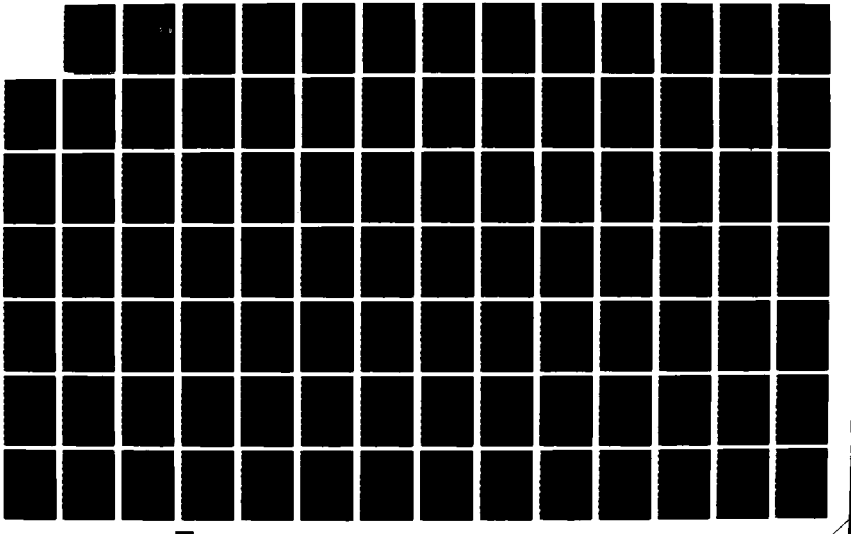
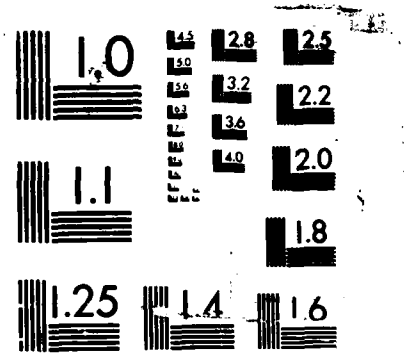
AD-A181 950 THE DESIGN AND SPECIFICATION OF PDL THE PROTOTYPE 1/2
DATAFLOW LANGUAGE(U) ARMY MILITARY PERSONNEL CENTER
ALEXANDRIA VA D J WOLFE MAY 87

UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART

(2)

The Design and Specification of PDL: The Prototype Dataflow Language

1LT Douglas J. Wolfe
HQDA, MILPERCEN (DAPC-OPA-E)
200 Stovall Street
Alexandria, VA 22332

Final Report
April 27, 1987

Approved for public release; distribution unlimited

A Thesis
Presented to
The Faculty and the Graduate School
of the
University of Southwestern Louisiana
In Partial Fulfillment
of the Requirements for the Degree
Master of Science

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A181950	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* The Design and Specification of PDL: The Prototype Dataflow Language	5. TYPE OF REPORT & PERIOD COVERED Final Report Approved: April 27, 1987 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 1LT Douglas J. Wolfe	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Douglas J. Wolfe, HQDA, MILPERCEN (DAPC-OPA-E) 200 Stovall Street, Alexandria, Virginia 22332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS HQDA, MILPERCEN, ATTN: DAPC-OPA-E, 200 Stovall Street, Alexandria, Virginia 22332	12. REPORT DATE 27 APR 87 13. NUMBER OF PAGES 138
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

Prepared in cooperation with Drs. Steve Landry, William Edwards, Margaret Montenyohl, and Joseph Urban

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Computer Science, Programming Languages, Formal Specification, Dataflow, Data-driven Computation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Several converging technologies have reached the point where they can be integrated and used to develop an advanced programming environment for writing parallel programs. These technologies include advanced graphics workstations, models of computation which can be used for parallel computation, and parallel architectures. Several manufacturers are providing commercially available parallel processing computers with potential for satisfying the performance requirements of many of today's computing problems. Unfortunately, these

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

computers usually do not have adequate, user-friendly programming environments, and the programming primitives supported by each machine are different. Thus, there is a need for a more general, user-friendly programming tool.

The dataflow model of computation has been receiving increasing attention to discover the model's potential use in parallel programming. Several textual languages have been developed for experimentation with parallel programming. Graphical base language representations have been proposed, but not developed because of the absence of graphics technology needed to implement graphical languages. The potential to support a programming language which permits the simultaneous existence of graphical and textual representations for programs has become practical with recent advances in interactive graphics technology and graphics workstations.

This thesis is concerned with the design and formal specification of a dataflow programming language which supports the simultaneous existence of graphical and textual representations for programs. The features of the language are synthesized from existing dataflow languages. The specification combines three formal specification techniques to formally specify the textual syntax, graphical representation, and semantics of the language. The specification serves as a rigorous, unambiguous description of the language.

Biographical Sketch

Douglas J. Wolfe was born in North Catasauqua, Pennsylvania on October 4, 1963. He attended the University of Pittsburgh in Pittsburgh, Pennsylvania and received the B.S. degree in Computer Science in April 1985. Mr. Wolfe was commissioned as a Second Lieutenant in the U.S. Army in April 1985.

Mr. Wolfe entered the masters program at the University of Southwestern Louisiana, Center for Advanced Computer Studies, during the Fall of 1985 with a U.S. Army Technical Enrichment Program scholarship. He is a member of the ACM and IEEE Computer Society.

Mr. Wolfe is currently a First Lieutenant in the U.S. Army.

The Design and Specification of PDL: The Prototype Dataflow Language

A Thesis

Presented to

The Faculty and the Graduate School

of the

University of Southwestern Louisiana

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Douglas J. Wolfe

May, 1987

The Design and Specification of PDL: The Prototype Dataflow Language

Douglas J. Wolfe

APPROVED:

Steve P. Landry, Chairman
Assistant Professor of
Computer Science

Margeret Montenyohl
Assistant Professor of
Computer Science

William R. Edwards
Associate Professor of
Computer Science

Joseph E. Urban
Associate Professor of
Computer Science

Joan Cain
Dean, Graduate School

Acknowledgements

My warmest thanks to Dr. Steve Landry for providing the motivating ideas for this thesis, and his enduring patience and continuing guidance throughout the work. His encouragement, professionalism, and friendship have made this work possible.

My thanks to Dr. Margaret Montenyohl for her constructive suggestions, continuous patience with all my questions, and guidance during the formulation of the formal specification.

My thanks to Dr. William R. Edwards and Dr. Joseph E. Urban for their constructive suggestions concerning this thesis and for serving on my committee.

My thanks to the U.S. Army for the opportunity to pursue this work.

My thanks and love to my mother and father, Mr. and Mrs. Donald R. Wolfe, for their constant love and support throughout the years.

My thanks to Mr. and Mrs. Curtis Hebert, for making me feel like part of their family and for all the fine home cooked meals. My thanks to Mrs. Marie Broussard for always being able to make me laugh.

My greatest love and appreciation to my fiancee, Miss Valarie Hebert, for all the love and support she has given me during this work. This thesis is dedicated to her.

My thanks to the many graduate students who have contributed through conversation, constructive criticism, and suggestions.

To Valarie...

who taught me the true meaning of love.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1. Introduction

Many of today's computing applications have performance requirements that cannot be fully satisfied by von Neumann architectures which are based on the control flow model of computation. Two examples of these applications are the weather problem [DENN84] and logic programming [BIC84]. Algorithms used in these applications have a high degree of parallelism which cannot be efficiently executed on the von Neumann computer. Thus, alternative models, such as the data-driven model of computation, have been of increasing interest to researchers. The dataflow model of computation addresses the performance requirements of computing applications, such as the weather problem, by exploiting the parallelism within algorithms used for solving problems within the application area. Languages and architectures which make use of the dataflow model of computation are thus desirable for use on these applications.

