A HIGH LEVEL LANGUAGE FOR SPECIFYING GRAPH-BASED LANGUAGES AND THEIR PROGRAMMING ENVIRONMENTS

APPROVED BY
DISSERTATION COMMITTEE:

To Dominique

with Love.

A HIGH LEVEL LANGUAGE FOR SPECIFYING GRAPH-BASED LANGUAGES AND THEIR PROGRAMMING ENVIRONMENTS

by

MICHIEL FLORIAN EUGENE KLEYN, B.Sc., M.Sc.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 1995

Acknowledgments

I would like to thank my committee chairman, my parents, friends, and colleagues for making this dissertation possible. Without their invaluable support it would have taken even longer than it did.

I would also express my sincere appreciation to all of the organizations that financially supported by work.

Bij het beëindigen van mijn academische studie maak ik garne van deze gelegenheid gebruik om mijn dank te betuigen aan allen, die to mijn wetenschappelijke vorming hebben bijgedragen, in het bijzonder aan mijn hooggeschatte Promotor.

A HIGH LEVEL LANGUAGE FOR SPECIFYING GRAPH-BASED LANGUAGES AND THEIR PROGRAMMING ENVIRONMENTS

Publication No. ______________

Michiel Florian Eugene Kleyn, Ph.D.

The University of Texas at Austin, 1995

Supervisor: James C. Browne

Abstract

This dissertation addresses the problem of creating interactive graphical programming environments for visual programming languages that are based on directed graph models of computation. Such programming environments are essential to using these languages but their complexity makes them difficult and time consuming to construct. The dissertation describes a high level specification language, Glide, for defining integrated graphical/textual programming environments for such languages. It also describes the design of a translation system, Glider, which generates an executable representation from specifications in the Glide language. Glider is a programming environment generator; it automates the task of creating the programming environments used for developing programs in graph-based visual languages. The capabilities supported by the synthesized programming environments include both program capture and animation of executing programs.

The significant concepts developed for this work and embodied in the abstractions provided by the Glide language are: an approach to treating programs as structured data in a way that allows an integrated representation of graph and text structure; a means to navigate through the structure to identify program components; a query language to concisely identify collections of components in the structure so that selective views of program components can be specified; a unified means of representing changes to the structure so that editing, execution, and animation semantics associated with the language can all be captured in a uniform way; and a means to associate the graphical capabilities of user interface libraries with displaying components of the language.

The data modeling approach embodied in the Glide specification language is a powerful new way of representing graph-based visual languages. The approach extends the traditional restricted mechanisms for specifying composition of text language structure. The extensions allow programming in visual languages to be expressed as a seamless extension of programming in text-based languages. A data model of a graph-based visual language specified in Glide forms the basis for specifying the program editing, language execution semantics, and program animation in a concise and abstract way.

Table of Contents

List of Figures

Chapter 1

Introduction

Interactive graphical software systems for supporting users in solving complex problems are now in widespread use. They are used in document publishing, computer aided design, and in many other synthesis problem domains. Such systems enhance a designer's effectiveness. They make it possible for the designer to explore design options by facilitating tasks such as construction, refinement, manipulation, modification, viewing, simulation, and analysis of models of the artifact that is to be created. One problem domain of significant and ever increasing complexity is that of specifying and developing programs. Interactive graphical systems are used, for the same reasons, in the domain of software specification and programming - to support software developers in similar tasks on programs (the artifact itself) or models of software artifacts. Within the domain of developing specifications and programs, there is a long history of research into the use of exploiting graphical techniques. Interest in so-called graphical languages or visual languages began almost as soon as programming languages themselves were developed. The use of visual languages is growing, driven by the increasing availability of the hardware and software execution environments that can support their implementation. This dissertation addresses the problem of creating interactive graphical programming environments for the predominant form of visual languages: graph-based visual languages.

1.1 Problem

Visual programming environments, *i.e.*, systems with graphical user interfaces which support a user in developing programs in visual languages, suffer from a problem which is common to all large complex software systems;

they are difficult and time consuming to develop. Every time a new visual language is designed, there remains the large task of creating the interactive environment that will support the development process for the new language. Development of the interactive graphical interface is often the largest single task in developing a programming system. The design of a visual language is aided by the ability to rapidly create a programming environment for it so that it can be easily evaluated and refined. Visual languages are difficult to use without the support of an interactive tool and so the creation of a new visual language requires the considerable time and effort for the implementation of the tool, above and beyond the design of the language itself and the implementation of its compiler. There is thus a frequent need for creating interactive graphical environments. Current support of the creation of these environments is inadequate because existing graphical tools and libraries are generic and do not support the abstractions that are specific to visual language programming environments.

1.2 Approach

The approach of this research was to design, implement, and experiment with a high level language which can be used to describe graph-based visual programming languages. A high level language is a means of simplifying the creation of programs in a particular problem domain by providing the abstractions necessary for solving problems in the domain as language constructs, so that solution programs can be formulated in a more concise and direct way. This description is analyzed by a high level language compiler in order to generate a program in a target language at a lower level of abstraction. The target may be in the form of a directly executable binary file or in the form of source code of a lower level language which can be further compiled. In order to reach the higher level of abstraction high level languages are mostly declarative. The compilation process transforms the declarative description into the procedural form required in the target language.

Glide, the high level language described in this thesis is a means through which the creation of programming environments for visual programming languages based on directed graph models of computation is automated. As such it is a *meta language*. The objects that can be described with it are languages. The compiler for Glide is a generator which analyses the high level specification of a particular visual language and emits a program which is the programming environment for that visual language. This environment is then used by a user wishing to develop programs in the visual language. The diagram in Figure 1-1 illustrates these relationships.

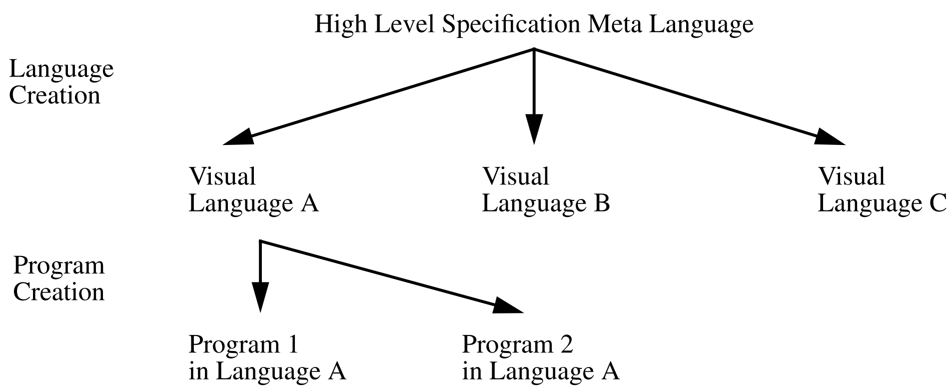


Figure 1-1 High Level Meta Language

Glide itself is not a visual language but a conventional textual language.

Within the context of using a high-level language, the approach developed in this dissertation is focused in two significant ways:

Graph-Based Visual Languages: First, the model of visual languages is restricted. No attempt is made to address all forms of visual programming languages. Taken as a whole there are very wide variations in the appearance, syntax, and semantics of visual languages and some lack of consensus on how to characterize them. Instead, the work in this dissertation focuses on the particular form of visual languages which is predominant. This form of visual languages can be termed “graph-based visual languages” since their common feature is that creating complex annotated graph structures is a central part of "writing" a program. There are two immediate advantages that accrue from focussing only these kinds of visual languages: (i) the notion of what a visual programming language is becomes clearer; and (ii) since graph-based structures and graph-based models are found in almost all areas of computer science computer and computer engineering, and many of these models have some form of execution semantics associated with them, the concepts developed in this dissertation are applicable to developing interfaces for them and the system developed in this dissertation can be used to generate interactive graphical programming interfaces for them.

Data-model Oriented: Second, within the context of generating a user interface from a specification, the approach that "the data model is the interface" is taken, *i.e.,* that a very effective way to design the graphical user interface for an application is to understand the structure of the model that is being manipulated by the application and being viewed through the interface. A simple example of this is a word processor; a good word processor must have a clear and consistent model of a document. The user can quickly understand the capabilities and limits of the word processing tool by understanding this model. With this model-oriented approach to visual language interfaces, the description of the structure of the language becomes the central means by which the interface itself is structured. The user's perception of the interface is one "through" which a data object (the program) is manipulated and viewed. For this reason it is possible to generate an interface for a visual language simply by creating a specification which is based in large measure on a description of the structure of the language itself. This approach, that the interface should not "get in the way" of the user, has repeatedly been advocated in the of user interface research community [AYC88, Sze93, Fol93, ShuFol].

1.3 Results

The results of this work are a set of concepts for characterizing graph-based visual languages and a set of abstractions embodied in the Glide specification language for specifying them and their programming environments. The concepts allow a unified view of the graphical/textual aspects of such languages and also a unified view of all the activity associated with them at the user interface: editing them, their on-screen execution/simulation and their animation. Secondly the abstractions and the design of the Glide language have been validated by an implemented compiler that has been used to create programming environments for several graph-based visual languages.

More concretely, the results of the work described in this dissertation are as follows:

- *Glide*: a high level language for specifying graph-based visual languages and their programming environments. Glide consists of several components:

  - A type composition component. The high level language described in this dissertation provides a means of modeling a visual language as a special form of composition of types. This specification of types is the basis for generating the interface.

  - A query language component. This query language provides a concise way of identifying a collection of components of a graph-based visual language program. Queries are used to specify the contents of views of programs.

  - An action language component. Actions are used to specify changes to parts of the structure that represents the program. It is used for specifying both editing operations and an abstract execution semantics of the language.

  - A graphical attributes component. This component is used to relate parts of the structure that represents the program to graphical properties of graphical objects that represent these parts on the screen.

  - A means to combine the action language and the graphical attributes to specify how animation of execution should occur.

- *Glider*: A functioning prototype generator. The generator is an implementation which validates the design of the Glide specification language. The Glider prototype consists of two parts: a generator which converts the high level specification into an executable program in a target language, and a run-time library which, when linked with the generated program forms the complete visual programming environment for a specific graph-based visual language.

- *Examples* Existing graph-based languages are specified in the Glide notation and it is demonstrated that Glider is can automatically produce the programming environment for the language. Though these interfaces are not as elegant and aesthetic as hand-crafted ones, they are effective and usable.
- *Extensibility*: It is shown that because of the way in which the Glide model uniformly integrates linear (text) and non-linear (graph) aspects of a graph-based language, a new framework for creating complex structures for representing computations is provided.

The system developed and described in this thesis is not a compiler for graph-based visual languages. The system does, however, allow the user to specify the execution of programs in an abstract way so that execution can be simulated for animation on the screen. If the language designer who provides the compiler for the graph-based visual language can instrument compiled programs to report changes when they execute, then the compiled programs can be coupled to the interface to drive animation from actual programs.

1.4 Contents of Dissertation

This thesis is divided into six major chapters. The next chapter, *Motivation*, justifies the need for a tool for automating the creation of programming environments for graph-based visual languages by showing that these visual languages are in widespread use. It also shows that many graph-based models of computation and specification exist, and that these benefit from the ability of a generator tool to quickly produce interactive programming interfaces for them, making experimentation and evaluation of models much easier. The following chapter, *Related Work*, identifies previous research that has addressed the problem of automatically creating programming environments and previous work in developing programming environments for graph-based visual languages. It shows that these interface generating tools are inadequate for creating programming environments for graph-based visual languages. Chapter 4, *The Glide Model*, provides a detailed exposition of the way in which graph-based visual languages are modeled with the Glide language. The exposition is a progressive one; Glide is described by introducing successive components of the language and relating them to familiar notations and languages. In Chapter 5, *Graph Types*, a data structure issue that needs to be addressed in representing graph-based languages is described and it is shown that this problem is fundamental in that it appears in the context of many other research areas. The way this problem is addressed in Glide is presented in the context of related work. This is followed by the chapter *Glider Design and Implementation*, which describes the design and implementation of the Glider generator and how the different components of a Glide specification are translated into a lower level of executable code by the generator. Finally in, *Examples,* three detailed examples are described to show how they are can be encoded in Glide and the resulting interfaces that were created are shown.

Chapter 2

Motivation

This chapter motivates the work of this dissertation by surveying and analyzing the widespread use of graph-based systems. The first section (Section 2.1), on visual languages, describes the basic ideas behind visual programming languages and then substantiates the claim that graph-based visual languages are the predominant form for visual languages. The next section (Section 2.2) surveys graph-based models. It is intended to give an appreciation for the large variety of ways in which graphs (as mathematical abstractions) have been used as vehicles for specifying, modeling, and understanding computational systems. The section makes the distinction between those graph models that exist to be seen and edited by a user vs. those that are internal graph-based structures that are never directly seen or manipulated by a user. It also makes the distinction between those directed graph models that represent some form of flow of execution vs. those that are not directly associated with execution. Those graph models which are intended to be edited by a user and also have an execution semantics fit the notion of graph-based visual language developed in this dissertation and are thus candidates for programming environments generated via the Glider generator. This chapter illustrates the range of graph-based languages and models for which a generator is intended to be able to create programming environments.

2.1 Visual Languages and their Programming Environments

In the broadest sense of the term, a visual language is any form of communication that is mediated through graphical means. Research into visual languages is aimed at finding ways of exploiting graphical displays to efficiently mediate communication. This involves making use of two or three spatial dimensions, color, icons, animation, and any other "meaningful graphic representations" [ShuVLdef] for which the human visual system has a natural affinity. Research into visual *programming* languages, however, is more specifically aimed at finding graphical means other than the conventional linear sequence of ASCII symbols to encode and display programs and specifications. The goal is to match this visual appeal with a "spatial" parser which can recognize the picture that is interactively created (*i.e.*, sketched, drawn, manipulated) by a user, so that a semantic interpretation can be derived from the recognized structure. An early example of this is the tabular specification interfaces of Query-by-Example database interfaces [ShuQBE]. This form of visual language makes use of vertical and horizontal alignment to convey common relationships between objects. At the time these interfaces were created they were considered visual language interfaces, but by current standards they would no longer be considered very graphical. Some visual languages go beyond exploiting the ability of users to perceive graphical relationships. They further exploit a human user's familiarity with objects in the physical world by giving the illusion of direct manipulation of physical objects by means of graphical objects on the screen. Examples of this are the programming by demonstration visual programming systems such as Programming by Rehearsal [ShuPBH], KidSim [Kidsim94], and the Alternative Reality Kit [AltReal88].

All visual languages share the need for a programming environment - an interactive graphical user interface which is used to create and manipulate visual language programs. Early visual language systems required extensive custom graphics software to be realized. Now many of these graphics capabilities are becoming more accessible with the more widespread use and standardization of higher level GUI (Graphical User Interface) libraries. Nonetheless, creating such an interface still requires considerable effort. This is because these libraries provide generic support for user interfaces of all kinds and do not provide the specific abstractions needed to support developing programs in visual languages.

A common feature of many visual language programming environments is the use of animation. Animation is the use of dynamic graphics to convey the execution behavior of a program. An example of a visual language in which programs are animated to illustrate execution is PICT [ShuPICT]. All visual languages share the need for graphics primitives to support the implementation of their programming environments, so it has been natural to further exploit the primitives to support animation. The programming environments allow the visual language to be executed in situ and the execution behavior is reflected as animations on the diagram (the visual language program) that the user created. This “immediate visual feedback” is a powerful aid to program comprehension [Bur94]. Visual language environments that provide animation are a specialized form of the more general concept of program visualization. Most program visualization systems have been developed for producing animations of algorithms and are independent of any particular programming language. Examples of such systems are BALSA [Bro88] and XTANGO [XTANGO]. These systems are used to illustrate and help explain programs and algorithms using a wide variety of graphical techniques.

Graph-based Visual Languages

Despite the very broad characterization usually given to visual programming languages, it becomes evident from surveying the many examples of visual programming languages that have been created, that those that are based on creating diagrams in which graphical objects are connected together into a graph structure are by far the most widespread. A simple tally of papers in the Proceedings of IEEE Visual Language Workshops [VLWks] reflects this. It shows the following approximate ratios of graph-based visual languages as a fraction of the total number visual languages: VL’84=7/11, VL’86=9/11, VL’88=16/25, and in the book on visual programming by Shu [Shu]=20/32. The recognition that graph-based visual programming is one of the most significant forms of visual programming is not new:

"..I think our paradigm is based on representing programs as graphs..."

(from "Is Visual Programming a New Programming Paradigm?" [CHVL 91])

There are too many examples to provide a complete list, but the following are three prominent examples of graph-based visual programming language environments: the network editor in AVS® [Sta91], the CODE2 parallel visual programming environment [Bro85,NB92,New94], and the graphical programming component of the LabVIEW® system [Nat87,Dye89]:

- *AVS*: The data visualization tool AVS contains a "network editor" subsystem. The network editor simplifies the creation of visualizations of data, allowing a user to compose a visualization program from predefined modules which process large numeric data sets. These modules are represented by nodes which are connected to each other with typed data flow links. The visualization programs can be immediately executed, evaluated and modified in the editor. The network editor allows rapid prototyping and reuse of modules. Animation in the form of highlighting of links and nodes gives the user an indication of the progress of the execution, the time spent in each module, and the order in which they were executed.
- *CODE*2: CODE2 is an interactive visual programming system for creating parallel programs at a high level of abstraction. The interface allows the user to specify a parallel program by interconnecting node icons with data dependency links. A node can represent a sequential computation or another parallel computation subgraph. Attributes associated with the nodes and the links are used to define the topology of data flow between nodes and the conditions under which nodes fire. The system is an integrated text/visual language system: the sequential code within a node is in a standard programming language (*e.g.* C) and can be edited with a text editor. The CODE2 system is the implementation of a directed graph-based model of computation (described in the next section).
- *LabVIEW*: LabVIEW is a system for processing data from and controlling data-acquisition instruments. It contains an embedded general purpose graph-based visual programming language called "G". G provides a range of general purpose programming constructs such as the typical forms of control structure (while and for loops), and complex data types (arrays) in

the form of complex nodes. The LabVIEW visual programming language is based on a data flow model of computation.

Many visual languages are based on the composition of objects into a graph but then also make use of 2D graphical features in order convey further semantics. Typical of these are the various object-oriented design notations such as Rumbaugh's Object Modelling Technique [OMT91], the Booch notation [Boo94], and the Shlaer/Mellor design language [ShM88]. These notations are all collections of graph-based visual languages which are overlaid with text and graphical features (e.g. iconic shapes). These features may have a particular meaning and their 2D spatial placement can be significant.

Some graph-based visual language are not intended to specify execution; syntax charts ("railroad diagrams") is an example. These charts do not specify the execution of a program but are intended for users of a text language who wish to understand the syntax of the language or to verify the syntax of programs in the language. A syntax chart is a visual language for communication between a language designer and its users.

Graph-based languages are used at all levels of design abstraction. For example, boolean circuit diagrams are a means to describe a computation at a very low level abstraction, since they can be directly implemented in hardware. The object-oriented design notations just mentioned, specify computation at a very high level of abstraction, since they record the initial design specifications for systems whose detailed components will be completed at a later stage.

There are only a few attempts to create general purpose graph-based languages which are analogous to the general purpose textual ones (C, Pascal) - LabVIEW and Prograph® [CGP89] come closest. Most are "little languages" which have specialized semantics for a particular domain. They are also often embedded as part of a larger system (AVS is a good example). In these systems a graph-based approach is used because the graph structure of the program is central to its design or the because the problem domain is not a naturally linear one (*e.g.,* parallel programming).

The Interconnection Paradigm

Underlying the large variation in visual appearance of graph-based visual languages is the same basic concept. In all these kinds of graph languages, it is an "interconnection paradigm" which is being exploited. Making textual programs by composing elements into a sequence is a very different program-creating mindset from making visual programs by composing elements into a graph. The key difference is that whereas in textual programs the tokens (characters or lexical elements) are *concatenated* or *inserted* into a linear sequence, in a graph-based program they are *interconnected.* It is often stated that the basic difference between textual languages and visual languages is that the former are "one-dimensional" while the latter are "two-dimensional". In the case of these graph-based visual languages however, it is more accurate to characterize the difference as "one-dimensional" vs. "many-dimensional", since in graph-based languages any graphical element can potentially be connected to (become adjacent to) *any number* of neighboring elements - not just *one* left neighbor and *one* right neighbor. This distinction is at the level of graphical elements that are manipulated on the screen; a user connects a "link" graphical element to a "node" graphical element in order to make the link and the node adjacent. A graphical element can even be transitively adjacent to itself through a cycle of adjacent graphical elements - this simply cannot happen in a text language. Even though the graph is graphically depicted as a two-dimensional geometric object, for most graph-based languages the particular physical location of connected elements is not meaningful. The graph has the same meaning no matter where the nodes and links are placed - it is only their interconnection that matters. However, just as in text languages with no formatting[1], it is difficult to read a poorly laid out graph[2].

[1] See Lamport [Lam90] for an illustration of the importance of formatting in text languages - the introduction discusses about how important formatting (i.e. layout) is in programs vs. in mathematical expressions.

The objects that are connected into a graph also often have further structure in the form of attributes or components. These are either displayed as graphical or textual annotations of the node or link, or they are made are accessible by opening a new window which displays these details. It is possible to implement an interface in which the objects can have their structure displayed and can be individually edited within the graph display itself, but it is only recently that this functionality has become available at the GUI level of abstraction and has not required detailed low level graphics coding to implement the visual language's programming environment.

The interconnection paradigm is a key concept that runs through this dissertation and it is reflected in the design of the Glide Language.

2.2 Graph Models

This section examines the use of directed graphs as abstractions which are the basis for models of computation. These models exist independently of whether or not they are created by a user with a graphical programming environment, but many do have an associated environment - in which case they can also be considered graph-based visual languages. Almost all the models of computation are *directed* graph models; the edges have an orientation. The edges are oriented in order to specify the direction of flow of data or tokens, or to indicate the direction of dependencies. Animation of programs of graph-based visual languages can be driven from the model of computation of the language; there is a flow of activity through the structure along the edges which can be graphically highlighted.

A great many of models and associated environments have been developed. There are too many to enumerate here, but the following list is a small representative sample of the main classes of graph-based models of computa-

[2] Providing efficient algorithms that produce readable graph layouts for applications such as graph-based visual programming environments is a difficult problem and an active area of research [ET89].

tion. It illustrates how widespread the use of a graph abstraction is in models of software systems.

- *Petri Nets*: A well known simple example of a graph-based model is Petri Nets [Pet81]. The Petri Net model is a parallel model of computation which is used to model the coordinated behavior of devices.

  Petri Nets are bipartite graphs; there are two kinds of nodes, transition nodes and place nodes, and there is one kind of directed link. A Petri Net executes as follows: Each place node may have zero or more tokens. If all the place nodes that are connected to links which are directed into a transition node have at least one token, then that transition node is enabled. One enabled transition node is chosen non-deterministically and fired atomically.This removes one token from each of the input place nodes and adds a token to the output place nodes (ones which are connected by links that are directed out of the transition node).

  There a great many variations of this basic theme that have been developed. For example, Colored Petri Nets is a model which adds structured typed tokens and more complex firing rules [CPN90], and PCM is an augmentation in which nets can be hierarchical, have time delays, and include parameters which specify replication of nodes at run time [BA88]. Others variations and many visual programing tools which provide simulation and animation are described in the proceedings for Petri Net workshops [F86].

- *State Transition Systems*: A graph-based way of representing finite state automata is through transition network diagrams. In these diagrams, every state of a system is represented by a separate node and edges between nodes are labeled with input symbols representing transitions that can occur between states. In this model the current state is represented by one node and not distributed over many nodes.

  More complex variations of this basic model have been developed. For example the system described in [ShuJLin, Jac85] is a transition network diagram model for specifying interaction in user interfaces. In this model there are several types of links which represent different types of transitions (*e.g.*, user input, interface output, function invocation). Another example is modelling communications protocols. Peers in communications protocols can be modeled as communicating finite state machines. The

Prospec system is an example of a visual programming tool developed for this kind of modeling [CL88]. *Hygraphs* is a model of computation based on hypergraphs. It is the model underlying the Statecharts visual language [Har87,Har88]. The model is based on state transition model and is intended to model so-called reactive systems. Reactive systems are systems which must respond to external events. Many variations of this model have been developed including, for example, Modecharts for modeling real time systems [JM88].

- *Data Flow:* The data flow model of computation is a graph-based model of computation which is intended to expose the maximum available parallelism. In this model of computation values flow along the arcs and are consumed by nodes which compute new values (*e.g.* simple arithmetic operations), and the new values are passed on out of the node . Again many more complex variations of this basic model have been developed [DF82].
- *Control Flow*: Control flow graphs are used to model the flow of execution in a program. Flowcharts is an early form of graph-based language for specifying program structure based on control flow.

Some graph models are intended to specify purely static relationships. For example, the data modelling language Entity Relationship Diagrams is the visual language for the Entity Relationship (ER) Model. A diagram is not directly associated with any execution model since it does not specify a computation to be performed. Instead it specifies static entities; the abstract structuring of data.

An example of a graph model which is not intended to be either seen or edited by a user is the global data flow graph that is used by an optimizing compiler to perform analysis of the use of variables [CCom88]. The flow analysis is based on graphs composed of nodes which represent basic blocks of instructions and edges which indicate possible succeeding blocks after branch points. These graphs are used compute how instructions which create, read, or write variables create dependencies between different blocks. The results are recorded as chains (paths) in the graph. The chains are then used decide if blocks are independent and can thus be executed out of order or in parallel. Despite the fact that a system that performs these kinds of analysis is not normally built to be graphical interactive system, a tool such as the one described in this dissertation, which could generate the interactive tool from a description of the data flow model could be very useful for understanding and experimenting which such graphs.

A graph model in which the editing operations are non-trivial is Binary Decision Diagrams [BDD92]. These diagrams are a means for encoding boolean functions in the form of a binary DAG (each node has two edges pointing out and any number pointing in). The nodes in the graph represent boolean variables and two edges represent the choice of assigning the value 0 or 1 to the variable. This representation sometimes has advantages over other representations such as truth tables for analyzing boolean functions. Graph manipulation operations exist which simplify the graph without changing the function it computes. This is a non-trivial task that requires skill and experience akin to being able to simplify algebraic expressions. These graphs can also be an internal representations manipulated by algorithms which preserve the function (*e.g.* merging).

Hypertext systems are systems which the structure of a document is a non-linear graph of interconnected pages instead of a conventional linear document. By themselves hypertext documents are static objects with no associated execution semantics. However the interaction of a user with a hypertext system or the process of traversing the system in order to retrieve information is dynamic, and models this activity, based on the graph structure of hypertext have been developed in the Trellis and PFG systems [Sto88,Sto90].

2.2.1 Graph Grammars

A more radical form of using graphs to model computational systems is to make the topology (the interconnection state) of the graph part of the execution model. In these *graph grammar* models, links and nodes can be created, deleted, connected, and disconnected during execution. The specification of these changes is in the form of graph rewrite rules. These rules contain a set of conditions on the attributes (“labels”) and connections of nodes or edges, and a set of changes to the attributes and connections that are to be applied if the conditions are satisfied. Execution of graph grammar programs means that the interconnection of the graph changes - the graph is dynamic. Graph grammars have been applied to model a great variety of activities, from database design to biological growth patterns [CER78,Ehr87,Ehr90]. A notion of dynamic graphs is sometimes also found in systems which are not directly thought of as graph grammars systems. For example, the CODE2 systems has a notion of replication and elaboration of graphs structure as part of its model of execution.

Though most the work in this area has dealt with classifying and finding properties of different classes of graph grammars, practical systems have also been built. The GraphEd system [Him89] is an interactive visual environment within which collections of graph rewriting rules of different classes can be defined and applied to sample graphs. The IPSEN system is a complete software development system (CASE tool) based on the graph rewriting concept [IPSEN92]. The Delta system is a model of parallel computation in which parallel programs can be defined with graph grammars [Delta91]. The ParaGraph system is an example of using graph grammars to concisely specify regular communications patterns used in parallel algorithms [BCL90].

Many of these models are visual languages since the rewrite rules and there effects are specified directly as graph diagrams, and not textually.

2.3 Summary

This chapter has given an overview of graph-based visual languages and the use of directed graph models of computation. When these languages or models are to be directly created and manipulated by a user, a graphical programming environment is needed to support this activity and to visualize the execution of the resulting program. The next chapter examines the extent to which previous work has addressed the problem of generating programming environments from descriptions of languages.

Chapter 3

Related Work

This chapter reviews previous work that is related to the system described in this thesis. It first describes previous research work performed in the area of programming environment generators. These are systems which allow the graphical user interfaces for developing programs in a language to be automatically generated from a specification of the language. It then examines some work that has indirectly addressed the problem of creating programming environments for graph-based visual languages. It finally presents work that has directly addressed the problem of creating programming environments for graph-based languages and discusses the different approaches taken. This chapter has a second important purpose. It also highlights the concepts developed in these areas of related work so that the design of Glide and Glider, described in the next chapters, can be seen as an extension of these concepts.

3.1 Programming Environment Generators

A significant amount of research work in the late seventies and early eighties was aimed at the automatic generation of language specific programming environments. These efforts ranged from the comprehensive approach of creating a complete set of language tools (editors, compilers, debuggers, profilers, etc.) to efforts targeted specifically at creating interactive editors tailored to a particular programming language. Some large systems were developed including Gandalf [ea85] and the Cornell Program Synthesizer (CPS) [ref TR81], and more recently Centaur [Klint93], and Pan [Grahm]. These systems are text language-based systems. Generation of the programming environment is based upon a specification of the syntax and semantics of the language in some meta language (the *meta-syntactic formalism* in the case of CPS and the *language description language* in the case of Pan). The use of a meta language embodies the idea that the language tools that had previously been created manually for each new language, could be synthesized from a language independent generator and a specification of the specific language.

These systems create editors that are based on the syntax of the language. These kinds of editors are known as *structure-oriented* or *syntax-directed editors*. It is possible to generate such an editor by providing a specification for the syntax of the language in the meta specification language (usually some variation or extension of BNF). The editors are interactive systems in which the user creates programs via editing actions which change a data structure which is the abstract syntax tree of a (partially completed) program. The editing actions are a specific set of tree editing operations. The tree may be displayed graphically as a tree or as text in a text editor which allows only tree editing operations. The motivation for using structure oriented editors is stated by Teitelbaum and Reps:

> "*Programs are not text: they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint...*[TR81]

No parsing is needed in such systems since the syntax tree is created directly. Because of this, there is more freedom in choosing the particular grammar to define the language syntax. Syntax directed editors have the advantage of making new users directly aware of the syntax of the language, but they have been criticized for making some kinds of program alterations more difficult. This is because any change to a program can only go through steps which leave the program in a syntactically correct state.

A more subtle effect of these editors is that they convey a feeling of composition and direct manipulation to creating programs. *Instances* of the *non-terminals* of the language are accessible by the user as templates which are partially instantiated when the program is being created. The non-terminal instances are first class editable objects which can be created and moved around in the tree in the same way as terminals: Terminals are placed into non-terminal instance templates; these can in turn be placed inside other non-terminal instance templates. This is different from using a plain text editor in which the user only manipulates arbitrary (semantics-free) character strings. In [Min92] it is argued that this kind of direct manipulation (of instances of terminals and non-terminals) is the correct approach to designing structure-based language editors and that reinforcing this approach in the view of the user (*e.g.*, by not insisting on a top-down or bottom-up order of editing operations) can overcome some of the awkwardness of using syntax directed editors.

The GRASE visual programming system is really a syntax-directed editor for a textual language (Pascal) which is graphically presented through variants of "Nassi-Schneiderman" diagrams [ShuGRASE]. Nassi-Schneiderman diagrams are recursive diagrams which divide a 2D rectangular space into triangular or rectangular parts. Each part represents a different syntactic component of a production of the syntax of a textual language. Each part can be recursively subdivided to represent the expansion of a production into components. GRASE is a top-down structure editor which makes the component oriented approach of syntax directed editing visually apparent.

The component based look and feel of structure editors is similar to the interconnection look and feel conveyed by using nodes or links in graph-based visual language programming environments - this resemblance is exploited and made concrete in the design of the syntax specification part of Glide and described in the next chapter.

The generators mentioned above were developed for text languages only. The underlying data model for the structure that represents programs is a tree, and it is only possible to specify editing actions in terms of manipulating nodes and branches in a tree. There is no *direct* way of representing the structure of graph-based languages within the meta-language models of languages used in these programming environment generators.

3.2 Programming Environments for Visual Languages

Though until very recently there have been no direct attempts to generate programming environments for graph-based languages, there has been some indirect work in this direction; some programming environments for specific graph-based languages have made their underlying execution engine accessible so that it becomes possible, in a limited way, to customize the structure of the graph language and the underlying model of computation (the graph language interpreter or compiler).

One example of this is the PFG (Parallel Flow Graphs) model for hypertext oriented systems mentioned in the previous chapter. The PFG system has an underlying Petri Net based model of computation. The primitives of the kernel computation engine have been made available to allow the creation higher level constructs to implement "higher-level" visual languages for particular applications of the system [Sto88]. Another example of this is the Olympus system [Nut91]. This system was originally a graphical programming environment for a specific graph-based model of computation: bilogic precedence graphs. The system was then made extensible to allow the model of execution to be customized while reusing the programming environment.

The existence of these systems further substantiates the argument that there is a need for a programming environment generator for graph-based languages.

3.3 Programming Environment Generators for Visual Languages

The recently developed Escalante system is a system that was developed as an off-shoot of Olympus. It is a system for generating visual languages that are based on graph models, and is thus the intention is similar to Glide. The Escalante system uses a different approach to defining the structure of a graph-based language. In Escalante, graphs are used as a representation medium for objects (nodes) and their relationships (edges). The underlying means for representing objects is then associated with graphical objects to describe appearances. Visual languages can be created which do not have a graph appearance but are represented by an underlying graph. Escalante uses classes to represent the graph objects (node and edges) and the graphical objects so that creating a new visual language is achieved by specializing these classes.

In [Got89,Got92] Gottler advocates the use of graph grammars as a basis for visual programming systems. He distinguishes between the user interface level diagram of a visual language and the underlying graph that represents it. The PAGG (programmed graph grammars) system that implements this approach is a graph grammar system in which the productions operate on attributed graphs. The productions are defined by directly drawing annotated graphs.

3.4 Summary

This chapter as described three threads of research towards the automation of creating programming environments for graph-based visual languages. No previous work has attempted to integrate the description of the textual structure and the graph structure of graph-based visual languages into a single uniform representation. This is unique to the Glide model and described in the next chapter. The next chapter will also show that the notion of a underlying graph representation to represent graph-based languages also exists in Glide.

Chapter 4

Glide Model of Graph Based Visual Languages

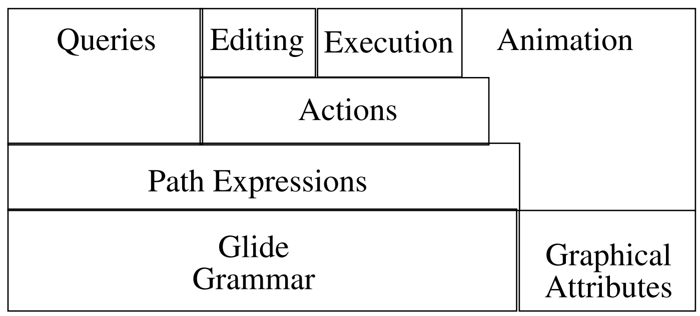
This chapter provides a complete description of the Glide approach to modeling Graph Based Visual Languages (GBVLs). The approach is presented by describing the design of the Glide high level specification language.

A user creates a programming environment for his/her GBVL by specifying a model of the language in Glide. This process involves first describing a data structure, and then describing sets of permissible changes to the data structure and associating graphical attributes with the data structure. Glide can be divided into six major components which play different roles in this process of modeling a GBVL. The components are:

- The *Glide Grammar*. This is the key underlying component of Glide. The Glide Grammar is a notation that provides the means to describe the syntactic structure of a graph based visual language as a set of types.
- The *Glide Path Expressions*. Glide path expressions are the basic mechanism for identifying and accessing parts of GBVL programs.
- The *Glide Queries*. This is a query language which provides a concise way of specifying the selection of parts of a program for display. Glide queries are used to specify the contents of views (windows) of programs.
- The *Glide Actions*. This component is used to specify changes of any kind - generally editing, execution, and animation.
- The *Glide Shape Predicates*. Shape predicates are used to further constrain the admissable instances of a data structure - they are used to specify the static semantics of a GBVL.
- The *Glide Graphical Attributes.* Graphical attributes provide information for the graphical rendering of the GBVL program on the display.

- The *Glide Animation* component. This is the means to tie changes as a result of execution to changes to the values of graphical attributes in order to reflect execution activity.

The following figure informally illustrates how these query, action and animation specifications are built up from the lower level components of the language.



Each component is described in turn in sections 4.1 through 4.7. Examples that illustrate how these components are used are provided in each section.

Main Ideas

The main ideas embodied in the design of the Glide language is that programs are view as data objects. Using this a starting point, Glide allows the structure of text parts of the language and its graph parts to be represented in a uniform way. In order to navigate and identify pieces of a program in the data structure path expressions are used. Queries, actions and animations are can then be expressed as accessing and altering the data in the data structure in terms of path expressions.

Graph Types

In Computer Science, it is often the case that a research issue or problem appears while investigating a particular area is found to be similar to problems encountered in other areas, because the problems are reflections of a more fundamental underlying issue. Part of this work on developing a model for GBVLs was an example of this phenomenon. In order to properly model the structure of such languages, the ability to represent “mutually defined” objects is needed. Such objects are also termed infinite structures, cyclic structures, shared structures, graph types, or multilinked types, depending on the area in which they are described. A full discussion of the issues surrounding the use of such objects in Glide is provided in the next chapter (Chapter 5, *Graph Types*). This chapter focuses on describing the design of Glide and how it is used, and shows how the issue of “mutually defined” objects arises in the context of representing graph based visual languages.

4.1 The Glide Grammar

This section describes the key underlying component of Glide, the Glide Grammar notation. The notation is used to describe the syntactic structure of a GBVL. It embodies the Glide approach of viewing the structure of a language in a way that unifies the description of the graph aspects of its structure and the text aspects of its structure.

