if it is inside the curve boundaries, according to the Even-Odd Rule.
  - `winding`
    A point is drawn if it is inside the curve boundaries, according to the Winding Rule.

  The default value is `evenOdd`.

- `arc_mode=`*term*
  Determines the type of arc. Possible values are:
  - `chord`
    The arcs are chord filled.

  - `pieSlice`
    The arcs are 'pie-slice' filled.

  The default value is `pieSlice`.

- `font=`*`font_name`*
  Determines the name of the font to use. Possible values are atoms with valid font names. The default font is `'9x15'`.

Appendix D

Event List

Events returned to Edipo include a substantial part of the X-Windows events, namely:

- `buttonPress(N,X,Y)`
  Occurs when a mouse button is pressed Instantiates `N` with the number of the button pressed (1 — leftmost button), and `X` and `Y` with the coordinates of the mouse.

- `buttonRelease(N,X,Y)`
  Occurs when a mouse button is released. Instantiates `N` with the number of the button released (1 — leftmost button), and `X` and `Y` with the coordinates of the mouse.

- `motionNotify(X,Y)`
  Occurs when the mouse moves around the window — a lot of these are generated! Instantiates `X` and `Y` with the coordinates of the mouse.

- `keyPress(C,K,X,Y)`
  Occurs when a key is pressed. Instantiates `C` with a list containing the ASCII code of the key, `K` with the scan code of the key pressed, and `X` and `Y` with the mouse coordinates.

- `keyRelease(C,K,X,Y)`
  Occurs when a key is released. Returns the same arguments as `keyPress`.

- `enterNotify`
  Occurs when the mouse pointer enters the visible part of the window.

- `leaveNotify`
  Occurs when the mouse pointer leaves the visible part of the window.

- `visibilityNotify(V)`
  Occurs when a part of the window changes its visibility status. `V` can be one of the following:
  - `partiallyObscured`
    A part of the window has been hidden.
  - `fullyObscured`
    The window has been totally hidden.
  - `unobscured`
    The window has been totally exposed.
- `expose(N)`
  Occurs when something that was hiding the window stops hiding it. Instantiates `N` with the number of `expose` events that will follow.
- `configureNotify(X,Y,W,H,B)`
  Occurs when the place or the size of the window is changed. Instantiates `X`, `Y`,`W` and `H` with, respectively, the coordinates and the dimensions of the window, and `B` with the border width.
- `unmapNotify`
  Occurs when the window has been unmapped.
- `mapNotify`
  Occurs when the window has been mapped.

Appendix E

Colours

The colours supported by **Edipo** depend on the X-server that is being used. For instance, on black and white screens the colours are converted to black or white according to its darkness. On gray scale and on pseudo colour monitors the the colour on screen may also be different from the colour specified. Colours can be specified in RGB format as an hexadecimal integer or using the colour names supported by the X-Windows system, such as:

```
white   black   gray  gray1...gray100
red     yellow  blue  green
orange  violet  pink  brown
```

To get a complete list of the valid `colour` names, consult the file `/usr/lib/X11/rgb.txt`.

References

[Abr89] Salvador Pinto Abreu. A Prolog interface to the x windows systems toolkit. In Lokhotia, D. Bowen, and R. Venken, editors, *Logic Programming Environments: The Next Generation*, Lafayette, 1989. NACPL.

[Alb91a] Henk Albas. Attribute evaluation methods. In Henk Albas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, number 545 in Lecture Notes in Computer Science, pages 215–233. Springer-Verlag, 1991.

[Alb91b] Henk Albas. Incremental attribute evaluation. In Henk Albas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, number 545 in Lecture Notes in Computer Science, pages 215–233. Springer-Verlag, 1991.

[Alb91c] Henk Albas. Introduction to attribute grammars. In Henk Albas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, number 545 in Lecture Notes in Computer Science, pages 215–233. Springer-Verlag, 1991.