1.2. Motivations

Several converging technologies have reached the point where they can be integrated and used to develop an advanced programming environment for writing parallel programs. These technologies include advanced graphics workstations, models of computation which can be used for parallel computation, and parallel architectures. A programming tool to experiment with a dataflow language which supports the simultaneous existence of textual and graphical representations for programs could be beneficial to programmers writing parallel programs. In this thesis, a dataflow language which supports the simultaneous existence of textual and graphical representations for programs is proposed.

Several manufacturers are providing commercially available parallel processing computers with potential for satisfying the performance requirements of many of today's computing problems. Unfortunately, these computers usually do not have adequate, user-friendly programming environments. In addition, as Ahuja, Carriero, and Gelernter [AHUJ86] state, "as parallel machines emerge commercially, there has been little effort spent on making high-level, machine-independent tools available on them. Young debutante machines are sometimes gotten-up in their own full-blown parallel languages; more often they come dressed in only a handful of idiosyncratic system calls that support the local variant of message-passing or memory-sharing". The programming environments and the primitives they support are different for each parallel machine. Programs which run on one machine may need to be totally recoded to be executed on another machine. Thus, there is a need for a more *general*, user-friendly programming tool. A programming tool should also be portable so that it could be used by more than one parallel computer.

Several alternative models of computation, including the control flow and dataflow models, have been explored to discover each model's potential use in parallel programming. A brief discussion of the dataflow model and its advantages will be given later. Several high-level, textual languages and graphical base language notations have been developed for experimentation with parallel programming. Because many dataflow languages use a graphical base language, a mapping between the textual languages and graphical base languages is possible. [RAED85] has given several reasons for the use of graphical representations of programs. These reasons include the potential for faster transfer rate of knowledge using graphics and the random access of information from graphical representations. A dataflow language which supports the simultaneous existence of graphical and textual representations for programs should be beneficial to programmers because, at each step of program development, a programmer could choose to write using the type of representation that is most convenient for writing that part of the program.

Widespread use of graphical languages has been constrained by the absence of graphics technology needed to implement graphical languages. The potential to use graphical representations for programs has become practical with recent advances in interactive graphics technology. Several graphics workstations, which support useful primitives that can be utilized within a graphical programming language, have been developed and made commercially available.

In this thesis we propose that a dataflow language which utilizes both graphical and textual representations for programs be made available for use in parallel programming. The features of a prototype language are synthesized from existing textual languages. The graphical representations of the language are incorporated from existing graphical base languages, as well as other graphical representations (e.g., Nassi-Shneiderman diagrams [NASS73]).

1.3. Overview

Most existing programming languages and architectures are based on the traditional (fetch-execute-store) control flow model of computation. In the control flow model, operations are executed in an explicit order determined by a control mechanism used by a programmer. Programs written in control flow languages are represented by a sequence of operations[1] consisting of an operator and its necessary operands. The programmer is totally responsible for explicitly specifying the order of the operations in a program. Data is passed between operations by the use of variables which denote memory locations in a shared memory. When an assignment is made in a program, the data in the memory is changed as is the global state of execution. The control flow model is implicitly sequential in nature. Parallelism can be introduced only after studying the operations in a program and discovering which can be performed in parallel. Explicit control structures (i.e., FORK - JOIN) must be used to specify which operations are to be performed in parallel. Thus, the control flow model of computation can introduce parallelism in programs, but the control information and program structures needed can become excessive. Backus [BACK78]

[1] In this thesis, an operation refers to a program instruction which consists of one operator and one or more operands. The operations include any arithmetic, boolean, and data-dependent operations.

describes many problems that plague control flow languages. These include the language's "gross size, complexity, inflexibility, and lack of useful mathematical properties" [BACK78].

The dataflow model of computation relieves the programmer's burdensome task of specifying parallelism in programs by utilizing an alternative representation for computation. The dataflow model does not utilize a shared memory or a locus of control mechanism, and is not history sensitive. In the dataflow model, operations are executed in an order determined by the data dependencies of a program, not by an explicit control mechanism. A programmer does not explicitly specify the ordering of operations in a program, only the data dependencies of the operations. Dataflow programs are conveniently described in terms of a directed graph. The nodes of the graph represent operations or actors that produce result tokens by performing transformations or tests on input tokens. The arcs, which connect the nodes of the graph, are used to carry data values in the form of tokens. These tokens flow along the arcs in a specified direction from one node to successor nodes. Thus, the arcs define the data dependencies between the operations. The basic dataflow model states that a node is enabled when all input arcs hold data values and the output arcs are empty.[2] An enabled node is executed or fired by removing a token from each input arc, performing the specified transformation on the input tokens, and producing one or more result tokens. An example of a dataflow program, which was taken from [LAND81] is shown in Figure 1.1. The program computes the factorial of N. The model also does not maintain a global execution state. Instead a local state is maintained which consists of the executing operation and its successors.

The dataflow model is implicitly concurrent in nature. Instructions which require a data value must wait for the instruction that produces the value to fire. The sequencing constraints and the execution order of the operations are determined by the data dependencies.

---

[2] Different forms of this basic model have been proposed and are discussed in [LAND81].