A Glide Grammar specification consists of a set of productions. Such a set of productions is similar in appearance to both a set of BNF productions (which are used for describing of the syntax of text languages) and to a set of “user defined type” definitions of standard programming languages such as “struct” definitions in C, “class” definitions in C++ or “record” definitions in Pascal (which are used for describing the structure of data used in a program). These similarities are intentional. The BNF-like aspect of the Glide Grammar notation allows a Glide user to describe the structure of the textual parts of a GBVL in the usual way (as a CFG), *and* to view the description of the graph structure of the language as an extension and generalization of the way BNF captures text structure. The type definition-like aspect of Glide Grammar notation allows the user to view a GBVL as defining the structure of data (*i.e.* programs) and the construction of a program as the creation, composition and connection of instances of data types. The productions that make up a Glide grammar will be mostly referred to here as "productions", though on some occasions the term "type" will be used to emphasize the latter point of view.

The *Metanot* notation was developed by Meyer in [Mey90] in order to specify language syntax and semantics. In Metanot, a text language is described by modelling a language as a collection of complex structured types. This forms the basis for defining their semantics. This is a data-oriented view of programs and is thus suited as the basis for representing language specific editing interfaces. The notation adds "tags" to identify particular pieces of the structure and it adds the notion of comoposing without order. The syntax of Glide itself was derived and inspired by Metanot.

Since Glide grammar notation is an extension and generalization of BNF, the notation is presented here by beginning with an example of the traditional way of describing the syntax of a component of a text language and then incrementally introducing examples which use the additional Glide Grammar constructs which increase the expressiveness of Glide over BNF. The relationship between these additions and BNF productions and C struct definitions is carefully discussed at each step. (Any standard imperative language with the ability to define user defined types could have been used; C was chosen because it is widespread and familiar.)

A simple production

A simple BNF production defining the abstract syntax of an "if-statement" might be:

```
<If-Statement>::= <BooleanExpr> <Statement> <Statement>
```

This same definition can be made in Glide, but the syntax of Glide Grammar notation is slightly different. The equivalent production in Glide Grammar notation uses "==" instead of "`::=`" to separate the production name (LHS) from the components (RHS) of the production. The angle brackets ("`< >`") used in BNF to identify non-terminals are omitted in Glide Grammar notation. Terminals in Glide are distinguished by being completely in uppercase. A semi-colon is used to terminate each production. Hence the equivalent definition of the if statement in Glide is thus simply:

```
If-Statement ==  BooleanExpr
                 Statement
                 Statement ;
```

Each component is usually written on a separate line for readability, but this is not a requirement.

Tagging

In order to identify each component of the production, Glide provides the option of prepending each component with a *tag*.

```
If-Statement ==  Cond    :BooleanExpr
                 ThenPart :Statement
                 ElsePart :Statement ;
```

Each component on the RHS of a Glide production then consists of a pair of identifiers separated by a colon. The tag is on the left of each colon, and a name of a production is on the right of each colon. Tags are needed because they are used in subsequent parts of the Glide language (queries, actions, path expressions) to specify accessing components of instances of productions.

A production definition with tags resembles the definition of a user defined type. The small syntactic differences between a Glide production and a C struct definition are: In Glide a semicolon (";") terminates the list of components in a production while in C/C++ curly braces ("{ }") enclose the "members" on the RHS of a definition; In Glide the components in the RHS are separated by whitespace while in C/C++ they are separated by semicolons (and whitespace); In each component of a Glide production, the name of a production follows the tag name (separated by a colon) while in C the syntax is reversed - the name of the member follows the type of the member (separated by whitespace). Standard BNF does not have tags, but descriptions of language semantics associated with a BNF grammar often make use of left-to-right integer indices to refer to each component of a production.

There is a subtle but very important semantic difference in the way in which this kind of a definition is interpreted in Glide as compared to C or BNF:

C - In C a struct can be instantiated, and when it is the values of the members of the instance that is created must either be initialized or have unpredictable values. The creation, destruction and existence of an instance of a struct is *inseparable* from the creation, destruction, existence of values for the members it contains.

BNF - In the case of BNF, a production is a statement about an object that can be recognized in parsing a sentence. In the process of parsing, a sequence of terminals and non-terminals is *replaced* by the non-terminal whose production is matched. In most uses of BNF there is usually no need to distinguish between an instance of the use of the production and the symbols that it matched. If a parse tree is created by a parser then the 'instance' of a production corresponds to the non-leaf node of the tree that is created by a particular match with that production (a reduction).

Glide - In interpreting a Glide Grammar production, the ability to distinguish between an instance and what the instance contains is critical. A Glide production is viewed as naming and defining a new *container* type. The contents of the type is defined in terms of other container types and primitive types. In contrast to C and BNF it is possible to create an instance of the container *without* creating instances of the types that the container instance contains; the container instance can exist independently of the instances it contains. This interpretation stems in part from the structure-oriented view of programs (discussed in the previous chapter). In structure-oriented editors, container instances (also called *template instances*) for different language constructs can be created and can be filled with container instances of other constructs or with terminals.

In C, one means to make the existence of values of members of an instance independent of the instance itself is to use *pointer* members rather than actual values. In this case the instantiation of the type creates pointers which have to be initialized (or have unpredictable values), but the instances they point to need not be created at the same time. They could have been created before, or could be created later. This notion of a composite type whose instances are temporally independent of the instances they contain is analyzed in further depth in next chapter.

Aggregation

A difference between a BNF production and a C struct definition is that the *order* of the components in a production defines the order of the sequence of terminals that should match each component, while the semantics of a struct is *not* affected by reordering of the members in its definition[1]. A Glide production such as the one above, in which components are separated by whitespace also indicates that the order of the components is significant and should be as stated (as in BNF). The Glide Grammar notation provides a third combining operator, *aggregation*, to model a set of components in some part of the structure of a GBVL that have no intrinsic relative order (as in a struct definition). Combination by aggregation is denoted by a dot ("."). The following Glide production is an example:

```
Node ==   Nm :Name .
          St :State ;
```

The aggregation operator is thus commutative; the following production is equivalent to the previous one:

```
Node ==   St: State .
          Nm: Name ;
```

The aggregation operator is used to combine components when their order is not significant, in contrast combination with whitespace (termed *concatenation*). The `Node` production is expressing the fact that a node instance *consists* of a `State` and a `Name` and there is no intrinsic ordering between the two

[1] In some languages the *implementation* of the struct might be assumed to allocate memory for each member in the order given but it is bad practice to rely on this ordering directly; the values they hold should only be accessed through their member names.

(`State` doesn't "follow" `Name`). BNF has no direct way of expressing this form of combination.

The `Node` production also illustrates how the container/component interpretation of a production just discussed allows expressing the fact that a node is an object which contains other objects. *In Glide, a parallel is drawn between the component oriented view of structured text in structure-oriented editors and the fact that objects such as nodes in a GBVL often have further structure*. The Glide approach is to view entities associated with a node not as *attributes* of the node but as *components* which together make up the node. This approach is a significant departure from other graph structure specification languages which usually view information associated with nodes or links as attributes or "labels".

Alternation

In Glide, as in BNF, it is possible to express the fact that a production consists of one or more alternative components (or concatenations of components) using the *alternation* operator ("|"). Hence the usual way to define the fact that a text language construct is of two possible kinds is also available in Glide:

```
Statement ==    if:If-Statement |
                wh:While-Statement ;
```

Alternation has a more general use in Glide. In Glide alternation is also the means to express the fact that a node in a GBVL can be of two different kinds:

```
Node ==         Cn:ControlNode |
                Dn:DataNode ;
```

Glide also has a containment/component interpretation of such an alternation production. Rather than viewing this production as stating that a node *is* either a control node or a data node, it is viewed as stating that a node container instance *contains* either a control node instance or a data node instance. The reason for this view is uniformity; the same operator ("|") is used to express alternatives in both the text structure and the graph structure parts of a GBVL. A tag can be used to identify the component, whichever type it may be:

```
Node ==          NodeType: (Cn:ControlNode | Dn:DataNode );
```

Operator precedence and expressing commonality

In BNF there are just two operators for combining components: concatenation and alternation. A space between components is used to denote concatenation and vertical bar is used to denote alternation. In Glide, as in BNF, alternation has lower precedence than concatenation. Hence the following production:

```
Statement ==    IfCond:BooleanExpr
                ThenPart:Statement
                ElsePart:Statement |
                WhCond:BooleanExpr
                Body:StatementList ;
```

is equivalent to:

```
Statement ==    (IfCond:BooleanExpr
                ThenPart:Statement
                ElsePart:Statement)|
                (WhCond:BooleanExpr
                Body:StatementList) ;
```

However, unlike the concatenation operator (whitespace), the aggregation operator (dot) has *lower* precedence than alternation (vertical bar). Hence the Glide production:

```
Node       ==   Cn:ControlNode | Dn:DataNode .
                Name:STRING .
                Stmt:Statement ;
```

is equivalent to:

```
Node       ==   NodeType:(Cn:ControlNode | Dn:DataNode) .
                Name:STRING .
```

```
Stmt:Statement ;
```

Thus an instance of this `Node` production has three components, tagged `NodeType`, `Name`, and `Stmt`. The component tagged `NodeType` can be either an instance of a `ControlNode` or an instance of a `DataNode`. In Glide this is the means for expressing *commonality* between objects. This form of combination doesn't exist in BNF. It is syntactically similar to the use of a "union" in a C struct - a member which can contain a value of different types at different times. However, a union is intended for economizing on memory space rather than for expressiveness. In this example a node, be it a control node or a data node, always has a name and a statement and so these properties are defined in the `Node` production, while the properties that are specific to control and data aspects are put in their respective productions.

It is useful to compare the Glide approach to capturing commonality to the approach used in object-oriented languages. Unlike most object-oriented languages, in Glide there is no *taxonomic* form of expressing commonality. Instead this combination of alternation and aggregation is the way in which commonality between types is expressed in Glide. Commonality is expressed through a composition hierarchy instead of through a taxonomic hierarchy. The containing type, `Node,` consists of all the common properties plus an alternation of all its "subtypes" (sub as in subcomponent rather than subclass). The reason for using this approach is simplicity: the same notion of containment described earlier (which comes from the BNF notations being essentially composition formalisms) is, in conjunction with operator "|", adequate. This "has-a" rather than "is-a" approach is similar to some variants of object-oriented programming models; these are sometimes called "component" or "composition" - based object-oriented programming.

Sharing

In addition to the differences between BNF, C structs, and Glide that have just been described, there is another very significant difference in the interpretation of a Glide production. The instances of Glide productions can contain instances which are *shared*. For example, if the following three productions:

```
Aaaa ==         B:Bbbbb ;
Cccc ==         B:Bbbbb ;
Bbbbb ==        D:Ddddd ;
```

were defined, then the semantics of Glide allows the creation of a single instance of type `Bbbbb` which is contained in *both* an instance of `Aaaa` and an instance of `Cccc`. This is often referred to as allowing *sharing* in data structures. Sharing is exploited in Glide as a means to express aspects of the structure of a GBVL. In order to allow sharing, the creation of components independently of their containers is a pre-requisite. The use of sharing is often hazardous and must be controlled carefully. Glide *Shape Predicates* (described in Section 4.5) are a means of specifying static semantics and are used to control the use of sharing. The issues surrounding the use of sharing is explored in the next chapter.

Cycles

In addition to the simple sharing just illustrated, Glide also makes use of a more subtle form of sharing: *cyclic* sharing. Cyclic sharing is a special form of sharing in which the sharing and shared objects are the same object. Cyclic sharing is used in Glide as a means to express the *interconnection* of GBVL objects. For example, the following two productions:

```
Node      ==    NodeName:STRING .
                Input:Link .
                Output:Link ;
Link      ==    LinkLabel:STRING .
                HeadNode:Node .
                TailNode:Node ;
```

when combined with admitting cyclic sharing, makes it possible to express the fact that node objects and link objects can be connected (attached) to each other. When an instance of the type `Node` contains an instance of the type `Link` and simultaneously the instance of `Link` contains the instance of the `Node` production, this represents the fact that they are connected. This is a cyclic structure. *Not only are the objects associated with a node or link represented as "components" in Glide, but also the objects to which they are connected.* The concept of "component" has thus been generalized to the concept "is a component or is attached to". The apparent mutual containment is only possible by decoupling the existence of the container object from the objects it contains. In the case of a C struct, the same effect can be achieved by using pointers to contained objects instead of the objects themselves (and this is effectively how the underlying implementation, that the Glider generator creates, operates). When instances mutually contain each other, this represents the fact that they are connected and so can be displayed touching each other in a typical "graph" display.

This approach to modeling connected structure has a number of advantages in the context of specifying GBVLs. It is a uniform integrated approach in which all objects: nodes, links, expressions, terminals, non-terminals, values are, from the point of view of the interface, peers.

Repetition: Sets and Lists

As do many extensions of BNF, Glide provides annotations to indicate that a component of a production consists of multiple instances of a type. There are two annotations: a single star ("*") annotation is used to denote a *list* of elements of a given type (*i.e.* ordered); and a double star ("**") annotation is used to denote a *set* of elements of a type (*i.e.* unordered). For example, the following Glide production states that a dataflow network consists of a set of nodes and a set of links:

```
DataFlowNet ==  N :Node** .
                L :Link** ;
```

The default case, no stars, means a single object of a given type. Another example: the simple `Node` production above specified that exactly two `Links` are attached to a node; a more general form allows an arbitrary number of input link and output links, *i.e.*:

```
Node       ==    NodeName:STRING .
                 Input:Link** .
                 Output:Link** ;
```

Note that ** is used to specify that there is no particular ordering amongst the links. A node which requires at least two links, but may have more can be captured as follows.

```
Node       ==    NodeName:STRING .
                 L1:Link .
                 L2:Link .
                 LMore:Link** ;
```

Terminals

Terminals are identified in Glide productions by being in all uppercase. Glide provides the primitive types `BOOLEAN`, `INTEGER`, `REAL`, `STRING`. Any other identifiers in all uppercase define constant terminal symbols. Hence an enumeration can simply be expressed as the alternation of constant types. For example:

```
State == choice: (e:ENABLED | d:DISABLED | f:FIRING)
```

where `ENABLED`, `DISABLED` and `FIRING` are constant terminal symbols. Constant terminals can also be specified by being enclosed in single quotes to allow them to be in mixed case:

```
Aterm === :'if' :BoolExpr :'then' :ThenPart
```

Note that because Glide admits cyclic structures, it is possible to define a language without using any terminals.

4.1.1 Complex GBVL structures

This Glide grammar-based approach to describing connected graph structure is very flexible, allowing more complex forms of GBVL structure such as ports, hierarchy, and hyperedges to be represented very naturally:

Ports - The Glide approach allows the notion of nodes with "ports" to be captured in a natural way; Nodes have ports as part of their components and it is these ports that are connected to links.

```
Node ==         NodeName:STRING .
                InputPort:Port .
                OutputPort:Port ;
Port ==         Label:STRING .
                L:Link** ;
Link ==         Head:Port .
                Tail:Port ;
```

HyperEdges - Edges or links which have multiple ends can be represented. For example, the following specifies a link which connects three nodes

```
Link ==         N1: Node .
                N2: Node .
                N3: Node ;
```

and a link attached to an arbitrary number of nodes can be specified by

```
Link ==         N: Node** ;
```

Hierarchy - Hierarchy, the ability to structure a complex object into many levels, is a feature provided by an increasing number of GBVLs. It can be represented quite naturally in Glide by including recursion through a production. In the simple example below, the type `Graph` is included as a component of the `Node` production to indicate the fact that nodes can themselves contain a complete `Graph` and this recursion can continue to and arbitrary depth. This is conceptually no different from the use of recursive productions in a BNF grammar.

```
Graph ==        N:Node**
                L:Link** ;
Node ==         NodeName:STRING .
                InputPort:Link** .
                OutputPort:Link** .
                SubGraph:Graph;
```

4.1.2 Example Glide Grammar Specification

The following is a complete example of a Glide Grammar that defines the structure of a simple form of Petri Nets:

```
Petrinet ==      Nodes : Node**.
                 Links : Link**;
Node ==          NT: (Pn:Placenode | Tn:Transnode) .
                 Inputs: Pnlink** .
                 Outputs: Pnlink** ;
Placenode ==     PLabel : STRING  .
                 NumTokens: INTEGER ;
Transnode ==     TLabel : STRING  .
                 State : State;
Pnlink ==        LLabel: STRING .
                 LinkHd : Node .
                 LinkTl: Node;
State ==         ch:(e:ENABLED | d:DISABLED | f:FIRING)
```

A typical instance of this type, containing 3 nodes and 3 links might be displayed to a user in an interactive graphical programming interface as the box shown here:

`Link` : **b, d, f**

`Transnode`: **c**

`Placenode`: **a, e**

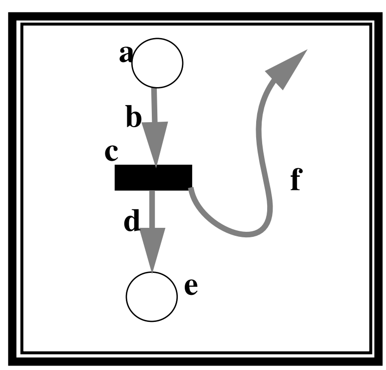


Figure 4-1 Displayed Petri Net

The `Link` instances **b, d,** and **f** are graphical elements in the same way as the nodes **a**, **c** and **e**. The interconnection of components is represented through cycles between instances of Node and Pnlink. The enclosing box represents the instance of `PetriNet` and contains a set of Links and a set of Nodes. A pointer implementation of the interconnection of the components is shown in this diagram:

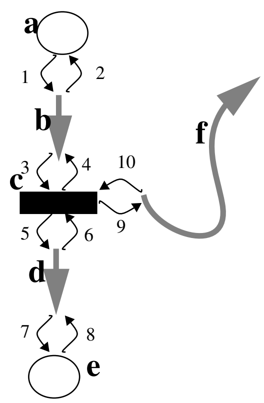


Figure 4-2 Implementation of a Petri Net

This diagram would not be displayed to a user of the system. It is shown to illustrate how the "interconnectedness" of graphical elements is represented through cyclic references. The following diagram illustrates the complete PIN for the Petri Net (with a box for each type instance) Note that the curved arcs are logically no different from the straight edged ones.

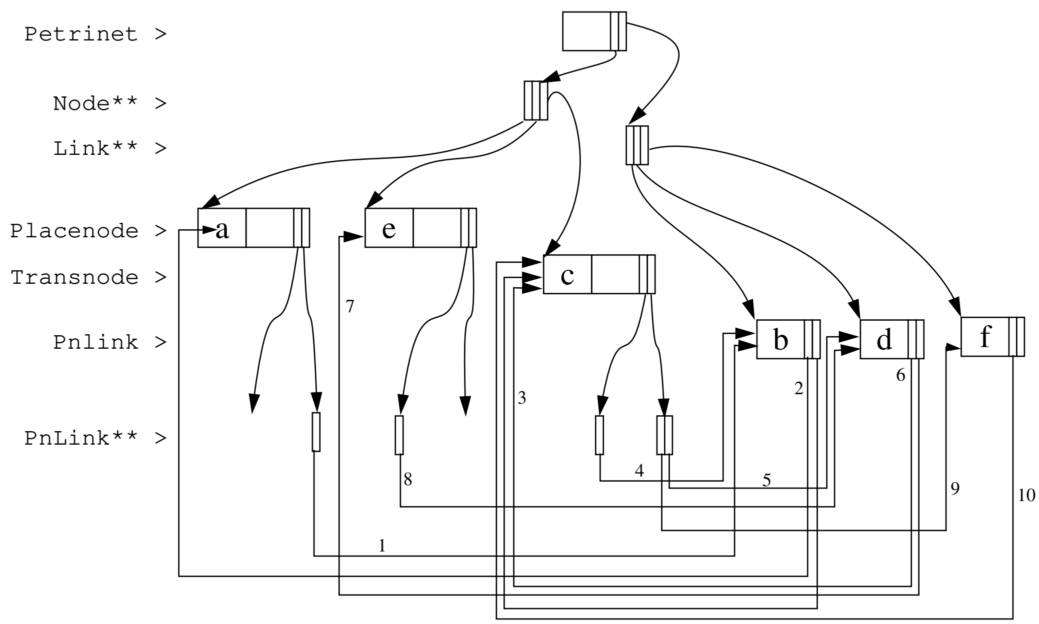


Figure 4-3 PIN of the Petri Net

This specification of Petri Nets can easily be extended to accommodate, say, a textual condition attribute by simply adding the BNF definition of the condition syntax to the current specification:

```
Transnode ==     TLabel : String  .
                 State : State .
                 Condition:Condexpr ;
Condexpr ==      Var1: STRING
                 RO: Relop
                 Var2: STRING ;
Relop == r : (eq:'=' | gt:'>' | lt:'<' | leq:'=<' | geq:'>=');
```

4.1.3 Summary

The Glide Grammar is a means of capturing the structure of a GBVL, *i.e.* how component objects of a GBVL are composed and connected. This structure then forms the basis upon which other aspects such as editing actions, execution semantics, and graphical appearance are specified. The Glide Grammar abstracts out the logical structure of a GBVL, separating it from issues of graphical appearance.

The key reason for wishing to represent the structure of GBVLs by extensions and generalizations of the way the text language structure is represented, is that is then possible to smoothly integrate the representation of graph syntax and text syntax into one uniform formalism, and hence to view graph based visual languages as a generalized form of text based languages. The way in which semantics, execution and animation is specified, as will be shown in the rest of this chapter, can then also be achieved through a uniform notation. The grammar also allows more complex forms GBVL structures such as ports, hyperedges and hierarchy to be captured.

Glide Grammar Syntax

A simplified description of the syntax of the Glide Grammar is listed below. The notation used to specify the syntax of the Glide Grammar itself is EBNF ({ } means zero or more, [] means optional) This description is abstracted to bring out the essential simplicity of the structure of Glide grammar specifications.

```
<glidegrammar>       ::= <productionlist>
<productionlist>     ::= <production> “;” { <production> “;” }
<production>         ::= <aggregationlist>
<aggregationlist>    ::= <aggelement> { “.”  <aggelement> }
<aggelement>         ::= [ <tagname> ] “:” <alternationlist>
<alternationlist>    ::= <altelement> { “|” <altelement> }
<altelement>         ::= [ <tagname> ] “:” <sequencelist>
<sequencelist>       ::= [<tagname>] “:” <seqelement> { “ ” <seqelement> }
<tagname>            ::= <identifier>
<seqelement>         ::= <identifier> |<identifier>“*” | <identifier>“**”
```

<identifier> is any alphanumeric string not beginning with a number.

4.2 Glide Path Expressions

This section describes the syntax and semantics of Glide Path Expressions. Glide Path Expressions are similar to the expressions used in standard programming languages to access components of instances of user defined types, such as those in C that specify accessing members of structs and/or following pointers to structs (*e.g.,* `a.b`, `a->b.c.f`, etc.). However, since the Glide Grammar allows definition of data models in a way that is more expressive than C structs, Glide Path Expressions are commensurately more expressive and their evaluation is more complicated.

Glide Productions can be viewed as type definitions. When a program in a GBVL is created (through the invocation of editing operations which will be described in Section 4.4), many instances of these types are created and interconnected to each other to form a network that represents the program. This network is termed a *Program Instance Network* (PIN). Glide path expressions are used to identify different parts of the PIN and to navigate through it. Path expressions are set-valued expressions that are used within Glide queries, actions, and shape predicates (described in the sections that follow: 4.4, 4.3, and 4.5). Since they are common to these components of Glide they are described here first.

For the purpose of simplifying the explanation of path expressions, this section uses identifiers for the instances that are created in a PIN (*e.g.* "`Node001`"). Path expressions in Glide queries and actions in fact never directly refer to such identifiers; they use variables which are bound to instances and so the actual identifiers of instances are never visible to either the Glide user or the GBVL user. They are exposed with identifiers here to facilitate explaining path expressions.

The syntax of Glide Path Expressions is simple. A path expressions consists of an instance followed by sequence of tags connected by a dots, *e.g.* "`Node001.b.c.g".`(The dot used is of course a different kind of dot from the aggregation dot used in productions). Path expressions are used to identify instances and components of the instances from the PIN. Consider the following simple Glide production:

```
If-Statement ==      Cond:STRING
                     ThenPart:STRING
                     ElsePart:STRING ;
```

If an instance of such a production, say `If001` has a condition component "`a<3`", a then part "`x:=2`", and an else part "`x:=3`"; then each piece can be extracted using a path expression:

The path expression `If001.Cond` identifies "`a<3`"

The path expression `If001.ThenPart` identifies "`x:=2`"

The path expression `If001.ElsePart` identifies "`x:=3`"

The term "identifies" is used here because, depending on context, a path expression may indicate returning the value found at a location in the PIN or identify it for alteration. However, for brevity an "=" is used in subsequent examples. The dot notation can be (and is often) used to arbitrary depth. If `If001` was the component tagged `IfS` of, say, `Node002` then

```
Node002.IfS.ElsePart = "x:=3"
```

If an instance of a production:

```
DataFlowNet ==  N :Node** .
                L :Link** ;
```

say, `DataFlowNet002`, has a set of 3 nodes (`N003 N004 N005`) and a set of 2 links (`LO06 L007`) then the relevant path expression identifies a list of instances:

```
DataFlowNet002.N = (N003 N004 N005)
DataFlowNet002.L = (L006 L007)
```

In a deeper path expression the evaluation is continued through each member of a set or list. Hence the three nodes have the names “alpha”, “beta”, and “gamma”, then

```
DataFlowNet002.N.Name = (“alpha” “beta” “gamma”)
```

Since the tag `N` identifies a set of instances of type `Node`, the semantics of path expression evaluation is that it iterates over each one to reach and the name of each `Node` instance.

Path Expressions and Alternation

In the case of a production with alternation as in:

```
Node       ==     NodeType: (Cn:ControlNode | Dn:DataNode) .
                  Name: STRING .
                  IfS:  IfStatement .
                  Input:Link .
                  Output:Link;
```

there are 3 tags that could be used to identify the first component (`NodeType`, `Cn`, or `Dn`). They are used as follows: using a tag inside the alternation will return the instance, if that instance is of the type associated with the tag, otherwise it will return `NULL`; a tag that identifies the whole alternation (*e.g.* `NodeType`) returns whatever instance is present. For example,

```
N001.Cn = ControlNode008
N001.Dn = NULL
N001.NodeType = ControlNode008
```

An path expression containing a tag that identifies a set or list component will have `NULL` elements removed, returning only the instances of the selected type:

```
DataFlowNet002.N.Cn = (ControlNode008 ControlNode010)
DataFlowNet002.N.Dn = (DataNode009)
DataFlowNet002.N.NodeType = (ControlNode008
                             DataNode009
                             ControlNode010)
```

Note that the result of the last path expression is not a homogenous list since it contains instances of more than one type.

Uppath

A single dot (".") is the "downpath" operator, indicating that the evaluation of the path expression "moves down" in a PIN into the component of an instance identified by the tag that follows the dot. It is also possible to the go the other way, "moving up" into the parent instance that contains a given instance. This is expressed with a "double dot" followed by a tag, for example:

```
Node002..N = DataFlowNet001
```

A double dot ("..") is used as symbol for the "uppath" operator. It is reminiscent of "cd .." in a file system command (UNIX/DOS/VMS). However, since a PIN is a graph and not a tree - some instances may have more than one "containing parent", a tag name is used to identify which parent is desired. In some cases there may be more than one parent that refers to (*i.e.* has as a component) a given instance through the same tag. In this case, all these parents are returned by an uppath.

In the case of using the uppath operator when an instance is on an alternation, the tag identifying the whole alternation or the tag identifying one of the alternatives can be used. For example

```
ControlNode008..Cn = Node001
ControlNode008..NodeType = Node001
ControlNode008..Dn = NULL
```

Glide Path expressions provide a simple but powerful way of extracting information from Program Instance Networks as these examples illustrate:

`DataFlowNet001.N.Stmt =` all the statements of all the nodes

`DataFlowNet001.N.Input =` all links that are input to some node

`DataFlowNet001.L.HeadNode =` all nodes at the head of any link

`IfStatement015..IfS =` the node which contains `IfStatement015`

Because of sharing, two different path expressions may identify two different paths that end up in the same place in a PIN. The different paths represent different roles that the identified object has; one path may express the fact that a node is part of some graph, the other that it is connected to some link. The up-path operator also has a loose correspondence to inheritance; it can be used as way of accesing the common propetries held in the containing object.

The result returned by a path expression is always a set instances (flat, and no duplicates). Unlike C there is no explicit notion of references or pointers in these path expressions; the component referred to by a tag is always identifies another object (or objects), never a pointer to it (or them).

4.2.1 Glide Path Expression Syntax

The simple syntax of path expressions is shown here. Extensions to this basic form of path expression are used within the other components of the Glide language (queries and actions) and are discussed in the relevant section.

```
<glidepathexpression> ::= <instance> <dotexpression>

<dotexpression> ::=
                  | "." <tagname> <dotexpression>
                  | ".." <tagname> <dotexpression>

<tagname>   ::= <identifier>
<instance>  ::= <identifier>
```

4.3 Glide Query Definitions

This section describes the query definitions component of Glide. This component allows the Glide user (the GBVL designer) to specify a set of queries which each determine the contents of a view of programs that he/she wants the GBVL user to have. Views are simply windows which display a part or parts of a program. Typical views that a Glide user might wish to provide are, for example: a top level view of the major structural components of the program, a top level view of the nodes and links as icons, a more detailed view of the nodes and links labeled with their more important attributes, detailed views of the contents of selected nodes and links with all their attributes, views of specific annotations of nodes or links, views of the entire program, etc. Views allow the GBVL user to inspect and edit programs. The query definition associated with a view is the means by which the contents of views is specified. Query definitions are used to identify the particular components of the program that are to be displayed in views, but they are not used to specify the graphical appearance of these components. The latter is determined by the specification of the graphical attributes of the productions and will described in Section 4.6.

Glide query definitions are based on Glide path expressions but they provide a higher level of expressiveness for specifying extracting information from a Program Instance Network than do simple path expressions. Query definitions also make use of a more complex form of path expression, *tree path expressions*. This section first describes tree path expressions. It then describes the syntax and semantics of query definitions by beginning with a simple example and progressing through to more complex forms of queries.

4.3.1 Tree path expressions

Glide query expressions use path expressions and an extended form of path expressions called *tree path expressions*. A tree path expression has the same syntax as a path expression, except that a star (“*”) may be used instead of a tag. For example

```
Node001.A.B.*.*
```

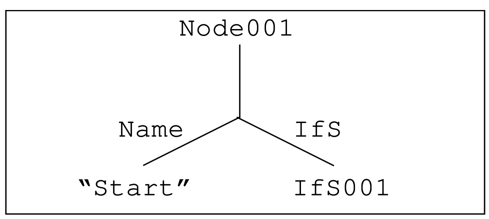
A star just denotes “all tags” of the production identified by the preceding portion of the path expression. The “tree” in tree path expressions refers to the fact that instead of specifying and returning a set of instances, tree path expressions specify and return a single tree that is made up of instances. The nodes of the tree correspond to all the nodes in the PIN that were encountered while traversing the PIN in order to evaluate the tree path expression. Such a tree structure is thus a subtree of the PIN graph. For example, if a GBVL specification contains the Glide productions:

```
Node ==          Name:STRING .
                 IfS:IfStatement ;
IfStatement ==   Cond :STRING
                 ThenPart:STRING
                 ElsePart:STRING ;
```

and there exists an instance `Node001`, then the tree path expression

```
Node001.*
```

returns a one-level tree of the form:[2]

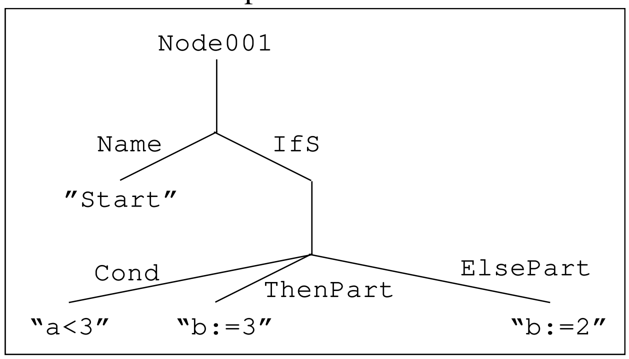


A tree path expression with two stars such as

```
Node001.*.*
```

[2] In these tree figures, the tag labels the branch that leads to the value or instance corresponding to that tag in a particular instance.

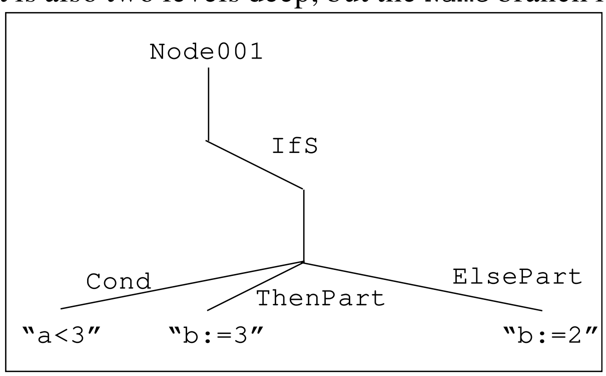
returns a tree that is two levels deep:



The expression

```
Node001.IfS.*
```

returns a tree that is also two levels deep, but the `Name` branch is pruned:



Each of these resulting trees is a tree which is derived from the PIN, which itself is a graph. A tree path expression is simply a way of stating which of the many possible trees in the graph should be extracted.

The tree path expression notation is reminiscent of “file globbing” expressions for identifying files in shell command languages (of UNIX, DOS, VMS, etc.), *e.g.*, “`ls */*/bin/*`”. Both Glide tree path expressions and globbing expressions are used to specify extracting objects from a structure. In globbing expressions, a *list* of file pathnames is returned. This list is simply an uncompacted representation of the pruned tree that the expression specifies. In Glide the more compact representation of a single tree is used, and the tree is extracted out of a graph (the PIN) rather than out of a tree (the directory tree).

4.3.2 Simple Queries

The simplest form of a Glide query definition is one that just relates an input parameter to a tree path expression:

```
ShowNode2Deep(n:Node) == { n.*.* }
```

The LHS of a query contains a "query name" followed by a list of typed "input parameters". In this example there is only one input parameter. The symbol on the left of the colon is the name of the input parameter and the symbol on the right is its type (*i.e.*, one of the productions previously defined in the GBVL's Glide grammar)[3]. On the RHS of this definition is a *return clause* which, in this simple case, consists of only one tree path expression. This query expression just returns the tree corresponding to the tree path expression evaluated for whichever instance of a node the parameter `n` is bound to.

The query is compiled into a procedure and its name is added to a menu of commands in the interface generated by Glider so that the user of the GBVL can invoke it and bring up a new view. The user invokes the command in concert with selecting one or more objects in existing views. These selections effect the binding of values to the input parameters of a query invocation. Invoking the command brings up a window displaying the specified contents of the node instance (in this query, all its components, two levels deep). The display window is created by the "Glider View Renderer" - a run-time module which takes the results of queries (one or more trees) as input.

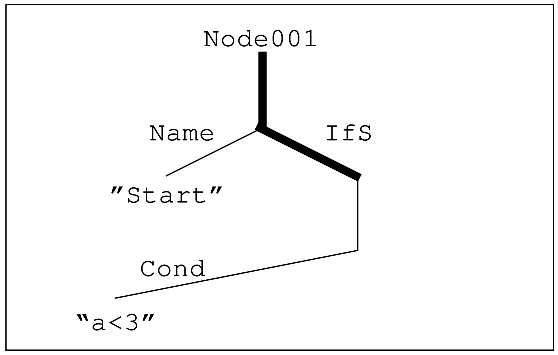
Merging trees

There can be more than one tree path expression in a query expression, for example:

```
ShowNodeCondition(n:Node) == { n.*, n.IfS.Cond }
```

[3] Note the colon in the input argument list is different from the colon in the Glide Grammar notation; here we are separating a variable from the name of its type.

This is a way of expanding specific branches of a tree - here only the `IfS` branch of the tree is extended down another level.



The procedure generated by the Glider queries compiler merges the two trees specified by the two tree path expressions because the trees overlap in the PIN graph - they have a common "stem" (illustrated in the figure with the heavy line).

4.3.3 More Complex Queries

The shape of the trees produced by the return clause of the queries can be controlled by using the more complex queries. These queries use the two other clauses available in Glide queries: the *suchthat-clause* for quantification and the *where-clause* for restrictions.

The following Glide grammar will be used to illustrate these more complex queries. The grammar is a simplified description of the Petri Net based GBVL "PCM" [Adi88]. PCM has nodes with structured text attributes and is hierarchical - a transition node can contain a further complete PCM net.

```
PCMStructure == PN: PCMNet;
PCMNet ==       N: Node** .
                L: Link** ;
Node ==         (P: PlaceNode | T: TransNode) .
                I: Link** .
                O: Link** ;
PlaceNode ==    NmTk: INTEGER .
```

```
                  Rp: RepParameter ;
TransNode ==      Name: STRING .
                  Ta: TransAttibutes .
                  SubNet: PCMNet;
TransAttributes ==     Pi : Pexpr .
                       TSte: State ;
State==    Ste:( :ENABLED | :FIRING | :DISABLED | :ACTIVE )
Link==     Head  :Node .
           Tail  :Node ;
```

Multiple trees

The simple queries return only one tree, but a query can also return a set of trees. For example, the query

```
ShowNode2Nodes(n1:Node,n2:Node) == { n1.*, n1.T.*, n2.* }
```

specifies displaying two objects, each one represented by its own tree. The trees generated by the first two tree path expressions have a common stem so they are merged into one, but the third tree specified by the third tree path expression does not overlap in the PIN with the previous one and so is not merged.

The set of trees specified by a query can also be dynamically generated by using a where-clause in a query definition. A where-clause allows the specification of one or more quantifications over sets or lists in a PIN. For example, the following query displays all the transition nodes labeled with their names.

```
ShowEnabledTokens(ps:PCMStructure) == {
           x, x.Name
           where x member-of ps.PN.N.T }
```

The query returns a set of trees, one for each value of `x` that was specified by the quantification in the where-clause. The path expression used in a where-clause is a standard path expression, not a tree path expression.