[AsU86] Alfred V. Aho, Ravi seth, and Jeffrey D. Ullman. *Compilers: Principles, Thecniques and Tools*. Addison-Wesley Publishing Company, 1986.

[Bar91] Lon Barfield. Editing tree structures. Technical Report CS-R9264, CWI, P.O.Box 4079, 1009 AB Amsterdam, The Netherlands, 1991.

[BC91] J. Barwise and R. Cooper. Simple situation theory and its graphical representation. In *Third Conference on Situation Theory and Its Applications*, Kanagawa, Japan, 1991.

[BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *Computer Grapics*, 18(3):177–186, 1984.

[BS86] Rolf Bahlke and Gregor Snelting. The psg system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[C$^+$] Mats Carlson et al. *Industrial SICStus Prolog Users's Manual.* Swedish Institute of Computer Science.

[Cal90] A. Callebaut. Carmen: A Prolog toolkit for generating user interfaces. In Tony Kusalik, Jacob Levy, Mirreile Ducasse, and Anne-Marie Emde, editors, *ICLP Preconference Workshop on Logic Programming Environments*, Eilat, Israel, 1990. ICLP.

[Car83] L. Cardelli. Two-dimensional syntax for functional languages. In *Integrated Interactive Computing Systems*, pages 139–151. North Holland, 1983.

[CdBTT92] R. F. Cohen, G di Battista, R. Tamassia, and I. G. Tollis. A framework for dynamic graph drawing. Technical Report CS-92-34, Brown University, Brown University Computer Science Box 1910, Providence, RI 02912, August 1992.

[Con88] John S. Conery. Logical objects. pages 420–434, Seatle, August 1988. Proceedings of the Fifth International Conference and Symposium on Logic Programing.

[CR94] Kenneth C. Cox and Gruia-Catalin Roman. A characterization of the computational power of rule-based visualization. *Journal of Visual Languages and Computing*, 5(1):5–27, March 1994.

[DC86] A.D. Dewar and J.G. Cleary. Graphical display of complex information within a Prolog debugger. *International Journal of Man Machine Studies*, (25):503–521, 1986.

[DC89] Luís Damas and Vítor Santos Costa. *Yap Reference Manual.* Centro de Informática, Universidade do Porto, 1989.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, pages 244–320. Elsevier Science, 1990.

[EB88] Marc Eisenstadt and Mike Brayshaw. Aorta diagrams as an aid to visualising the execution of Prolog programs. In *Symposium on Visual Programming and Program Visualization*, London, England, March 1988.

[E.Y91] E. Shapiro E.Yardeni. The type system for logic programs. *Journal of Logic Programming*, (10):125–153, 1991.

[FR91] Thomas Fruchterman and Edward Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21(11):1129–1164, November 1991.

[FW94] Michael Frohlisch and Mattias Werner. *daVinci V1.4 User Manual.* University of Bremen, Departement of Computer Science University of Bremen - FB 3 P.O.Box 330 440 D-28334 Bremen Germany, second edition, November 1994.

[GCH92] L. Gomez, M. Carro, and M. Hermenegildo. Implementation of an event driven scheme for visualizing parallel execution of logic programs. In *Proceeding of the Workshop on Parallelism at IJCSLP'92*, Washington DC, November 1992.

[GNS87] J. Gettys, R. Newman, and R.W. Scheifler. *Xlib - C Language X Interface, Protocol Version 11.* M.I.T., Laboratory for Computer Science, 1987.

[GNS89] Jim Gettys, Ron Newman, and Robert W. Scheifler. *X11 MIT Reference Manual.* MIT Press, 1989.

[Hag92] Olof Hagsand. A framework for generating language-oriented environments for visual programming languages. Technical Report R92:06, SICS, BOX 1263, S-164 28 Kista, SWEDEN, March 1992.

[HO80] Gérard Huet and Derek Oppen. Equations and rewrite rules: A survey. In Ron Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic, 1980.