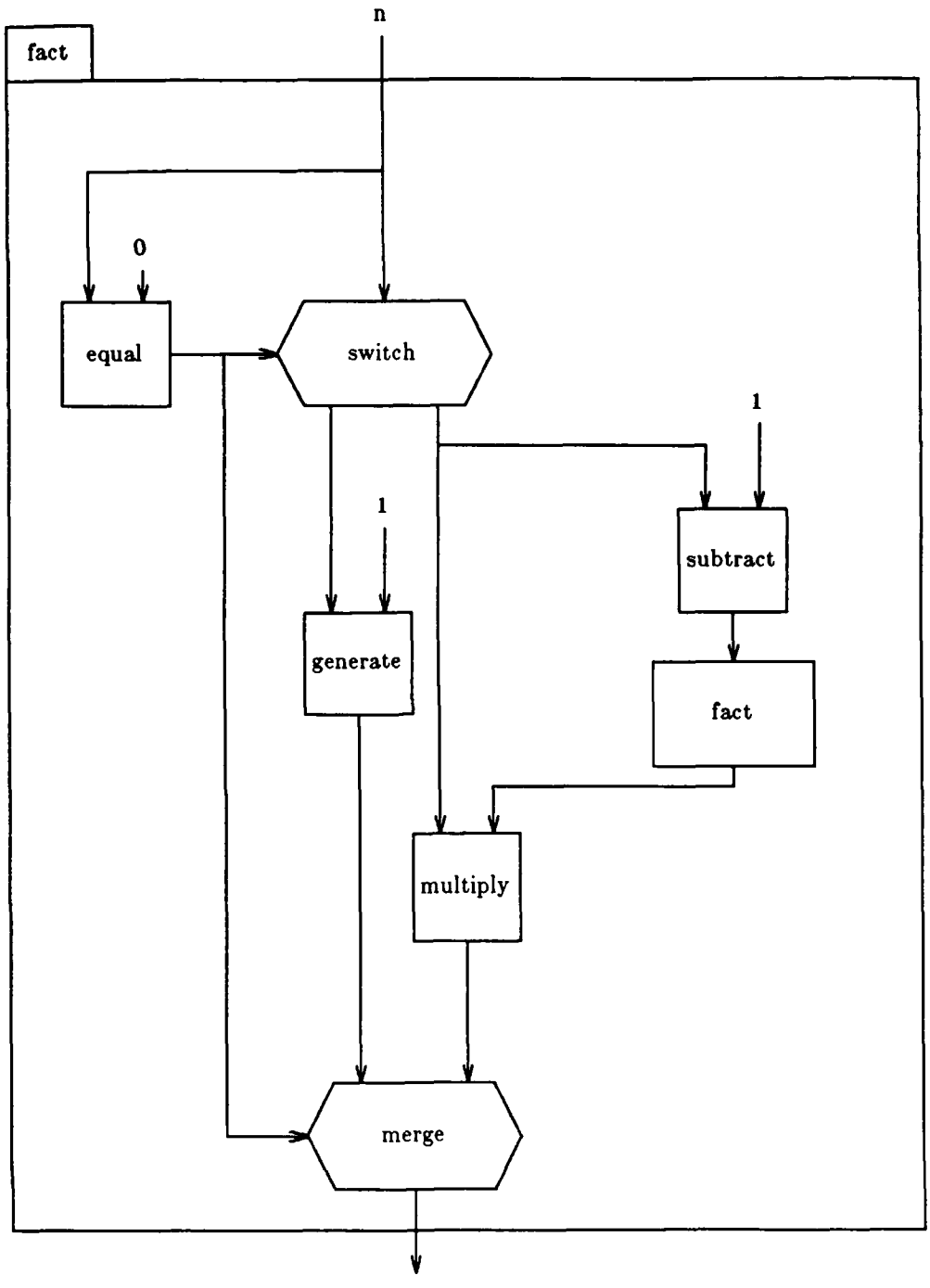


Figure 1.1 - example of a dataflow program graph

1.4. Review of Related Research

The fundamental concepts upon which the dataflow model of computation are based upon are described in [KARP69, DENN74, ARVI78, SHRI80, TREL82, and SHAR85]. Several research efforts [KARP69, RODR69, ADAM68, DENN74, KOSI73, ARVI78, and DAVI78] have resulted in alternative models of computation which share the same basic principles. Landry [LAND81] presents an overview of these and other models. Several novel architectures [GURD85, DENN80] and applicative languages [ACKE79, HANK81, and ARVI78] have been developed in association with the different models. An overview of the architectures appears in [TREL82 and SRIN86]. The language proposed in this thesis is based on Val, Id, and Lucid which are summarized in Section 2.2.

Formal specification of dataflow languages has received limitied attention by researchers. McGraw [MCGR82] discusses two formal specifications of Val. The first specification, presented by Gehani and Wetherall, who gave a complete denotational specification of Val. The second specification, presented by Ackerman, who gave an axiomatic specification of a "Val-like toy language". Brock [BROC78] gives an operational specification of a subset of Val. Brock's specification utilizes a translation algorithm to map program constructs into a corresponding graph and a semantic function to map the graph into formal semantics.

Raeder [RAED85] presents an excellent overview of the research in visual programming and gives several reasons for the emergence of visual programming[3]. Visual programming is concerned with the creation, manipulation, and/or execution of programs in a graphical form. Graphical programming uses an arrangement (picture) of graphical icons to represent a program. The picture is then translated into machine instructions using a strict mapping mechanism.

Brooks [BROO87] presents several reasons why "nothing even convincing, much less exciting, has emerged" from visual programming efforts. These reasons include that graphical

[3] One of three major areas of visual programming as defined by [GRAF85] is the development of graphics-based, very high-level programming languages.

representations used to describe conventional software, such as the flowchart, are a very poor abstraction of the software; and that software is very difficult to graphically visualize. Brooks was mainly concerned with *conventional* software, not dataflow programs which can be naturally represented using either a graphical or textual representation. Textual dataflow programs are translated into graphical representations. Thus, graphical representations are not a poor, but a natural abstraction of their textual equivalents. Graphical representations of dataflow programs are also not hard to visualize; in fact, the graphical representations show the parallelism within dataflow programs.

Few high-level graphical languages have been developed. Pagan [PAGA87] presents a programming environment for an FP-like language which has a graphical syntax. The language, which is a modified subset of Backus' FP, uses a graphical syntax similar to Nassi-Shneiderman diagrams. A program is depicted by nested boxes. Matwin and Pietrzykowski [MATW85] present a functional language, PROGRAPH, with semantics which are based on the dataflow model. Programs are expressed in the form of pictographs. Graphical representations are presented for conditionals, loops, user-defined functions, parallel computation operators, and an apply operator.

1.5. Statement of the Problem

This thesis contains a description of a dataflow language, PDL[4], that supports the simultaneous existence of graphical and textual representations for programs, and the formal specification of the syntax, graphical representation, and semantics of the language. Several existing textual dataflow languages have been studied and features of these languages were selected to synthesize the nucleus for the new language. A description technique, which utilizes three specification techniques, was developed to specify the graphical representation, textual syntax, and semantics of the language. A new graphical data type and a complete axiomatic specification are presented.

---

[4] PDL denotes the Prototype Dataflow Language.

1.6. Synopsis of Thesis

In Chapter 2, an overview of the textual dataflow programming languages in use today is presented. The features and programming constructs of each language are explained. Several graphical base languages associated with existing dataflow models are also presented.

In Chapter 3, an overview of the specification techniques used to specify PDL is provided. The techniques include the axiomatic and algebraic specification techniques and BNF Grammar.

In Chapter 4, an informal discussion of the prototype dataflow programming language, PDL, developed in the thesis work is presented. Specific language features, which were synthesized from the languages described in Chapter 2, are discussed in terms of both their textual and graphical representations.

In Chapter 5, the formal specification of the prototype language is given. A specification for the textual syntax, graphical representation, and semantics of each language construct is given using the techniques introduced in Chapter 3.

In Chapter 6, conclusions and suggestions for future research are given.

CHAPTER 2

Dataflow Languages

2.1. Introduction

A dataflow program is represented by a directed graph composed of nodes connected by arcs. The nodes of the dataflow graph represent individual operations (instructions) which receive, perform transformations upon, and transmit data tokens. The tokens flow along the arcs from the producer of the token to the consumer. Hence, the arcs define the data dependencies in the dataflow program. Since a dataflow system performs its operations asynchronously, with each node firing when all of its input tokens are available, several operations can be performed concurrently. Hence, dataflow languages can represent parallelism in a natural manner.

Most dataflow languages developed to date [ACKE79, ARVI78, and HANK81] use a textual representation for programs. Several researchers have proposed alternative dataflow models which have as their basis a graphical notation for programs. In Section 2.2, several relevant properties of dataflow languages are discussed. In Section 2.3, several graphical notations for programs proposed by Dennis [DENN74], Rumbaugh [RUMB77], and Kosinski [KOSI73] are discussed. In Section 2.4, three predominant languages, Val, Id, and Lucid, currently being used for dataflow programming research, are described. Other textual languages, some of which were patterned after Val, Id, and Lucid are also discussed.

2.2. Language Issues

Several relevant properties of dataflow languages are discussed in [ACKE82, SHAR85, and HAN85]. Most dataflow languages exhibit some, but not necessarily all of these properties. Six relevant properties of dataflow programming languages are identified by Ackerman [ACKE82] and

Sharp [SHAR85]. These include freedom from side effects, locality of effect, equivalence of instruction scheduling constraints with data dependencies, a "single-assignment" convention, an unusual notation for iteration, and a lack of history sensitivity in procedures.

Side effects arise in programs through the use of global variables and by reference parameter transmission. Global variables and by reference parameter transmission are not present in a dataflow programs. Further, dataflow languages utilize a value-oriented programming philosophy rather than a variable-oriented philosophy. All identifiers of a program, including data structures, denote values rather than memory locations. [GAUD86] discusses several solutions for the treatment of structured values.

Locality of effect implies that instructions do not have far reaching data dependencies. The effect that the instructions have is restricted to the block of code in which the instructions appear. Identifiers used in a program are active only within their defining block and have no effect beyond that block.