Filtering trees

Queries can be further extended to include a logical expression in a suchthat-clause which filters the trees based on the value of the logical expression for each value of a quantified variable. In the evaluation of the query

```
ShowEnabledTokens(ps:PCMStructure) == {
        x, x.Name
        where x member-of ps.PN.N.T
        suchthat x.Ta.TSte.Ste = ENABLED }
```

the branches of the tree which have enabled transition nodes only are kept, the others are omitted. The path expression `ps.PN.N.T` has already selected only those nodes which are transition nodes. The test in the suchthat -clause is used to keep the transition nodes which have a `State` attribute which has the value `ENABLED`. The logical expressions in the suchthat-clause are standard logical expressions except that the operands that can be variables (quantified or input) or path expressions which use these variables, or constants of the base or enumerated types. In this example the value of the enumerated type `State` of each transition node `x` is tested. The operators available in standard programming languages for the base types (integer, real, boolean, string) are also available in Glide.

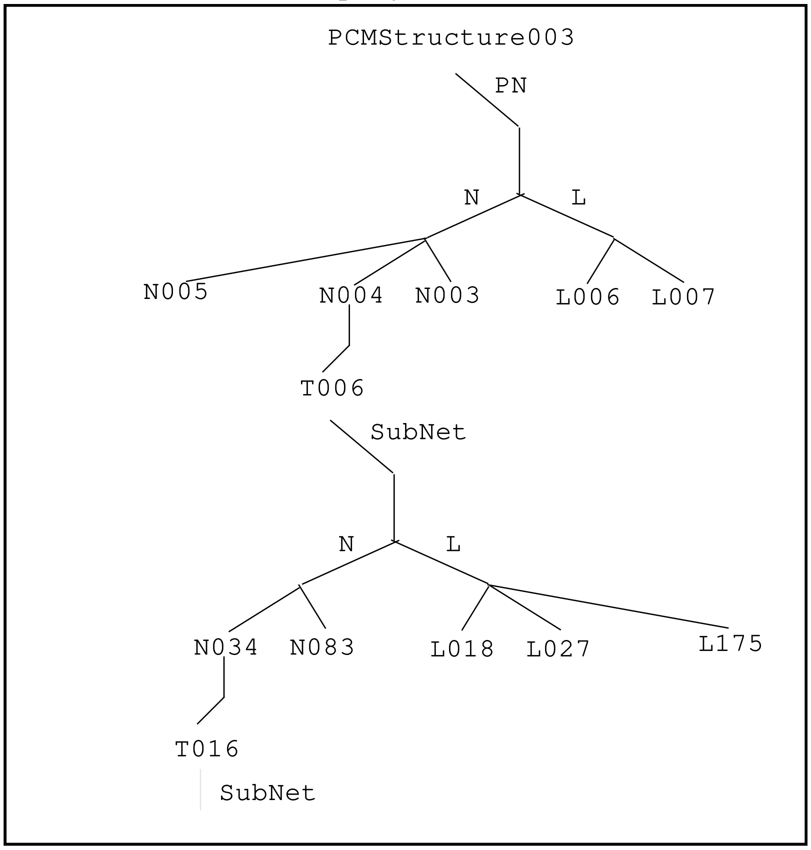
Recursion in tree path expressions

A tree path expression can include a recursive expression to denote the fact that a tree within the PIN should be traversed to its full depth. The annotation in a tree path expression for recursion is an "`@`" followed in parentheses by the path expression that links the levels of the recursion. For example, with the `PCMStructure` grammar listed above, the following query can be used to specify a single view which shows node and links at all the levels of the hierarchical PCM net.

```
ShowAll(ps:PCMStructure) == { ps.PN@(.N.T.SubNet).* }
```

The value preceding the recursion expression, in this case `ps.N`, is of type `PCM-Net`, and so are the objects found by following the path expression

`(.N.T.SubNet)` three levels down. Hence the objects `ps.PN.*`, `ps.PN.N.T.SubNet.*`, `ps.PN.N.T.SubNet.N.T.SubNet.*`, etc. are returned (in a single tree) as a result of this query:



Again, the way the resulting trees are eventually depicted is left to the graphical attributes and the view renderer. The query is only concerned with identifying components.

Recursion is also useful to specify that a complete piece of text should be displayed. For example, the PCM transition nodes have associated predicate expressions. The Glide style BNF syntax of PCM predicate expressions, is shown here (tags are omitted for clarity):

```
Pexpr    ==    Pterm | Pexpr 'OR' Pterm ;
Pterm    ==    Pfactor | Pterm 'AND' Pfactor ;
Pfactor  ==     '(' Pexpr ')' | Patom | 'true' | 'false' |
```

```
                'NOT' Pfactor;
Patom    ==     Aexpr '=' Aexpr | Aexpr '≠' Aexpr |
                Aexpr '≤' Aexpr | Aexpr '≥' Aexpr |
                Aexpr '>' Aexpr | Aexpr '<' Aexpr ;
Aexpr==         Aterm | Aexpr '+' Aterm | Aexpr '-' Aterm ;
Aterm ==        AFactor | Aterm 'MOD' Afactor |
                Aterm '/' Afactor | Aterm '*' Afactor ;
Afactor ==       '(' Aexpr ')' | VAR | NUM | '+' NUM | '-' NUM ;
```

In order to specify that a PCM predicate expression should be completely displayed, the following query can be used:

```
ShowTransNodePredicate(n:Node) = { n.T.Pi@(.*) }.
```

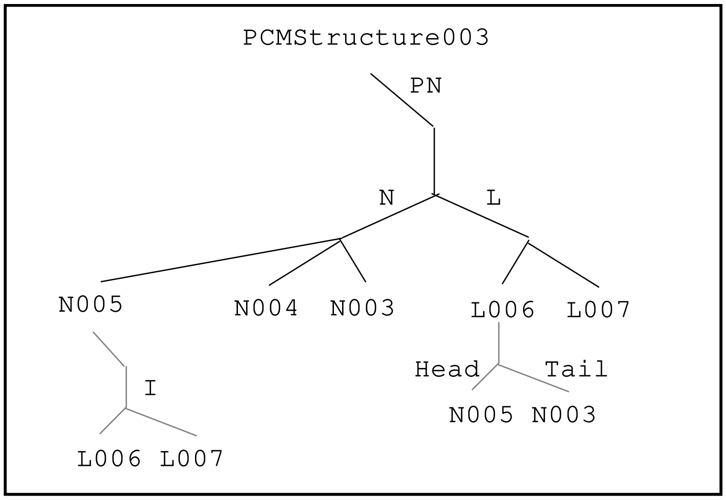
This query specifies recursing down and returning the entire syntax tree of the current PCM predicate expression (`Pi`) of the node `n`. The systems generated by Glider always represents such expressions by a syntax tree which is part of the PIN, but the Glide user has the option of specifying whether the GBVL (in this case, PCM) user should edit this tree with a structure editing interaction or allow the GBVL user to edit the expression as a piece of text which is parsed when the user has finished editing it.

Graph displays

A query can return a tree which, because of the cycles in the graph, contains replication of some instances. For example, the query

```
ShowGraph(ps:PCMStructure) == { ps.PN.*.* }
```

This specifies a tree, in which node replication implicitly describes a graph (part of the original PIN graph). The following diagram illustrates part of this tree, showing identifiers of some of the instances involved.



In this example the cycles exist because the node and link instances refer to each other. When this kind of a tree is passed to the view renderer, the implicit graph is recognized and it can be displayed as a connected graph; one in which PCM links and nodes are shown touching each other.

4.3.4 Glide Query Syntax

The following is a simplified abstract syntax of Glide queries which describes their essential structure. The path expression syntax, `<dotexpression>`, has already been defined in Section 4.2.1. `<const>` is an integer, real, boolean, or string. String and list equality can also be tested in suchthat -clauses.

<glidequery> ::= <querylhs> "==" "{" <queryrhs> "}"

<querylhs> ::= <queryname> "(" { <queryinputparams> } ")"

<queryname> ::= <identifier>

<queryinputparams> ::= <variable> ":" <typename>

<variable> ::= <identifier>

<typename> ::= <identifier>

<queryrhs> ::= <returnclause> [<suchthatclause>]

<returnclause> ::= <variableexpr> { ","<variableexpr> }

<variableexpr> ::= <variable> <treepathexpr>

<treepathexpr> ::= | "." <stagname> <treepathexpr>

```
                        | "." <stagname> "@(" <dotexpression> ")" <treepathexpr>
<stagname>   ::=        <identifier>
                        | "*"
<suchthatclause> ::=    "suchthat" <quantifications> [<whereclause>]
<quantifications> ::=   <quantification> { "," <quantifications }
<quantification> ::=    <variable> "member_of" <variable> <dotexpression>
                        | <variable> "=" <variable> <dotexpression>
<whereclause> ::=       "where" <logicalexpr>
<logicalexpr> ::=       <pexpr>
                        | <pexpr> <logop> <pexpr>
<logop>      ::=        "||" | "&&"
<pexpr>      ::=        "(" <pexpr> ")"
                        | "~" <pexpr>
                        | "true"
                        | "false"
                        | <patom>
<patom> ::=             <expr> <relop> <expr>
                        | <stringexpr> <stringrelop> <stringexpr>
                        | <listexpr> <listrelop> <listexpr>
<expr> ::=              "(" <expr> ")"
                        | <expr> <arithop> <expr>
                        | <eatom>
<arithop> ::=           "+" | "-" | "/" | "*"
<eatom> ::=             <variable> <dotexpression>
                        | <const>
<relop> ::=             "=" | "≠" | "≤" | "≥" | ">" | "<"
<stringexpr> ::=        <eatom>
<stringrelop>  ::=       "=" | "≠"
<listexpr> ::=          <variable> <dotexpression>
                        | "(" { <const> } ")" )
<listrelop>    ::=      "=" | "≠"
```

4.4 Glide Action Definitions

This section describes Glide *action definitions*. Just as query definitions are used to specify extraction of information out of a PIN, action definitions are used to specify updates to a PIN. Action definitions are used to capture different categories of activity, *i.e.* semantics, associated with a GBVL. The three main categories of activity are *editing*, *execution*, and *animation*, and a complete Glide specification of a GBVL contains a set of action definitions for each category.

A basic idea that influenced the design of Glide was the desire to provide a uniform way of specifying these different activities. This was achieved by incorporating data related to each one into a single structure and using a single notation for defining changes to data values. The Glide Grammar is thus used to specify a data structure which is *based* on the GBVL, but it is not *only* the language; it also contains other components. The syntax of the GBVL provides the organizing framework for extra components related to execution and animation. The different kinds of activity are characterized through action definitions which specify possible changes to instances of this structure (the PIN). The different categories of update usually (but not always) affect separate sets of components of the structure. Editing activity is the category of actions in which the user changes the PIN during program creation and modification. Execution and animation activity are the categories of actions which change the PIN at run-time. An action definition specifies an atomic set of changes to the PIN. The action definitions for the three categories share a single uniform notation, but each has a few additional constructs. The Glide model provides an integrated approach, in which all these activities are regarded as updates to one data structure. This not only simplifies the specification language but it also promotes the view that these activities are all interleaved and part of the overall activity of a user using a GBVL interface in order to develop programs.

This section introduces action definitions with examples of specifying editing actions (Section 4.4.1) and then shows how similar definitions are used to specify execution actions (Section 4.4.2). The description of how animation actions are specified is left until after the description graphical attributes (Section 4.6), since these are used to specify how changes to the values of graphical attributes are coupled to those changes in the PIN associated with execution.

4.4.1 Editing Actions

An action definition consists of an *action name* and zero or more typed *input parameters* on the LHS (same as queries) and a RHS that consists of a conjunctive set of *action expressions*. Like queries, action definitions make use of path expressions to navigate through the PIN and identify instances in it. An editing action definition relates the states of the PIN before and after an occurrence of the action (an *action event*).

Basic action expressions

A simple action expression is:

```
n.P.NmTk′ = 3
```

It specifies that the value of the number of tokens of a given node (`n`) which is a place node, is 3 after the occurrence of an action which contains this expression. In general, these simple action expressions have of the following syntax:

```
<actionexpression> ::= <pathexpression> “′” “=” <newvalue>
<newvalue> ::=  <constant>
                | “new” “(“ <type> “)”
                | <pathexpression>
                | <pathexpression> “∪” <pathexpression>
                | “lins” “(“ <pathexpression> “,” <pathexpression> “,” <index> “)”
```

As before, a path expression can be either just a variable (a trivial path expression) or a variable followed by a dotted sequence of tags (a variable followed by a *dot expression*). In an action expression, a path expression is annotated with a prime (′) to indicate its value in the *next* time step, *i.e.* after the event. The first form simply associates a constant value (integer, real, string or enumerate) with the place in the PIN specified by a path expression. In the second form the `new` function specifies the creation of a new instance of the specified type. In the third form the $\cup$ operator is the usual set union operator used to define a result which is the union of two existing sets or elements. The `lins` function specifies list insertion; it defines the result of an element or list inserted within another list after a specified index position.

Example - connecting objects

The following example editing action definition uses these simple action expressions to specify an update to the PIN in which a node and a link are created and are connected together.

```
AddNodeAndLink(dfw:DataFlowNetwork) == {
            n1 = new(Node) ∧
            l1 = new(Link) ∧
            n1.Input′ = l1 ∧
            l1.HeadNode′ = n1 ∧
            dfw.N′ = dfw.N ∪ n1 ∧
            dfw.L′ = dfw.L ∪ l1 }
```

The definition consists of a conjunction of six action expressions. The first and second specify the creation of instances of the types `Node` and `Link` in the `new` statements. The two local variables `n1` and `l1` are used to identify the new instances within the scope of the definition so they can be used in other action expressions. The fact that the two objects that are created are to be attached to each other is specified by the third and fourth expressions. Finally, the last two expressions specify that the new objects are to be attached to the existing PIN structure by adding them to the set of nodes (`N`) and links (`L`) of the dataflow network. The input variable `dfw` simply provides a starting point in the PIN for the path expressions such as `dfw.N` and `dfw.L` to identify places in the PIN that are to be changed. Note that the operation has *grown* the PIN, by both adding new instances and increasing the number of connections within it.

Semantics of action expressions

The semantics of *prime-equals* ('=) in an action expression superficially resembles that of assignment, but the two are very different. First, action expressions are not procedurally ordered; they are not imperative operations performed in sequence. Instead, the action definition is as a whole a declarative, atomic statement of the relationship between the state of the PIN before and after an event (an invocation of an editing action). Second, a prime-equals action expression represents a more general concept than assignment in that it specifies a connection operation *in the PIN*. It indicates how the references (pointers) between instances in the underlying PIN network are to be added, removed, or altered. A connection operation at the level of the PIN may be part of a visible action at the GBVL level dsiplayed in the user interface (e.g., attaching a node to a link, editing text tree structure, or setting a value). The assignment of a value is just a special case of such a connection operation. In other words, though the state of a PIN can be viewed as being in two parts: (i) the values of components which are terminal types (integers, strings, enumerated values, etc.) and (ii) the connection of component instances of non-terminals to each other (the topology of the PIN graph), the syntax of action expressions is such that it allows these two forms of change to be expressed in a uniform way.

The semantics of glide action definitions are thus close to the various graph grammar[4] languages for specifying graph rewriting (*i.e.* changes to a graph based on its current state). There are both textual and graphical graph grammar notations; Glide actions definitions is one in which the changes are expressed textually.

Editing vs. structure

Glide editing actions provide a *decoupling* of the creation of structure from its Glide grammar description; it is up to the Glide *user* to choose which

[4] The word "actions" was used instead of "grammar" so as not to confuse it with its use in "Glide Grammar".

actions are provided for his/her particular GBVL. For example, an editing action may create instances of several types and connect them together appropriately because he/she wishes this to be an atomic editing action for the *GBVL user*. It is quite possible to add nodes or links without connecting them to any other existing ones. For example, a single editing action could consist of adding `n1` as component of `dfw.N`, but not connecting it to any of the links in `dfw.L`. The action definition below is a further example which illustrates how to specify an editing operation that connects both ends of an existing unconnected link to two existing nodes:

```
ConnectLink(n1:Node,n2:Node,l1:Link) == {
            n1.Input’ = l1 ∧
            l1.Head’ = n1 ∧
            n2.Output’ = l1 ∧
            l1.Tail’ = n2 }
```

It is the decoupling that enables the needed flexibility to provide editing operations which are based on, but separate from, the basic structure of the GBVL embodied in its Glide grammar.

The Glider generator processes editing definitions in the same way as it does queries; the definitions are compiled into procedures and their names are added to a menu of editing commands in the interface generated by Glider, so that the user of the GBVL can access them to invoke the operation. The input arguments are bound by the user selecting the appropriate objects in existing views and then selecting the editing action from a command menu.

User input actions

There are two actions which prompt the user for information which is needed to complete the action, `newchoose` and `newparse`.

```
n.NT’ = newchoose(Node.NT)
```

The `newchoose` function denotes the fact that the user must choose which one of a set of alternate types is to be instantiated, or must provide the value of a terminal type. In this case the `NT` tag identifies an alternation in the Glide production of `Node`, an alternation which can contain either a `Placenode` or a `Transnode`. Typically, the prompt by the interface to the user would be through a dialog menu of all the possible types for the given alternation, and in the case of a primitive type (string, integer), a text dialog box to enter in a value for the primitive type with the keyboard.

An expression using the `newparse` function also denotes an interaction. It provides a means to instantiate a complete parse tree part of a PIN by prompting a user for a string which will then be parsed in accordance with the subset of the glide grammar types (productions) which specify a text tree. For example, the action expression,

```
n.Test' = newparse(BoolExpr)
```

might be used to prompt a user for a boolean expression to be associated with the component tagged `Test` of a node `n`.

Disconnection actions

The following action expressions are examples of disconnection actions:

```
p.T.Ta' = NULL

pn.L' = ps.L - l1
```

The first expression specifies that the instance that used to be at the place indicated by the primed path expression is no longer reachable in the PIN by following that path expression. The second expression is used to specify that an instance or set of instances is removed from a set or list. These expressions do *not* specify destroying the instance or instances found at the place identified by the path expression, only their disconnection. The syntax of these actions expressions is:

```
<actionexpression> ::= <pathexpression> "' " "=" <newvalue>
<newvalue> ::=   "NULL"
               | <pathexpr> "-" <pathexpr>
```

| “lrem” “(“ <pathexpression> “,” <index> “)”

Deletion actions

A separate deletion operation is needed for destroying instances, *i.e.*, for specifying that they no longer exist in a PIN after an action event. This operation is `old`, it removes one or more instances from the PIN, and is the inverse of the `new` operation.

```
old(p.Ta)
```

This operation destroys the instance found at `p.Ta`. The object that was at this position no longer exists in the time step after an action event that includes this expression. At the PIN level, all references to a deleted instance are removed, so that at the GBVL level a shared object disappears from the many places it used to be. The `old` operation can take a path expression or a tree path expression as argument, to specify the deletion of many instances at once.

Conditional action expressions

Actions expressions can be made conditional with a boolean expression which acts as a guard.

<ConditionalActionExpression >
::= “(“ <BoolExpression> ”)” “⇒” “(“ <ActionExpressionsConjunction> ”)”

The syntax and semantics of these boolean expressions is the same as that for the suchthat-clause of Glide queries. The following example illustrates its use in specifying an action that only completes if the node is a place node.

```
(n.P.NmTk = 3) => (n.P.NmTk’ = 4)
```

Conditional expressions can be used to differentiate between the types of components that could be present, in a tag which identifies an alternation. For example:

```
(n.NT = n.P) => (n1.P.NmTk’ = 4)

(n.NT = n.T) => (n1.T.Ste’ = ENABLED)
```

This completes the description of editing action definitions. The following section describes the use of action definitions for modeling execution.

4.4.2 Execution Semantics

The purpose of an execution semantics specification in Glide is to represent, at some level of abstraction, a description of the execution behavior of a GBVL. This description captures the dynamic relationships that exist during execution between components specified in a Glide grammar. As was mentioned at the beginning of this section, a key idea embodied in the Glide model is that the values associated with the execution or evaluation of components of a program (*e.g.* arithmetic expressions, logical expressions, rule firings, flows of data, etc.) are represented by adding further components to the structure of a Glide grammar. All actions specify changes to the PIN. The original structure of the language provides the framework within which to add components whose values represent the execution state of the system. A *structure-oriented* organization for specifying execution semantics is thus used in the Glide approach.

Execution action definitions are intended to provide a way to describe an *abstracted* execution semantics. This is a description of the changes that occur during execution at a level chosen by the GBVL designer. By adding more detail to a Glide grammar specification, a finer grained and lower level of description of the execution semantics can be made. For example, three successive levels of description in GBVL might be: (i) the binding of values of variables inside rules associated with nodes, (ii) whether or not the rule conditions are satisfied, (i) whether or not a node was executed. The abstract execution can either be driven by the interface itself, to give the user (animated) views of the execution of programs, or it can be coupled to the execution of an actual program that was generated by extracting a description of the program from the PIN and passing it to a compiler for the particular GBVL. This compiler is provided by the GBVL designer (it is not part of Glider) and must be able to instrument the program so that it can report and confirm expected state changes back to the interface. Though it might be possible to describe a complete execution semantics of simple GBVLs in Glide, in general this is not the case - many GBVLs have very complex semantics, which often include the invocation of routines written in existing standard languages such as C or Fortran.

As mentioned in Chapter 2, in some GBVLs the program graph is altered only during the development (editing) of the program and it is static during execution. Execution of the static graph involves the movement of data or state through the nodes and links. This form of execution is modeled by execution action definitions which only change terminal values in the PIN. Other GBVLs however are dynamic; their graph topologies change at run-time. Such GBVLs can be captured in a simple way with Glide because of the uniform approach in Glide to specifying (i) editing and execution *and* (ii) value and topology change. In some cases the same change can be both part of execution actions and part of user editing actions - such as the setting of the number of tokens in Petri Net place nodes.

Execution action definitions are similar to editing actions except that they are in the form of *condition⇒action* rules which can be quantified over sets or lists of components in the PIN. These rules test the state of a PIN and then change it if their condition is satisfied. The set of rules are compiled into a set of corresponding procedures. These procedures are invoked by an underlying run-time execution engine which chooses satisfied rules to fire according to one of several policies specified by the user, called *regimes*. Execution action expressions include all the expressions used for editing except those that specify user input (`newchoose`, `newparse`).

Example - Transition node state update

In the following simple Petri Net grammar:

```
PetriNet ==     N: Node** .
                L: Link** ;
Node ==         (P: PlaceNode | T: TransNode) .
```

```
                    I: Link** .
                    O: Link** ;
      PlaceNode ==  NmTk: INTEGER ;
      TransNode ==  Name: STRING .
                    St: State ;
      State ==  Ste:( :ENABLED | :FIRING | :DISABLED | :ACTIVE );
      Link ==       Lp:PlaceNode .
                    Lt:PlaceNode ;
```

The following action definition is part of the execution semantics of Petri Nets.

```
Enable(pn:PetriNet) == {
      ∀t:pn.N.T (∀p:t..NT.I.Lp (p.NmTk >0) ⇒ t.Ste′ = ENABLED )
}
```

It states that all the transition nodes of the net (`pn`) that have at least one token in all the place nodes which are input-linked to them (`..NT.I.Lp`) should change their state to `ENABLED`. The "`∀t:pn.N.T`" expresses quantifying over the set of transition nodes within the PIN and "`∀p:t..NT.I.Lp`" expresses quantifying over all the place nodes which are inputs to a given transition node (`t`). Since execution rules are invoked by the interface itself (rather than the user), execution action definitions have a single input parameter which is bound to the whole PIN.

The rule for firing a transition node is:

```
Fire(pn:PetriNet) == {
      ∃t:pn.N.T (t.Ste = ENABLED ⇒
                    t.Ste′ = FIRING ∧
                    ∀p:t..NT.O.Lp (p.NmTk′ = p.NmTk + 1)
                )
}
```

This action definition states that one of the transition nodes that is in the state `FIRING` be chosen and that the number of tokens in all of the place nodes that are output-linked to it (`∀p:t..NT.O.Lp`) should have their number of tokens incremented. The restricted existential quantifier denotes the non-deterministic choice of a `t` which satisfies the condition.

Example - value update

Execution action definitions can be used to specify the computing of values associated with expressions. For example, in the following simple productions of a GBVL which contains nodes that compute an integer value:

```
Node ==          NodeLabel:String .
                 NodeValue:Aexpr ;
Aexpr ==         Val:INTEGER .
                 Exp:(Var '+' Var) ;
Var ==           Val:INTEGER .
                 Name:STRING ;
```

the grammar represents both the expression (`Aexpr.Exp`) and its value (`Aexpr.Val`), as two parts of a component. The following unconditional execution action rule captures the execution semantics of evaluating the expression by updating the values of all these expressions:

```
ComputeAexprVal(dfw:DataFlow) == {
    ∀a:dfw.N.Aexpr (a.Val' = a.Exp.0.Val + a.Exp.2.Val)
}
```

`dfw.N.Aexpr` specifies all arithmetic expressions of all the nodes. Index tags 0 and 2 are used for identifying items in a sequence. The grammar holds *both* the expression *and* the value it computes. From the interface point of view these are both displayable objects. The expression is accessible during editing, and its value is accessible for inspection during execution (and possibly also for the user to modify during execution). This approach is different from representing values of expressions as *attributes*, as exists, for example, in *attribute grammar* representations for specifying language semantics. The reason for this integrated approach is because, in this application of generating programming interfaces, it is necessary to make the state of the program visible and accessible in the interface in the same way as the program itself.

Example - data flow

The following simple example illustrates how the semantics of data flow type GBVLs can characterized with action definitions that specify the movement of data values along nodes and links. For a grammar:

```
DataFlowNet ==    N:Node** .
                  A:Arc** ;
Node ==           Name:INTEGER .
                  Val:INTEGER .
                  Operation:Expr .
                  I:Arc .
                  O:Arc ;
Arc ==            I:Node .
                  O:Node .
                  TransVal:INTEGER ;
```

the following action definition expresses the movement of the value into the arc and out of it:

```
MoveValIn(dfw:DataFlow) == {
      ∀n:dfw.N (n.O.Val′ = n.Val ∧ n.Val′ = 0)
      }
MoveValOut(dfw:DataFlow) == {
      ∀n:dfw.N (n.Val′ = n.I.Val ∧ n.I.Val′ = 0)
      }
```

Here 0 is being as used to record the absence of a value. If needed, an alternation construct could be used so that the value in the `Val` slot could contain either an integer or “ABSENT”. More complex forms of flow, with a list of values in the link and/or node can be used to represent queues, LIFOs, etc.

Execution firing regimes

Execution action definitions are similar to editing action definitions, but their invocation is not driven by the user, it is either driven internally by Glider's rule execution engine or is triggered by the execution of the external compiled, instrumented program. In the former case, the collection of condition-action rules is run under some regime suitable for the particular model of computation of the GBVL. The basic firing regimes provided are the following:

- `round_robin` - Of the all the action rules matched, fire each in turn in the order they were originally declared (in the Glide specification), repeat.
- `first_and_restart` - Of the all the action rules matched, fire the first one of the order they were originally declared, repeat.
- `random_and_restart` - Of the all the action rules matched, pick one at random to fire, repeat.

These regimes are similar to different firing strategies in "rule-based systems" (*e.g.,* [OPS5-85]). They provide a simple and transparent underlying control for execution that ensures that the step-by-step logic of execution remains easy to follow for the user. The firing rules themselves provide the GBVL designer the means by which to implement more complex control logic for his/her particular GBVL.

This completes the description of the use of actions for editing and execution. Similar actions which also access graphical attributes, *animation actions*, will be described in Section 4.7.

4.4.3 Glide Action Syntax

The following abstract syntax captures the essential elements of the syntax of Glide actions. Definitions of non-terminals previously defined in the query syntax (Section 4.3.4) are omitted, for example, the conditions in conditional action execution rules are the same as that for queries.

Common

<actiondefinition> ::= <actionlhs> "==" "{" <actionrhs> "}"

<actionlhs> ::= <actionname> “(” { <actioninputparams> } “)”

<actioninputparams> ::= <variable> “:” <typename>

<actionname> ::= <identifier>

<actionrhs> ::= [<rquantifiers> “(“] <conjunctiveexpr> [“)”]

<conjunctiveexpr> ::= <conjunctexpr> { ”∧” <conjunctexpr> }

<conjunctexpr> ::= [<rquantifiers> “(“] <compactionexpr> [“)”]

<compactionexpr> ::= <simpleactionexpr>

| <condaction>

<condaction > ::= [<rquantifiers>] “(“ <logicalexpr> ”)” “⇒” “(“ <conjunctiveexpr> ”)”

<actionexpr> ::= <newvalexpr>

| <destructionexpr>

<creationexpr> ::= <pathexpr> “′ ” “=” <newvalue>

<newvalue ::= <addnewvalue>

| <remnewvalue>

| <ednewvalue>

<addnewvalue> ::= <constant>

| “new” “(“ <type> “)”

| <pathexpr>

| <pathexpr> “∪” <pathexpression>

| “lins” “(“ <pathexpr> “,” <pathexpr> “,” <index> “)”

<remnewvalue> ::= “NULL”

| <pathexpr> “-” <pathexpr>

| “lrem” “(“ <pathexpr> “,” <index> “)”

<destructionexpr> ::= “old “(“ <pathexpr> “) “

| “old “(“ <treepathexpr> “) “

Editing specific

<ednewvalue> ::= “newchoose” “(“ <typename> “)’”

| “newparse” “(“ <typename> “‘)”

4.5 Glide Shape Predicates

This section describes Glide *shape predicates*. Shape predicates provide a means of specifying those invariant structural properties of the PINs of a given GBVL which cannot be captured syntactically with the glide grammar. Foremost among these static semantics are restrictions on sharing and cycles that can exist in a PIN. Shape predicates are used to express these restrictions and cause updates to the PIN to ensure that the invariant properties are maintained. This section provides examples of shape predicates to show how they are expressed and used. The next chapter discusses this Glide approach to handling sharing and cycles in greater depth and compares the approach to other existing techniques addressing the same issue in the context of other problem domains.

As discussed earlier in Section 4.1, the Glide grammar type specifications are unconventional in that shared structures and cyclic structures such as instances which “contain” themselves are permitted. This latitude provides an elegant, concise, and uniform way of describing the interrelationship of objects in a GBVL, but there is a price to paid for this elegance. Though the type specification does identify the desired PINs, it also admits instance networks that do not correspond to a program (either partial or complete). For example, the following diagram illustrates a Petri Net as a collection of type instances and the references between them (as small arrows, →).

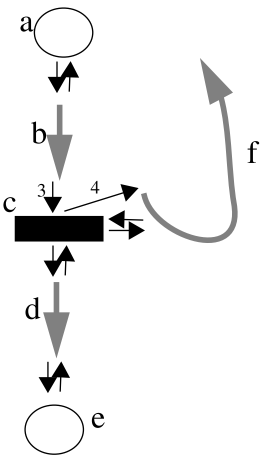


Figure 4-4 Incorrect Instance Network

Though this PIN does not violate any of the type compositions prescribed by the glide grammar productions, it should clearly be excluded because references 3 and 4 are not correctly “paired”. It is thus necessary to provide an external means to characterize the particular sharing and cycles that are admissable and separate them from those which are not. Shape predicates are the means to express these constraints in Glide. Shape predicates make use of path expressions to identify the desired cases of cycles and sharing and distinguish them from the undesired ones.

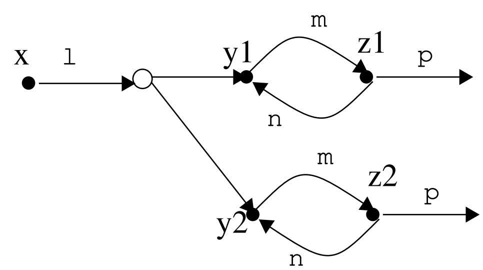
Examples

The following three examples illustrate how shape predicates can be used to characterize patterns of cycles and sharing in PINs.

Graphical illustrations of parts of PINs are used to illustrate each pattern. In these illustrations, a full circle (●) represents some instance of a type (hence its out-degree, corresponding to the number of tagged components of the type, is fixed). The labelled arrows emanating from full circles represent the tagged references of instances to each other. An arrow pointing to a hollow circle (○) represents a set or list component of a type (hence the circle’s out-degree is not fixed). Elements of a list or set are labelled by the quantified variable used in the shape predicate, and so are subscripted with integers to differentiate each value.

(i) Simple Fixed - A simple pattern of cycles occurs when instances two types always point to each other:

```
X ==   l:Y** ;
Y ==   m:Z ;
Z ==   n:Y .
       p:W ;
```

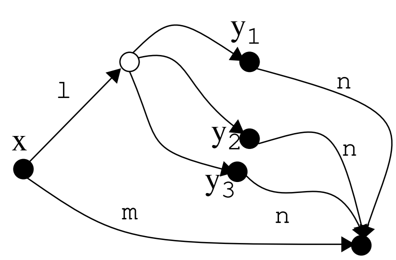


Shape Predicate: `SimpleCycle(x:X) =={∀y:x.l (y.m.n = y) }`

The shape predicate is captures the restriction that only the simple cycles in which the instance of `a Z` that is referred to by a `Y`, refers back to that instance of `Z`. Hence an instance network such as the one above except that `y1` referred to `z2` and `z1` to referred `y2` would be inadmissable. Note that this invariant holds only for the instances of types `Y` and `Z` referred to by `X`. It may well be that types `Y and Z` elsewhere in the PIN do not have this constraint.

(ii) Single Collection - In this example instances of a set of one type refer to the *same* instance of another type:

```
X ==   l:Y** .
       m:Z ;
Y ==   n:Z ;
```

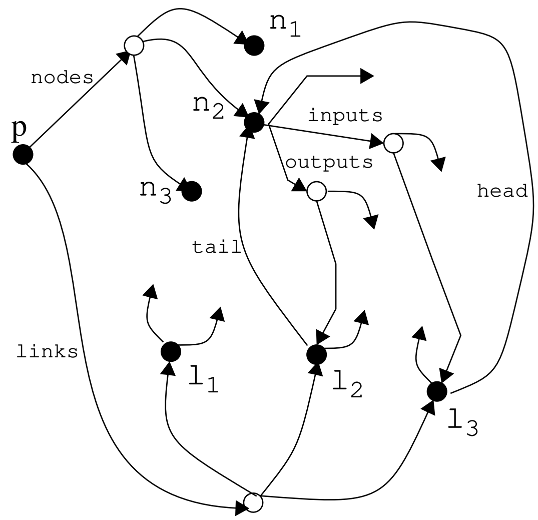


Shape Predicate: `SameSink(x:X) =={∀y:x.l (x.m = y.n )}`

In this example the shape predicate is not describing a cycle. This part of the PIN is only a DAG.

(ii) Triple Collection - This example is a simplified case of the one needed for the Glide Grammar descriptions Petri Nets used in previous sections.

```
P ==  nodes:N** .
      links:L** ;
N ==  inputs:L** .
      outputs:L** ;
L ==  tail:N .
      head:N ;
```



Shape Predicate:

`Paired(p:P)=={∀n:p.nodes,l:p.links(l∈n.inputs ⇔ n=l.head)}`

The predicate involves 3 sets: nodes of a net, links of a net, and input links of a node. It captures the pairing pattern mentioned in the introduction. It is also similar to example (i) except that part of the cyclic path goes through a list and hence there is a membership test in the predicate. For clarity, the diagram is not complete, only cycles involving just node n_2 with links l_2 (inputs) and l_3 (outputs) are shown. A similar predicate is needed to capture the cycles going through `outputs`.

The syntax and semantics of shape predicates are similar to actions. They are also compiled into procedures. Operationally, shape predicates can be used in many ways: They may be provided as interface commands and applied at the users discretion to verify that the invariant holds. They may also be triggered after each editing operation to flag an incorrect state of the PIN, or correct it automatically. It is possible to simplify the specification of many of the editing operations by using shape predicates to complete the references between instances. Shape predicates can also be used to specify simple semantic checks for the GBVL.

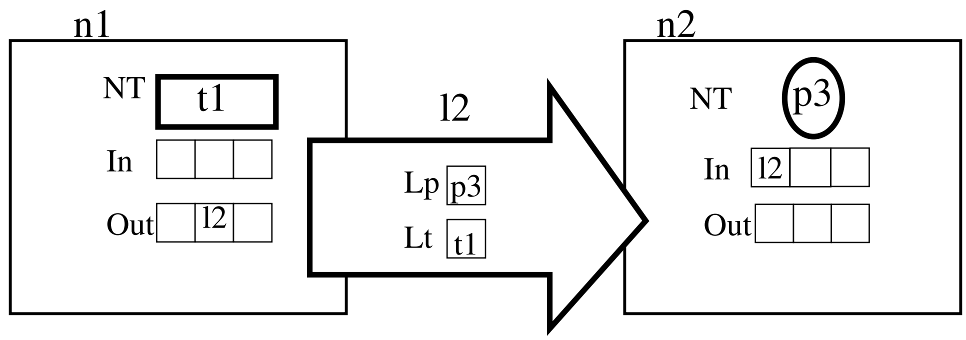
The predicates shown in these examples are simple, but GBVLs with more complex forms of interconnection structure, such as ones with nodes which have many ports, or ones with hyperlinks may require more complex shape predicates. The path expression-based scheme of these predicates allows these shapes to be specified in a straightforward and concise way.

Design variations - bipartite vs. input-output.