[HT86] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on attributes and relations. *ACM Transactions on Programming Languages and Systems*, 8(4), July 1986. in CIUP library.

[Hud91] Scott E. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems*, 13(3), July 1991.

[Hud94] Scott E. Hudson. User interface specification using an enhanced spreadsheet model. Technical report, Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, 1994.

[Kim90] Ted Kim. *XWIP Reference Manual Version 0.5, UCLP 1990.* UCLP, 1990.

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, (31):7–15, April 89.

[KP93] Antony J. Kusalik and Steven D. Prestwich. Program-oriented visualization of parallel logic program execution. Technical report, ECRC, Kobe, Japan, October 1993.

[LD87] E. Lusk and T. Disz. A graphical tool for observing the behavior of parallel logic program. In *Proceedings of the 1987 Symposium on Logic Programming*, San Fransisco, Agust 31 - Setember 4 1987.

[Lea90] José Paulo Leal. The Ytoolkit: a Prolog approach to an users interface. In Tony Kusalik, Jacob Levy, Mirreile Ducasse, and Anne-Marie Emde, editors, *ICLP Preconference Workshop on Logic Programming Environments*, Eilat, Israel, 1990. ICLP.

[Lea94] José Paulo Leal. A system to visualize and interact with logic programs. In Manuel Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, Madrid, Spain, 1994. PLILP'94.

[LJ89] P. Schachte L. Jenkins. Integrating Quintus Prolog and the X11 windowing system. In Lokhotia, D. Bowen, and R. Venken, editors, *The Next Generation. Proc. NACPL Workshop*, Lafayette, 1989. NACPL.

[MA] J. McCormack and P. Asente. *Using the X Toolkit or How to Write a Widget.* Western Software Laboratory, Digital Equipment Corporation.

[MA88] Joe MaCormack and Paul Asente. Using the X toolkit or how to write a widget. Usenix, Summer 1988.

[MW91] Ulrich Moncke and Reinhard Wilhelm. Grammar flow analysis. In Henk Albas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, number 545 in Lecture Notes in Computer Science, pages 151–186. Springer-Verlag, 1991.

[Mye89] B. A. Myers. *The Garnet Toolkit Reference Manuals: Support for High-Interactive graphical User Interface in Lisp.* Carnegie Mellon University, 1989.

[Mye90] B. A. Myers. Taxonomies of visual programming and program visualization. 1(1):97–123, 1990.

[O'D85] Michael O'Donnell. *Equational Logic as a Programming Language.* MIT, 1985.

[Ous94] John K Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing, 1994.

[PP92] Marian Petre and Blaine A. Price. Why computer interfaces are not like paintings – the user as a deliberate reader. In *The St. Peterburg International Conference on Human-Computer Interaction*, volume I, page 217, Moscow, Russia, 1992. Eat-West HCI.

[pT90] Frances Newbery paulish and Walter F. Tichy. Edge: An extensible graph editor. *Software-Practice and Experiece*, 20(S1):S1/63–S1/88, June 1990.

[Qui91] Quintus Corporation. *Quintus Prolog Reference pages*, 1991.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transations on Programming Languages and Systems*, 5(3), July 1983.

[SB89] Michael Spenke and Christian Beilken. A spreadsheet interface for logic programming. In *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems*, pages 75–80, 1989.

[ST83] E. Shapiro and A. Takeuchi. Object oriented programming in cuncurrent prolog. *New Generation Computing*, 1:25–48, 1983.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109–125, February 1981.

[TR81] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[War80] David H. D. Warren. Logic programming and compiler writting. *Software-Practice and Experience*, 10:97–125, 1980.

[Wat94] Richard C. Waters. Cliché-based program editors. *ACM Transactions on Programming Languages and Systems*, 16(1):102–150, January 1994.

[WCS96] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, 2nd Edition.* O'Reilly, 1996.

[Win83] T. Winograd. *Language as a Cognitive Process*, volume 1. Addison Wesley, 1983.

Index