In a sequential language, instruction scheduling is determined by the order in which instructions are listed. However, in a dataflow programming language, the order of execution is determined by the data dependencies of the program. Therefore, the instruction scheduling constraints are equivalent to the data dependencies of the instructions. Instruction scheduling constraints and data dependencies must be equivalent so that the instruction firing rule holds. Freedom from side effects and locality of effect assure that the scheduling constraints and data dependencies of a program are equivalent.

The *single assignment* convention states that "a variable may appear on the left side of an assignment statement once within the area of the program in which it is active" [ACKE82]. The assignment binds the identifier on the left side to the value on the right. An example of the single assignment convention and the violation of this convention are show in Figure 2.1. Each identifier in the block of code, except J, is assigned once and used only after it has been assigned. Identifier J violates the single assignment convention because J appears on the right side of the assignment

```
S:=X + Y;
D:=3*S;
E:=S/2 + F(S);
J:=J+1;
```

Figure 2.1 - considerations for the single assignment convention

statement before it is bound to a value (J may not appear on both sides on an assignment statement).

Dataflow languages use a different notation for iteration than that used in conventional control flow languages. The need for a new notation is caused by the fact that side effects and program state information are absent from dataflow languages. Loops have four distinct parts: initialization, a test for loop completion, the redefinition of loop variables, and the production of result values. Special operators are used to redefine the values of the loop variables between each iteration of a loop. Redefinition of the loop variables occurs only at the loop boundaries. In Val, redefinitions are allowed to occur only after the word *iter*. An example of Val's notation for iteration is shown in Figure 2.2. The values of K and J on the left and right sides of the assignment refer to different identifiers. The identifiers on the left side refer to the values of the identifiers on the next iteration of a loop. The identifiers on the right side refer to the values on the current iteration of the loop. Therefore, the single assignment convention is still enforced during each

```
for J,K:= N,1; do
 if J = 0 then K
  else iter J:=J-1;
       iter K:=K*J;
  end
end
```

Figure 2.2 - example of Val's notation for iteration

cycle of the loop. If the values of loop variables depend upon the values of those variables in previous cycles, a sequential iterative loop must be used. If the values are not dependent on each other, some form of a parallel loop, such as Val's FORALL loop [ACKE79], can be used.

The lack of history sensitivity means that the value of the output of each procedure is dependent only on its current input values[5]. The procedure cannot remember past input values. The lack of history sensitivity means that the languages are functional in nature. Each node of a dataflow program represents a true mathematical function in that the output produced is only dependent upon the input it receives. In Lucid, arbitrarily long streams and special operations are used to achieve history sensitivity. For example, Lucid's *first* operation remembers the first element in an input stream and produces a stream of tokens, each of which has the value of the first input token.

The dataflow languages Val and Id exhibit all of the properties discussed. Lucid, which was not developed for dataflow programming, can be used for dataflow programming because the language exhibits many of the above cited properties.

2.3. Graphical Languages

A dataflow program maybe represented by a directed program graph. The nodes of the graph represent either arithmetic, logical, or run-time, decision-making operations. The arcs represent the flow of data in the form of tokens from the producer to the consumer of the token. Individual operations may be combined to form complex functions and finally program graphs. Three graphical, base languages [DENN74, RUMB77, and KOSI73] have been studied and are now briefly discussed.

2.3.1. Dennis' Language

Dennis [DENN74] presents a graphical base language into which Val programs can be translated. The language has two types of nodes, links and actors. Links are used to produce

[5] Streams and operations to manipulate streams may be introduced to achieve history sensitivity.

multiple copies of a data token. Two types of links are used, one for data values and one for control values. Eight actors are used, an operation actor (for all functions), a decider actor which produces control values, a true gate, a false gate, a merge actor, and three boolean actors (and, or, and not) which act upon control values. The merge, true-gate, and false-gate nodes are used for run-time data-dependent decisions to affect the flow of data tokens. These nodes are used to implement the graphical base language equivalents of high level program constructs. A node can execute only when all input arcs have tokens present and the output arcs are empty. Dennis' language allows for cyclic graphs, so a token-tagging scheme is introduced to distinguish tokens of different cycles of a procedure. An explicit application node, the ***apply node***, is used to perform a run-time binding of an argument list to a named procedure. Rumbaugh's language [RUMB77] is similar to Dennis' except that only two data-dependent decision nodes are used, the merge and switch.

2.3.2. Kosinski's Language

Kosinski [KOSI73, KOS73b] presents a dataflow programming language designed for use in operating systems programming. Programs in the language are composed of function definitions which are based on a set of programming primitives. Programs are determinate in nature unless indeterminacy is explicitly introduced. The language uses two types of nodes, computational and administrative. The computational nodes include constants, predicates, and the typical arithmetic, boolean, and string operations. Administrative nodes include forks, switches, function applicators, and several loop nodes. The inbound and outbound switch nodes are used to implement the base language equivalent of the *if* and *for* constructs of high-level dataflow languages. Loop nodes are used to implement the different types of loops present in high level languages. The loop node also introduces a memory structure in programs. Forks are used to replicate data tokens of any type. Function applicators are similar to Dennis' *apply* node. Kosinski allows any sub-graph to be named and used as a function (the sub-graph is replaced by a single node in the graph). Operations can execute with partial inputs, and can take more than one input token from

the same arc. The arcs carry not only data values, but PRESENCE and DONE signals. These signals are used for operation synchronization. Kosinski's inbound and outbound switches are incorporated into the design of PDL to implement the base language equivalent of PDL's program constructs.

The success of graphical languages has been impeded by the lack of adequate, user-friendly graphical programming tools. Therefore, dataflow researchers have given limited attention to graphical languages and have concentrated on developing several high level textual programming languages to support research efforts.

2.4. Textual Languages

In this section, some high level textual programming languages are presented. Val [ACKE79] and Id [ARVI78], are applicative languages that have been developed by Jack Dennis' group at MIT and the Arvind group at the University of California at Irvine, respectively. Since Val and Id are similar, they will be discussed together. The biggest difference in the languages is the type of dataflow model the languages use. Lucid [WADG85], was not developed for, but is currently being used for, dataflow programming. Several example programs and a discussion of each of the language's features are given in the following sub-sections. These three languages were used as a foundation for developing PDL which is introduced in Chapter 4.

2.4.1. Val and Id

Val [ACKE79,MCGR82] and Id [ARVI78,ARVI80] were designed for highly concurrent numerical applications. Since the languages are applicative, they exhibit many of the properties discussed in Section 2.2. The languages are free from side effects, have locality of effect, use a single assignment convention, and utilize completely functional, block-structured features.

Since the languages are functional in nature, all functions or operations compute specific output values depending upon the input given to the module or operation. The functionality of the languages guarantees that side effects do not occur. Although the languages are similar in

their features, they are based on entirely different models. Val is based on a static model and its features exhibit static behavior. In a static model, only one occurrence of an node (instruction) is enabled for firing at one time. Concurrent invocations of an instruction are not permitted. Instructions are loaded into memory before computation begins. The static model also allows only one token can reside on an arc at one time. Id is based on a dynamic model and most of its features exhibit dynamic behavior. In a dynamic model, several instances of a node can be executing concurrently and more than one token may reside on an arc at one time. The instances of a node can be dynamically generated at run-time. Id tags data elements to isolate data elements for each instance of a node and assigns an activity name with each instance of a node. The instance and its set of input data tokens must have the same names for the instance to fire. Id uses several special operators and the Unfolding Interpreter [ARVI82] to manipulate the tags and activity names.

Val was designed to meet two design goals: implicit concurrency and ease of program construction. Most concurrency in Val is implicit, but the language also supports one explicit form of concurrency, the FORALL loop. An Id programmer relies on the Unfolding Interpreter to discover the concurrency in programs and generate several independent activities which can execute concurrently.