The grammar and the shape predicates together specify the set of allowable PIN graphs for a given GBVL. As is often the case in specifying languages, there is a design choice as to which aspect of a GBVL structure to capture directly in the grammar and which to capture as static semantics of a shape predicate. The following two variations of representing Petri Nets illustrate this issue. A Petri Net is both a directed graph and a bipartite graph; all links connect either a place node to a transition node or a transition node to a place node. This can be directly encoded in the grammar by representing links as objects which are connected to one place node and one transition node, as shown below.

```
PetriNet ==     N:Node** .
                L:Link** ;
Node ==         NT:(P:PlaceNode | T:TransNode) .
                I:Link** .
                O:Link** ;
Link ==         Lp:PlaceNode .
                Lt:TransNode ;
```

A graphical illustration of three instances conforming to this grammar is the following:

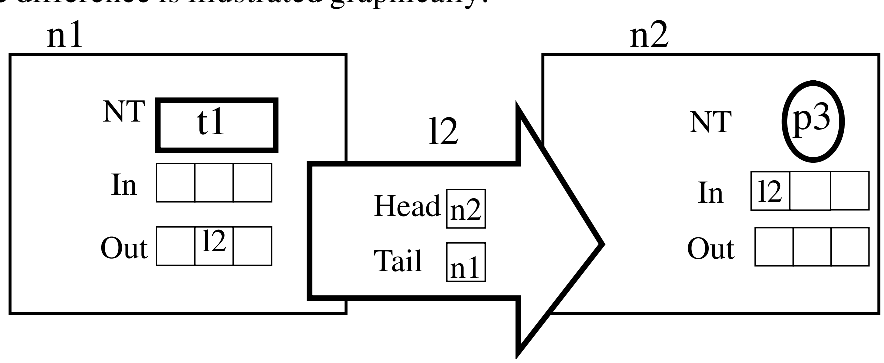


In this grammar, the bipartite property of the net is captured directly in the representation of the link, but direction of the link is not represented in the link itself. Given a link, its direction can only be deduced indirectly, from the information in the nodes the link is attached to.

An alternative representation is to capture the input-output relationship directly in the link and let the bipartite nature of the net be implicit, by changing the definition of `Link` in the grammar to:

```
Link ==          Head:Node .
                 Tail:Node;
```

The difference is illustrated graphically:



The bipartite property of the net can then be captured by using a shape predicate to state the that a link always has one node that is a place and one node that is a transition:

```
Bipart(pn:PetriNet) == {
      ∀l:pn.L ((l.Head.NT = l.Head.P ⇔ l.Tail.NT = l.Tail.T)
             ⊕  (l.Head.NT = l.Head.T ⇔ l.Tail.NT = l.Tail.P)
}
```

(⊕ - exclusive or) These predicates use the Glide idiom of comparing the equality of path expressions values (`.NT` and `.P`) to identify the type in alternation (as was already illustrated earlier in this chapter).

Both representations can be used; it is up to the designer as to how partition representing properties of the language between the grammar and the predicates. The effect of the of this partitioning choice on editing interaction is one aspect that needs to be taken into account. By moving the "bipartitedness" out of the grammar and into a shape predicate, editing operations can be provided in which the user is free to put a link between two place nodes. Though this may create an incorrect structure (which will subsequently be flagged by the verification with the `Bipart` shape predicate) it is often the case that this "looser" form of interaction is preferred by some classes of user. The key advantage of the Glide model is that it provides the flexibility of being able support *many* choices of design. Indeed, in Glide it is quite possible to have both properties aspects represented explicitly,

```
      Link ==         Head:Node .
                      Tail:Node .
                      Lp:PlaceNode .
                      Lt:TransNode ;
```

and use shape predicates to enforce consistency. The design principles should be the ease and clarity with which the semantics can be expressed and the form of editing interaction that is desired. It is often the case that the more that is captured in the grammar, the easier it is to express semantics, but the complex shape predicates may also be needed.

4.5.1 Glide Shape Predicates Syntax

Shape predicate syntax is the same as that for action specification except that that bi-directional implication (iff) can be used.

4.6 Glide Graphical Attribute Definitions

This section describes the *graphical attributes* component of Glide. Up to this point the description of Glide has dealt with: (i) the grammar and the actions, which provide a complete but abstract logical description of the structure and semantics of GBVLs; and (ii) the queries, which allow the designer to specify *which* parts of a structure to display in a particular view. The purpose of the graphical attributes component of Glide is to specify the graphical appearance of these parts as they are displayed in views. The next section (4.7) will describe how animation specifications can couple changes in the values of these graphical attributes with execution.

The main design concepts behind the way graphical attributes are specified and used in Glide are the following:

(i) There is close correspondence between what appears on the screen and the underlying data structure specified with the Glide grammar. The interface directly exposes the data model for reasons that were already discussed in the introduction chapter. Because of this approach, graphical attributes are simply attributes of the types and tags specified in the Glide grammar. Each type or tag can have one or more graphical attributes which indicate how instances of a given type are to appear on the screen or how the tag labelling an instance should appear.

(ii) The Glide graphical attributes are *hierarchical*. The syntax of graphical attributes is recursive. Attributes have a name and a list of values, and a value can itself be an attribute name and a list of values. Thus specific graphical attributes may contain arbitrary levels of further specification detail.

(iii) The use of graphical attributes can be *progressive*. As graphical attributes are added, or more levels of specification are added to a given attribute, the views produced by the run-time rendering system will be progressively more distinctive in graphical appearance. Indeed, if *no* graphical attributes are provided in a Glide specification, the rendering system will still be able to produce a default generic display, based solely on the structural description in the grammar. As attributes are added, more of this default appearance will be superseded with the specified graphical appearance.

(iv) Many of the graphical attributes are taken *directly* from the graphical capabilities of the GUI library that supports the interface. There are still quite a few GUI libraries (or *toolkits*) in existence and the capabilities of these libraries are still evolving. Rather than providing a static generic set of attributes which would reduce the possible displays to the lowest common denominator of these libraries, and fail to make available their particular capabilities, the attributes reflect the capabilities of a GUI directly. An advantage of this approach is that graphical attributes are "open", allowing more attributes to be added when underlying facilties GUI support them. A disadvantage of this approach is that if a new underlying GUI is targeted to support the Glider generated interfaces, parts of this component of a Glide specification would need to change to match the new GUI. However the changes are limited to just this component of a Glide specification and in practice there is a large overlap in the kinds of graphical primitives provided the various GUIs. In this chapter specific attributes of a particular toolkit are used[5]. This approach also means that graphical attributes are more concrete than the those of the preceding sections; specific names of shapes, colors, widths and patterns of links, etc. are typical.

(v) The Glide model uses a composition-oriented approach to capturing commonality between types. Hence, as has been discussed earlier, a `Node` type can consist of a collections of types, one of which is an alternation of `transition` or `place`. In order to accommodate this composition-oriented approach, graphical attributes can contain path expressions which provide a means to reflect the graphical properties of the subobject (in this case, `transition` or `place`) in the superobject (in this case `Node`).

---

5 In this case the widget properties of the Tk API of the Tcl/Tk GUI toolkit [Oust94].

Operationally, the values of graphical attributes are combined with the result of a query (query trees) and passed as more complex data structure (the glider *display tree*) to the run-time rendering module. The latter interprets this information and performs the appropriate invocations of GUI library calls to produce the view on the screen. This process is analogous to the use of “display lists” (which are really trees) in classical graphical rendering models (e.g. PHIGS). This process will be described in detail in the Chapter 6 on the Implementation of Glider.

Default display

If no graphical attributes are provided, the renderer will produce a default display of a PIN based only on its type structure specified in the grammar. There are in fact two forms of default display, *textual* and *graphical*:

In the case of the textual display (when no GUI whatsoever is available), the program will be displayed as a large list of records structured according to the individual grammar productions and will contain the internal system-generated identifiers for all the instances in the PIN. These identifiers are also then used for all the references between instances. This display is in effect a textual display of the PIN. The very large number of identifiers used to encode references makes this form of display difficult read and understand, and it is the fact that graphical approaches display these relationships by graphical adjacency and nesting that can make them so much more concise and effective.

The default graphical mode uses *nesting* of boxes according to the structure of the grammar to show the same information. Each box is labelled with the type name of the instance and the relevant tag. Shared objects will appear repeated at the multiple places they are shared. As graphical attributes are added to the Glide specification these boxes for instances of different types will instead have their own specific shapes, colors, and icons, and sharing will be displayed as objects adjacent to each other (*e.g.* node and links touching each other).

Simple example

Graphical Attributes are specified as hierarchically nested lists with the keyword “`Appearance`”. For example:

```
Appearance(PetriNet) == {Box     {BackgroundColor Red}
                                 {Border
                                      {Size 3}
                                      {Color Blue}
                                 }
                        }
```

This is a simple example of hierarchical specification. It states that instances of the type `PetriNet` should be displayed as a red box with 3 pixel blue border. Several `Appearance(PetriNet)== ...` definitions might appear in a single Glide specification, each would simply be added to the list of graphical attributes for the type `PetriNet`. Tag graphical attributes are specified by including a tag name after the type, e.g.:

```
Appearance(PetriNet.Name) = {Text {Font Helvetica)}
```

4.6.1 List Graphical Attributes

The following is the list of the graphical attributes and their meanings. An attribute without choice of values simply indicates it is boolean set to true. Graphical attributes listed here can be nested when appropriate (e.g. `color` inside `text`)

- `Terminal` - Used to indicate the leaf of a display tree - default to a box with the name of the type unless further attributes are provdided.
- `Text` - Used to indicate that a terminal should appear as editable text.
- `Texttree` - Used to indicate that portion of a PIN is a parse tree which should be displayed as a string of text obtained by putting all the leaf values of the tree into a single text string
- `GRAPH` - Used to indicate that compoents of this type can be displayed connected together (*e.g.,* `Node**` and `Link**` as components of `PetriNet`) This graphical attribute requires additional information which is basically

the same as that which is embedded in the shape predicate, i.e. the paths by which the objects are shared.

- `Port` Indicate that this component is a port of a node and should be displayed at the on the edge of a node or inside the node and links can be show connected to the port. Nested attributes: `(IN ON)` to indicate whether the ports are displayed inside the perimeter of the node or outside touching.
- `Shape` Indicates shape of object

  Nested attributes: `Arc, Bitmap, Line, Oval, Polygon, Rectangle`
- `ForegroundColor Color`
- `BackgroundColor Color`
- `Border` Properties of shape borders

  Nested attributes: `Width, Color`
- `Icon` - If no futher detail is being shown, the bitmap is used, iconically indicating the type of the object (*e.g.* firing rule, dataflownode, token, etc.). If more detail is being shown, the bitmap is used as a title of the box.
- `Line`

  Nested attributes: `(Straight Manhattan SplineCurved, Directed) Width, Color`
- `Arrow`

  Nested attributes: `Shape, Size, Color`
- `Shading`
- `Labelling (Inside, Beside, Above, Below)`
- `Font`

  Nested attributes: `Size, Color, Family, Style)`

Note that the `SIZE` attribute is only ever used for fonts. This is because of the nested display approach to displaying objects. As more detail is presented the size of the enclosing object will grow, and hence it is not necessary to assign specific sizes.

4.6.2 Glide Graphical Attributes Syntax

<graphicalttributedef> ::=

“Appearnce” “(“ <type name> “)” “=” “{“ <graphicalattribute> “}”

| “Appearnce” “(“ <type name>”.”<tagname> “)” “=” “{“ <graphicalattribute> “}”

<graphicalattribute> ::= <graphicalattributename> <graphicalattributevalue>

<graphicalattributename> ::= <gattributeidentifier>

<graphicalattributevalue> ::= <identifier>

| <constant>

| “{“ <graphicalattribute> “}”

4.7 Glide Animation Definitions

The final component of a Glide specification is a set of *animation definitions*. The purpose of animations is to provide the GBVL user with useful dynamic graphics. These dynamic graphics can be used to convey the flow and control activity of an executing program, highlighting where and when changes are taking place so that a user can more quickly comprehend (i) the execution semantics of the language and (ii) whether or not the program that the user has created is matching the users specific expectations about its execution. Animations can be a very effective means of diagnosing problems because anomalous behaviour can be made instantly apparent with the right animation.

The basic means by which animations are specified with Glide is by rules which are similar in form to execution actions. The only difference is that they relate particular tests on the state of the PIN structure to changing the values of graphical attributes. In other words they specify invariant relationships between PIN components and the values of graphical attributes. In this way graphical features just described, such as the color, border size, font, shape, will change in response to changing PIN state. The same test syntax as execution rules is used, so that graphical changes can be made to occur only when very specific conditions on the state of the PIN occur. In addition, animation conditions can also trigger on specific *changes* of state of the PIN between adjacent steps in the execution. Finally, it is also possible for animation, to trigger *active* graphical attributes. These are simply graphical attributes which are in fact procedures which execute in a given state in order to create a dynamic animation of a given state (*e.g.* the cyclic movement of a value along an arc, to illustrate that the system has reached the state where the value has arrived) .

Simple example

Simple examples of animation are the following:

```
EnableGreen(pn:PetriNet) ==
      { ∀t:pn.N.T (t.Ste = ENABLED ) ⇒ (t..NT<Color> = Green)}


FiringRed(pn:PetriNet) ==
      { ∀t:pn.N.T (t.Ste = FIRING) ⇒ (t..NT<Color> = Red)}
```

The graphical attributes are accessed via a special form of path expression terminating with angle brackets enclosing the particular graphical attribute. The enclosed expression may itself be a dot expression to indicate accessing a nested graphical attribute.

More complex example

The conditons in the invariants can be made as complex as desired, so that very specific animation can be used to detect and indicate and a very specific condition. Though at the moment these animation invariants are designed to only be created and specified by the designer, opening such facilities to the user is quite straightforward.

```
    {∀t:pn.N.T (t.Ste = FIRING ∧ t.input.Lp.NmTk < 2 )⇒
                (t..NT<Color> = Red)
      }
```

This animation definition specifies that transition nodes should trun red when they are in the state `FIRING` and all their input place nodes have less than two tokens.

Transition animation

In addition to producing graphical reflections of the state of the PIN, It is also possible to create animations of *change* of state in the PIN.

```
      { ∀t:pn.N.T (t.Ste = FIRING) ∧ (t.Ste′ = DISABLED)
          ⇒ (t..NT<Color>′ = Red)}
```

This animation will cause transition nodes which have fired and become disabled to turn red.

Animation procedures

In addition to the tying a graphical attribute to the value in the PIN (or, more generally, the state of the the PIN), it is also possible to register attributes which are procedures. This is similar to the path transition paradigm of [XTANGO]. Such animation provides no more information that simply having a color valuereflect the state, but the activity is does provide a more noticeable effect which can be used to make some animations more obvious than others.

```
{ ∀t:pn.N.T (t.Ste = FIRING) ⇒ (t..NT<Flash>)}
```

This example invokes the graphical attribute which is a routine which flashes the node momentarily. The semantics of these active animations are blocking ones; the execution of the program itself stops, the animation procedures runs to completion before the next step of program execution.

4.8 Summary

This chapter has described the Glide model. It has described how it is based on an extended notion of composition of types which accepts and exploits cycles and sharing to encode a concise and rich description of the components graph-based visual language and their inter-relationship. Building on this underlying data structure specification component and path expressions to navigate through such structures, Glide provides queries for extraction, actions for capturing semantics, graphical attributes for specifying graphical rendering of the structure, and animation definitions for visualizing execution.

The Glide query language is a query language which provides a succinct and simple way to extract parts of a program from the PIN that represents the whole program. The queries allow the Glide user to specify both what objects should be displayed in a view and at what level of detail those objects should be shown. The query language is one that is adapted to the Glide data model for graph-based visual languages.

The various forms of Glide action definitions provide a unified notation for describing changes associated with the language. Simple forms of action just test and change terminal values associated with objects (*e.g.* firing state, number of tokens, etc.). More complex forms change the configuration of the graph structure of the PIN itself, gluing or ungluing objects. Execution semantics are encoded as rules that access and test the PIN and then alter it.

The appearance is derived by a generic display generator in conjunction with attributes which can progressively refine the graphical apperance of instances of given types.

The next chapter provides a more detailed analysis of the issues surrounding the use of shared and cyclic types. Chapter 6 describes the implementation of the Glider compilation process and how it translates a complete Glide specification in order to generate and interface for a GBVL.

Chapter 5

Graph Types for Graph Based Visual Languages

This chapter provides a deeper analysis of the Glide data model by relating it to work in a diverse range of other research areas within each of which the issue of recursive and cyclic data types has also been addressed. The areas are: functional languages, abstract data types, data base models, object-oriented data models, formalization of pointers, compiler data structure dependency analysis, and graph grammars. The need to address the issue arises for a variety of reasons in each of these areas. In the case of Glide the need arises from wishing to support the "interconnection paradigm" of GBVLs.

This chapter first examines and reviews the use of these special data types at the lower level of abstraction of *imperative languages and pointers* (Section 5.1) (this has already been partially discussed in Section 4.1, the Glide Grammar). It then examines the higher level approaches to characterizing these types that are found in the *formal and declarative languages* used in the areas mentioned (Sections 5.2 through 5.5), so that they can be compared to Glide. Finally, in Section 5.6, an illustration of why this issue is so closely related to the connection-based paradigm of GBVLs is provided.

5.1 Data Types in Imperative Languages with Pointers

This section presents a series of examples of definitions of complex data types and illustrates the implementation of instances of these types with pointers in imperative languages. Each successive definition represents a class of data type which is more expressive. The purpose of this series is to show how the final examples correspond to the types found in Glide.

Basic composite data type

The simplest form of composite data type is one in which primitive types are composed and can be manipulated and as a unit. For example, the coordinates of objects in 2D-space might be represented through the following composite type definition (using Glide-style notation).

```
COORDINATE ==   X-coordinate: INTEGER
                Y-coordinate: INTEGER
                Label: STRING ;
```

Such a type can be instantiated for all combinations of values of each of its component primitive types. Thus the set of values (instances) of the type `COORDINATE` is the set of values of the cartesian product of the domains of each component (`INTEGER` **x** `INTEGER` **x** `STRING`). All modern programming languages provide at least this level of type definition facility.

Recursive types I: lists

A next higher level of expressiveness in composite types is to allow type definitions to be *recursive*. In this case the definition of a new type includes the type being defined. The simplest example of a recursive type is a list. For example, the type definition

```
LIST ==     Label: STRING
            Tail: LIST ;
```

is a composite type in which one of the fields contains a value also of type `LIST`. Instances of this type are lists whose elements are strings.

In an imperative language it is possible to make use of *pointers* to represent instances of this type. A pointer is a memory location which contains a value which identifies another memory location. Instead of assigning the value of a primitive type to the component field `Tail`, a pointer which is the address in memory of the first element of the tail of the list is assigned to the field. A picture of such a list instance might be the following

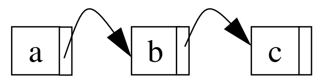


Figure 5-1 Pointer implementation of a list

This informal graphical depiction illustrates the concrete pointer-based implementation of an example list. A *dereferencing* operation must be used to access the tail. Imperative languages such as C or Pascal require the use of a special annotation in the type definition (*e.g.*, `*LIST`) to indicate the presence of a pointer to a given type. The dereferencing operator and this annotation bring to the surface an aspect of the implementation which would ideally be hidden [Hoare75]. A recursive type does not restrict the size (length) of the instances. This introduces the issue of dynamic structures whose size is only known at run-time and may vary during the course of program execution. A list defined in this way can be viewed either as a linked list or as a recursive containment as illustrated by the two diagrams in the figure below (taken from [BL86] p. 83).

(i)

(ii)

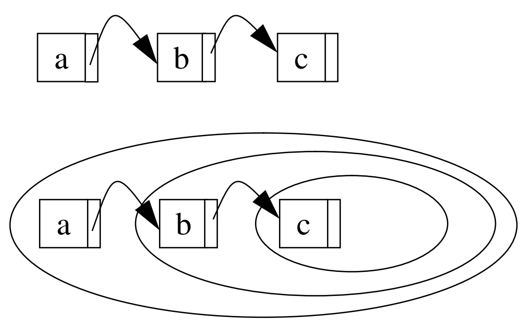


Figure 5-2 Two views of a recursive type : linked list or containment (from Guttag and Liskov p. 83)

Recursive types II: trees

A simple generalization of the list type is the tree type. Trees can also be conveniently represented using recursive type definitions. For example, the following defines a binary tree type.

```
BTREE ==  Label: STRING
          Lbranch: BTREE
          Rbranch: BTREE ;
```

Instances of such a type can be similarly implemented with pointers, as is illustrated here.

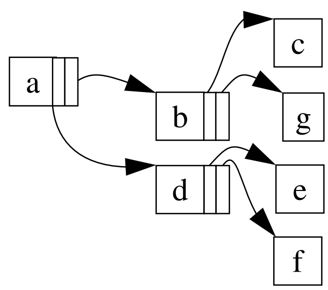


Figure 5-3 Implementation of a Binary Tree

These definitions can be naturally extended to trees with other branching factors. So called *general trees* in which the branching factor varies can also be handled in a pointer-based implementation by using (dynamic) *lists* of pointers, *e.g.*:

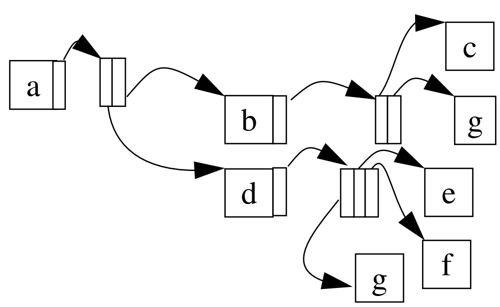


Figure 5-4 Implementation of a General

(A ▯ represents some implementation of a list.)

Recursive types III: grammars

A different extension of expressiveness is to allow a composite type to contain different combinations of types. These are sometimes termed *variants* or *constructors*:

```
EXPR ==   Fexpr :(Flt1:FLOAT Op:OPERATOR Flt2:EXPR)
```

```
| Iexpr :(Int1:INTEGER Op:OPERATOR Int2:EXPR);
```

In this example an instance of type EXPR can be either the combination of a FLOAT, an EXPR, and an OPERATOR *or* an INTEGER, an EXPR, and an OPERATOR. The variants are distinguished here by the tags Fexpr and Iexpr. This extension provides a type definition language with the same level of expressiveness as context free grammars, the notation being equivalent to BNF. A type definition corresponds to a production and each direct or mutual recursion between type definitions corresponds to a recursive use of productions in a grammar. A data type of this class can be used to specify a set of instances which correspond to the parse trees of sentences of a given language. A program in the language thus corresponds to a tree instance of such a type. Hence, the specification meta-languages for automatic programming environment generators discussed earlier (Chapter 3) allow the user to specify a language as a data type from this class.

Shared Structures

A different form of generalization is to allow instances of recursive types in which parts of the instance structure are *shared*. It is possible to take a type definition that is of the same form as the previous one for binary trees,

```
BDAG ==    Label: STRING
           Branch1: BDAG
           Branch2: BDAG;
```

but create an instance for the type which is not a tree but an acyclic *graph*. The diagram below illustrates an example (again, at the level of implementation with pointers).

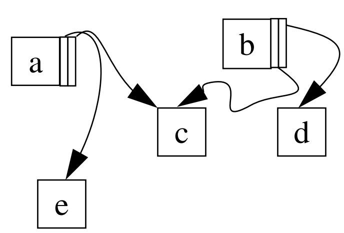


Figure 5-5 Implementation of a Shared

Admitting such structures to be legal instances of the type introduces the first problem for the formal specification of changes to such instances. If the value (in the sense of the previous examples) of the DAG rooted at **a** is modified by modifying **c** it has the *side effect* of modifying the value of the DAG rooted at **b**. The component labelled **c** is shared by both the instance rooted at **a** and the one rooted at **b**. Admitting side-effects violates the requirement of referential transparency of a declarative formal description. Instances implemented as DAGs encounter the problem of side effects through structure sharing. This problem of structure sharing occurs whenever a component object has more than one "parent".

Cyclic shared structures

A further step in increasing expressiveness is to allow *cyclic* structures. Using the same form of type definition:

```
BDG ==      Label: STRING

            Branch1: BDG

            Branch2: BDG ;
```

the following instance could be admitted:

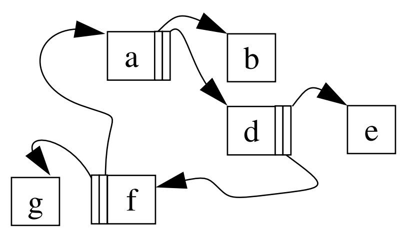


Figure 5-6 Implementation of a Cyclic

In order to admit the structure that is implemented with pointers as illustrated, it is necessary to remove a further restriction. In the previous categories only the type *definition* was recursive. In the case illustrated above the *instance* of a type can also "contain" itself as a component. This is recursion in the instance rather than in the type - a *recursive instance*, though the terms *cyclic structure* or *circular structure* are more commonly used. Circularity can be thought of as a special case of sharing in which the shared item and sharing item are the same [Lev78]. The cycles in such structures cause further problems for declarative formal descriptions of structure manipulation. This is essentially because such descriptions rely on the ability to traverse a well-founded data structure, which a cyclic graph is not. A naive structural equality test of two cyclic instances, for example, would not terminate because it will get caught in a loop.

Note that cyclic structures can only occur as instances of a set of types that are recursive, but it is not necessary that a set of types be recursive for simple, non-cyclic sharing to occur.

GBVLs and shared/cyclic types

The use of structures with sharing and cycles are surprisingly common, especially in interactive systems which need a way to represent the interconnection of objects. For example, hypertext systems allow links between components of a document without restriction, so that the structure of the document as a whole can be an arbitrary directed graph. CAD/CAM modelling packages contain data structures for 2-D and 3-D geometric modeling. These models have cyclic pointers to represent the topological properties such as the interconnection between volumes, faces, edges, vertices. Graphical user interface toolkits allow an interactive graphical interface to be created by composing small interactive objects (“widgets”) into a complete interface for given application. The neighboring relationships between such objects are often captured with inter-object references which may be cyclic [ET++89].

The area of concern for Glide, interfaces for GBVLs, is another application area in which these kinds of structures are needed, as the description of Glide in the previous chapter has already illustrated. Classical notations for defining the structure of programming languages are essentially describing tree-based structures. The program is viewed as a hierarchical decomposition of the text string. The Glide view of GBVLs is, however, that components are not only parts of other components, but they may also be connected to each other. This adjacency is defined mutually: If two objects A and B are connected, this can be represented by the fact that A has a property identifying its connected neighbor as B and B has a property identifying A. When properly exploited, this mutual definition approach provides very compact, direct, versatile, and elegant way of capturing the logical structure of connected and composite objects of a GBVL. Such cyclic definitions must be considered with care because they can cause problems, as has already been alluded to and will be shown in more detail in the next sections.

The complete Glide Grammar type system is an extension of the examples that have just been described. The notion of variants is added with the alternation operator (|); the distinction between ordered and undordered combination is added with the aggregation operator (.); a means of expressing commonality is added by combination of alternation and aggregation; and finally the postfix annotations (* and **) are added to express lists and sets directly.

Having described how these kinds of data types can be captured at the lower level of abstraction of imperative languages, the different ways they have been characterized in formal declarative languages is now examined. This essentially involves avoiding the explicit use of pointers, as is done in Glide and is advocated in [Hoare75]. Section 5.2 examines functional languages, Section 5.3 abstract data types, Section 5.4 data base models, and Section 5.5 notes a few other areas in which this issue has also been addressed.

5.2 Functional Languages

Functional languages (such as ML, Miranda, Pure Lisp, etc.) allow a more abstract view of data types than the pointer implementations just illustrated. Functions accessing and manipulating instances of the types (values) can be defined abstractly and in a way independent of their implementation. Two functional languages are used here as illustrative examples, first ML [Paul] and then Miranda [BW88].

ML

As its name suggests (Meta Language), one of the primary purposes of ML is to describe languages, and thus it is well-suited to applications centered on representing language structure. The data type definition constructs of ML can be used to describe sets of trees which correspond to programs of a language.

If ML were used to capture the structure of the nodes and links of Petri Nets in the same fashion as Glide, the following set of ML recursive data type definitions might be appropriate:

```
datatype
Node  =  NODE    of NodeName
                  * I
                  * O            and

 I    = INPUTS   of Edge list    and

 O    = OUTPUTS  of Edge list    and

 Edge = EDGE     of Node
                  * Node ;
```

In ML, the keyword **datatype** introduces a data type definition and the keyword **and** is required in ML when defining a set of types which are recursive. For simplicity this definition has omitted the use of variants, but ML does provide them (they are known as type *constructors*) to allow creation of a given type out of different combinations of other types.

The set of type definitions above is accepted by the ML compiler and there is no need for any visible use of pointers for the user of ML, either in the definition of the type itself or for functions using the types. The problem with this composite type definition is that instances that are cyclic can *not* be created; only instances which correspond to trees can be instantiated. In order to preserve referential transparency as required in ML, an instance of a type can be created but cannot be changed. A small change to part of an instance can only be modeled by constructing a completely new copy of the structure with the small change. The problem of side effects in shared structures does not arise since update-in-place is not available, but the consumption of time and space due to copying structure for every change soon becomes prohibitive. A further consequence of this requirement for referential transparency is that it is not possible to create cyclic structures: one of the components of a cyclic instance is the instance itself which, by definition, hasn't been created yet. These problems can also be viewed as a consequence of the *strict* functional semantics of ML.

It is because cyclic structures are so useful in real programming problems, however, that ML does include an imperative extension, *reference types*, which are similar to pointers. The **ref** keyword is used to specify a reference type. The following changes to the previous definition is how the Petri Net type could also be defined in ML.

```
datatype
Node      = NODE          of      NodeName
                          *       I
                          *       O                and

 I        = INPUTS        of      ref Edge list    and

 O        = OUTPUTS       of      ref Edge list    and

 Edge     = EDGE          of      Node
                          *       Node ;
```

The only difference between this definition and the previous one is the addition of the **ref** qualifiers in the definitions of the INTPUTS, OUTPUTS, and EDGE types. The reference type extension to ML can be used in conjunction with destructive assignment (**:=**) to create cyclic structures. This requires a notion of change of state and thus lies outside the functional model. In ML, support for imperative programming is closely associated with the use of reference types. It is not possible to create cyclic instances of recursive types without using **ref**. The solution to the need for cyclic structures is in ML to stop being declarative.

Miranda

Miranda is a lazy functional language with an elegant minimalist syntax. The analogous definition of nodes and links in Miranda is shown here:

```
node     ::=   Makenode        nodename
                               inputs
                               outputs

inputs   ::=   Makeinputs      [edge]

outputs  ::=   Makeoutputs     [edge]

edge     ::=   Makeedge        node
                               node

nodename ::=   Makenodename    [char]
```

In Miranda square brackets (`[]`) are used to indicate a list of elements of a given type.

In contrast to ML, cyclic instances of this type *can* be created in Miranda, without requiring an imperative extension. There are two ways to explain why this works in Miranda. At the abstract level, Miranda is a lazy functional language, so a value can be used before it is created, and hence it is possible to create an object out of itself (the cyclic case). At the underlying implementation level, all values in Miranda are accessed by always dereferencing from a pointer. Thus the pointer to an object can be created and used before the object itself is created. Structures which are implemented as cycles in Miranda are viewed as *infinite* structures, because at the functional level of abstraction they can be regarded as such - the underlying pointer structure is not visible and not accessible. Traversal operations (such as printing of an infinite structure) do not terminate. A problem with this approach is that a cyclic list composed of two elements (containing the same value) cannot be distinguished from a cyclic list of one element (containing the same value). Both have the same value of being an infinite list of the value.

In contrast to ML, Miranda does not "break symmetry". When the **`ref`** construct is used in ML, the definition of `INPUTS` and `OUTPUTS` is different from the other type definitions. In Miranda and Glide this does not happen.

The issues of sharing and cyclic structures are well known problems of functional languages and various solutions have been proposed. One proposed solution that avoids "recreating the entire data structure" after any change to a data structure is described by Burton and Yang on the use of multilinked data structures in functional languages [BY90]. This solution "hides" the change of state as auxiliary heap data structure which must then be passed as an extra argument between function invocations. It is possible to reduce the amount of copying by carefully sharing structure between the successive heaps associated with each function invocation. Though this approach does permit cyclic types, it is still far from and ideal solution since it simply defers the problem of structure re-creation to this auxiliary structure which in effect represents the state of the system. The problem of dealing with state and functional languages continues to receive attention in research on functional languages [FLS93].

5.3 Abstract Data Types

Work on the formal specification of abstract data types is similar to the work in functional languages in that both strive for concise declarative abstract descriptions of data types and functions operating on them. In the same spirit as functional descriptions, the *algebraic* approach to specifying abstract data types (ADTs) allows the semantics of such types to be defined via algebraic equations interrelating the functions [Gut78]. The intent behind the algebraic specification of abstract data types is to characterize the behavior of a data type without needing to appeal to any particular implementation (pointer-based or otherwise)[1]. The problems of dealing with shared and cyclic structures have also been investigated here:

---

1 This ideal is in some sense at odds with visual programming and program editing in general, since the data type instance (the program) is intended to be *visible* for direct manipulation by the user, not hidden.

The thesis of M.R. Levy has directly addressed the problem of extending the ideas of algebraic specification of abstract data types to incorporate shared and circular structures [Lev78]. Two solutions are proposed to providing an abstract description: (i) providing an explicit algebraic characterization of the use of references and assignment by relating references to a special notion of *congruence* and (ii) by relating circular structures to infinite objects in continuous algebras. Sets of equations are used to characterize sharing in instances of a type. These sets of equations play a similar role to that of the shape predicates of Glide in characterizing the loops.

Moller [Mol85] also addresses the problem of shared and cyclic structures from an abstract mathematical perspective on algebraic specification. He proposes a solution to the inability of algebraic formalisms to describe graph structures in which a graph is characterized as a repeating pattern in an infinite tree. This resembles the Miranda view of cyclic structures as infinite structures. The repeating pattern is similar to the notion of congruences of Levy. Solutions of Moller and Levy require a high degree of comfort with abstract algebraic techniques.

Graph Types

More recent work by Klarlund and Schwarzbach [K&S93] also examines the problem of capturing cyclic structures as a data type (which they term *graph types*). The authors provide a simpler yet effective way of formalizing at least a subset of graph structures as abstract data types. The key ideas are to first identify an underlying *spanning tree* of a graph structure and then to use a simple language based on *regular expressions* to define the remaining links which span the tree. These regular expressions are termed *routing expressions* and are similar to Glide path expressions. The authors demonstrate that this technique makes it possible to describe a wide class graph structures including doubly linked lists, threaded trees, and binary trees with the leaves linked to the root. They also note however that some structures are not amenable to this form of description. The graphs defined by their routing expressions are *func-*

tions of the underlying tree, *i.e.*, for a given set of routing expressions and a given tree it is not possible to have more than one graph. A further disadvantage that this technique shares with ML references is that symmetry is broken since the one path to a node on the spanning tree is distinguished from the others described by routing expressions. The contribution of this work is that it is possible to identify subclasses of general graphs which, because of some unique property, can be characterized in a simple way - in this case through regular expressions on paths in a tree.

It is instructive to compare the syntax and semantics of lexical components of K&S routing expressions with those of Glide path expressions. The rough correspondence between them is shown in the following table:

K&S	Glide	Meaning
↓x	.x	access component tagged x
↑	..x	access parent (identified through x)
^		test if node is root
$		test if node is leaf
T		test if parent of type T
T:v	x.w = x.v	test if component is of type T variant v
(expr)*	@(expr)	repeat zero or more times
()		regular expression composition
+		regular expression alternation

Though the syntax of K&S routing expressions is richer, the Glide data model is more abstract than the data model of K&S. Glide incorporates representing sets and lists directly in the type, hence the extra routing expression constructs are not needed in Glide path expressions.

5.4 Database Models

The issue of shared and cyclic structures also arises in work on developing models of data bases, especially when the goal is to find more expressive data models than the relational one.

5.4.1 Model of Network Data Bases

Network model data bases have faced similar issues; these data bases represent an early case of attempting to deal with references/pointers. The work by Gangopadhyay [DG83] provides a formal model of network data bases ("DML"). In this model the same problem of shared mutable data structures is explicitly considered. Here, the solution used is to distinguish between the concept of *identity* of a data item as distinct from its *value*. The identity of a shared component of a data structure does not change, only the value associated with it. An auxiliary look-up data structure (the "*type-state*") is used to associate identities with values. This auxiliary structure is described using a state transition model. This solution is thus similar to the Barton's solution in functional programming.

5.4.2 Object-Oriented Data Models

Object-oriented data models are useful for representing data that has a more complex structure than can be naturally represented by the more rigid relational model. Object-oriented programming languages have been described as languages which provide extensive support for dealing with and controlling the use of pointers. The following comments by Stroustrup [ARM91] also allude to the issue under consideration:

> *"One of the most powerful intellectual tools for managing complexity is hierarchy, that is, organizing related concepts into a tree structure......Naturally, this organization has its limits. Sometimes even a directed acyclic graph seems insufficient for organizing concepts of a program; some concepts seem to be inherently mutually dependent. If a set of mutually dependent classes is so small that it is easy to understand, then cyclic dependencies need not be a problem."*

K&S also note the close relationship between their formal specification of graph types and work on formal models of object-oriented programming, because of the need to deal with mutual reference between objects in the latter.

5.5 Other Areas Addressing Cyclic and Shared Types

This section notes a few other areas in which the issue of sharing a cycles in data structures also appears.

5.5.1 Parallelizing Compilers and Pointer Structures

The issue of sharing is also of concern to designers of parallelizing compilers. These compilers attempt to identify various forms of *dependencies* in programs. If the compiler can identify the type or the absence of dependencies it may then be possible to partition a program into independent pieces which can be executed separately. Most of this work has focused on array dependencies but Hummel, Hendern and Nicolau [HH&N92] have examined the problem in the context of arbitrary (pointer-based) C data structures. In these structures, dependencies due to sharing (the term used here is *aliasing*) can preclude parallelization.

The scheme used by HH&N is in effect the dual of Glide shape predicates. HH&N define a language (orginally ADDS and subsequently ASAP) which allow the definition of predicates (called aliasing axioms). They are used to specify where sharing that does *not* occur. The reason for this is that in the task of parallelizing programs the goal is to identify those parts of a structure which are not shared, do not carry dependencies, and therefore the program statements accessing might be safely executed in parallel. These references may be named, in which case dependency detection is quite easy, or they may be anonymous in which case dependency detection is more difficult. The languages based on aliasing axioms are used as by the programmer to describe his/her data structures so that the compiler can exploit the information contained in them to produce parallel code which will not violate the dependencies. The aliasing axioms of ADDS and ASAP use path expression which are also based on regular expressions of names of members of C structs.