A VAL program is composed of a number of function modules. An example of a Val program module which performs *quicksort* is shown in Figure 2.3. The code has not been syntactically verified. The example is presented only to show some of Val's program constructs. Each function module consists of a header, optional type definitions, and a *result* expression. Function modules are called by using the name of the module with a list of actual parameters. Function definitions may not be passed as parameters to a function module.

A program in Id is a list of expressions. An example of an Id program which performs *quicksort* is shown in Figure 2.4. This example is taken from [CARL85]. Type definitions are not utilized by Id; rather the identifier type is inferred from the context in which the identifier is

```
*****************************************************************************
Procedure quicksort receives an array of elements and the size of the array, and produces a sorted
array. The procedure uses two other procedures; one (qsort1) to produce two other sorted arrays
(Above and Below) and one (qsort2) to merge the elements of the two sorted arrays and a pivot
element (A[n/2]) into one sorted array.
*****************************************************************************
 1 procedure quicksort(a:array[int],n:integer; returns array[integer])
 2   below:array[integer],j:integer,above:array[integer]:=qsort1(a,n/2,n);
 3   qsort2(below,j,above,a[n/2],n)
 4 end
*****************************************************************************
Procedure qsort1 is used to divide an array (a) into two arrays (above and below) and either re-
turn these arrays (if the size of the array is one), or return the array produced by recursively cal-
ling quicksort on these arrays (line 8). The two arrays are formed by testing each element of the
input array to see if the element is less than or equal to the pivot element a[m] (line 10). If the
element is less than or equal to the pivot element, the element is appended to array below and the
index of this array (j) is incremented (line 11). Otherwise, the element is appended to array
above and its index (k) is incremented (line 13). This process occurs for each element in the input
array (a) (lines 6-16). In line 6 of the program, the arrays (above and below) are initialized (both
are empty arrays) and the indices of the arrays are set to zero.
*****************************************************************************
 5 procedure qsort1(a:array[integer],m,n:integer; returns array[integer}, integer, array[integer])
 6   for i:integer:=1; below,above:array[integer]:empty[integer]; j,k:integer:=0,0   do
 7     if i > n then
 8       if j > 1 then quicksort(below,j) else below, j, if j > 1 then quicksort(above,k) else above;
 9      elseif i <> m then
10           if a[i] <= a[m] then
11               iter below:=below[j+1:a[i]];   j:=j+1
12            else
13               iter above:=above[k+1:a[i]];   k:=k+1
14           endif
15      endif
16    endfor
17  end
*****************************************************************************
Procedure qsort2 merges two arrays (below and above) and the pivot element (mid) by appending
the pivot element and each element of array above onto array below to form the sorted array.
*****************************************************************************
19  procedure qsort2(below:array[integer], j:integer, above:array[integer],mid:integer,n:integer;
20                                        returns array[integer])
21    for i:integer:=j+2;  sorted:array[integer]:=below[j+1:mid]  do
22     if i > n then sorted
23      else
24       iter sorted:= sorted[i:above[i-j-1]]
25     endif
26    endfor
27   end
```

Figure 2.3 - quicksort written in Val

```
**********************************************************************************************
```
Procedure quicksort receives an array of elements and the size of the array, and produces a sorted array. The procedure uses two other procedures; one (qsort1) to produce two other sorted arrays (Above and Below) and one (qsort2) to merge the elements of the two sorted arrays and a pivot element (A[m]) into one sorted array.
```
**********************************************************************************************
```
```
 1  procedure quicksort(A,n)
 2   (m ← n/2;
 3    below, j, Above ← qsort1(A, m, n);
 4    return qsort2(Below, j, Above, A[m], n);
 5   )
```
```
**********************************************************************************************
```
Procedure qsort1 is used to divide an array (A) into two arrays (Above and Below) and either return these arrays (if the size of the array is one), or returning the array produced by recursively calling quicksort on these arrays (lines 16-18). The two arrays are formed by testing each element of the input array to see if the element is less than or equal to the pivot element A[m] (line 11). If the element is less than or equal to the pivot element, the element is appended to array Below and the index of this array (j) is incremented (line 12). Otherwise, the element is appended to array Above and its index (k) is incremented (line 13). This process occurs for each element in the input array (A) (lines 9-15). In lines 7 and 8 of the program, the arrays (Above and Below) are initialized (both are empty arrays) and the indexs of the arrays are set to zero.
```
**********************************************************************************************
```
```
 6  procedure qsort1(A, m, n)
 7   (initial Below ← Λ; j←0;
 8    initial Above ← Λ; k←0
 9    for i from 1 to n do
10      ( if i ≠ m then
11              (if A[i] ≤ A[m] then
12                   new Below←append(Below, j+1, A[i]); j←j+1;
13               else new Above←append(Above, k+1, A[i]); k←k+1;
14              )
15      )
16       return (if j > 1 then quicksort(Below,j) else Below),
17              j,
18              (if k > 1 then quicksort(Above,k) else Above)
19  )
```
```
**********************************************************************************************
```
Procedure qsort2 merges two arrays (Below and Above) and the pivot element mid by appending the pivot element and each element of array Above onto array Below to form the sorted array (sorted).
```
**********************************************************************************************
```
```
20  procedure qsort2(Below, j, Above, mid, n)
21   ( initial sorted←append(Below, j+1, mid);
22    for i from j+2 to n do
23      new sorted←append(sorted, i, Above[i-j-1]);
24    return sorted
25   )
```

Figure 2.4 - quicksort written in Id

used. Procedures may be passed as parameters to other procedures. Val's function modules and Id's procedures behave like true mathematical functions: they compute a specific output value for a given input value. Thus, the languages are not history sensitive, although history sensitivity is introduced in Id through the use of streams and feedback loops.

Val and Id utilize expressions and values as the basic units of computation. All language constructs are expressions which can return several values. The basic expressions include constants, value names, and the basic mathematical and logical operations applied to other expressions. Both languages support the same basic scalar data types and operations, and use a value-oriented programming philosophy.

Val's scalar types include boolean, integer, real, and character. Three types of structures are used: the array, the record, and the oneof. Type oneof allow discriminated union data types and is discussed in Section 4.2.1. Val allows the definition and use of constructed types. Val uses an extensive error handling system which associates error values with each data type, and extensive type checking rules to check for correctly typed arguments to each operation and function. The error handling system simplifies the treatment of errors in programs by allowing errors to propagate. Thus, computation is allowed to continue after an error arises.

Values associated with identifiers in Id are typed, not the identifiers themselves. Identifiers can assume values of any type. There are ten types of Id values which include integer, real, boolean, string, structure, procedure definition, manager definition, manager object, programmer-defined data types, and error data types. A structure value is either an empty structure or a set of <selector:value> ordered pairs. Resource manager definitions and objects allow Id to be used for operating systems programming.

The compound constructs supported by Val include the *begin* construct, the *if* construct, the *tagcase* construct, the *for-iter* construct, and the *forall* construct. Figure 2.1 shows the use of the *begin*, *if*, and *for-iter* constructs. An example of the *forall* construct is shown in Figure 2.5. The example produces four results, one integer and three arrays. The *forall* construct is explained in

Chapter 4.

Id supports similar program expressions. Four basic expressions which include blocks, conditionals, loops, and procedure applications are utilized by Id. Block expressions are similar to Val's *begin* construct. Conditional expressions are similar to the *if* construct of Val. If the result of one iteration of a loop expression does not depend upon previous iterations, the loop expression can be unraveled into concurrent executions. The unraveled loop is similar to the *forall* construct of Val. Both Val and Id use a different notation for loops than is used in conventional languages. An explicit operator is used to update the values of identifiers between iterations of a loop. Id's procedure applications have been incorporated into the design of PDL and are explained in Chapter 4.

2.4.2. Lucid

Lucid [ASHC76, ASHC77, and WADG85] is a functional programming language in which dataflow programs can be written. The use of Lucid for dataflow programming is justified by two features of the language. One, the language has no side effects; and two, the sequence of operations is determined by the data dependencies within a program. The order of statements in a program is irrelevant.

Lucid is also a formal system in which dataflow programs can be written and proofs of the programs can be deduced. Using the formal system, proofs of Lucid programs are derived directly

```
forall J in [1,N]
 X:real:=square_root(real(J));
 eval plus J*J
 construct J,X,X+1.0
end
```

Figure 2.5 - example of Val's *forall* loop

from the program text using logical reasoning. Lucid has a number of special-purpose functions which are used within a program. These special-purpose functions, called filters, require extra axioms and rules in order to prove Lucid programs.

A program written in Lucid can be thought of (operationally) as infinitely reading a stream of input values, performing computation upon these values, and producing another stream of result values. Lucid programs are composed of functions, *where* clauses, and a number of conditional expressions. The conditional expressions utilized by Lucid include the *if* expression, the *case* expression, and the *cond* expression. The *where* clause is the block structuring mechanism of the language. The *if* expression is similar to the conditional constructs of Val and Id. The *case* and *cond* expressions provide an alternative way of writing nested *if* expressions. An example of a Lucid program which performs quicksort is shown in Figure 2.6. The example is taken from [WADG85].

Identifiers in Lucid programs denote arbitrarily long sequences of data entities called histories. Lucid program statements are considered to be true mathematical assertions about the histories of these identifiers. The end of a sequence is denoted by a special eod marker.

Lucid is "typeless" in the sense that no syntactical type checking is performed and there are no type declarations. The language does support data types though, including integer, real, boolean, word, character strings, and finite lists. The language uses the special object error to denote a stream of error values. Each data type has operations, called filters, associated with it. These filters or operations behave the same as operations in most dataflow programming languages, except that filters perform *pointwise* computation on streams of data values. Pointwise filters compute one output token set for each input token set.

Lucid has a number of *special* filters used to build streams and extract elements from streams to produce new streams. These filters can be pointwise or *non-pointwise* in their operation. Non-pointwise filters can have some internal memory associated with them and can maintain their identities between each token set. Non-pointwise filters need not produce an output

```
**********************************************************************************************
```

Procedure quicksort accepts a stream of data values and produces either a stream with one element or a stream which is formed by merging the results of recursively applying quicksort on two parts of the input stream. The first part contains values of the input stream which were less than the first value in the stream. The second part contains value which were greater than or equal to the first element in the input stream. The expression in line 4 checks to see if the current input value is less than the first input value. If so, the value is placed into stream b0 (line 5). Otherwise, it is placed into stream b1 (line 6). Quicksort is then recursively called on these two streams (line 2). Finally, procedure follow (lines 7-10) is used to merge the elements of the two sorted streams to produce a final result stream (line 2). Procedure follow merges the two streams by producing all of the elements of the first input stream followed by all of the
elements of the second input stream.

```
**********************************************************************************************
```

```
 1 quicksort(a) = if iseod(first a) then a
 2                else follow(quicksort(b0),quicksort(b1)) fi
 3   where
 4     p = first a < a;
 5     b0 = a whenever p;
 6     b1 = a whenever not p;
 7     follow(x,y) = if xdone then y upon xdone else x fi;
 8         where
 9           xdone = iseod x fby xdone or iseod x;
10         end
11   end
```

Figure 2.6 - example of a *Lucid* program which performs quicksort

token set for each input token set. Thus, the filters introduce history sensitivity in programs. The special filters supported by the language include first, next, fby, whenever, asa, and upon. These filters are explained in Section 4.3.3. An example of the use of the *special* filters is presented in Figure 2.7. The example program performs *mergesort* on a stream of data values. Lucid also utilizes the special *is current* declaration which allows programmers to write programs with nested iteration.

When a user defines a function in the language, the user is actually defining a new filter which behaves the same as the special filters of the language. The function continuously accepts streams of input values, performs computation upon the input values, and produces streams of

****

Procedure msort accepts a stream of data values and performs mergesort by dividing the input stream into two parts, performing msort on each of those parts, and merging the results into one result stream. If there is only one value in the input stream (the second value in the stream is the special object eod), then that value is returned as the result of the procedure (line 1). Otherwise, the input stream is divided into two parts by using a boolean value (p) to place alternative values in the input stream into two new streams, b0 and b1 (lines 5-6). Mergesort is then performed on each of the streams and the results are merged. The value of p alternates by using an ***fby*** filter to produce a false value followed by a stream of alternating true and false values (line 4). In lines 5 and 6, two ***whenever*** filters are used to place the values into the new streams. If p is true, then the current input value is placed into stream b0; otherwise, it is placed into b1. Procedure merge (lines 7-15) is used to merge the values of the two streams into one result stream. The function uses a boolean value (takexx) which is true whenever the value of stream xx is less than the current value of stream yy or the current value of stream yy is eod. Otherwise, the value of takexx is false (lines 11-12). Procedure merge produces the current value of xx if takexx is true; otherwise, the value of yy is produced (line 7). Stream xx is produced by using an ***upon*** filter with x and takexx as inputs. The value of xx is initially the first value in stream x. Then, if takexx is true, a new value of stream x is produced; otherwise, the last value produced by the ***upon*** filter is produced again (line 9). The same is done with stream yy, except the value of not takexx is used to control the output of the ***upon*** filter (line 10). Procedure just is used to place an eod object at the end of its input stream (lines 13-14).

****

```
1   msort(a) = if iseod(first next(a)) then a
2                   else merge(msort(b0),msort(b1)) fi
3     where
4       p = false fby not p;
5       b0  = a whenever p;
6       b1  = a whenever not p;
7       merge(x,y) = if takexx then xx else yy fi
8         where
9           xx = just(x) upon takexx;
10          yy = just(y) upon not takexx;
11          takexx = if iseod(yy) then true elseif
12                      iseod(xx) then false else xx < yy fi;
13          just(a) = ja where ja = a fby if iseod ja then eod
14                                          else next a fi; end;
15        end;
16    end;
```

Figure 2.7 - example of the use of *Lucid's* filters

result values. Lucid programs are also filters which continuously accept input values and produce output results.

2.4.3. Other Dataflow Languages

Several other dataflow languages have been proposed including SISAL and DFL. SISAL (Streams and Iteration in a Single-Assignment Language) [MCGR83] is a functional dataflow language which was developed in a cooperative effort by the Lawrence Livermore National Laboratory, Colorado State University, DEC, and the University of Manchester. The language's main application is numerical computations. The language was designed after Val and has many of the same features as Val. SISAL supports streams and several operations which manipulate streams. The language uses only one error value in every data type in contrast to Val's extensive error values.

[FAUS86] presents a real-time dataflow language which is being developed as an extension of Lucid. Lucid is extended by associating a stream of time windows with each stream of data values. Time windows are needed and used because a real-time language is concerned with both data and time dependencies. The time window defines exactly when a data value can be produced by an operation.

[PATN84] discusses a high level language, DFL (Data Flow Language) whose syntax closely resembles Pascal. DFL borrows much of its languages features from Val. The language is a block-structured, single-assignment language which uses strong type checking. DFL has no provisions for records, only supports two dimensional arrays, and does not support user-defined or stream data types.

Val, Id, and Lucid were studied to discover specific useful features of each language. These features were incorporated to form the foundation of PDL. Val and Id have many similar language features, including data types and language constructs. Since the features are similar, they can be extracted from only one of the languages. Val was chosen. The language features extracted from Val include its data types and the treatment of those data types, all of the language's program constructs, and most of the operations of each data type. PDL also uses Val's static model. The features of Val were augmented by features taken from Id and Lucid. The

only feature incorporated exclusively from Id is the apply operator. The features incorporated from Lucid include the use of streams and operations to manipulate the streams. These operations include all of Lucid's special filters.