The aliasing axioms fulfill a similar role to Glide shape predicates in that they provide the extra higher level description of the structures that is needed to characterize them more accurately.

5.5.2 Galois

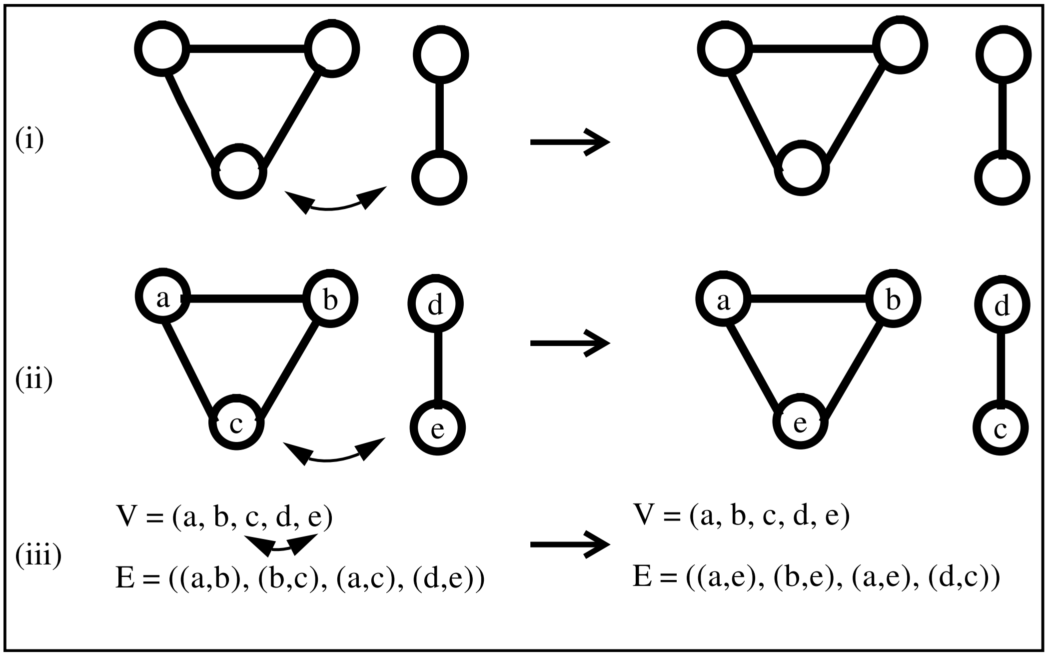
Recent work by Turpin combines the use of pointer-based data structures with a declarative logic-based language. Though the approaches are very different, this is similar in intent to the references of ML in that they both make pointers first-class objects in the language and then attempt provides a formal semantic characterization of these objects as part of the language.

5.5.3 Graph Grammars

The work in this area takes a somewhat different approach to characterizing graph structures. Classes of graphs are described by collections of graph rewriting rules. A graph grammar is a collection of rules to change graphs by first matching and then altering parts of the graph structure. These rules are often graphically depicted. Graph grammars sidestep the issue of cyclic types, by avoiding representing data structures in a linear text form. The problem of circular structures is hidden by dealing with pictures of graphs directly. The difficulties encountered in attempting to find a concise standard textual notation for graph rewriting is an indication of the underlying problem of cyclic structures [Ehr90].

5.6 Dynamic Structures, Anonymity, and Connections

This survey of the different approaches to describing shared and cyclic structures shows that there is an underlying issue of whether or not to associate explicit names (a pointer is a kind of name) with objects so that these names can be used to refer to the objects in the graph. The desire to *avoid* using names can best be illustrated by the following diagram showing a simple manipulation in a simple graph - the exchange of two nodes in it.



Consider the node swapping operation indicate by the . In the top diagram (i) it is evident that the before and after graphs are isomorphic; without labels; no discernible change has occurred. However, if the objects are given identifiers then a change *has* occurred (ii). Any linear text representation (iii) of graphs requires some form of identifiers, and then it is not immediately obvious that the graph has not changed. In the graphical case (i) the abstract "identity" of the objects comes *purely* from their mutual attachment to other objects. The intent behind the pointerless, nameless approach of Glide to cyclic structures is to preserve this abstract quality. It is by assigning identifiers the anonymous quality of objects in the graph diagrams is lost.

5.7 Summary

This chapter has put the Glide data type model into context by relating it to work on other declarative models of data types. These models either do not admit sharing and cycles, do allow them with special annotations, or do allow them with auxiliary means of specifying where cycles and sharing are permitted. Glide is of the latter form. The Glide grammar is used to specify the basic compositional structure of types and then shape predicates are used to define and constrain the use of cycles and sharing. The uses of various forms of path expressions for expressing paths through the structure on which the characterization of cycles is based was also described. The chapter has also demonstrated that the issue of managing cycles and sharing in data structures arises in a wide variety of application areas. The reason the issue arises in Glide is that the components of graph-based visual languages that are manipulated by a user in the interface are inherently mutually defining; each gets its identity from the other components it is attached to. Simple predicates based on path expressions (as others have found as well) allow a concise characterization of many shapes of sharing and cycles.

Chapter 6

Glider Design and Implementation

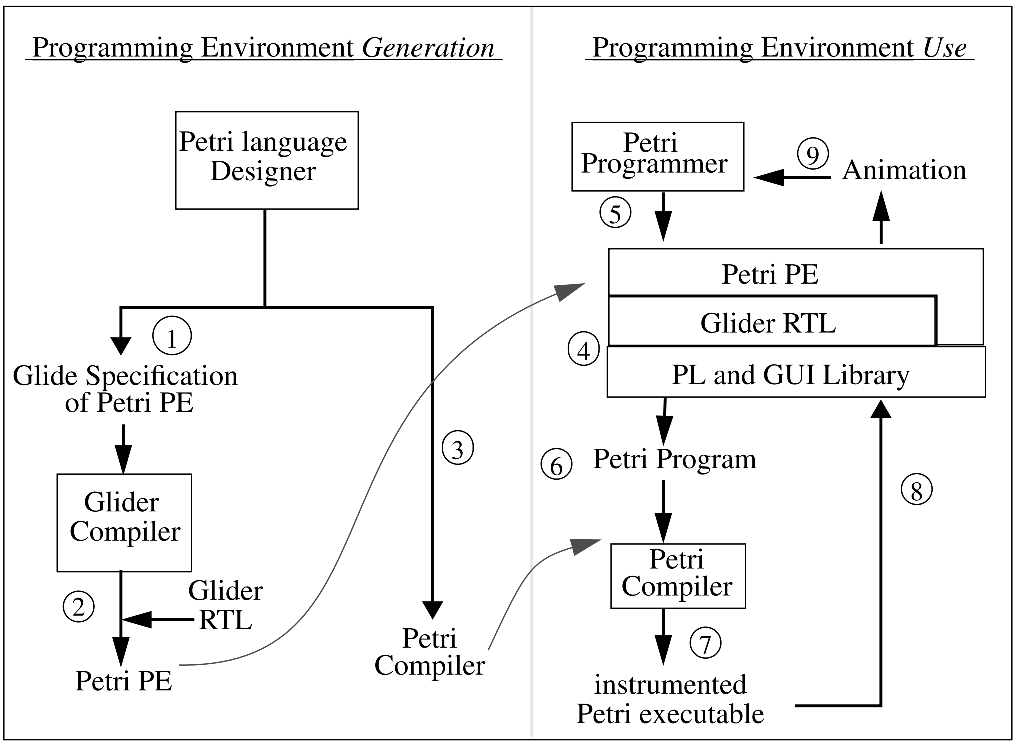
This chapter describes the design and implementation of Glider, the system that generates interactive graphical programming environments for GBVLs from their Glide specifications. Such a programming environment allows the user to edit, view, execute, and animate programs "written" in a GBVL. Glider consists of two main parts, a *compiler* which translates the various parts of a Glide specification into programs, and a *run-time library* (RTL) which provides a set of procedural abstractions which are used by the programs.

Section 6.1 provides an overview of how the Glider system is used. Section 6.2 describes the high level design of the Glider compiler and of the programming environments (composed of the programs generated by the compiler and the RTL they are linked to). Section 6.3 discusses the choice of target language and GUI library used. Sections 6.4 and 6.5 present the Glider implementation in detail: Section 6.4 describes the procedural abstractions and a collection of run-time components, each handling specific tasks, that form the Glider RTL; and Section 6.5 describes the program generator components that form the Glider compiler. The structure of Section 6.5 follows that of Chapter 4 since there are separate compilation algorithms used for each part of the Glide specification language.

Overall, the last three sections provide a *bottom-up* description of Glider: they start with the languages and libraries needed to support the RTL, then progress up through the levels of abstraction within the RTL, and finish by showing how the Glide high level specifications are compiled into procedures which make calls to the procedural abstractions provided by the RTL (its API).

6.1 Overview of Glider

It is important to clearly distinguish the two phases of activity associated with Glide: *generating* the programming environment for a given GBVL and *using* the end product that is generated - the user interface for developing programs in the GBVL. The left and right halves of Figure 6-1 below illustrate the relationship between these two activities.



PL = Programming Language
PE = Programming Environment
GUI = Graphical User Interface
RTL = Run-time Library

Figure 6-1 Generation of a Programming Environment vs. its Use

Programming Environment Generation

In the generation phase, the Glider compiler is invoked in order to compile a specification, written in Glide, that a language designer has developed to specify the programming environment for his/her particular GBVL. In the diagram above the example GBVL is called “Petri” ①. Executing Glider generates an executable program②. This program is the programming environment for Petri. The program, the output of Glider, is in the form of source code in a standard programming language which can subsequently be compiled and linked to the Glider RTL in order to create the complete executable. Parts of the RTL access routines of a standard programming language library and GUI library ④.

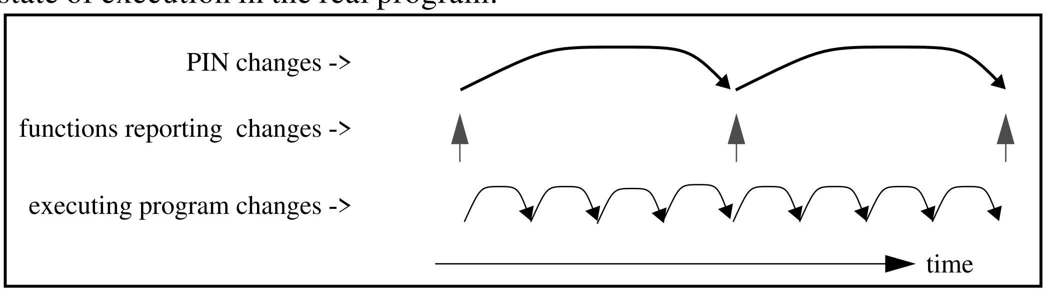
Glider does not itself compile programs of a given GBVL, it only generates a programming environment for developing them. It is the separate responsibility of the language designer to write the compiler for the language itself ③. In order to allow programs to be animated, they must be appropriately instrumented by the compiler with functions that report changes relevant to the abstracted execution of the GBVL (that the designer provided in Glide specification), so that the changes in the executing program are communicated back to the interface. (However, abstract execution of programs, based on only the GBVL Glide execution semantics, is possible, and is described below).

Programming Environment Use

In order for a user to program in the GBVL (Petri, in this case), the Glider-generated programming environment is executed, causing an interactive graphical user interface to appear on the screen ready for user interaction ⑤. The user can then create and manipulate Petri language programs through the editing commands that the designer specified. These commands update the underlying Program Instance Network (PIN) that represents the program. The different parts of the program are visible through views of parts of the PIN that the designer specified as Glide queries. When the user has completed editing a Petri program ⑥, a representation of the program appropriate for the Petri compiler is extracted from the PIN and is passed to the compiler to generate an executable of the program ⑦. This executable is run, having been instrumented with reporting functions. These functions are invoked as the program executes so as to relay changes of program state back to the PIN ⑧. The changes trigger animations which illustrate the behavior of the executing program to the programmer ⑨. The programming environment thus provides a "closed loop" which allows the programmer to interactively develop programs and modify them after viewing their behavior in animations.

Internal or External Execution

The execution within the PIN is driven by a combination of the abstract execution specification and changes to the PIN that represent the state of executing program. The reporting functions invoked during program execution maintain a consistent "mirroring" of the state of the executing program and the PIN. Procedures derived from the Glide actions that specify the abstracted execution semantics are executed in response to changes in the mirrored state. These ensure that the executing program and the program as represented in the interface stay in step. The figure below illustrates this basic idea of maintaining consistency between the state of execution as modeled in the PIN and the state of execution in the real program:



The actual program may undergo many low level state transitions while the PIN only undergoes one "high level" transition (since it has an abstract model of the program execution). Each change in the PIN is triggered by state changes relayed by the reporting functions.

The abstract execution specification can be used in two ways, internally or externally:

(i) *Internal Simulation* - It is possible to specify the execution semantics of a language completely within Glide. In this case the language designer is effectively specifying a (slow) interpretive simulator and no separate language compiler (or instrumented executable) is required. The Glide execution semantics condition-action rules specification is compiled into a set of procedures that access and alter the PIN data. If the conditions on values in the data of a particular rule are met, the rule can fire and operations are invoked to update values in the PIN data structure.

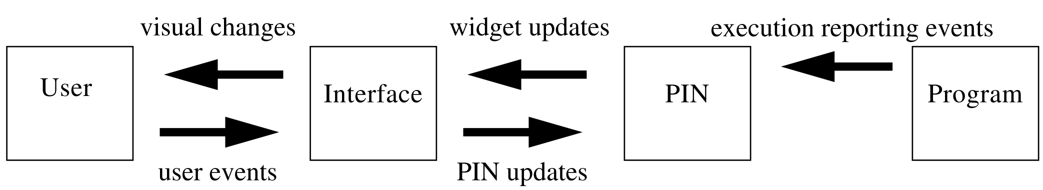
(ii) *Tracking External Execution* - If the execution semantics of the language are not fully specified within Glide, the abstract execution specification is used to track and to verify the execution of the actual program. After each high level transition, the interface waits until the reported state variables change as expected. If the abstract specification is non-deterministic - at some points during execution one of several possible transitions may occur - then the reporting functions are used to indicate which particular transition actually occurred in the real program (*e.g.*: the actual node that fired out of the several enabled nodes, or which branch of a condition was taken).

6.2 Design of Programming Environments and Compiler

From a functional point of view, a complete programming environment system program consists of (i) the routines generated by the Glider compiler, (ii) the routines in the RTL, and (iii) the routines provided by the libraries of the target language and the GUI system. This section first describes this architecture, *i.e.*, the contents of (i), (ii), and (iii) and the interfaces between them. It then describes the design of the compiler itself, which generates (i).

6.2.1 Programming Environment Design

From a data structures point of view, a complete programming environment system consists of (i) sets of user interface widgets which are on-screen visual representations of instances in the PIN (representing parts of, say, a Petri program), and (ii) the PIN itself. Front-end user events (*e.g.*, mouse and keyboard) are translated into widget-specific events by the GUI. These events then invoke commands whose actions were defined in the Glide specification. The actions are implemented as (generated) procedures which, descending through several layers of abstraction, perform updates to the PIN. The PIN updates are then reflected back on the screen by passing updated information back to the widgets (creation, deletion, modification of values of their properties). Back-end execution events from a running program also cause PIN updates:



Various components of the RTL maintain the consistency between instance creation/deletion and value changes in the PIN, and the widgets and widget property values that represent them. Changes occur first to the PIN and then to the user interface widgets. The widgets are thus functionally *dependent* on the PIN; they always reflect the current state of the PIN. The interface widgets themselves are supported by a data structure which is internally managed by the GUI library. The management of the PIN is achieved through a set of layered components in the RTL. There are effectively three layers of abstraction in the RTL. Each will be described in turn in Section 6.4.

6.2.2 Compiler Design

The Glider compiler consists of several sub-compilers, one dedicated to translating each part of a Glide specification, as is illustrated in Figure 6-2, an expanded view of the lower left of Figure 6-1 ②. Each sub-compiler translates its particular part of a specification by creating an abstract syntax tree (AST) for it and then generating code from the AST.

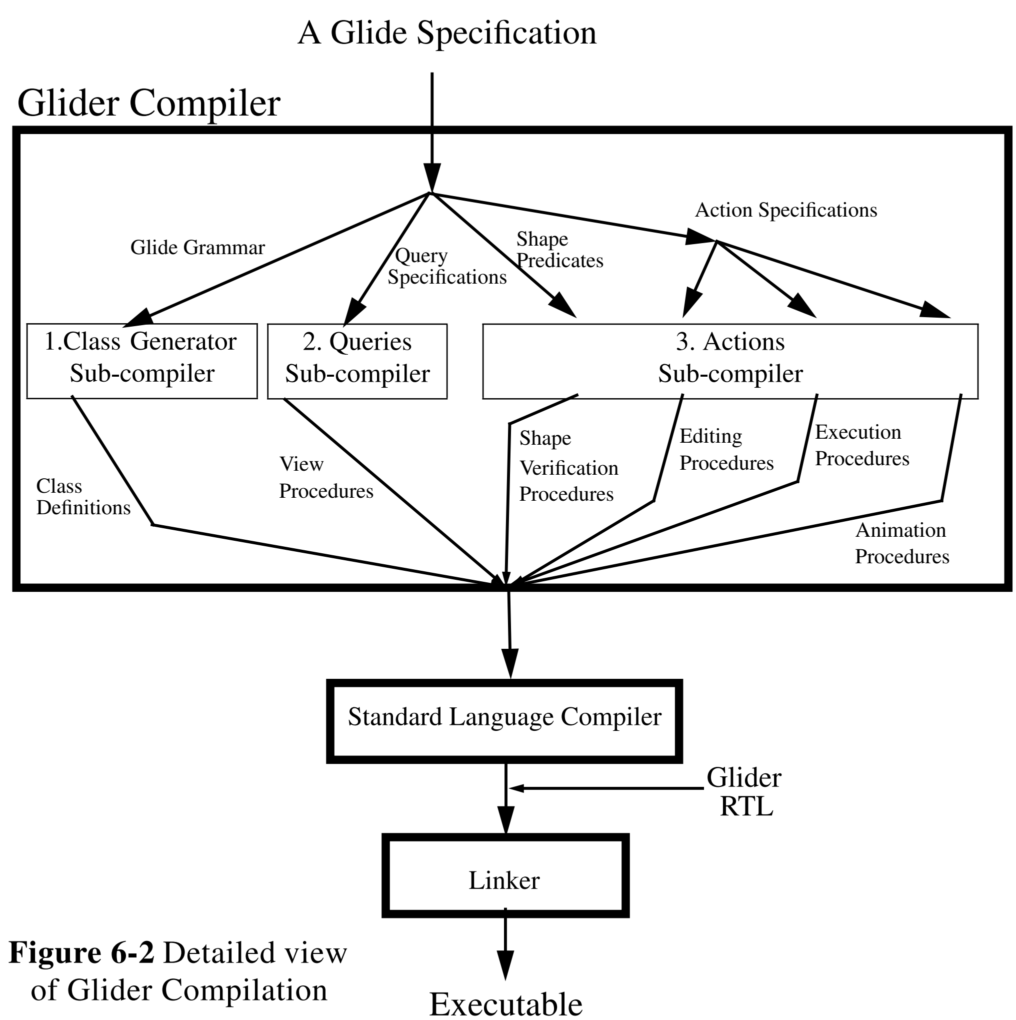


Figure 6-2 Detailed view of Glider Compilation

The most significant part of the sub-compilers are their code generators. All the code generators share a common object-oriented design. Each AST is composed of a set of AST nodes which are objects created from class definitions which encapsulate the semantics of Glide language constructs. These AST node class definitions contain methods for computing variable bindings, semantic checks, symbol table manipulation and access, and program templates used in the generation of source code associated with the particular Glide language construct (Figure 6-3). Depending on the particular sub-compiler, the AST may be traversed (passed) one or more times, invoking specific methods from the classes during each pass. Each AST node class contains methods which may be invoked for the particular pass.

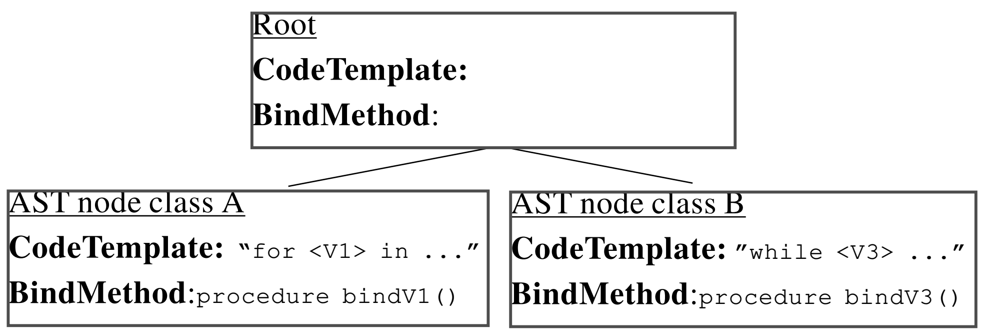


Figure 6-3 Object-Oriented Compiler Design

Using an object-oriented design for a compiler is becoming more common with the widespread availability of object-oriented programming languages (OOPLs) such as C++ (*e.g.,* [New94]). This modular design has several benefits. The standard benefit of programs written in an object-oriented manner, that they are more easily modified and extended, makes changing and extending the Glide language simpler. The design also allows the generator to be more easily *retargetable*. It makes it possible to create executables which are written in new languages and/or use new GUI libraries, by just modifying the code templates stored in the AST node class definitions, and porting the RTL to the new language.

6.3 Target Run-time Environment

In the demonstration system that was built, the Glider compiler generates programs in the interpreted language *Tcl* and the Glider RTL consists of programs implemented in a combination of C and Tcl and they use the GUI Library *Tk* [Oust94]. Though the demonstration system has been implemented to target this particular substrate, the compiler is designed, as has just been described, to allow easy retargeting to other substrates (*e.g.*, C/C++/Motif). It is possible to target both compiled and interpreted languages. In the former case, source code that is generated is further compiled and linked as is illustrated at the bottom of Figure 6-2. The latter case is more simple since no secondary compilation is needed; the source code generated by the Glider compiler is simply concatenated with the RTL source and fed to the interpreter at run-time. The example segments of generated code and the algorithms listed in following sections are given in a simple pseudo-code, reflecting the fact that almost any standard language/GUI library could be targeted by simple modifications to the code generators.

Implementation Language

Tcl is an interpreted language which, in combination with its GUI widget library Tk, provides a high level target for the generator and the RTL. Tcl is a simple high level language that provides built-in support for string and list manipulation. Tcl also provides a tight and elegant interface to its associated GUI library Tk. This substrate allows the generated programs to be small and fairly simple. Tcl also has various object-oriented extensions, one of which, *itcl*, was used as the implementation language for PINs in the demonstration system [ITCL94]. The Tcl/Tk target environment was chosen for speed and ease of development of the demonstration system.

GUI Libraries

GUI libraries are still undergoing rapid change, but a number of standard libraries that are in widespread use have emerged over the last few years. These standard libraries, such as (*Motif* [Mot93], *Microsoft Windows*[MSWin], and *Macintosh System 7.x*) have a great deal in common. They provide similar forms of user interaction widgets such as command menus, pop-up menus, check boxes, type-in dialog boxes, etc.

Newer GUI libraries that are currently being developed seek to extend the breadth of forms of interaction with new widgets, and raise the level of abstraction provided for programming or specifying a complete interface. The Tk system provides a core of the common forms of widgets but it also and allows extensions to be easily incorporated. A number of GUIs developed and described in the research literature have included a "graph widget" extension [PT90,RDM$^+$87,GNV88,Him89,daV93,Dea92]. The Appendix provides a short summary of these extension packages. These widgets support the display of connected objects (nodes and links). Efficient layout algorithms for graphs, which are needed for these displays have also been the subject of extensive research [GDr93,DH89,ET89]. A basic requirement for supporting the interconnection paradigm for GBVL programming environments is good support for such graph displays. Good support includes the ability to perform fast automatic layout of nodes and links of arbitrary sizes, the display of ports, the labeling of nodes, links and ports, support for hierarchical graphs, multiple links between the same nodes, etc. After an evaluation of available graph widgets, the dot/dag graph display libraries were integrated into Tk for the demonstration system [GNV88].

The Tk library has good support for dynamically creating and modifying nested widgets through the use of "frame" widgets for composing widgets together and treating them as a unit. These nested widgets are exploited to directly reflect the recursive structure of GBVL specifications in the Glide Grammar.

6.4 Glider Run-time Library

The Glider RTL consists of a set of components and it has a three-level layered set of procedural abstractions for implementing these components and supporting the compiler-generated programs. This section first describes each level (Sections 6.4.1-6.4.3) and then the components that are built using them (Section 6.4.4). In addition to providing a clean structuring of the RTL, these levels of abstraction also define interfaces for portability - the RTL can be moved to a new run-time environment by re-implementing it at the level most suitable for the new target environment.

6.4.1 Level-0: Access and Alteration of Objects

Viewed at the lowest level of abstraction, a PIN is implemented as a set of objects defined by classes. The class definitions are generated from the Glide Grammar specification; there are one or more classes for every Glide production (type). The objects contain the following data:

- *PIN Data*: The classes contain data members corresponding to tags of the Glider production. These members can contain either the value of a primitive type (integer, real, etc.), a reference (id, name, pointer) to another object, or a list of references to other objects.
- *Meta-Data*: The classes contain data members which store *meta-data* about the PIN data. This meta-data includes the names and types of the PIN data members, whether the member represents a single item (SINGLETON), a collection (SET or LIST), whether the member is from a Glide alternation (CHOICE) or part of an ordered sequence (SEQUENCE), and the number of alternation choices and their tags. The including of meta-data in classes is a very common practice in object-oriented systems. Some object-oriented languages provide direct support for it (*e.g.*, Smalltalk), while in others, such as C++, it must be implemented by the programmer. In the case of Glide this information is simply a re-representation of some of the symbol table data that was created when parsing the Glide productions. It is included in the classes so that it is available to the next higher level routines which make use of it.

- *Back Pointers*: The objects also contain back pointers. Back pointers implement an inverse mapping from an object to each of the objects that refers to the given object (contains a pointer to it). Each back pointer has associated with it the name of the tag through which an object references the given object. Note that more than one back pointer may have the same tag. These back pointers are used, for example, in implementing the double dot “uppath” Glide path expression operator. Back pointers are automatically maintained by the RTL as objects and references are created and destroyed.
- *Widget Pointers*: These are back pointers to widgets. Each object contains a list of all the widgets currently displayed on the screen that represent the object (in one or more windows). This list is used, for example, to implement highlighting of all the widgets that represent an object when one of the widgets representing it is selected, or for animating all the widgets when the object is updated during execution. There is a one-to-many mapping of objects to widgets.
- *Graphical Attributes*: The graphical attributes provided in a Glide specification are simply implemented as additional members of the classes. The graphical attributes can be associated with a Glide production or with specific tags in the production.

Since all the information used is “flattened” out into data members of the classes at this level, the following small core of procedures is all that is needed to manipulate and access the objects[1]:

`create: class → objectid`

Creates an object a given class, returns an identifier for the object.

`destroy: objectid →`

Deletes the object associated with the object identifier.

`set: objectid, member, {pos-in-list}, (objectid|val|NULL) →`

Sets the value of a member to be an object identifier, the value of a primitve type, or NULL. If the member contains a list, the insertion position in the list can be specified (pos-in-list).

`get: objectid,member → (objectid|objectid-list|val|val-list|NULL)`

Returns the value associated with the member.

Figure 6-4 Level-0 - Object Access and Alteration

At this level-0 the object pointers (`objectid`) are visible in the sense discussed in Chapter 5. This interface is imperative; the procedures have the side effect of assigning values to members or of creating or deleting objects.[2]

The class hierarchy is in essence flat. Though there is a “root” class from which all the generated classes are derived, it only serves to simplify implementation. The reason the *is-a* inheritance structuring of OOPLs is not used is because the Glide language uses an aggregation approach to expressing commonality (see Section 4.1) rather than a taxonomic one. The target object-oriented language is being exploited simply as a high level implementation language providing encapsulation, polymorphism, and managing inter-object referencing, not for inheritance.

6.4.2 Level-1: Access and Alteration to the PIN

At the next level up from the basic procedures just described is a larger collection of more specific procedures for accessing and altering the data structure at the higher PIN level of abstraction. The *classes* and *objects* of level-0

---

[1] “{}” indicates optional arguments and “(|)” indicates alternative types of arguments.

[2] Depending on the implementation OOPL, the objectids may be typed (*e.g.*, pointers in C++). In this case the four procedures should viewed as being polymorphic, and instantiated for each combination of input and output object types.

are used to implement the *types* and *instances* at level-1 respectively. The procedures that define level-1 are:

(1)
```
gCollectionType:instance tag→ CHOICE | SINGLETON |
                 SET | LIST | SEQUENCE
gTagChoices:instance tag→ tag-list
gTagChoice:instance tag→ tag
gTagsAll:instance→ tag-list
```
(2)
```
gGetType:instance→ type
gIsOfType:instance type→ boolean
gIsInstance:instance→ boolean
gIsSameInstance:instance instance→ boolean
```
(3)
```
gMake: type   → instance
gDestroy:instance→ instance
gConnect:instance tag instance {pos}→ PIN
gDisconnect:instance tag instance→ PIN
```
(4)
```
gConnectedTo:instance tag→ instance | instance-list
gConnectedBy:instance tag→ instance-list
gIsConnectedTo:instance tag instance→ boolean
```
(5)
```
gAddConnectedBy:instance tag instance→ PIN
gRemConnectedBy:instance tag instance→ PIN
```
(6)
```
gGAGetValue:instance {tag} ga→ ga-value
gGASetValue:instance {tag} ga ga-value→ PIN
```

Figure 6-5 Level-1 - PIN Acces and Alteration Procedures

Set (1) are used to access tag information about a given instance. Set (2) are used to access information about instances themselves - such as which type they are an instance of. Set (3) are used to alter the PIN by creating or deleting an instance of a type, or by associating (connecting) an instance with the tag in another instance. The procedures (4) provide information on how instances are interconnected. This includes the `gConnectedBy` procedure which returns the list of instances that have the given instance has as a component - this procedure is implemented via the back pointers of level-0. The two procedures (5) are used to implement the automatic maintenance of the “back pointers” when instances get connected and disconnected. Procedures (6) return and change the values of graphical attributes. These attributes may be associated with an instance or more specifically with a tag in an instance.

The `set` procedure of level-0 is used to implement the “connect/disconnect” semantics of altering the PIN. Associating a value of a primitive type with a tag is viewed, at this level, as *connecting* it rather than *assigning* it. The only difference between instances of composite types and of primitive types is that after completely disconnecting an instance of a composite type, it can subsequently destroyed, but a primitive value is only connected in *one* place and cannot be shared, making the need to “destroy” it superfluous. Procedures of this level which change the PIN can be viewed as functions which map PIN$\rightarrow$PIN; a modification to any single instance is a change to the value of the PIN as a whole. The functions in effect define an abstract data type specification for the semantics of a set of PINs as a graph type.

The following diagram of a typical instance of a typical PIN, similar to the figures used in earlier chapters, illustrates the level-0 vs. level-1 view.

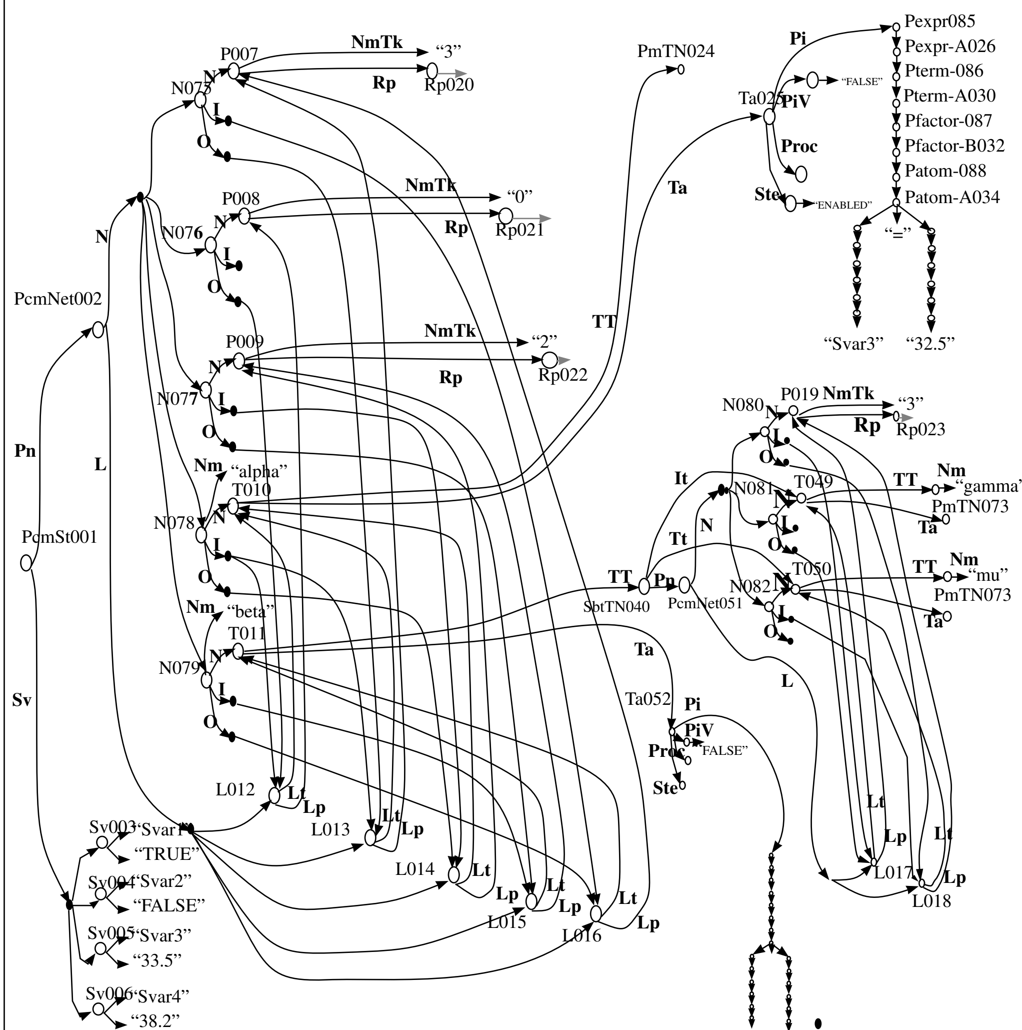


Figure 6-6 Example Program Instance Network (PIN)

This is part of an example of the PIN of a PCM program; one which contains 5 Nodes (three place and two transition nodes) and 3 links at the top level, and a subgraph of the (bottom right) with 3 nodes and 2 links. The instances (empty circles) are labelled with their objectids or their values (if they are of a primitive type), links are labeled with the relevant tags (bold). Full circles are used to indicate tags which contains sets or lists. The two dangling trees (top right and bottom right) are the syntax trees of node attributes. At level-1 the objectids neither are not visible or accessible, and altering the PIN is equivalent to creating and deleting edges and filled circles in the diagram.

The set of procedures of level-1 also represent an interface of portability; if an OOPL is expressive enough to directly support the implementation of these procedures (*e.g.*, an OOPL that provides meta-data automatically) then this would obviate the need for implementing level-0. In the Glider RTL of the demonstration system all these routines were implemented as calls to the level-0 primitives with parameters bound appropriately.

6.4.3 Level-2: Path Expressions

Finally, the next higher level of abstraction is one providing the path expression evaluation functions:

```
EvalPathExpr:       instance pathexpr → (instance|instance-list)

EvalStarPathExpr:   instance starpathexpr → PINTree
```

These are implemented using the procedures of level-1. `EvalPathExpr` is used to in both queries and actions, `EvalStarPathExpr` is used in queries. Path expressions are the expressions of the form `x.A.B.C`, where `x` is a variable whose value is an instance of some type, and `A`, `B`, `C` are, names of tags of successive complex types. Star path expressions are the similar forms of expression which can also include stars ("`*`") to indicate all tags of the corresponding type. `EvalPathExpr` returns a single instance or a flat list of instances. `EvalStarPathExpr` returns a tree (`PINTree`) corresponding to all the instances encountered in traversing the PIN graph when evaluating the expression. Algorithms for these two functions are shown below. The pseudo-code description below is meant to show the essential simple recursive structure of the algorithms. The actual implemented algorithms are somewhat more complicated - they include the checks for avoiding looping in the graph and include the ability to evaluate recursion in path expressions ("`@(...)`").