CHAPTER 3

Specification Techniques

Formal specification techniques have been widely used to specify "non-graphical" general purpose programming languages. Specification techniques can be separated into two categories, syntactic and semantic. Syntactic techniques define the syntax of a language, but specify nothing about the semantics of the language. On the other hand, semantic specifications define the formal semantics of a language, but are not concerned with the syntax of the language. One of the most widely known and used syntactic specification techniques is the BNF Grammar. Many semantic specification techniques have been developed and are discussed in [PAGA81]. The specification of PDL is based on the algebraic technique developed by Mallgren [MALL82] and the axiomatic technique developed by Hoare [HOAR69].

Formal specifications serve three purposes for the language designer and user. First, the specification provides a rigorous unambiguous description of the language at some level of abstraction. Second, the specification serves as a foundation for reasoning about programs written in that language. Last, the specification serves as a reference document for users.

The specification technique adopted in this thesis combines three specification techniques to formally describe the graphical representation, textual syntax, and semantics of the language. The three techniques are Mallgren's algebraic specification technique used to describe the graphical representation of the language, a BNF grammar used to describe the textual syntax of the language, and an axiomatic specification technique used to describe the semantics of each language feature. In this chapter, a brief overview of the three specification techniques is presented.

3.1. Mallgren's Algebraic Technique

Limited work has been done in the area of the specification of computer graphics programming languages. [MALL82] notes that there are problems with applying existing specification techniques to graphical languages. These problems include:

(1) Graphics applications utilize several special constructs which are not found in general purpose languages. Many of these special constructs are inadequately analyzed in the literature to provide a sound basis for their specification; and

(2) Graphics applications involve user interaction. The formal specification of user interaction is difficult using existing techniques.

Mallgren's specification technique utilizes graphical data types, an algebraic specification technique, and a base language to define a graphical programming language. An axiomatic specification technique is used to describe the base language statements, an algebraic data type specification technique is used to describe the graphical data types, and a new technique based on the algebraic specification technique is used to describe user interaction.

Graphical languages have been traditionally specified in an ad-hoc manner by building structures out of general-purpose data types (i.e., integer and character). Formal, structured techniques have not been developed or used to describe the diverse concepts of graphical languages. These concepts include pictures, points, graphical transformations, and user interaction. Mallgren uses the algebraic specification technique for abstract data types, developed by Guttag [GUTT78], to incorporate these concepts into a graphical programming language. An abstract data type is a collection of values and operations which manipulate the values. A specification of an abstract data type is composed of two parts: a syntactic specification and a set of axioms. The syntactic specification provides the syntactic and type information of each operation associated with the data type, including the operation's name, domain, and range. The set of axioms defines the meaning of the operations by stating relationships between operations. The axioms are expressed

using algebraic equations.

Graphical data types encapsulate the concepts of a graphical programming language in a manner that allows existing specification techniques to formally specify the data types and the graphical programming language. Mallgren states that graphical data types make it easy to write formal specifications of graphical languages and provide tools for the verification of programs written in these languages. Implementation details of each data type are *abstracted away* by allowing a programmer to use the data types only with a pre-defined set of operations. Examples of an abstract graphical data type are given in Appendix C.

User interaction is extremely important to interactive computer graphics. Mallgren specifies user interaction by extending the algebraic specification technique to handle shared data types. These data types are used to specify the interaction between the computing machine and the programmer in terms of concurrent processes.

Mallgren's technique uses several steps to specify each data type used by the graphical language. The steps are:

1. Show the operations associated with each data type in terms of algebraic equations.

2. Define the semantic portion of the specification by listing the axioms of each data type.

3. Divide the operations into three categories: generator, inquiry, and basic generators.

    a. Generator: Operations which produce objects of the type being defined. These operations are added for convenience.

    b. Inquiry: Operations which produce objects of other types.

    c. Basic Generator: Operations which are *necessary* to generate any object of the data type being defined.

4. Give meaning to each of the inquiry operations by:

a. Writing axioms for reducing each generator to an expression involving only basic generators; and

b. Provide axioms that give the result of applying each inquiry operation to each of the basic generators.

5. Define synonyms for some of the operations providing an infix notation for binary operations which are frequently used.

Mallgren has defined several data types which are used as a nucleus of a simple programming language. Many of these data types will be used to help define the graphical dataflow programming language developed in this thesis. These data types include region, point, name, user interaction, and continuous picture. Each of these data types are presented in Appendix C.

3.2. Hoare's Axiomatic Technique

The axiomatic specification technique (Hoare Calculus) was developed by C. A. R. Hoare to prove the partial correctness of programs [HOAR69]. The technique is based on the specification of axioms and rules of inference used to prove programs correct. The axioms are used to prove simple program statements, while the rules of inference are used to prove the structured statements of programs. By using rules of inference, properties of the structured statements are deduced from the properties of its constituents.

Apt [APT81] presents an excellent overview of relevant issues concerning the axiomatic specification technique including the specification of procedures with parameters, recursion, and variable declaration. Apt also discusses the soundness, completeness, incompleteness of axiomatic specifications, and gives a comprehensive reference list of other work concerning axiomatic specifications. Hoare [HOAR73] presents a complete axiomatization of the programming language Pascal, including the first proof rule for the assignment of array elements. The programming language Euclid [LOND78] was designed with the idea of axiomatization in mind. In designing Euclid, it has been seen that axiomatizing a language during its development is easier than writing axioms for the language after it has been developed. We chose to write the specification during the design phase and found that this exercise had a positive influence on PDL's design.

The axiomatic technique associates assertions about the values of variables at the beginning and end of the execution of a program or program statement. Pre-conditions are the assertions about the variables at the beginning of execution while post-conditions are the assertions at the end of execution. The assertions are expressed in a strict first-order predicate logic notation. Assertions of the form P{S}Q are used to state the relationship between the pre-condition (P), post-condition (Q), and the program or program statement (S).

The axioms and rules of inference are used to describe the semantics of a programming language up to termination (the axioms and rules do not ensure that the program will terminate). The assertion P{S}Q states that "if P is true before the initiation of program S, then Q will be true on its completion" [HOAR69]. Rules of inference permit the deduction of new assertions from one or more assertions previously proven correct. The rules of inference have the two forms shown in Figure 3.1. The first rule states that if H_1, ..., H_n are true assertions, then H is a true assertion. The second rule states that if H_{n+1} can be proven correct from assertions H_1, ..., H_n, then H is a true assertion.

Hoare initially developed one axiom for assignment and three rules of inference for a conventional imperative language. The notation used for the axiom and rules is shown in Figure 3.2. The axiom of assignment states that if *P(x)* is true after the assignment, then *P(f)* must have been true before the assignment. *X* is an identifier and *f* is an expression of the programming language which may contain *x*. P[f/x] is denoted by substituting *f* for all free occurrences of *x*.

$$\frac{H_1, \cdots, H_n}{H} \qquad\qquad \frac{H_1, \cdots, H_n \vdash H_{n+1}}{H}$$

Figure 3.1 - the forms of Hoare's rules of inference

The axiom is actually a schema which defines an infinite number of axioms which share a common form. The rules of consequence are straightforward and need no explanation. The rule of composition states that if the result of the first program statement is the same as the pre-condition of the second statement, then the composition of the two will produce the result of the second statement. The rule of iteration states that if an assertion is true on the initiation of a loop, and it is true after any number of iterations of the loop; and that on the final iteration of the loop, the boolean test (B) will be false.