Path Expression Algorithm

Level-1 and Level-2 procedures are in bold, the functions `append` creates a list composed of its inputs.

```
procedure EvalPathExpr (instance pathexpr) {
   dottype :=  DotType(pathexpr)
   headtag :=  PathExprHeadTag(pathexpr)
   tailexpr := PathExprTail(pathexpr)
   returnlist := empty
   if (headtag is empty) then // termination: end of path expr, just return value itself
     returnlist := instance
   else
     case dottype in
      DOUBLEDOT                             // go up, iterate and collect over each parent
      { foreach elem in gConnectedBy(instance,headtag)
         returnlist := append(returnlist,(EvalPathExpr(elem,tailexpr)))
        endfor }
      SINGLEDOT                                                                 // go down
      { case gCollectionType(instance,headtag) in
        SINGLETON                                 // a single element, return the element
          { returnlist := EvalPathExpr(gConnectedBy(instance,headtag),tailexpr)}
        LIST, SET, or SEQUENCE                           // iterate and collect over list
          { foreach elem in gConnectedTo(instance,headtag)
              returnlist := append(returnlist,(EvalPathExpr(elem,tailexpr)))
            endfor }
        CHOICE                          // head tag is a single element, return the element
          { choice := gTagChoice(instance,headtag)
            returnlist := EvalPathExpr(gConnectedTo(instance,choice),tailexpr)}
        endcase
     endcase
   endif
   return returnlist }
```

Figure 6-7 Recursive Path Expression Evaluator

Tree Path Expression Algorithm

The algorithm to evaluate tree path expressions is similar, but instead of returning a list of instances, a tree structure (`PINTree`) composed of instances is returned. The recursive construction of the tree expands into branches if either (i) a tag containing a set or list is encountered, or (ii) a "*" is encountered in descending the star path expression. The function `mknode` creates a tree node composed of its inputs.

```
procedure EvalStarExprTree (instance,pathexpr) {
 headtag :=  PathExprHeadTag(pathexpr)
 tailexpr := PathExprTail(pathexpr)
 returnnode := empty
 if (gIsInstance(instance) is true and pathexpr is not empty) then
   if (headtag is a "*") then        // star => expand path into tree and follow each branch
      foreach val in gTagsAll(instance)
         nodelist := append(nodelist,EvalStarExprTree(instance,tailexpr))
      endfor
      returnnode := mknode(instance,nodelist)
   else                                               // no star =>  keep going down the path
      tagval := gConnectedTo(instance,headtag)
      case gCollectionType(instance,headtag) in
       SINGLETON                        // head tag is a single element, return the element
        { returnnode := mknode(instance,gEvalStarExprTree(tagval,tailexpr))) }
       LIST, SET, or SEQUENCE                // head tag contains a list, collect results
        { foreach val in tagval
            nodelist := append(nodelist,EvalStarExprTree(instance,tailexpr))
          endfor
          returnnode := mknode(instance,nodelist)) }
       CHOICE                           // head tag is a single element, return the element
        { choice := gTagChoice(instance,headtag)
          returnnode := mknode(instance,
                          (EvalStarExprTree(gConnectedTo(instance,choice),tailexpr))}
      endcase
   endif
 else         // termination: end of the path expr,just return the instance or value itself
   {returnnode := mknode(instance)}
 endif
return returnnode }
```

Figure 6-8 Recursive Tree Expression Evaluator

This completes the description of the three levels of abstraction of procedures for accessing and manipulating the PIN. The functions provided by the last two levels are used both in implementing the RTL components, which will be described in the following Section 6.4.4, and they are also used in the programs created by the Glider sub-compilers, described in Section 6.5.

6.4.4 Run-time Library Components

This section describes RTL components that perform specific run-time tasks. They are: the *display trees interpreter*, the *main frame*, the *selection manager*, the *object-widget mapping manager* and *execution and animation manager* components. All are built on the PIN access and alteration procedures just described.

Display Trees Interpreter

Glide queries evaluate to a set of sub-trees of the PIN (*e.g.*, sub-trees of the PCM PIN of Figure 6-6). After a query is evaluated, the resulting trees are augmented to incorporate graphical attribute information in order to create a more complex tree data structure called a Glider *display tree*. These display trees are passed to a Glider RTL component, the *display tree interpreter*. This component traverses a display tree and invokes the appropriate GUI library widget creation calls, with the appropriate bindings of graphical properties, in order to render a set of widgets on the screen. Such a process is similar to the use of a “display list” data structure in some 3D graphics systems such as PHIGS.

A display node of a display tree is a complex nested data structure containing both *what* is to be displayed and *how* (graphically) it is to be displayed. The following diagram shows the structure of a node that represents an instance which contains a set of component instances:

TAG	N								
TYPE	Node								
COLLTYPE	SET								
VALUE	**DDTV**	COMPOS/GA		**DDTV**	SIMPLE/GA	**DDTV**	COMPOS/GA		
	OBJECTID	P001		**OBJECTID**	P002	**OBJECTID**	P003		
	COMPS	C2	C3	**COMPS**		**COMPS**	C4	C5	C6

In this example the instance contains a set of three component instances (P001, P002, P003); in other cases there may be only a single instance (in which case, COLLTYPE = SINGLETON). Each component instance may itself be composite (next level down in the tree) or simple (a leaf). The display directive (DDTV) field indicates whether each component is composite or simple, and it also contains graphical attributes and values (which may themselves be nested). The values C1.....C6 are nodes and the next level down in the display tree - they have the same structure as the whole diagram.

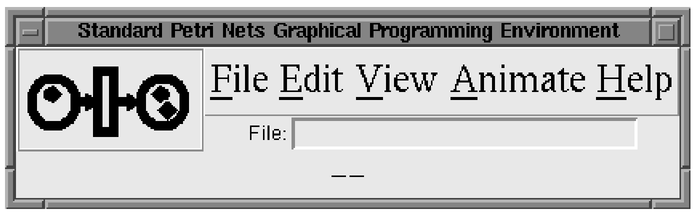
At the top level, the task of the display tree interpreter is to recursively traverse down the display tree containing such these complex nodes, examine the display directives embedded in the nodes and then perform a *dispatch*, based on the directives, to routines specialized to display according to the particular directives, passing to it the branch of the tree at that level. For example, if a tree contains the syntax tree of, say, an arithmetic expression, the display tree interpreter will invoke a dispatch routine, `DisplayTextTree,` which will collect all the leaves of the expression syntax tree and display them as a string of text in a text dialog box widget. This design for creating views makes the display tree interpreter extensible - new specialized dispatch routines can be added. The dispatch routines implemented in the demonstration system are:

- `DisplayConnected` - Provides a classical “graph” display of nodes and links.
- `DisplayList` - Displays the instances as a horizontal row.
- `DisplayText`- Displays a value of a primitive type in a text dialog box
- `DisplayTextTree` - “Unparses” a syntax tree and displays it as a string of text. This allows the string to be edited as piece of text.

The routines may recursively invoke the display tree interpreter again - for example the `DisplayList` routine might display a row of objects, each of which is a graph.

Main Frame

When a new interface is started, a simple menu bar which provides access to commands appears. Some of these commands are generic to all interfaces (*e.g.*, loading and saving program files, controlling speed of animation, etc.) and the others - those generated from their Glide specification - are specific to the particular GBVL. The Glider RTL component *main frame* contains all the procedures for the generic commands - since they do not need to be generated. The main frame window has the following appearance.



A logo bitmap identifying the particular GBVL is displayed on the left of the main frame.

Selection Manager

The Glider RTL *selection manager* component handles maintaining the set of instances selected by the user and their consumption by commands invoked by the user. This module also handles the highlighting of widgets to indicate which instances have been selected. The selection manager implements a *stack* of instances that records which instances the user has selected by clicking on the widgets that represent them. Commands (viewing or editing) can then consume (pop) one or more items and use them as arguments when they are invoked:

`gPushOnSelectionStack: instance` $\longrightarrow$ `stack`
Pushes instance on stack.

`gPopOffSelectionStack: instance` $\longrightarrow$ `stack, instance`
Pops instance off the stack.

`gFetchOutOfSelectionStack: type` $\longrightarrow$ `stack, instance`
Extracts instance of given type out of the stack.

Object-Widget Mapping Manager

A separate Glider component maintains the one-many mapping between instances and widgets (the widget back pointers). The mapping is updated as instances are created or deleted and widgets representing them are created or deleted from the screen. Entries are generally added during the execution of the display tree interpreter, as widgets representing instances in the display trees appear on the screen, and are generally deleted when the user closes and destroys a window full of widgets:

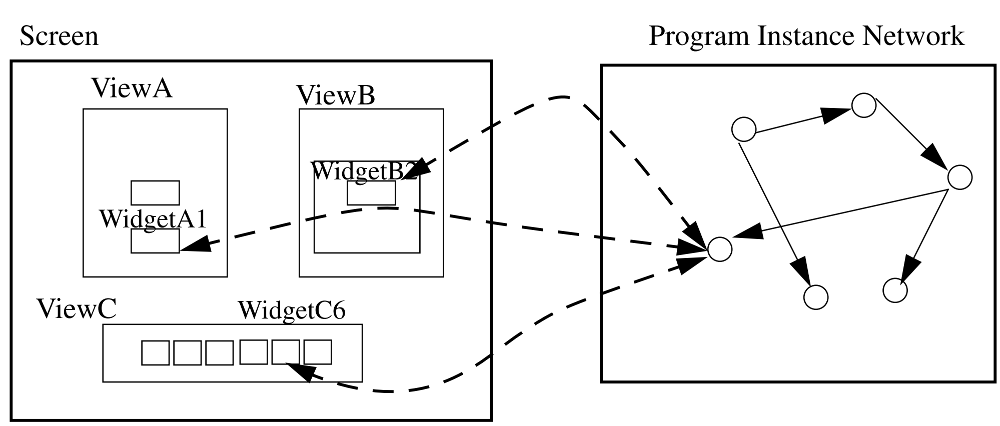


Figure 6-9 Mapping between screen widgets and PIN instances

Execution and Animation Manager

The RTL *execution and animation manager* component controls the execution of programs and the animations they cause. The component contains the procedures to select and invoke the execution actions, to read in PIN update events from the reporting functions, and to invoke the animation procedures which update the graphical appearance in response to changes in the PIN.

This component allows the user, through the menu in the main frame window, to select which of the animations provided by the designer are to be enabled.

The speed with which a program is executed and animated can be controlled by the user. The speed control is in two parts: The user can choose the time delay between the application of successive abstract *execution actions* and can also choose how fast the animation *themselves* run. Animation consists of values of graphical attributes being updated to reflect updates in execution state, and of *animation functions* which execute when the execution is (stopped) in a particular state. The animation functions have their own speed of running (*e.g,.* rate of a flashing animation function) which is independent of the speed of the execution of the program itself (be it internal or tracking external).

6.5 Glider Compiler Components

This section describes how the Glide specifications are translated into programs. The generated programs make use of the RTL procedures of level-1 and level-2 and they are invoked through the RTL manager components just described. The Glide productions are translated into classes which define the PIN. The queries, actions (editing, execution, and animation), and the shape predicates are translated into procedures which access and alter the PIN. Though parts of a Glide specification are compiled by separate sub-compilers (as was illustrated in Figure 6-2) the sub-compilers do share information through using a single symbol table. The graphical attributes part of a Glide specification does not need to be compiled. The attributes and their values are simply placed into the appropriate class definitions. During execution they are read from the objects into the display trees and passed into the display system for interpretation.

Since the Glider compiler produces code in a high level language and the Glider RTL provides and even higher level API, the "amount" of translation required to get from the Glide specification to procedural code is relatively small. For this reason, the compilation algorithms are relatively simple and straightforward. This section provides an outline of the algorithm used in each of the three sub-compilers, the *class generator*, the *queries compiler*, the *actions compiler*, and examples of small sample translations.

6.5.1 Class Generator

The Glide grammar part of a specification is translated into class definitions for the objects that implement the PIN data structure. The *class generator* sub-compiler parses a grammar specification and creates an AST which is then traversed to create class definitions in the OOPL. In the demonstration system, the object-oriented extension of tcl, itcl, was used as target OOPL, but an example of generated C++ code is also shown in this section, for comparison. The class generator component also generates a symbol table of the names of all the productions and tags - this is subsequently used for variable type checking in the query and action sub-compilers.

Class Generation Algorithm

Class generation consists of the following steps:

1. Since providing tag names for components in a Glide grammar specification is optional, any missing tags are first filled in with unique names. They are generated by the compiler and added into the AST. This is so that names for all tags are present for the following generation steps.
2. One class definition is created for each production in the Glide grammar. Data members are defined for each component of the production.
3. Members for the additional properties described earlier (meta-data, back pointers, widget pointers, graphical attributes) of each member are added to the class definition.
4. For productions containing sequences (component combined with whitespace “ ”) inside alternations, additional class definitions are created; one for each sequence. Each of these extra classes contains a PIN data member for each component in the sequence.
5. A predefined class “`Root`” of which all the generated class are subclasses is added to complete the class definitions file.

Class Generation Example

The following shows a simple example. Three productions from the Glide Grammar of the PCM GBVL are:

```
PCMStructure ==
      PN:PCMNet .
      SV:StateVariable** ;
PCMNet       ==
      N:Node** .
      L:Link** ;
Node         ==
      NT:( P:PlaceNode | T:TransNode ) .
      I:Link** .
      O:Link** ;
Pexpr        ==
      Pe:( :Pterm | :(Pexpr ‘or’ Pterm));
```

These productions are translated into the following six classes in itcl:

```
itcl_class PCMStructure {
 inherit Root
 constructor {config} {}
 protected Tags-List {PN SV}
 protected PN-type "SINGLETON PCMNet"
 protected SV-type "SET StateVariable"
 public PN
 public SV
}

itcl_class PCMNet {
 inherit Root
 constructor {config} {}
 protected Tags-List {N L}
 protected N-type "SET Node"
 protected L-type "SET Link"
 public N
 public L
}

itcl_class Node {
 inherit Root
 constructor {config} {}
 protected Tags-List {NT I O}
 protected NT-type "CHOICE P T"
 protected P-type "SINGLETON PlaceNode"
 protected T-type "SINGLETON TransNode"
 protected I-type "SET Link"
 protected O-type "SET Link"
 public NT-choice
 public P
       public T
 public I
 public O
}
```

```
itcl_class Pexpr {
 inherit Root
 constructor {config} {}
 protected Tags-List Pe
 protected Pe-type "CHOICE C0 C1"
       protected C0-type "SINGLETON
Pterm"
 protected C1-type Pexpr-C1
 public Pe-choice
 public C0
       public C1
}

itcl_class Pexpr-C1 {
 inherit Root
 constructor {config} {}
 protected Tags-List {S0 S1 S2}
 protected S0-type "SINGLETON Pexpr"
 protected S1-type "SINGLETON OR"
 protected S2-type "SINGLETON Pterm"
 public S0
 public S1 "or"
 public S2
}

itcl_class Root {
 constructor {config} {}
 public GraphicAttribs
}
```

The class and member names with numbers (`S0`, `C1`, etc.) are the system-generated names. The protected members are the meta-data, and the public members are the PIN and graphical attribute data. By using different templates and different AST traversals, the compiler can create equivalent C++ class definitions. The equivalent C++ class definitions for this example are:

```
class PCMStructure : public Root {
private:
 enum mTags {PN, SV};
 const int m_NumTags = 2;
 TagSort m_PN_Sort = SINGLETON ;
 TagClass m_PN_type = PCMNet;
 TagSort m_SV_type = SET;
 TagClass m_SV_type = StateVariable;
public:
 PetriNet* PN;
 StateVariable** SV;
};

class PCMNet : public Root {
private:
 enum mTags {N, L} ;
 const int m_NumTags = 2 ;
 TagSort m_N_Sort = SET ;
 TagClass m_N_type = Node ;
 TagSort m_L_Sort = SET;
 TagClass m_L_type = StateVariable;
public:
 Node** N ; // ptr to list of ptrs
 Link** L ;
};

class Node : public Root {
private:
 enum mTags {NT, I, O};
 const int m_NumTags = 3 ;
 TagSort m_NT_Sort = CHOICE ;
 mTags m_NT_Choices[2] = {P, T};
 mTags m_NT_Choice;
 TagClass m_P_type = PlaceNode;
 TagSort m_P_Sort = SINGLETON;
 TagClass m_T_type = TransNode ;
 TagSort m_T_Sort = SINGLETON;
 TagSort m_I_Sort = SET;
 TagClass m_I_type = Link;
 TagSort m_O_Sort = SET;
 TagClass m_O_type = Link;
public:
  PlaceNode* P;
  TransNode* T;
  Link* I;
  Link* O;
  GraphicAttribsPtr GA;
};
```

```
class Pexpr : public Root {
private:
 enum mTags {Pe C0 C1} ;
 const int m_NumTags = 1;
 TagSort m_Pe_Sort = CHOICE ;
 mTags m_Pe_Choices[2] = {C0, C1};
 mTags m_Pe_Choice;
 TagClass m_C0_type = Pterm;
 TagSort m_C0_Sort = SINGLETON;
 TagClass m_C1_type = Pexpr-C1;
 TagSort m_C1_Sort = SEQUENCE;
public:
 Pterm* C0;
       Pexpr-C1* C1;
};

class Pexpr-C1 : public Root {
private:
 enum mTags {S0 S1 S2};
 const int m_NumTags = 3;
       TagClass m_S0_type = Pexpr;
 TagSort m_S0_Sort = SINGLETON;
       TagClass m_S1_type = String;
 TagSort m_S1_Sort = SINGLETON;
       TagClass m_S2_type = Pterm;
 TagSort m_S2_Sort = SINGLETON;
public:
 Pexpr* S0;
 char*  S1 = "or";
 Pterm* S2;
};

class Root {
public:
 GraphicAttribsPtr GA;
};

enum TagClass { PCMStructure PCMNet
      Node Pexpr Pexpr-C1 };
enum TagSort { SINGLETON LIST
          SET CHOICE SEQUENCE};

typedef struct
 GraphicalAttribTree *GraphicAttribsPtr;
typedef struct GraphicalAttribTree {
  char* Attribute;
  GraphicAttribsPtr value_composite;
  char* value_primitive;
  GraphicAttribsPtr next;
} GraphicalAttribs;
```

Here the private members (prefixed with `m_`) store the meta-data. The itcl classes are a little more concise than the C++ ones because itcl, being an interpreted language, allows some meta-data to be derived. Hence they do not have to be explicitly defined in separate members (*e.g.,* the number of choices associated with a CHOICE tag can be computed from the length of the tcl list). Also, itcl is a weakly-typed language so that typing declarations for members and the auxiliary enums and structs are not needed.

6.5.2 Graph Language Queries Translation

The *queries sub-compiler* analyzes the queries in the specification and produces a parameterized procedure for each one that computes the specified collection of PIN trees. The basic query evaluation algorithm is to evaluate the tree expressions in a query in the context of all the desired combinations of input parameter values and quantified variable values in the query. The queries sub-compiler generates a procedure for each query; the body of the procedure implements this evaluation.

Queries Compilation Algorithm

The queries compilation algorithm consists of the following steps:

1. Create a procedure header with a new name for the query procedure. Open a new procedure definition.
2. Insert statements for consuming from the selection stack, and binding to local variables, values for the input parameters of the query.
3. If there is a **where** clause in the query then insert, for each quantification expression, statements for evaluating the quantification and for collecting all the results in a list of lists of values.
4. Insert a statement computing *all combinations* of values of the input variables of step 2 and quantified variables of step 3.
5. If there is a **such-that** clause in the query then insert a for-loop that filters the list of combinations of values computed in 4 with the test expression in the **such-that** clause. This loop removes all elements (combinations of values) for which the test does not hold.

6. Insert a sequence of statements for each tree path expression in the query **return-clause.** Each sequence contains a for-loop over the combinations in the filtered list. Insert statements in the body of the for-loop that:

   - Evaluate the tree path expression for the particular combination of values.

   - Call a function that collects into the list of trees (forest) collected so far the new PIN tree, merging the tree if it overlaps (has a common stem).

7. Insert code to return the forest and close procedure definition.

In addition to the basic algorithm code, additional bookkeeping statements are added as well *(e.g.*, to register the fact that when the procedure is invoked with a particular set of input values that the corresponding view exists on the screen). The forest returned by the query procedure is subsequently augmented with graphical attribute values and display directive information and passed to the display tree interpreter. The execution of the display tree interpreter produces the view on the screen.

Query Compilation Example

The following example illustrates this translation with a simple query for the PCM GBVL. The input query:

```
ShowBusyPlaces(ps:PCMStructure)==    { x.NmTk, x.Name
                                       where x member-of ps.PN.N.P
                                       suchthat x.NmTk > 3 }
```

is translated into the following procedure:

```
procedure queryproc_ShowBusy () {
   PINForest := empty // local variable in which PIN trees are collected
   // (1) get input instances from the selection stack
   ps_inst := gFetchOutOfSelectionStack("PCMStructure")
   local_vars := append(local_vars,"ps")
   local_vars_values := append(local_vars_values,ps_inst)
   // (2) generate sets of values from  where-clause quantification expressions
   x_qvalset := gEvalPathExpr(ps_inst,".PN.N.P")
   local_vars := append(local_vars,"x")
   local_vars_values := append(local_vars_values,x_qvalset)
   // (3) compute a list of all the combinations of local variable values
   combinations_list := gCrossProduct(local_vars_values)
   // (4) remove all combinations that fail the restriction test
   foreach comb in combinations_list
     testexprval := gBindAndEval(local_vars,comb,"gEvalPathExpr(x,\".NmTk\") > 3")
     if (testexprval) then
       filtered_combinations := append(filtered_combinations,combination}
     endif
   endfor
   combinations_list := filtered_combinations
   // (5) collect and merge tree expression evaluation in return list
   foreach comb in combinations_list
     idx := search(local_vars,"x")
     PINForest := gMergeTF(PINForest,gGetStarExprTree(index(comb,idx),".NmTk"))
   endfor
   foreach comb in combinations_list
     idx := search(local_vars,"x")
     PINForest := gMergeTF(PINForest,gGetStarExprTree(index(comb,idx),".Name"))
   endfor
   return PINForest}
```

Figure 6-10 Generated Query Procedure

`append` composes its arguments into a list. The routines `gMergeTF`, `gCrossProduct`, and `gBindAndEval` are support functions provided by the RTL. `gBindAndEval` takes as input a list of variables, a list of value bindings for the variables, and a boolean expression; it returns the value of the expression with the variables bound to the values. The function `gMergeTF` creates the collection of trees (`PINForest`) by merging in new trees.

6.5.3 Actions Translation

This section describes the *actions sub-compiler*. This compiler is used for the remaining parts of a Glide specification: the editing and execution actions, the shape predicates, and the animations. These parts all contain *action expressions* that are translated into sequences of statements that update the PIN. An action expression is either simple or a *conditional action expression*. The latter is composed of a boolean test expression and action expressions guarded by the test. The boolean expression can contain *access expressions* which access values in the PIN (same as **such-that** clause in a query).

All expressions are translated to appropriate calls of level-1 procedures and of the path expression evaluation procedure of level-2. After actions are invoked (either by the user or the system itself), the changes to the PIN caused by the actions are immediately propagated to any dependent views on the screen (*i.e.*, ones that contain widgets representing the relevant parts of the PIN). The basic process of translating the different categories (editing, execution, shape predicates, and animations) is the same, with some variation for each particular category. In this section, examples of translation in each category are shown, followed by a description of the specific variation of the general algorithm.

Editing Actions Translation

The editing actions specify the set of interactive operations available to the user to create and modify programs. Input parameter bindings are obtained via the selection stack in the same ways as with queries. Editing actions are usually just a conjunction of action expressions, though conditional actions can, and are, sometimes used.

Editing Action Translation Example

The following editing action specification is one which just adds a transition node to a PCM PIN, unconnected to other nodes and links:

```
AddTransNode(pm:PCMNet) == {
                    t1 = new(TransNode) ∧
                    n1 = new(Node) ∧
                    n1.T′ = t1 ∧
                    pm.N′ = pm.N ∪ n1  }
```

It is translated into the following procedure:

```
procedure editingaction_AddTransNode {} {
   // get input arguments from the selection stack
   pm_inst := gFetchOutOfSelectionStack("PCMNet")
  // perform actions
   t1_inst := gMake("TransNode");
   n1_inst := gMake("Node");
   gConnect(n1_inst,"T",gEvalPathExpr(t1_inst,"."))
   gConnect(pm_inst,"N",n1_inst)
   gUpdateViews()
}
```

Though the source and target forms are superficially similar, their semantic interpretations are quite different. As has already been discussed in Section 4.4.1, the Glide specification is a declarative statement relating the PIN before and after the editing event - hence the ordering of the conjoined expressions has no significance. On the other hand, the target code consists of ordered sequential imperative statements. The compiler re-orders the conjunction to satisfy the data dependencies implied by the action expressions before generating the sequential code. Through this analysis the compiler also catches semantic errors, such as associating a prime-equals expression with the same PIN place (*e.g.*, `n1.T′ = t1 ∧ n1.T′ = t2`).

Editing Actions Translation Algorithm

1. In the AST, perform dependency re-ordering of action expressions.
2. Open a new editing procedure definition.

3. Insert code to bind input parameters to values on the selection stack. (same as queries).

4. [Recursive step] For each action expression:

   - If the action expression is simple, insert appropriate level-2 connect statements.

   - If the action expression is a conditional expression, insert an evaluation of the boolean expression in the condition-part of an if-statement and open the then-part of the statement. Repeat this step on actions inside conditional action expression, inserting the results. Close the then-part of the statement.

5. Insert call to a procedure to update views and then close procedure definition.

The boolean expressions are evaluated in the same way as the tests in **such-that** clauses in queries.

Execution Actions Translation

Execution actions are similar to editing actions, but there are the following differences:

- The execution actions do not take user-originated input parameter bindings. Instead, they are provided with the value of the top level type of the GBVL (*e.g.*, `PCMStructure`, or `PetriNet`) through which any part of the whole PIN can be addressed via the appropriate path expression.
- Conditional actions can be existentially quantified over sets or lists in the PIN. In this case the code generated attempts to find a value in the set or list that satisfies the boolean expression and, if it succeeds, executes the guarded actions for the given value.
- Conditional actions can be universally quantified over sets or lists in the PIN. In this case the actions are executed for those values that satisfy the boolean expression.
- Simple action expressions can be universally quantified over sets or lists in the PIN. In this case the action expressions are executed for all values of quantified variables.

The quantifications are evaluated by introducing for-loops over the values of the variables and recording the boolean value of the quantified expression.

Execution Action Translation Example

The following two execution actions for Petri nets:

```
Enable(pn:PetriNet) == {
      ∀t:pn.N.T (∀p:t..NT.I.Lp (p.NmTk >0) ⇒ t.Ste' = ENABLED )}

Fire(pn:PetriNet) == {
      ∃t:pn.N.T (t.Ste = ENABLED ⇒
                      t.Ste' = FIRING ∧
                      ∀p:t..NT.O.Lp (p.NmTk' = p.NmTk + 1))}
```

are translated into the following two procedures:

```
procedure exec_action_petri_enable {
  pn_inst := gGetInstance("PetriNet")
                                        // iterate over set of forall quantification
  foreach t_inst in gEvalPathExpr(pn_inst,".N.T")
                                  // evaluate and iterate over existential quantification
    FAFlag := TRUE
    foreach p_inst in gEvalPathExpr(t_inst,"..NT.I.Lp")
      if (gBindAndEval(p,p_inst,"gEvalPathExpr(p,\".NmTk\") > 0")) then
        FAFlag := FALSE
      endif
    endfor
    if (FAFlag) then
     gConnect(gEvalPathExpr(t_inst,"."),"Ste",ENABLED)
    endif
  endfor
}
procedure exec_action_petri_fire {
  pn_inst := gGetInstance("PetriNet")
                                          // evaluate there exists quantification
  TEFlag := FALSE
  foreach t_inst in gEvalPathExpr(pn_inst,".N.T")
    if (gli_EvalPathExpr(t_inst,".Ste")==ENABLED) then
      candidates := append(candidates,t_inst)
      TEFlag := TRUE
    endif
  endfor
  if (TEFlag) then
    chosen_t_inst := gPickOne(candidates)
    gConnect(gEvalPathExpr(chosen_t_inst,"."),"Ste",FIRING)
    foreach p_inst in gEvalPathExpr(chosen_t_inst,"..NT.O.Lp")
      gConnect(gEvalPathExpr(p_inst,"."),"NmTk",(gEvalPathExpr(p_inst,"NmTk")+1)
    endfor
 endif
}
```

It is these procedures that are selectively invoked by the execution and animation manager according to the one of the firing regimes described in Section 4.4.2 (for internal simulation) or according to the reporting functions (for tracking an external execution). An existential quantification is translated into invoking the function `gPickOne`, which picks at random one of the values that made the test true, or, if the execution is tracking, follows the choice reported by the executing program.

Execution Actions Translation Algorithm

1. In AST, perform dependency re-ordering on actions, check for violations.
2. Open new execution procedure definition.
3. Insert code to bind input parameter to top level instance.
4. [Begin recursive steps]
5. If there are universal quantifiers, open a for-loop for each one. For each action expression, insert appropriate level-2 connect statements. If the action expression is a conditional expression, insert an evaluation of the boolean expression in the condition-part of an if-statement and repeat this steps from 4 on actions inside conditional action expression, inserting the results in the then-part. Close loops.
6. If there are existential quantifiers, open a for-loop for each one. These loops try all values in the range, collecting those that satisfies the condition. Insert code to chose one combination satisfying condition and repeat steps from 4 on actions inside conditional action expression, inserting the results in the then-part. Close loops.
7. Insert call to function to update views and close procedure definition.

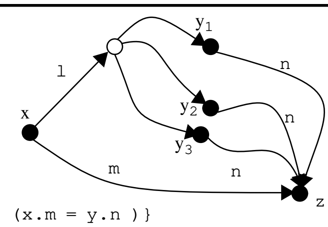
Shape Predicates

Shape predicates are used to verify consistency of the structure and they are translated into procedures which check and maintain the shape of the PIN. The time at which the procedures are applied can be set to be (i) immediately after any editing interaction with the programmer, or (ii) deferred to when requested by the programmer. The main use of shape predicates is to ensuring cyclic and shared references are up to date and consistent, but they can be used to express other static semantic constraints as well. Shape predicates state invariants. Shape predicates consist of action expressions without any primed variables and they do not use existential quantifications.

Shape Predicates Translation Example

In this example (from Chapter 4) instances of a set of one type must point to the same instance of another type:

```
X ==     l:Y** .
         m:Z ;
Y ==     n:Z ;
```



`SameSink(x:X) =={∀y:x.l (x.m = y.n )}`

It is translated into the following procedure that maintains the shape invariant:

```
procedure Same_Sink (x_inst) {
    foreach y_inst gEvalPathExpr(x_inst,".l")
      x_m := gEvalPathExpr(x_inst,".m")
      y_n := gEvalPathExpr(y_inst,".n")
      if ((x_m!=NULL)||(y_n!=NULL)) then
        if (x_m==NULL) then
          gConnect(gEvalPathExpr(x_inst,".m"),"m",gEvalPathExpr(y,".n"))
        elseif (y_n==NULL) then
          gConnect(gEvalPathExpr(y_inst,".n"),"n",gEvalPathExpr(x_inst,".m"))
        endif
      else error()
      endif
    endfor
}
```

Shape predicates are bi-directional - both connect ways to re-establish the invariant are attempted by the procedure. Note however that this procedures makes the assumption that either the `x` is pointing to a `z` or the `y`'s are.

Shape Predicate Translation Algorithm

1. Open new shape procedure definition
2. Insert code to input parameter to top level instance.
3. If there are universal quantifiers open for-loops for each one.

4. Insert connect code to maintain equality, membership, or implication, in both directions. Close loops.
5. Close procedure definition.

Animation Compilation

Animation procedures in the programming environment are triggered in response to changes in the abstracted, mirrored, execution state. The animations are specified as constraints which relate one or more PIN instances and/or values to one or more graphical attribute values, so that any change to the causes changes to the graphical attributes. The animation specifications are compiled into procedures which are invoked when the relevant parts of the PIN are changed.

Animations can not only specify changes to the values of graphical attributes, but can also specify the execution of animations functions - these usually cycle through a set of values for a graphical attribute. The compilation of animations is similar to that for shape predicates, since both ensure that the invariant properties they state are maintained. However, since the graphics are always dependent properties, it is not necessary to attempt the bi-directional updates.

Animation Compilation Example

The animation invariant:

```
EnableGreen(pn:PetriNet) ==
        { ∀t:pn.N.T (t.Ste = ENABLED ) ⇒ (t<Color> = Green)}
```

is translated to:

```
procedure EnableGreen {pn_inst) {
   foreach t_inst gEvalPathExpr(pn_inst,".N.T")
     if (gEvalPathExpr(t_inst,".Ste")==ENABLED) then
       gSetGA(gEvalPathExpr(t_inst,".NT"),"Color",GREEN)
     endif
   endfor
   gGraphicalUpdateViews()
}
```

The `gGraphicalUpdateViews` call causes invocation of calls to the GUI so that views that contain widgets representing the instance that has the attribute, is updated. Note that the user must also supply further animation invariants the color again.

Action Compiler Translation Algorithm

1. Open new shape procedure definition
2. Insert code to input parameter to top level instance.
3. If there are universal quantifier open for-loops for each one.
4. Insert connect code to maintain truth of invariant by modifying the graphical equality specified. Close loops.
5. Insert call to propagate *graphical attribute* updates to views and close procedure definition and close procedure definition.

6.6 Summary

This chapter has described the design and implementation of the Glider system. The system consists of a run-time library and a compiler. The RTL provides the interface to the GUI library and supports the programs generated by the compiler with the operations for manipulating PIN data structures. The compiler translates Glide type definitions that characterize the syntax of a GBVL into classes of an OOPL and it then generates procedures for accessing and altering the objects of the classes from Glide the query and action definitions. The graphical attributes of the GBVL are stored in the classes and passed to the run-time component which creates views. Animation is accomplished by routines which reflect the execution state of the program represented in the PIN in the values of graphical attributes. RTL manager components invoke the compiled-generated procedures and maintain consistency between the GUI interface, the PIN and the executing program being animated.

The next chapter shows results: it provides three examples of specifying typical GBVLs and resulting displays.

Chapter 7

Examples

This chapter presents three detailed examples which demonstrate the use of Glide in specifying programming interfaces for GBVLs. The three examples are:

- Boolean Circuits,
- PCM - a variant of Petri Nets for modeling parallel systems performance, and
- LabVIEW - a part of an early version of this commercial GBVL for expressing computations associated with data acquisition instruments.

The three examples were chosen to illustrate both the breadth of structures and of models of execution that can be captured in Glide, and to illustrate a progressive increase in complexity of structure and semantics with each example. Each section presents an example, discusses variations on ways in which the GBVL can be represented and notes those aspects of the semantics of the GBVL which are easy or difficult to capture in Glide.

7.1 Simple Boolean Circuit

This section shows one way in which simple combinational boolean circuits can be described in Glide as a GBVL. This particular Glide specification includes a complete execution semantics - the values that a boolean circuit computes - as well as all the intermediate computed values on the wires.

7.1.1 Glide Grammar for Boolean Circuit

This Glide grammar represents boolean circuits:

```
Circuit   ==  G:Gate** . W:Wire** ;
Gate       == GT:(Or:OrGate | And:AndGate | Inv:Inverter ) .
              Lbl:TEXT ;
AndGate   ==  In1:InPort .
              In2:InPort .
              Out:OutPort ;
OrGate    ==  In1:InPort .
              In2:InPort .
              Out:OutPort ;
Inverter  ==  In1:InPort .
              Out:OutPort ;
InPort    ==  Vol:BOOLEAN .
              W:Wire ;          // an input port can have only one wire
OutPort   ==  Vol:BOOLEAN .
              W:Wire** ;          // an output port can have many wires
Wire      ==  Sr:OutPort .                // wire links source and sink
              Si:InPort ;
```

Though this grammar is small and simple, there are already significant design choices and some semantics of boolean circuits reflected in it:

- There are two kinds of ports on the gates, `InPort` and `OutPort`. The distinction captures the fact that a gate's output can be attached to many wires, but a gate's input to only one.
- All ports are associated with a gate. It would be possible to change the grammar so as to include the set of all ports in the top level `Circuit` production. This is useful for representing ports which are attached to ends of wires without being associated with any gate. Ports "at the edge of circuit board" could then be included, allowing the user to set input values to the "board".
- It is possible to have a more compact specification. For example, it is possible to use aggregation (.) and an intermediate production to "factor" out the fact that gates are either one-input or two-input. In a language as simple as this one it is not really needed, but as a GBVL becomes more complex such factoring is becomes useful.

7.1.2 Editing Semantics for Boolean Circuit

The following list of edit operations provide a simple set of actions for creating and deleting gates and wires and for connecting them together.

```
AddAnd(c:Circuit) == { g = new(Gate.And.*) ∧ c.G′ = c.G ∪ g }
AddOr(c:Circuit) == { g = new(Gate.Or.*) ∧ c.G′ = c.G ∪ g }
AddInv(c:Circuit) == { g = new(Gate.In.*) ∧ c.G′ = c.G ∪ g }
AddWire(c:Circuit) == { w = new(Wire) ∧ c.W′ = c.W ∪ w }
Connect(i:InPort,o:OutPort) ==  { w = new(Wire) ∧
                                  i.W′ = w ∧ o.W′ = o.W ∪ w ∧
                                  o..Out..GT..W′ = o..Out..GT..W′ ∪ w }
Disconnect(w:Wire,g:Gate) == {
  (w = g.GT.In.W ⇒ g.GT.In.W′  = NULL ∧ w.Si′ = NULL) ∧
  (w = g.GT.In1.W ⇒ g.GT.In1.W′ = NULL ∧ w.Si′ = NULL) ∧
  (w = g.GT.In2.W ⇒ g.GT.In2.W′ = NULL ∧ w.Si′ = NULL) ∧
  (w ∈ g.GT.Out.W ⇒ g.GT.Out.W′ = g.GT.Out.W - w ∧ w.Sr′ = NULL)}
DeleteWire(w:Wire) ==  { old(w) }
DeleteGate(g:Gate) ==  { old(g) }
SetVoltage(i:InPort, v:BOOLEAN) ==  { i.Vol′ = v}
```

The first four commands just create individual circuit objects and add them to a circuit. These single-object creation edit operations are the kinds of operations that can be associated with a “palette” from which circuit objects drag-and-drop into a view showing the circuit. The disconnect operation detaches a selected wire from a selected gate. The wire itself is not deleted, since it is still attached to its other gate. The disconnect operation is complex because it ensures the that mutual references between ports and wires are properly maintained (removed). An alternative means of maintaining them is through shape predicates. The last command allows the user to set input voltage values.

7.1.3 Execution Semantics for Boolean Circuit

In this simple system, the complete execution semantics of the language of boolean circuits can be captured within Glide. Four execution action rules, one for each type of gate and one to propagate the signal along the wire are sufficient.

```
EvalAnd(c:Circuit) ==  ∀a:c.G.And(a.Out.Vol' = a.In1.Vol && a.In2.Vol)
EvalOr(c:Circuit)  ==  ∀o:c.G.Or(o.Out.Vol' = o.In1.Vol || o.In2.Vol)
EvalInvert(c:Circuit) == ∀i:c.G.Inv(i.Out.Vol' = !(i.In.Vol))
Propagate(c:Circuit) == ∀w:c.G.W (w.Si.Vol' = w.Sr.Vol)
```

Selecting the action firing regime *round_robin* allows all the gates to compute their outputs (one atomic execution step for each type of gate). These are followed at the end by an execution step in which the values at the output ports are propagated along the wires to the input ports. Alternative forms of execution control are possible:

- The first three actions could be combined into a conjunction in order to have all the gate output values produced in one atomic execution step.
- “Clocked” execution of the gates could added by introducing a boolean component to represent the clock in the Glide grammar - one which is shared by all the gates, so that all the gates only compute their outputs when the clock value is true.
- A *data-driven* execution semantics would require a more extended representation of state of execution - one in which additional variables associated with each gate were added to the grammar and used to record the arrival of data.

7.1.4 Graphical Attributes for Boolean Circuit

The graphical attributes below specify that in a graph display the wires should be shown as lines, the gates as boxes and they identify icon images for the different types of gate.

```
Circuit  ==   {{GRAPH {N W}}}
Gate     ==   {{INGRAPH {SHAPE Box}
                     {ConnectedThrough {.GT.Out.W .GT.In1.W
                                         .GT.In2.W .GT.In.W}}
               {ICON gate.xbm}}
AndGate ==    {{ICON and.xbm}}
OrGate ==     {{ICON or.xbm }}
Inverter ==   {{ICON inv.xbm }}
Wire ==       {{INGRAPH {SHAPE Line}
                     {ConnectedThrough {.Sr .Si}}}}
```

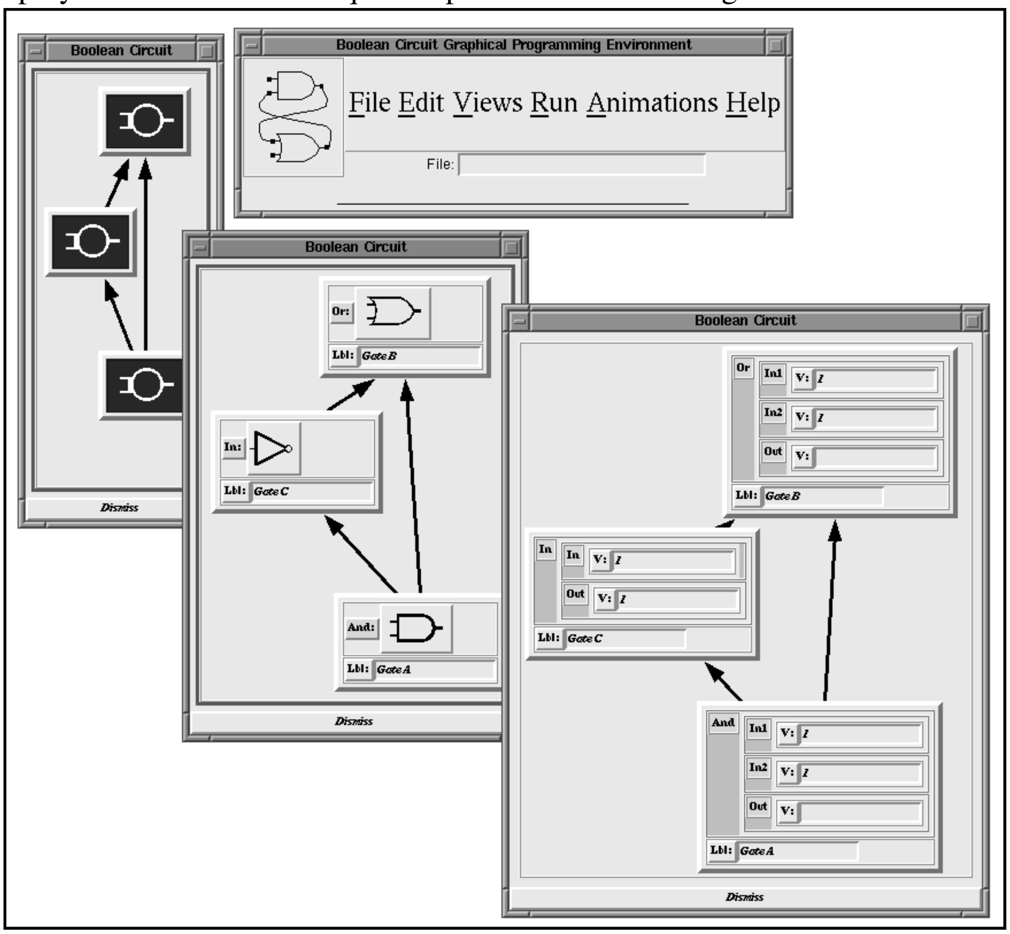
Note that an icon for a generic gate is associated with the type `Gate`. Additional attributes specify how to create the graph display. The path expressions in the `ConnectedThrough` attribute indicate to the graph display renderer how the widgets representing gates and wires are to be displayed interconnected - the path expressions are derived from the relevant shape predicates (see 7.1.7).

7.1.5 Queries for Boolean Circuit

The following four queries provide progressive levels of detail in displaying Boolean Circuit GBVL programs.

```
ShowTop(c:Circuit)== { c.*}
ShowWhole(c:Circuit)== { c.*.*}
ShowValues(c:Circuit)== { c.*.*.*.*}
ShowOneGate(g:Gate)== { g.*.*.*}
```

With only these simple queries and the small number of graphical attributes - all other graphical features of the views being left to defaults of the display renderer - the three queries produce the following views on a circuit.



The ShowValues view allows direct user access to changing the voltage logic values of the ports (of the gates). The fourth query can be used in conjunction with selecting a single node out of ShowTop or ShowWhole views in order to display it to the level of detail of the ShowValues view (rightmost in figure), but in its own separate widow.

7.1.6 Animation of Boolean Circuit

The following two simple animation actions specify that the voltage logic value at the output (sink) end of the wires should be reflected on the wires by coloring the wires.

```
TrueRed(c:Circuit) == {∀w:c.W {w.Si.Vol = 0 ⇒ (w<fill> = red)}
FalseGreen(c:Circuit)== {∀w:c.W {w.Si.Vol = 1 ⇒ (w<fill> = green)}
```

`fill` is an attribute of the (Tk) widget for specifying its color. The sink value is chosen so that the wire color indicates that a value *has* propagated. The animations take place in all view windows in which wires appear.

7.1.7 Shape Predicates for Boolean Circuit

The following list of shape predicates describe the cyclic structure of mutual references between gates, ports, and wires.

```
OutAttached(c:Circuit) ==
        ∀w:c.W,∀o:c.GT.Out{w.Sr = o ⇔ w ∈ o.W}
InAttached(c:Circuit) ==
        ∀w:c.W,∀i:c.GT.In{w.Si = i ⇔ w = i.W}
In1Attached(c:Circuit) ==
        ∀w:c.W,∀i:c.GT.In1{w.Si = i ⇔ w.Si = i}
In2Attached(c:Circuit) ==
        ∀w:c.W,∀i:c.GT.In2{w.Si = i ⇔ w.Si = i}
```

The set of editing actions shown earlier do not allow the structure to become inconsistent so these shape predicates serve to chracterise the legal structures but are not needed for creating the system.

7.2 Complex Petri Nets - PCM

The second example consists of a Glide specification of an extension of basic Petri Nets, “PCM” (Parallel Computations Model). PCM is a directed graph based language intended for performance modeling of parallel computations [BA88] [Adi88]. PCM is one of a very large number of variations on the basic Petri Net model that have been developed over the years. The major extensions provided by PCM over Petri Nets and shown here are the use of hierarchy and the use of additional (textually specified) attributes. The textual attributes are used to specify the semantics of PCM program nodes and give PCM more expressive power than simple Petri Nets for describing parallel systems. In PCM, hierarchy is through the transition nodes, *i.e.*, a PCM transition node may either be a primitive transition node or a composite transition node which contains a lower level PCM subnet.

7.2.1 Glide Grammar for PCM

The Glide Grammar for PCM consists of the specification of the different types of nodes and links, and their textual attributes. The top level productions capture the basic PCM graph structure:

```
PCMStructure ==      PN:PCMNet .
                     SV:StateVariable* ;
PCMNet ==            N:Node** .
                     L:Link** ;
Node   ==            NT:(P:PlaceNode|T:TransNode) .
                     I:Link** .
                     O:Link** ;
PlaceNode ==         NmTk:INTEGER .
                     Rp:INTEGER ;
TransNode ==         TT:(Prt:PrimitiveTransNode|Sbt:SubnetTransNode).
                     Nm:TEXT .              // Name for labelling node
                     Ta:TransAttributes ;
PrimitiveTransNode==  ;
SubnetTransNode ==   It:TransNode .
                     Tt:TransNode .
                     PN:PCMNet .
                     Rp:INTEGER ;             // replication parameter
Link   ==            Lt:TransNode .
                     Lp:PlaceNode ;
```

The recursion of `PCMNet` type through `SubnetTransNode` is the way in which the hierarchical nature of the PCM language is represented. The `Rp` components are parameters which specify run-time replication of nodes.

The following lower level productions capture the syntax of the textually specified parts of PCM programs. These are mostly attributes associated with the transition nodes. Transition nodes have: a predicate expression (`Pi`) on state variables which must be true for the node to fire; a sequence of procedures (`Proc`) which are executed when the node fires to assign new values to the state variables, and a delay (`Tau`) - a fixed amount of time before the node fires. The Glide syntax of the text expressions were trivially derived from their BNF [BA88].

```
TransAttributes == Pi:Pexpr .
                   PiVal:PexprVal .       // Predicate result value
                   Phi:Proc .
                   PhiVal:REAL           // Procedure result valuel
                   Tau:REAL .                         // Time delay
                   Ste:State ;
State ==     St:( :ENABLED | :DISABLED | :ACTIVE )
Pexpr ==     Pe:( :Pterm |:(Pexpr '||' Pterm) )
Pterm ==     Pt:( :Pfactor | :(Pterm '&&' Pfactor) )
Pfactor==    Pf:( :Pexpr |:Patom |:TRUE |:FALSE |:('!' Pfactor))
Patom ==     Pa:(:(Aexpr '=' Aexpr) | :(Aexpr '!=' Aexpr) |
              :(Aexpr '>=' Aexpr) | :(Aexpr '=<' Aexpr) |
              :(Aexpr '>' Aexpr)|:(Aexpr '<' Aexpr) )
Aexpr ==      Ae:(:Aterm | :(Aexpr '+' Aterm) | :(Aexpr '-' Aterm) )
Aterm ==      At:(:AFactor |
                  :(Aterm 'MOD' Afactor) |
                  :(Aterm '/' Afactor) |
                  :(Aterm '*' Afactor) )
Afactor== Af:(:('(' Aexpr ')') |:StateVariable |
                   :REAL |:('-' REAL)
Proc  ==      Pr:( :Assnmt | :(Proc ';' Assnmt))
Assmt ==      As:( StateVariable ':=' Aexpr)
StateVariable==  VarName:STRING .
                  VarVal:REAL
```

The execution state of a node during execution is represented by the values of the enumerated type `State`. This includes the `ACTIVE` state which does not exist in standard Petri Nets. It is used to indicate that execution is currently active within the node's subnet .

The admissibility of sharing in Glide is being exploited in this specification in two ways, beyond its use in representing interconnection of nodes and links:

- In the `SubnetTransNode` production, the components tagged `It`, and `Tt` specify the distinguished nodes in the subnet `PN` that are the "initiating" transition and the "terminating" transition nodes respectively. These two nodes are shared components with the nodes in the subnet `PN`.
- The top level `SV` component refers to a global list of PCM "state variables". These variables are present in the expressions associated with transition nodes. Glide sharing is used in order to represent the fact that state variables are *PCM* shared variables accessible by all expressions in transition nodes. This means that the text expressions are no longer strictly pure BNF text expressions since (i) one of expression components uses the aggregation operator (.) and (ii) they contain composite components which are shared (with other expressions and the list of all the state variables, in `SV:StateVariable*`).

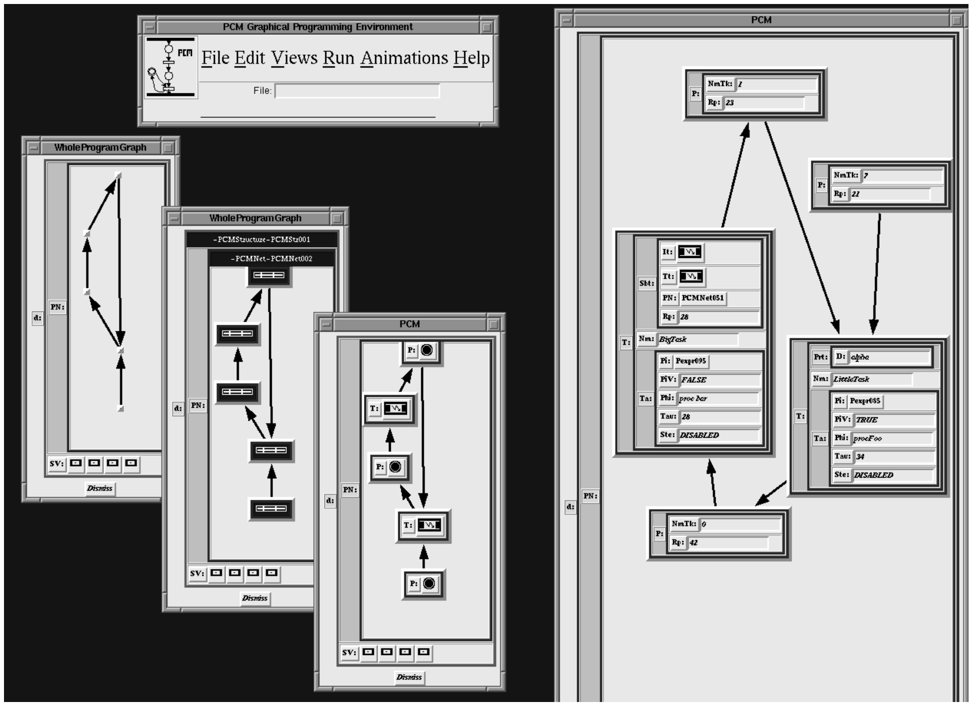
The extra uses of sharing is a style of use of Glide in which more of the semantics of a GBVL is reflected in the specification of its structure. Use of sharing is at the discretion of the Glide user. Its disdavantage is that it makes the Glide grammar more dense, less hierarchical and thus more difficult understand. It's advantage is that the specification of structure and semantics of a GBVL can become very compact.

7.2.2 View Queries for PCM

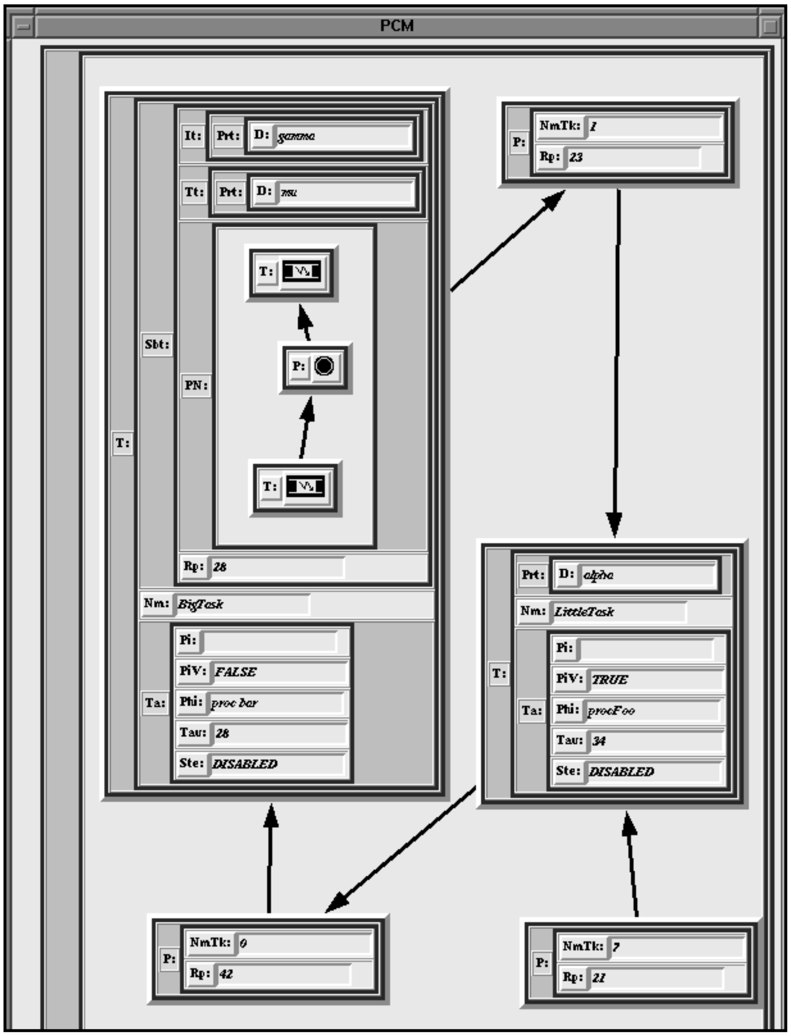
The following is a set of queries, which, like those used for boolean circuits, provide progressively more detailed views of PCM programs.

```
ProgramTop (p:PCMStructure) == { p.*, p.PN.*}
ProgramNodeTypes(p:PCMStructure) == { p.*, p.PN.*.*}
ProgramNodesDetail(p:PCMStructure)== {p.PN.*.*.*}
ProgramSubNets(p:PCMStructure)=={p.PN.*.*.*, p.PN.T.Sbt.PN.*.*}
```

The first three queries provide the views shown below. Note the list of icons along the bottom, one for each PCM state variable (`SV`).



The fourth query produces the views shown below. The hierachical nature of PCM can be seen directly, with one of the transition nodes being a subnet transition node containing a net of three nodes. The two nodes in the entries `It` and `Tt` at the top of that transition node are the initiating and terminating transition nodes which are also in the subnet. This is made directly evident to the interface user; selecting either one will highlight the node in both places, because it is the same object.



7.2.3 Execution Semantics for PCM

The following rules capture an abstract description of the PCM execution semantics. The execution model is abstract in that the full semantics of PCM predicate and procedure evaluation is not included, only the completion of the their evaluation is represented. The semantics of PCM also incorporates time - a delay time is associated with each transition node. This is outside the Glide model. These lower level details can be provided by an external real

PCM program if it is available. In the abstract execution semantics below the fact that a predicate is true is represented in a separate component (`PiV`) .

```
Enable(pn:PCMNet) ==
      { ∀t:pn.T (∀p:t..NT.I.Lp (p.NmTk > 0)) ∧ t.Ta.PiV = TRUE
                     ⇒ t.Ta.Ste′ = ENABLED }
Disable(pn:PCMNet) ==
      { ∀t:pn.T(∃p:t..NT.I.Lp(p.NmTk = 0) ∨ t.Ta.PiV = FALSE)
                     ⇒ t.Ta.Ste′= DISABLED }
PrimitiveFire(pn:PCMNet) ==
      { ∃pr:pn.T.Prt(pr..TT.Ta.Ste = ENABLED )
                     ⇒ pr..TT.Ta.Ste′= DISABLED ∧
                     ∀p:pr..TT..NT.I.Lp(p.NmTk′ = p.NmTk + 1) ∧
                     ∀p:pr..TT..NT.I.Lp(p.NmTk′ = p.NmTk - 1)}
SubNetActivate(pn:PCMNet) ==
       {∃st:pn.T.Sbt{ st..TT.Ta.Ste = ENABLED
                     ⇒ st..TT.Ta.Ste′ = ACTIVE ∧
                     ∀p:st..TT..NT.I.Lp(p.NmTk′ = p.NmTk - 1) ∧
                     st.It.Ta.Ste = ENABLED}         // start the subnet
SubNetComplete(pn:PCMNet)==
      {∃st:pn.T.Sbt{ st..TT.Ta.Ste = ACTIVE ∧ st.Tt.Ta.Ste = ENABLED
                     ⇒ st..TT.Ta.Ste′ = DISABLED ∧
                     st.Tt.Ta.Ste′ = DISABLED ∧
                     ∀p:st..TT..NT.I.Lp(p.NmTk′ = p.NmTk + 1)}
```

These execution rules reflect the more complex execution model for PCM as compared to standard Petri Nets. The values of predicates associated with Transition nodes must be true for a node to fire or become active. A primitive transition node fires immediately; tokens are removed from input place nodes and added to output place nodes in one event. In the case of a subnet transitions, the net within the node is run to completion in between removing and adding tokens. Note the use of the expression `st.It.Ta.Ste` and `st.Tt.Ta.Ste` to express accessing the initiating and terminating transition nodes of a subnet.

7.2.4 Animations for PCM

The following list shows a collection of four simple animations. The first two reflect the state of the nodes by their color. The second two animation actions color and change the width of links according to whether the place nodes they orignate from have any tokens and thus whether the links are contributing to enabling or preventing the transition node they are connected to from firing.

```
EnabledGreen(pn:PCMNet) ==
        {∀t:pn.N.T(t.Ta.Ste = ENABLED⇒ t<background> = green)}
DisabledRed(pn:PCMNet) ==
       {∀t:pn.N.T(t.Ta.Ste = DISABLED⇒ t<background> = red)}
ToksForLink(pn:PCMNet)   ==
       { ∀l:pn.N.P..NT.O
               ( l.Lp.NmTk > 0 ⇒ l<width> = 3∧ l<fill> = green) }
NoToksForLink(pn:PCMNet) ==
       { ∀l:pn.N.P..NT.O
               (l.Lp.NmTk = 0 ⇒ l<width> = 1 ∧ l<fill> = red }
```

7.3 The LabVIEW GBVL

This section provides a final example, that of the LabVIEW graphical programming language. More specifically, the Glide specification presented in this section was derived from [Dye89], which provides an informal english description of the syntax, semantics, and graphical appearance of an early version of LabVIEW. The commercial system has evolved, but the compact and self-contained Dye description provides an ideal source from which to derive a Glide specification and compare it.

In essence LabVIEW is a hierarchical dataflow language, but it also integrates typical sequential programming control constructs. It also provides support for composing and decomposing collections of data values through nodes specific for that purpose. LabVIEW is a more complex language and has a large Glide specification that reflects this. The overall structure of LabVIEW as represented with Glide grammar productions is presented here. Only some salient aspects of the specification are shown here, with a more complete specification is provided in the Appendix. Some of the important Grammar productions are shown here, and some representative definitions of the semantics as a representative example of LabVIEW node execution.

7.3.1 Glide Grammar for LabVIEW

The following top level productions identify the major types of independent components in LabVIEW. These are: "Terminals" the LabVIEW term for ports, "Signals" the LabVIEW term for links through which data flows. Terminals have values and connect signals to Nodes. Nodes perform computations on the values. The values passed have user defined numeric-based data types (scalar, array) .

```
BlockDiagram ==
             S:Signal** .
             N:Node** .
             T:Terminal** .       /* only the 'independent' terminals */
            TL:TextLabel** ;     /* set of comments */

Node  ==
      MT:(IUN:InstrumentUseageNode |    /* attached to virtual instrument */
           SN:StructureNode |
          AMN:ArrayManipulationNode |
           BN:BundleNode) .           /* gather and split values */
           Enable:BOOLEAN .                /* for execution semantics */
            Cmmnt:STRING;             /* to add a comment */
```

Diagrams can contain five major types of nodes. Of these, the `StructureNode` node type which provides different kinds of control flow and hierarchy. These nodes are one of four kinds: for loop, while loop, select (similar to a switch/case type statement), sequence (for executing a fully ordered sequence of computation in order).

```
ForLoopNode ==
             LLT:LoopLimitTerminal .       /* Loop Limit Terminal */
            ItCnt:IterationCountTerminal .  /* Iteration Count */
             InTn:TunnelTerminal** .       /* Inputs */
            OutTn:TunnelTerminal** .       /* Outputs */
             Enable:BOOLEAN .              /* For execution semantics */
             SbD:BlockDiagram ;            /* Sub Diagram, body of loop */
ForLoopLimitTerminal ==
            Val:INTEGER .                   /* Loop Limit Value */
            Name:STRING ;
WhileLoopNode
      ==    ItCnt:IterationCountTerminal .
            LCond:LoopCondition .
            ShftR:ShiftRegsiter** .
            ItCnt:INTEGER ;
WhileLoopConditionTerminal
      ==    Val:BOOLEAN .
            Name:STRING ;
/* auxilliary productions used in both loop productions */
IterationCountTerminal                /* also used in while loop */
      ==    Val:INTEGER .
            Name:STRING ;
ShiftRegister ==
          Val:Value /* records previous value */
```

Terminals in LabVIEW are more complex than ports for a simpler language such as Boolean Circuits. They are used in LabVIEW for binding signals to different parts of the computation nodes. The grammar reflects the different types of terminal which depend on their role in the computation. Signals connect terminals.

```
Terminal ==
        TT:( FPCT:FrontPanelControlTerminal |
               BDC :BlockDiagramConstants) .
               Val :Value
           Cmmnt:STRING ;         /* comment for describing terminal */
FrontPanelControlTerminal ==
         FPC:FrontPanelControl .   /* Input from the User Interface */
         DT: DataType ;           /* adapts to front panel */
BlockDiagramConstants
      ==   PT:(Pi:3.141 | e:2.71 | TRUE:0 | FALSE:1) ;
```

The next level of detail of the grammar describes the structure of the array manipulation nodes, the bundling nodes and the structure node. The array manipulation nodes are used to extract values from arrays or to compose them back into arrays.

```
                                                     // gather and split values
ArrayManipulationNode==
                    :ArrayElementReplacement |
                    :ArrayIndexerNode |
                    :ArrayBuilding
ArrayElementReplacement==
                    :ArrayInputTerminal .
                    :ScalarValueTerminal .
                    :NumericIndex** .
                    :ArrayOutputTerminal ;
                          // slicing data out of multidimensional array
ArrayIndexerNode== :ArrayInputTerminal .
                    :Index** .
                    :ArrayOutputTerminal ;
ArrayBuilderNode== :ArrayOutputTerminal .
                    :ArrayInputTerminal* ;
```

The Bundler nodes are used to pack and unpack values.

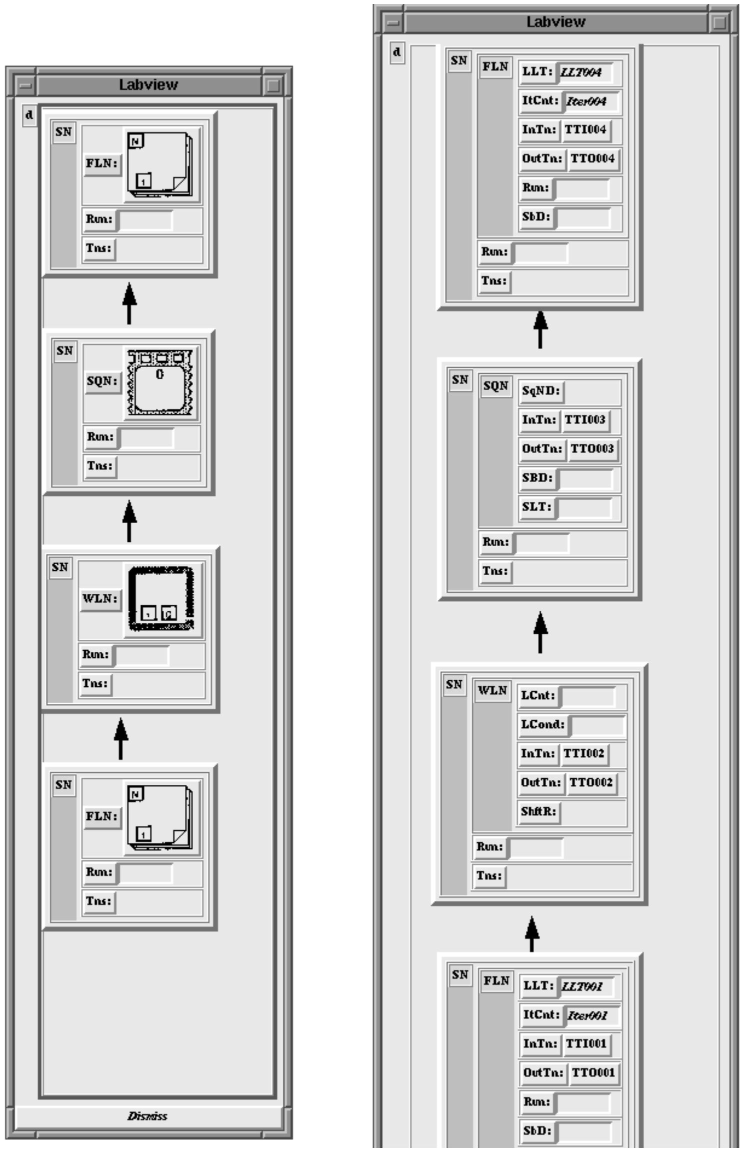
```
BundleNode   ==     :BundlerNode |
                    :UnBundlerNode ;
BundlerNode  ==     :InputTerminals** .
                    :OutputTerminal .
                    :DataType ;
UnBundlerNode==     :OutputTerminals** .
                    :InputTerminal .
                    :DataType ;
```

7.3.2 Queries for LabVIEW

These pictures are graphically much more primitive than the actual LabVIEW User Interface, but the Glider generated version is quite useable and its specification much more compact.

```
ProgramLevel_1 (bd: BlocDiagram) == { bd.*.*.*}
ProgramLevel_2 (bd: BlocDiagram) == { bd.*.*.*}
ProgramLevel_3 (bd: BlocDiagram) == { bd.*.*.*.*}
```

The queries produce various levels of detail in views:



7.3.3 Execution Semantics for LabVIEW

The execution semantics of LabVIEW is quite large and complex. This section just focuses on the execution semantics of loop node types as representative examples. The execution actions shown here are again derived from their description in the Dye thesis.

The execution action that specifies its behavior is the following.

```
ExecForLoop(bd:BlockDiagram) == {
 -TE-:fl:bd@(.N.SN.CT).N.SN.FLN {
    (fl..CT.Enable = TRUE)
      => (fl.Enable = FALSE // loop start
                    => fl.Enable' = TRUE &&
                     fl.ItCnt' = 0 )
         (fl.Enable = TRUE && fl.ItCnt < fl.LLT.Val      /* during loop */
                    => fl.ItCnt' = fl.ItCnt + 1 &&
                       fl.SbD.Enable' = TRUE )
          (fl.Enable = TRUE && fl.LLT.Val = fl.ItCnt     /* loop termination */
                    => fl.Enable' = FALSE &&
                     fl..Sn.Enable' = FALSE)
      }
}
```

This execution action rule checks if the loop has been enabled (by a another rule which sets the `Enable` attribute of the enclosing structure node to `TRUE`). If so, the `Enable` attribute is set to true and three possible actions can occur: initializing the loop, performing an iteration, or terminating the loop. The enabling of the node is controlled by other execution action rules which enable a node if values have arrived on all inputs. The incremented iteration count (`ItCnt`) is available as input (more specifically as an input terminal) to the block diagram enclosed by the for loop.

The while control loop has a similar execution semantics specification. In this case the loop termination condition is a boolean terminal that takes its a boolean value.

```
ExecWhileLoop(bd:BlockDiagram) == {
 -TE-:wl:bd@(.N.SN.CT).N.SN.WLN {
    (wl..CT.Enable = TRUE )
      => (wl.Enable = TRUE && fl.LCond = 1                    /* during loop
             => fl.Active' = TRUE )
         (wl.Enable = TRUE && fl.LCond = 0                      /* end loop
                    => wl.Enable' = FALSE &&
                    wl..Sn.Enable' = FALSE)
      }
}
```

The semantics of bundle and array index nodes are examples of semantics which difficult to represent in Glide. This is simply because the primitive types provided by the Glide language itself does not include arrays. There are two remedies: (i) use Glide lists to simulate arrays so that arrays exist as extensions of the PIN structure; or (ii) expose array manipulation primitves if they are provided by the implementation langauge (C or Tcl in the case of the prototype).

7.4 Summary of Results

This chapter has given examples which demonstrate how the Glide Language concepts described in earlier chapters allow the specifier to achieve direct compact and integrated descriptions of GBVL language syntax and semantics and associated interface editing and animation, and that this can be done in a broad range of GBVLs. This is possible because of the unified approach integrating text and graph structure, and because of the unified approach to specifying execution and editing semantics.

Chapter 8

Conclusions and Further Work

This final chapter summarizes the contributions of the work described in this dissertation and suggests avenues of further research which can now be pursued based on it.

This dissertation has described a high level language for specifying graph-based visual languages and their programming environments. The essential idea developed in this dissertation is that of a model for representing the graph-based visual languages. The structure of GBVLs, in which programs consist of a combination of text and graph structure, is modeled with a small but powerful set extensions over BNF, so that the combination of graph and text structure is captured in a seamless way. The model then also allows the basic definition of the structure of the language to be augmented with additional semantic components and with graphical attributes. The data model provides the foundation upon which specification of access and alteration to data is built. These are constructs for identifying pieces of data (the path expressions, the tree path expressions, and the queries) and constructs to express change to pieces of data (the editing, execution, and animation actions). The constructs allow the specification of views of programs, and editing, execution, and animation semantics to be captured so that a structure-oriented graphical programming environment for the GBVL can be generated automatically.

The way of characterizing GBVLs has been validated by creating a working compiler that implements the translation algorithms for each part of the specifications, and by creating a run-time library to support the generated code. Several existing graph-based visual languages have been modeled and programming environments for them produced.

The work on Glide and Glider also represents solving a problem by drawing on concepts and solutions to problems from different areas of research. The design of the Glide language and the Glider generator has drawn on a wide variety of areas including: graphical user interface design, language specific programming environments, programming environment generators, specification of the semantics of programming languages, specification of complex data types, graph grammars, data models and query languages, and functional languages. In this respect it represents a synthesis of the ideas taken from these areas.

The Glide language and Glider systems now open several new avenues for further exploration. These can be divided into two main categories: (i) relatively straightforward enhancements and variations to the current implementation of the Glider system and (ii) pursuing the ideas embodied in Glide and Glider further:

(i) For the short-term:

- An obvious immediate extension to Glide would be to allow the user to formulate his/her own view queries, so that views were not only tailored to a particular language but could also be created add-hoc. For example, the user could define highly diagnostic views by specifying particular conditions to be met by the executing program for components to appear in the view - an advanced form of animation. These would be expressed as suchthat-clause boolean expressions on the execution semantics components. There is very little that stands in the way of this since the compilation of queries is fairly simple - performing their compilation at run-time would not be too difficult.
- Glide in Glide: A programming environment for Glide could be created by specifying Glide in Glide. A graph-based view of the Glide grammar is fairly straightforward - along the lines of the way railroad diagrams are related to BNF. The way in which queries and actions could be recast as GBVLs is less clear but worthy of investigation. Such a system would then allow visual programming environments to be created with a visual programming environment.

- More experiments: Glide offers up a new framework within which to experiment with new graphical languages. The examples of the previous chapter show that Glide and Glider form a "workbench" that provides the opportunity to better, more quickly and more frequently explore the utility of a GBVL as a means for solving problems. Previously, the time and expense of such an endeavour would too often have been deemed too costly. Glide is intentionally open-ended so that completely new variations of GBVL structure which have not been used very much can be explored, *e.g.*, Graph languages with subgraphs associated with edges, edges with more than two end points, even graphs as one component in a text sequence.

(ii) For longer term investigation, there are several extensions to the syntax and semantics of Glide that could be explored:

- GBVLs with user defined data types. Glide is weak in the ability express GBVLs in which new data types can be defined. Examining how easy it would be to add this to Glide merits further study.
- There are several ways in which the path expressions could be made more expressive, *e.g.*, limits on depth of recursion.
- Glide is a general purpose programming language. The data modeling available in Glide is very expressive it may well be a useful model for specifying graph-based computations independent of whether a graphical user interface for it.

In conclusion, the work on Glide and Glider represents an advance on formalizing a significant segment of visual forms of specification. By focussing on the graph-based visual forms that are so frequently used in communicating about computing systems designs it has been possible to put them on a more sound and formal footing.

Appendix

LabView in Glide

The following is a LabVIEW specification in Glide, derived from Rob Dye's original thesis, pages 38-59. Comments are enclosed in /*...*/ and italicised.

```
/* LabView Syntax represented in Glide Grammar */
BEGIN_GLI_GRAMMAR
/* ---- TOP LEVEL ------------------------------ */
/* Root of block diagram hierarchy. States that a Labview program  consists of
Nodes, Terminals, and Signals. The user can add text labels for documentation
comments. Only self-standing, "independent", terminals are listed in TL, there
are other "dependent" terminals which are components of nodes since they only ex-
ist with the nodes */

 BlockDiagram ==
              S:Signal** .
              N:Node** .
              T:Terminal** .              /* only the 'independent' terminals */
             TL:TextLabel** ;                                /* set of comments */

 TextLabel ==                       /* used for adding comments to the diagram */
    Txt:STRING ;

/* ---- Level1: MAJOR NODE TYPES ------------------------------ */
/* This level 1 identifies the major kinds of nodes that exist in a labview pro-
gram. Of  these, the structure node is a type of Node that can be one of a further
four types of control flow node. */

 Node  ==
       MT:(IUN:InstrumentUseageNode |        /* attached to virtual instrument */
            SN:StructureNode |
           AMN:ArrayManipulationNode |
            BN:BundleNode) .                          /* gather and split values */
            Enable:BOOLEAN .                         /* for execution semantics */
             Cmmnt:STRING;                                   /* to add a comment */
 InstrumentUseageNode==
       InstIdent:INTEGER .                          /* Instrument Identifier  */
        InstType:STRING ;


/* -- Level2: CONTROL STRUCTURE NODES ------------------------------ */
```

```
/* Structure nodes have tunnels - these are points at which signals cross bound-
aries into nodes. Hence if they are inputs viewed from outside the node they are
outputs viewed from inside the node, and vice-versa.*/

 StructureNode ==
         CT:( FLN:ForLoopNode   | WLN:WhileLoopNode |
              SLN:SelectionNode | SQN:SequenceNode ) .
                Tns:Tunnel**  ;
 TunnelTerminal ==
          TnTm:(InpT:InputTunnel | OutT:OutputTunnel) ;
 InputTunnel  ==              /* An input tunnel is an entry port for a gate*/
           AIF:AutomaticIdexingFlg .
          Sins:Signal** .                           /* signal inside fan out */
         Souts:Signal ;                                  /* signal outside */
 OutputTunnel ==                /* An output tunnel is an exit port for a gate */
          Sins:Signal .                              /* signal inside node */
          Souts:Signal** ;                   /* signal outside node fan out */

/* ---- Level 3: 4 Types of Control Flow ------------------------------ */
/* Two types of loop control flow */
/* ---- For Loop ------*/
/* The for loop enables repeated execution of the sub block diagram it contains.
The Terminals are not listed under the top level list of terminals since they
only exist as part of the for loop */

 ForLoopNode ==
               LLT:LoopLimitTerminal .                /* Loop Limit Terminal */
             ItCnt:IterationCountTerminal .               /* Iteration Count */
              InTn:TunnelTerminal** .                              /* Inputs */
             OutTn:TunnelTerminal** .                             /* Outputs */
              Enable:BOOLEAN .                   /* For execution semantics */
               SbD:BlockDiagram ;              /* Sub Diagram, body of loop */
 ForLoopLimitTerminal ==
             Val:INTEGER .                                 /* Loop Limit Value */
             Name:STRING ;

/* ---- While Loop ------ */
/* Contains Iteration Count, test expression evaluated after executing loop ,
zero or more shift registers, While loop also has Iteration Count terminal? */

 WhileLoopNode
      ==     ItCnt:IterationCountTerminal .
             LCond:LoopCondition .
             ShftR:ShiftRegsiter** .
             ItCnt:INTEGER ;
 WhileLoopConditionTerminal
       ==    Val:BOOLEAN .
             Name:STRING ;
/* auxilliary productions used in both loop productions */
 IterationCountTerminal                    /* also used in while loop */
       ==    Val:INTEGER .
             Name:STRING ;
 ShiftRegister ==
          Val:Value /* records previous value */
```

```
/* ---- Select ------*/
 SelectNode ==
              :CaseSelectorTerminal .
              :SelectNodeDiagram** ;
 SelectNodeDiagram ==
              :DiagramNumber .
              :BlockDiagram ;

/* ---- Sequence ------*/
/* The sequence local terminals are for getting values between diagrams in  se-
quence */

 SequenceNode ==
              SND:SequenceNodeDiagram** .
               BD:BlockDiagram .
              SLT:SequenceLocalTerminal ;

/* ---- TERMINALS ------------------------------ */
/* Terminals are the labview version of ports. Tunnels are terminals which have
an inside and an outside. The distinction between fixed and adaptive terminals is
omitted */

 Terminal ==
         TT:( FPCT:FrontPanelControlTerminal |
                BDC :BlockDiagramConstants) .
                Val :Value
            Cmmnt:STRING ;                 /* comment for describing terminal */
 FrontPanelControlTerminal ==
          FPC:FrontPanelControl .            /* Input from the User Interface */
          DT: DataType ;                             /* adapts to front panel */
 BlockDiagramMathFunctions
       ==   PT:(Add:'+'|Times:'*' |Div:'/' | Sub:'-' ) ;
 BlockDiagramConstants
       ==   PT:(:Pi | :e :TRUE :FALSE) ;

/* ---- SIGNALS ------------------------------ */
/* Signals are the labview version of wires */

 Signal ==
              DSr:Terminal .                         /* data source terminal */
              DSi:Terminal** .         /* data sink terminal arbitrary fanout */
              Dim:INTEGER .                /* indicates number of dimensions */
              Val:Value ;
 Value ==
               V:(:INTEGER |:REAL) .                          /* Actual value */
              DT:DataType ;                         /* Data type of the value */

 /* ---- ARRAY HANDLING ------------------------------ */
 /* Labview provides support for flow and manipulation of arrays of values */

 ArrayManipulationNode
       ==     :( :ArrayElementReplacement |
                 :ArrayIndexerNode |
                 :ArrayBuilding ) ;
 ArrayElementReplacement ==
              :ArrayInputTerminal .
              :ScalarValueTerminal .                    /* thing that replaces */
```

```
              :NumericIndex** .                /* can replace at many places */
              :ArrayOutputTerminal ;
 ArrayIndexerNode ==
              :ArrayInputTerminal .       /* slicing data out of multidim data */
              :Index** .
              :ArrayOutputTerminal;  /*output is of lesser rank by # of indices /
 ArrayBuilderNode
       ==     :ArrayOutputTerminal .                        /* of dimension n */
              :ArrayInputTerminal* ;                   /* of dimension n or n-l */

/* ---- BUNDLE HANDLING  ----------------------------- */
/* Labview provides support for flow and manipulation of arrays of values */
 BundleNode ==
              :BundlerNode |:UnBundlerNode ;
 BundlerNode ==
              :InputTerminals** .      /* invariant # of wires in = array size */
              :DataType .                                /* adapts to wire */
              :OutputTerminal;                                 /* only one */
 UnBundlerNode ==
              :OutputTerminals** .    /* invariant # of wires out = array size */
              :DataType .                                /* adapts to wire */
              :InputTerminal ;                                 /* only one */
 END_GLI_GRAMMAR

 BEGIN_GLI_QUERIES
 Levels1(b:BlockDiagram) == {b.*}
 Levels2(b:BlockDiagram) == {b.*.*}
 Levels3(b:BlockDiagram) == {b.*.*.*}
 END_GLI_QUERIES


/*
6.2  LabView Execution Semantics
================================

This section provides a list of the execution actions that specify the executions
semantics of Labview. The basic idea is to use a boolean component as a token
(e.g. Enable) to represent the passing of control, form node to node. e.g. from
calling to called nodes, selecting in a case statement etc. Such specifications
are similar to the operational semantics of a high level langauge (like C) into a
low level one (like assembler).A node is enabled only when values are available
at all of its inputs.
*/

 BEGIN_GLIDE_EXEC_ACTIONS

/* propagate a value along a signal. The Labview semantics is that a copy of the
value at the source terminal is made  available at the sink. Since a primitive
value is beiing 'assigned' it is a copy. Since signals have arbitrary fanout, the
movement of the dat values to all the data sink terminals */

 PropagateSourceToSignal(d:BlockDiagram) == {
   -FA-s:d.@(????).S { s.Val' = s.DSr.Val &&
                       s.Dsr.Val' = NULL
 }

/* This quantification is over all the terminals of all the signals */
```

```
 PropagateSignalToSink(d:BlockDiagram) == {
   -FA-sink:d.@(????).S.DSi { sink.Val' = sink..DSi.Val  }
 }
```

/ nodes are activated when data is avaliable at all inputs this action is for all for loops */*

```
 Activate(d:BlockDiagram) == {
   -FA-sn:d.@(????).SN { -FA-:v:Term.Val != NULL => sn.Activate = TRUE }
 }
```

/ Semantics of ForLoop. The three guarded actions captures the semantics of initializing, incrementing and terminating of a for loop. Perform update action on all ForLoops.*/*

```
 ExecForLoop(bd:BlockDiagram) == {
  -TE-:fl:bd@(.N.SN.CT.SbD).N.SN.FLN {
     (fl..CT.Enable = TRUE)
       => (fl.Enable = FALSE // loop start
                     => fl.Enable' = TRUE &&
                        fl.ItCnt' = 0 )
          (fl.Enable = TRUE && fl.ItCnt < fl.LLT.Val             /* during loop */
                     => fl.ItCnt' = fl.ItCnt + 1 &&
                       fl.SbD.Enable' = TRUE )
           (fl.Enable = TRUE && fl.LLT.Val = fl.ItCnt       /* loop termination */
                     => fl.Enable' = FALSE &&
                      fl..Sn.Enable' = FALSE)
       }
 }
```

/ Semantics of sequential control execution. Keep a counter which increments. The the node with in the list which is the same as the current vaue of the counter is enabled. Need some way of counting into the list in Glide...*/*

```
 ExecSequence(bd:BlockDiagram) == {
  -TE-:sq:bd@(.N.SN.CT.SbD).N.SN.SQN {
     (sq..CT.Enable = TRUE)
       => (sq.Enable = FALSE // sequence start
              => sq.Enable' = TRUE &&
                 fl.SqCnt.Val' = 0 )
          (sq.Enable = TRUE && sq.SqCnt.Val < fl.SqTotalNum.Val  /+ during loop */
         => -TE-n:sqn.Diags (sq.DiagNumber = sq.SqCnt.Val')
                   => sq.SqCnt.Val' = fl.ItCnt.Val + 1 &&
                      fl.Active' = TRUE )
          (sq.Enable = TRUE && fl.LLT.Val = fl.ItCnt.Val       /* loop termination
                     => fl.Enable' = FALSE &&
                            fl..Sn.Enable' = FALSE)
       }
 }
```

```
/* Semantics of while loop execution. expression. */

 ExecWhileLoop(bd:BlockDiagram) == {
  -TE-:wl:bd@(.N.SN.CT.SbD).N.SN.WLN {
     (wl..CT.Enable = TRUE )
       => (wl.Enable = FALSE // start loop
              => wl.Enable' = TRUE &&
                 fl.ItCnt.Val' = 0 )
          (wl.Enable = TRUE && wl.LCond = 1                      /* during loop
              => wl.ItCnt.Val' = wl.ItCnt.Val + 1 &&
                 fl.Active' = TRUE )
          (wl.Enable = TRUE && wl.LCond = 0                       /* end loop
                     => wl.Enable' = FALSE &&
                      wl..SN.Enable' = FALSE)
       }
 }

/* Semantics of SelectNode - Select nodes are analogous to switch statements.
Uses there-exists to find the selected of case type statement */

 ExecSelect(bd:BlockDiagram) == {
  -FA-:sln:bd@(.N.SN.CT.SbD).N.SN.SLN {
    (sln..CT.Enable = TRUE )
     => (-TE-:acase:sln.Cases {
           (acase.val = sln.Switch.DiagramNumber)
                     => acase.SubDiag.Enable' = TRUE)
       }
 }

 ExecBundle(bd:BlockDiagram) == {
    -FA-bun:bd.Nodes.Bundle()
       -FA-inputs:BundleInputs ( bun.Bundle = append(new)

 }
END_GLIDE_EXEC_ACTIONS

/* The graphical attributes add icons to the major labview node types.*/

BEGIN_GLI_GRAPHICALATTRIBUTES
 {BlockDiagram  { GRAPH {N S}}}
 {Node        { SHAPEINGRAPH Box ConnectedThrough {
                     {{.MT.CT.InTn.TnTm.Sins .MT.CT.OutTn.TnTm.Sins} HEAD}
                     {{.MT.CT.InTn.TnTm.Souts .MT.CT.OutTn.TnTm.Souts} TAIL}}
                      WRAP TRUE BITMAP node.xbm }}
 {PrimitiveNode      { BITMAP transnode.xbm }}
 {InstrumentUseageNode{ BITMAP placenode.xbm }}
 {ForLoopNode        { BITMAP forloop.xbm }}
 {WhileLoopNode      { BITMAP whileloop.xbm }}
 {SequenceNode       { BITMAP sequence.xbm }}
 {SelectionNode      { BITMAP selection.xbm }}
 {Signal             { SHAPEINGRAPH Line ConnectedThrough}}
END_GLI_GRAPHICALATTRIBUTES
```

Graph Toolkits

The graph editor toolkits that have been developed in recent years are noted here. In general, the display, manipulation, and editing of graphs has remained just outside being included in a standard GUI system and has be created as a separate layer on top. The following ones have either been described in the literature or are publicly available:

- *EDGE* - The EDGE system was first described in [Pau88] and forms the basis of a thesis by Paulisch [EDGE90]. EDGE is object-oriented (implemented in C++) so that the classes, “node” and “edge” can be specialized, adding attributes which define the application. Both edges and links can be labelled. The package also supports hierarchy and incorporates a number of automatic layout algorithms. EDGE has been used for displaying call graphs, makefile dependencies, simple logic circuits. It also contains some support for animation (routines for highlighting nodes and links).
- *XGRAB* - XGRAB is a package has had several implementations, the latest being on top of the InterViews GUI library. It has been used to display “program call graphs, module dependency, finite state automaton graphs and database designs”. XGRAB (originally GRAB) evolved from the original work by Carl Meyer. The current version leverages the higher level of sophistication of the Interviews GUI library (*e.g.*, zooming commands). [RDM+87]
- *TGE* - This package is also developed on Interviews. The paper by Karrer and Scacchi provides a description of the essential requirements for a general tool graph interface library and point out the difficulties of trying to use one of the existing packages to implement a real system (*e.g,*. the need to be able to store graphs). [KS90]
- *XSIM* - A simple but effective package. XSIM provides support for flagging syntactic errors and generating the textual representation of a graph suitable for input to other programs. XSIM has been used as a front end a Generalized Timed Petri Nets (GTPN) simulator. [Tho90]
- *DAG* - The DAG system is described by Gansner, North, and Vo in [GNV88]. It describes various layout strategies and performance statistics for them. This package and it successor was used in Glider.

- *GMB* - The GMB graph display system was created in the context and managing software in the Faust software programming environment for developing supercomputing applications. GMB was used for such things as the display of file compilation dependencies (makefiles). [J88] [JG89] [Jab90].
- *GUIDE II* - Guide is an internal product of Scientific and Engineering Software Inc. It is one of the few systems that provides a high level of functionality for both graph display and manipulation *and* other more standard interactors such as tables and menus, integrated in the same package.
- A number of earlier systems exist, notably the ones developed on interactive lisp workstations such as the ISI Grapher [Rob87] and the Interlisp Grapher [Int85].
- Some experiments have been performed by MacKinlay et al. at Xerox with the display of graphs in 3D. This has the advantage of being able to display quite large graphs (which is a problem in 2D because readable layouts take up large amounts of screen space), but it is not yet clear how useful they are.

Systems 1, 2, and 4 are publicly available. All these packages face the graph layout problem. *i.e.,* the need to provide a readable arrangement of nodes and links, preferably in the smallest amount of space. The problem of efficient of 2D layout algorithms for readable graph displays has received extensive attention (see surveys by Harel [DH89] and Tammasia [RTB88], and an analysis of the problem in [ET89]. Many of these algorithms have already been incorporated into some the graph GUI tools such as EDGE and XGRAB.

References

[ACR+89] Bowen Alpern, Alan Carle, Barry Rosen, Peter Sweney, and Kenneth Zadeck. Graph attribution as a specification paradigm. *ACM SIGPLAN Notices*, 24(2):121–129, February 1989.

[ACS90] B. Alpern, L. Carter, and T. Selker. Visualizing computer memory architecture. In *Proceedings of the First IEEE Workshop on Visualization*, pages 107–113, October 1990.

[AI89] H. Ammar and S. Rezaul Islam. Time scale decomposition of a class of generalized stochastic Petri net models. *IEEE Trans. on Software Engneering*, 15(6):809–820, June 1989.

[AltReal88] R. Smith. The Alternative Reality Kit. *Workshop on Visual Languages* 1988

[ARM91] B. Stroustrup. *The C++ programming language 2nd ed.* Addison-Wesley, 1991.

[AYC88] R. Akscyn, E. Yoder, D. MacCracken. The data model is the heart of interface design. *CHI* 1988

[BA88] J.C. Browne and Ashok Adiga. *Performance Evaluation of Supercomputers*, Chapter: Graph Structured Performance Models. Elsevier, 1988, pp. 239-281.

[BCL90] Duane Bailey, Janice Cuny, and Craig Loomis. Paragraph: Graph editor support for parallel programming environments. *International Journal of Parallel Programming*, 19(2):75–110, 1990. parallel programming environment graph grammar.

[BDD92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. In *ACM Computing Surveys*, Vol 24, No.3 Sept. 1992.

[BDG+91] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing '91*, pages 435–444, Nov 1991.

[BH92a] M. H. Brown and J. Hershberger. Animation of Geometric Algorithms: A Video Review. Technical Report 87a, DEC Systems Research Center, Palo

Alto, June 1992.

[BH92b] M. H. Brown and J. Hershberger. Color and Sound in Algorithm Animation. Technical Report 76a, DEC Systems Research Center, Palo Alto, June 1992.

[BL86] B.Liskov, J.Guttag, *Abstraction and specification in program development*, MIT Press, Cambridge, Mass.1986.

[Boo94] G. Booch. *Object-oriented analysis and design with applications.* 2nd ed. Redwood City, 1994.

[Bow89] Jonathan P. Bowen. Formal specification of window systems. Technical Monograph PRG-74, June 1989.

[Bro85] J. C. Browne. Formulation and programming of parallel computers: A unified approach. In *Proc. Intl. Conf. Par. Proc.*, pages 624–631, 1985.

[Bro88] Marc H. Brown. *Algorithm Animation*. ACM Doctoral Dissertation Award. MIT Press, Cambridge, Mass, 1988.

[Bro92] M. H. Brown. Zeus: A System for Algorithm Animation and Multi-view Editing. Technical Report 75, DEC Systems Research Center, Palo Alto, February 1992.

[BS84] M. H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, July 1984.

[BSS84] D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors. *Interactive Programming Environments*. McGraw-Hill, New York, 1984.

[BS84] M. H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, July 1984. (BALSA)

[Bur94] M. Burnett, R. Hossli, T. Pulliam, B. VanHoorst, X. Yang. Toward Visual Programming Languages for Steering Scientific Computations. *In IEEE Computational Science and Engineering*, Winter 1994.

[BY90] W Burton, HK Yang. Manipulating multilinked data structures in a pure functional language. *Software Practice and Experience* Vol 20 (11) 1167-1185 Nov 1990.

[BW88] R. Bird, P. Wadler. *Introduction to Functional Programming*. Prentice-Hall 1988.

[CComp88] C. Fischer, R. LeBlanc, *Crafting a compiler*. Benjamin/Cummings, Menlo Park, CA, c1988.

[CER78] Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors. *Graph-grammars and their application to computer science and biology: international workshop*. Springer-Verlag, Bad Honnef, October 1978.

LNCS73.

[CGP89] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming form textual conditioning. In *IEEE Workshop on Visual Language*, pages 150–156, Rome, Italy, October 1989.

[Che91] D. Cheng. A survey of parallel programming tools. Technical Report RND-91-005, NASA Ames, May 1991.

[Chi85] Uli H. Chi. Formal specification of user interfaces: A comparison and evaluation of four axiomatic approaches. *IEEE Transactions on Software Engineering*, 11(8):671–685, August 1985.

[CHVL91] C. Holt in Report on E-mail Panel: Is Visual Programming a New Programming Paradigm? *IEEE Workshop on Visual Languages 1991*.

[CL88] C. Chow and S. Lam. Prospec: An interactive programming environment for designing and verifying communication protocols. *IEEE Transactions on Software Engineering*, 14(3):327–338, March 1988.

[CLU93] B. Liskov. A history of CLU *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*. ACM SIGPLAN Notices. V28, Number 3, March 1993.

[Cou90] Bruno Courcelle. *Handbook of Theoretical Computer Science*, volume B - Formal Models and Semantics, chapter 5 - Graph Rewriting: An Algebraic and Logic Approach, pages 194–242. Elsevier, Amsterdam, 1990.

[CPN90] K Jensen, Colored Petri Nets: A High Level Language for System Design and Analysis. In *Advances in Petri Nets 1990 LNCS no.483*, Springer, Berlin NY 1990, pp. 342-416. "*..In our opinion, all users of CP-nets (and other kinds of Petri-nets) are forced to make simulations - because it is impossible to construct a CP-net without thinking about the effects of the individual transitions. Thus the proper question is not whether the modeller should make simulations or not, but whether he wants computer support for this activity. With this rephrasing the answer becomes trivial: Of course we want computer support..*"

[CS90] L. Chang and B. Smith. Classification and evaluation of parallel programming tools. Tech Rept CS90-22, Dept. of Comp. Sci. Univ. of New Mexico, 1990.

[daV93] M. Frohlich, M. Werner. daVinci. Ver 1.1 User Manual. Tech Report Univ. Bremen August 1993

[Dea92] Nate Dean. Viewing and analyizing graphs with netpad. In *Proceedings of the DIMACS Workshop*, March 1992.

[DF82] Davis, Keller. Data Flow Program Graphs. *In IEEE Computer Special Issue on DataFlow Langauges*. 1982

[DH89] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. Technical Report CS89-13, Weizmann Institute of Science, July 1989.

[Dil90] Antoni Diller. *Z - A Introduction to Formal Methods*. John Wiley and Sons, Chichester, 1990.

[Dix91] Alan John Dix. *Formal Methods for Interactive Systems*. 1991.

[DP83] Dipayan Gangopadhyay. *A formal system for network databases and its applications to integrity based issues.* Ph.D. Thesis, University of Texas, Dept. of Computer Sciences 1983.

[Dye89] R. Dye. Labview : A visual data-flow programming language and environment. Master's thesis, Dept. of Elec. and Comp. Eng. University of Texas at Austin, 1989.

[E. 90] E. Kant, F. Daube, W. MacGregor, J. Wald. Synthesis of mathematical modelling programs. In *Mathematica Conference Proceedings*, Redwood City, CA, January 1990.

[ea85] D. Notkin et al. The GANDALF Project. *The Journal of Systems and Software*, 1985.

[ea91] J. Werth et al. The interaction of the formal and practical in parallel programming environment development: Code. Tech. Rep. TR-91-09, Dept. Comp. Sci., Univ. Texas at Austin, 1991.

[EGF91] M. Blattner E. Glinert and C. Frerking. Visual tools and languages: Directions for the 90's. In *Workshop on Visual Languages*, pages 89–95. IEEE, 1991.

[Ehr87] Hartmut Ehrig, editor. *Graph-grammars and their application to computer science : 3rd international workshop*. Springer-Verlag, Warrenton, Virginia, USA, December 1987. LNCS291.

[Ehr90] Hartmut Ehrig, editor. *Graph-Grammars and their Application to Computer Science : 4th International Workshop*. Springer-Verlag, Bremen, Germany, March 1990. See discussion page 41.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W Schafer, and A Schurr. Building integrated software development environments part1: Tool specification. *ACM Transaction on Software Engineering and Methodology*, 1(2):125–167, April 1992.

[ENR82] Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors. *Graph-grammars and their application to computer science : 2nd international workshop*. Springer-Verlag, Haus Ohrbeck, October 1982. LNCS153.

[Env92] Interactive Development Environments. Software through pictures, 1992. 595 Market Street, San Fransisco CA94105.

[ERL90] H. El-Rewini and T. G. Lewis. Task grapher: A tool for scheduling parallel program tasks. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 1171–1178, Charleston So. Carlonia, April 1990.

[ET89] P. Eades and R. Tamassia. Algorithms for automatic graph drawing: an annotated bibliography. Tech Report CS-89-09, Dept. of Comp. Sci., Brown Univ., 1989.

[ET++89] A. Weinand, E. Gamma, R. Marty. Design and implementation of ET++ a seamless object-oriented application framework. *Structured Programming*, Vol1 No. 2 1989.

[F86] F. Feldbrugge and K. Jensen. Petri Net Tools Survey. Petri Nets: Central Models and Properties. In *Advances in Petri Nets*, pages 20–61, Berlin, 1986. Springer-Verlag. LNCS 254.

[FA90] D. Workman F. Arefi, C. Hughes. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3), March 1990.

[FLS93] Workshop on State in Programming Languages (SIPL), June 12 Copenhagen, Denmark..

[Fol93] J. Foley. A Second Generation User Interface Design Environment: The Model and the Runtime Architecture. *Proceedings of CHI* 93.

[GDr93] *Graph Drawing '93*. ALCOM International Workshop on Graph Drawing and Toplogical Graph Algorithms. September 1993. Paris.

[GH80] J. Guttag and J. Hornig. Formal specification as a design tool. In *Proc. 7th Symp. Principles of Programming Lang.*. ACM, 1980.

[GH78]. J. Guttag and J. Horning. *The Algebraic Specification of Abstract Data Types*. *Acta. Inform.*, vol. 10. 1978.

[Got89] H. Gottler, Graph grammars, a new paradigm for implementing visual languages. In *Eurographics'89*, pages 505–516, 1989.

[Got92] H. Gottler, Diagram editors = graphs + attributes + graph grammars. *Int. J Man-Machine Studies* (1992) 37,481-502.

[GNV88] E. Gansner, S. North, and K. Vo. Dag - a program that draws directed graphs. *Software Experience and Practice*, 18(11):1047–10621047–1062, November 1988.

[Grahm] M. Vanter, S. Graham, R, Ballance. Coherent user interfaces for language-based editing. *Int. J. of Man-Machine Studies* No 37 1992 pp. 431-466.

[GS92] E. P. Glinert and P. D. Stotts. Special issue on visual languages and concurrent computing. *Journal of Visual Langauges and Computing*, 3(2), June 1992.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Communications of ACM*, 31(5):514–529, May 1988.

[HC88] A. Hough and J. Cuny. Initial experiences with a pattern-oriented parallel debugger. In *Proceedings of the 1988 Workshop on Parallel and Distributed Debugging*, pages 195–205. ACM, 1988. also SIGPLAN Notices 24(1).

[HC90] A. Hough and J. Cuny. Perspective views: A technique for enhacing parallel program visualization. Coins Technical Report 90-02, University of Massachusetts at Amherst, January 1990.

[HE91] Michael Heath and Jennifer Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[HH&N92] J. Hummel, L. Hendren, A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. *International Conference on Parallel Processing* 1992.

[Him89] Michael Himsolt. Graphed: An interactive graph editor. In *STACS '89*, 1989.

[HO] Jack Hagemeister and Paul Oman. Powerful CASE on the Mac. product review, *IEEE Computer* July 1992.

[Hoare75] C.A.R. Hoare. Recursive data structures. *Intl. Journal of Computing and Information Sciences*. Vol. 4, No. 2 1975.

[Hun90] N. Hunt. Idf: A graphical data flow language for image processing and computer vision. In *IEEE Conf. on Systems, Man, and Cybernetics*. IEEE, Nov 1990.

[Int85] Xerox Artificial Intelligence Systems, Pasadena, CA. *Interlisp-D Reference Manual*, October 1985.

[IPSEN92] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, A. Schurr. Building Integrated Software Development Environments Part1: Tool Specification. In *ACM Transactions on Software Engineering and Methodology*, April, Vol. 1, No. 2, 125-167, 1992.

[ITCL94] M.. McLennan, Object-Oriented Programming with [incr Tcl]. Seminar at *Tcl/Tk Workshop* June 20-25, 1994, in New Orleans, LA.

[J88] David Jablonowski et al. GMB - In *ACM SIGGRAPH Symposium on User*

Interface Software, Banff, Canada, March 1988.

[Jab90] David Jablonowski. Gmb: Graph manager / browser. Technical Report CSRD 968, Center for Supercomputing Research and Development, Univ. of Illinois, February 1990.

[Jac85] R. J. K. Jacob. A state transition diagram language for visual programming. *Computer*, 18(8):51–59, August 1985.

[JG89] David Jablonowski and Vincent Guarana. Gmb - a tool for manipulating and animating graph structures. *Software Practice and Experience*, 19(3):283–301, March 1989.

[JM88] A. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 1988.

[K&S93] N. Klarlund, M. Schwartzbach. Graph Types. *POPL* 1993.

[Kan90] Elaine Kant. Automated program synthesis. Industry Leaders in Computer Science and Electrical Engineering Distinguished Lecture Series Video, May 1990. University Video Communications.

[KidSim94] D. Smith, A. Cypher, J. Spohrer; KIDSIM: Programming Agents without a Programming Langauge. *Commuincations of the ACM* July 1994, pages 55-66.

[KKS88] Hyoung-Joo Kim, Henry F. Korth, and Avi Silberschatz. Picasso: A graphical query langauge. *Software Practive and Experience*, 18(3):169–203, March 1988.

[Klint93] P. Klint A Meta-Environment for Generating Programming Environments. ACM Transaction on Software Engineering Methodology Vol 2, April 1993.

[KN90] D. Kimelman and T. Ngo. Program visualization for rp3: An overview. Technical Report RC 15917, IBM T.J. Watson Research Center, July 1990.

[KS84] P. Kruchten and E Schonberg. The ada/ed system: A large scale experiment in software protoyping. *Technique et science informatiques*, 3(3):179–185, 1984.

[KS90] A. Karrer and W. Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *Proceedings of ACM Third Annual Symposium on User Interface Software and Technology*, pages 84–91, 1990.

[Lam90] L. Lamport. A temporal logic of actions. Tech. Report 57, Digital Research Center, 1990.

[Lan87] D. Lange. A formal approach to hypertext using post-prototype formal specification. INTERCHI

[LD85] Ralph L. London and Robert A. Duisberg. Animating Programs Using Smalltalk. *IEEE Computer*, pages 61–71, August 1985.

[LER92] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*, chapter 12. Prentice-Hall, 1992.

[Lev92]. M.R. Levy Data types with sharing and circularity *Ph.D. Thesis* Dept. of Computer Science, University of Waterloo 1978.

[LMCF90] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. Analyzing parallel program executions using multiple views. Technical Report TR90-110, Rice University, January 1990.

[Loy91] Joseph Patrick Loyall. *Specification of Concurrent Systems Using Graph Grammars*. PhD thesis, Dept. of Comp. Sci., Univ. of Illinois, Urbana-Champaign, May 1991.

[MDB87] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Special issue on visualization in scientific computing. *Computer Graphics*, 21(6), November 1987.

[Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.

[MK92] J. Magee and J. Kramer. Mp: A programming environment for multicomputers. In *Proc. of Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992. Springer-Verlag. IFIP WG10.3.

[Min92] Interacting with structure-oriented editors. *Int. J Man-Machine Studies* (1992) 37,399-418

[Mol85] B. Moller On the algebraic specification of infinite objects - ordered and continuous models of algebraic types in *Acta Informatica* 22 p.537-578 1985.

[Mot93] V. Quercia, T.O'Reilly Volume 3M: X Window System User's Guide:Motif Edition 2nd Edition January 1993 ISBN: 1-56592-015-5.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. Tech. Rpt. CMU-CS-92-113, Carnegie-Mellon, School of Comp. Sci., February 1992. To appear in Proceedings SIGCHI'92.

[MSWin] C. Petzold, *Programming Windows 3.1, Microsoft Press Books - ,*ISBN: 1-55615-395-3, 1992.

[Nat87] National Instruments Corp., 12109 Technology Blvd. Austin, Texas. *LabVIEW: a demonstration*, 1987.

[NB92] P. Newton and J. C. Browne. The Code2.0 graphical parallel programming language. In *Proc. ACM Intl. Conf on Supercomputing*, July 1992.

[New94] Peter Newton. *A Graphical Targetable Parallel Programming Environment and its Efficient Implementation*. PhD thesis, Dept. of Comp. Sci., The University of Texas at Austin, 1994.

[NUH89] H. Kawata N. Uchihira and S. Honiden. A concurrent program synthesis using petri net and temporal logic in mendels-zone. Tech. Rep. 449, ICOT, Jan 1989.

[Nut91] G. Nutt. A simulation system architecture for graph models. In *Advances in Petri Nets*, Berlin, 1991. Springer-Verlag.

[OMT91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-oriented modeling and design*, Prentice Hall, 1991.

[OPP89] Joseph Oliger, Ramani Pichumani, and Dulce Poceleon. A visual object-oriented unification system. Technical report, Center for Large Scale Computing, Dept. Comp Sci., Stanford Univ., 1989.

[Oust94] J. Ousterhout. *Tcl and the Tk Toolkit*. Adison-Wesley, Reading MA, 1994

[Pan91] Bhalchandra Shankar Pandit. A syntax directed editor for code. Master's thesis, Dept. of Comp Sci., Univeristy of Texas at Austin, 1991.

[Pau88] Frances Newberry Paulisch. An interface description language for graph editors. In *Workshop on Visual Languages*. IEEE, 1988.

[Pau] L. Paulson. *ML for the Working Programmer*. Cambridge University Press.

[Pet81] J. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliff, 1981.

[PL88] Uwe Pleban and Peter Lee. An automatically generated, realistic compiler for an imperative language. In *SIGPLAN '88 Conference on Language Design and Implementation*, pages 222–227, Atlanta Georgia, June 1988. ACM.

[Pou94] A. Poulovassilis, M. Levine. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Informations Systems* Vol 12 No. 1 Jan 1994 pages35-68.

[PT90] Frances Newberry Paulisch and Walter F. Tichy. Edge: An extendible graph editor. *Software-Practice and Experience*, S1(20), June 1990.

[RC89] G. Roman and K. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 1989.

[RDM+87] L. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software Practice and Experience*, 17(1):61–76, 1987.

[Rob87] Gabriel Robins. The ISI grapher: A portable tool for displaying graphs pictorially. In *Symboliikka 87*, Helsinki, Finnland, August 1987. Also Chp 12. Multicomputer Vision, 1988 Academic Press.

[RTB88] Enrico Nardelli Roberto Tamassia and Carlo Batini. Automatic Graph Drawing and Readability of Diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, January-February 1988.

[SBN88] D. Socha, M. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. In *Proceedings of the 1988 Workshop on Parallel and Distributed DebuggingIEEE Workkshop On Visual Languages*, pages 206–215, New York,NY, 1988. ACM. also SIGPLAN Notices 24(1).

[SBY87] R. G. Smith, P. S. Barth, and R. L. Young. A Substrate for Object-Oriented Interface Design. In B. Shriver and P. Wegner, editors, *Research Directions In Object-Oriented Programming*, pages 253–315. MIT Press, Cambridge, MA, 1987.

[ShM88] S. Shlaer, S. Mellor. *Object-oriented systems analysis : modeling the world in data*, N.J.,Yourdon Press, Englewood Cliffs, 1988.

[Shu] Nan C. Shu, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuQBE] Nan C. Shu, Chapter 11, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuFol] J.D. Foley in Visual Programming, Nan C. Shu ed., page 17: *"When a person uses an interactive graphics system to do real work, he wants the system to virtually disappear from his conciousness so that only his work and its ramification have a claim on his energy"*

[ShuGRASE] Nan C. Shu Chapter 9, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuJLin] Nan C. Shu Chapter 9, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuPBH] Nan C.. Shu, Chapter 5, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuPICT] Nan C. Shu, Chapter 10, Visual Programming, Van Nostrand Rheinhold, NY, 1988.

[ShuVLdef] Definition of Visual Programming in Nan C. Shu, Visual Programming: : *"Visual Programming - the use of meaningful graphic representations in the process of programming"*

[Sil92] Silicon Graphics Inc. *Iris Explorer' User's Manual*, Jan 1992.

[Spi88] J. Spivey. *Understanding Z - A Specification Language and its Formal Semantics*. Cambridge Univ. Press, 1988.

[SS92] S. Sistare. Data visualization and programming in the prism programming environment. In *Proc. of Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992. Springer-Verlag. IFIP WG10.3.

[SSW+92] D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu, and C. Smith. The enterprise distributed programming model. In *Proc. of Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992. Springer-Verlag. IFIP WG10.3.

[Sta90] John T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1:213–236, 1990.

[Sta91] Stardent Computer Inc. *AVS Reference Manual*, 1991.

[Sto88] D.P. Stotts. The PFG Language: Visual Programming for Concurrent Computation Expressing High-Level Visual Concurrency Structures in the PFG Kernel Language. In *Int. Conf. on Par. Proc.*, pages 72–79, August 1988. Vol2: Software, Univ Maryland.

[Sto90] P. Stotts Graphical Operational Semantics for Visual Programming. in *Visual Languages and Visual Programming*, S-K Chang 1990

[STP93] Interactive Development Environment's "Software Through Pictures". 595 Market Street San Fransisco CA94105. System reviewed by P. D. Stotts in Tools Review: 'Software through Pictures' from IDE Inc. in *Journal of Visual Languages and Computing* (1993) 4, 201-209.

[Sur82] Bernard Surfin. Formal specification of a display oriented text editor. *Science of Computer Programming*, (1):157–202, 1982.

[Sze93] P. Szelky, P. Luo, R. Neches Beyond Interface Builders: Model Based Interface Tools. *Human Factors in Computing Systems INTERCHI* '93 pages 383-390. Amsterdam 24-29 April 1993

[Szw87] Gerd Szwillus. Cegs - a system for generating graphical editors. In H. Bullinger and B. Schakel, editors, *Human-Computer Interaction - Interact '87*. Elsevier, 1987.

[Tho90] Gregory S. Thomas. Xsim 2.0 user's guide. ftp cs.washington.edu, April 1990. Dept. of Comp Sci. Univ Washington.

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

[Tur92] Russel Turpin. *Programming Data Structures in Logic*. PhD thesis, Dept. of Comp. Sci., The University of Texas at Austin, 1992.

[VLWks] *Proceedings of the IEEE Workshop on Visual Languages*: 1990, *October Skokie Illinois; Proceedings of the IIEE Workshop on Visual Languages* 1991, October Kobe Japan; 1994 (10th) *IEEE/CS International Symposium on Visual Languages*, October 1994.

[vZMC92] Lynette van Zijl, Deon Mitton, and Simon Crosby. A tool for graphical network modelling and analysis. *IEEE Software*, pages 47–54, January 1992.

[XTANGO] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer* v 23 n 9 Sep 1990, pp 27-39.

J. T. Stasko. Simplifying algorithm animation with Tango. *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pp 1-6.

[YNTL88] S. S. Yau, R. A. Nicholl, J. J.-P. Tsai, and S.-S. Liu. An Integrated Life-Cycle Model for Software Maintenance. *IEEE Transactions on Software Engineering*, 14(14):1128–1144, August 1988.

Vita

Michiel Florian Eugene Kleyn was a born a citizen of the Netherlands in Tripoli, Libya on August 15, 1961, the son of Henri F. Kleyn and Johanna M-C. Kleyn-Hillen. After graduating from St. Paul's School in London in 1979, he enrolled at Imperial College, University of London, where he received the degree of Bachelor of Science in Electrical Engineering in 1983 and of Master in Computing Science in 1984. After four years in the Systems Science Department at the Schlumberger Research Laboratory in Connecticut on graphical user interfaces and object-oriented programming, he entered the Graduate School of the University of Texas in September 1989. While enrolled at the University of Texas, he also worked at Schlumberger Austin Systems Center on computer graphics and parallel programming.

Permanent address: 3A Bennett Park, Blackheath, London.

This dissertation was typed by the author.