Assignment Axiom:	$P_0\{x:=f\}P$
Rules of Consequence:	If $\vdash P\{S\}Q$ and $\vdash Q \supset R$ then $\vdash P\{S\}R$ If $\vdash P\{S\}Q$ and $\vdash R \supset P$ then $\vdash R\{S\}Q$
Rule of Composition:	If $\vdash P\ \{S_1\}Q$ and $\vdash Q\ \{S_2\}R$ then $\vdash P\ \{S_1;S_2\}R$
Rule of Iteration:	If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\text{while B do S}\}\neg B \wedge P$

Figure 3.2 - Hoare's Axiom and Rules of Inference

3.3. BNF Grammar

The BNF (Backus-Naur Form) grammar was originally developed to syntactically describe ALGOL and has been widely used in language definition. The grammar consists of several productions or BNF grammar rules which specify allowable sequences of character strings in the language being described. The grammar defines a programming language's legal syntax but describes nothing about the semantics of the language.

The grammar rules have the form:

<syntactic category> ::= <definition>

where the syntactic category is the name of the language construct or feature defined by the grammar rule. The '::=' means that the syntactic category is defined by the expression on the right side. The expression can be as simple as a list of objects, (e.g., a list of letters) or can consist of other syntactic categories. Recursive definitions are also allowed. Once a syntactic category has been defined, it can be used in other syntactic categories to build more complex language constructs. A complete BNF grammar of a programming language is defined by a hierarchy of grammar rules (syntactic categories). The top-level syntactic category of the hierarchy is the program.

Each of the specification techniques discussed in this chapter were combined into a *multi-dimensional* specification technique introduced in Chapter 5. The specification will describe the semantics (behavior), syntax, and graphical representation of each language construct of PDL.

CHAPTER 4

PDL: the Prototype Dataflow Language

In this chapter, an overview of PDL, with its language features, and an explanation of the design goals and motivations of the language, is presented. In Section 4.1, the design goals of PDL are presented. In the design of PDL, Val is used as a core language extended with other useful features such as Id's *apply* operator and Lucid's filters and streams. In Section 4.2, the features of Val incorporated into PDL's design are presented. Much of the textual syntax of PDL follows Val. A detailed description of PDL's syntax is given in Appendix B. In Section 4.3, the features of Lucid incorporated into PDL's design, including the concepts of streams and filters, are presented. In Section 4.4, the *apply* operator, a feature incorporated from Id is presented.

As each syntactic language construct is presented, the *abstraction* icons[6] used to represent the construct and the lower-level dataflow diagram of the construct are presented. Specific graphical icons were taken from the graphical base languages introduced in Section 2.2 to represent the base language representations of the constructs. In Section 4.5, the motivations for the features of PDL are presented. Icon design was not one of the objectives of this work. Sample icons are proposed and presented to give the reader an appreciation of the potential of the prototype language. Programmers may define and use their own abstraction icons.

4.1. Design Goals

Three design goals have been followed in the development of PDL. The main design goal for PDL is that the language support the simultaneous existence of graphical and textual

[6] In this chapter, an icon refers to a node in the program graph. The icon (node) represents an operation in the program graph. Thus, an icon is allowed to perform some action.

representations for each language construct. Most dataflow languages express programs in a textual notation which are translated into a graphical base language (a directed program graph). A user-friendly programming environment would support the coexistence of graphical and textual representations. For this reason, both representations are incorporated into the design of PDL.

The second design goal is for PDL to be extensible. Because the language was designed to be a prototype language, it is anticipated that the language will evolve. As the language is used, programming constructs may be added or deleted to improve the usefulness of the language. New features can be added to the language by defining the textual and graphical representations of the construct along with its semantics.

The final design goal for the language is to make abstraction a major part of the language. Each function of the language is thought of as an abstraction of its sub-parts. Each sub-graph (function) is represented as a single node in a graph (program). A programmer needs to view the lower-level diagram only when the diagram is created or manipulated. The programmer needs to know only what a function does, not how it performs its operations. Abstraction of all language constructs eases the programmer's task of program construction by allowing an abstraction icon to represent a complex function graph, and allows for easy use of the language since the programmer need not be concerned about the complex details of a program graph. Abstraction also allows for easy program readability since the complexity of a program graph can be reduced through the use of abstraction icons. Abstraction icons are utilized by PDL to represent each language construct, where appropriate, and user-defined operation. An example of the abstraction icon used to represent the *forall* construct is shown in Figure 4.1. The lower-level dataflow diagram of the construct is shown in Figure 4.2. Each *fat arc* in the diagram represents a collection of arcs from one node to another. The functions of each node in the diagram will be explained in Section 4.2.4.

4.2. Features Taken From Val

The language features of Val which were incorporated into PDL's design include its data types and the treatment of those data types; all of the language's program constructs, including

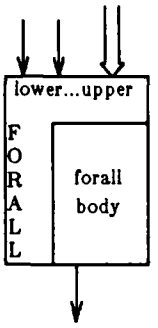


Figure 4.1 - graphical representation of the *forall* construct

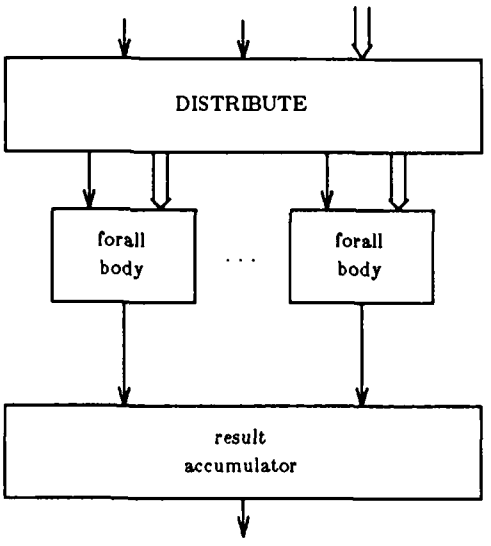


Figure 4.2 - lower level diagram of the *forall* construct

the *begin*, *if*, *for-iter*, and *forall*; and most of the operations of each data type. These operations are used in a different manner explained in Section 4.4.2. Finally, most of the textual syntax of PDL is taken from the Val language reference manual [ACKE79].

4.2.1. Data Types

The primitive types supported in PDL are integer, real, boolean, and character. The compound data types supported include: array, record, union, and function. The union data type of PDL is Val's oneof data type. PDL also supports programmer-defined data types which are constructed using the primitive and compound types. Array types have no bounds associated with them. The type of the array is the type of the constituent elements which must all have the same type. Strings are represented as an array of characters. Union types allow discriminated union data types. A union type is composed of several tags and the types associated with each of the tags. When a value of type union is created, a programmer supplies the tagname and a constituent value whose type is the type associated with the tagname. Identifiers of type union will be bound to one of the constituent values and will have that value's type. The *case* construct is used to access the constituent values of a union type. An example of this type will be given when the *case* construct is discussed. Values of type function are procedure definitions which are utilized with the apply operator which is described in Section 4.4. Associated with each type is a value domain, which consists of the proper elements of the type, one error element, and several operations to manipulate values of the type.

4.2.2. Values

PDL is value-oriented; all identifiers (value names), including compound objects, are as treated values. The language also uses the single assignment convention, but in an different manner. Identifiers denote not just a single value, but an arbitrarily long sequence of values. The use of sequences is fully explained in Section 4.4.1.

Identifiers can be bound to values of any type. The type of the identifier is specified when the identifier is first used in the program or in the type definitions section of the program. When a value is bound to the identifier, the type of the value must be equivalent to the identifier's type. If the two types are not equivalent, no automatic type conversion will occur and an error value of the appropriate type is bound to the identifier. Once a binding is made, that binding remains in

force for the entire scope of the identifier. The scope of an identifier is the body of the function or program construct in which a reference to the value name denotes its value.

Like Val, PDL incorporates extensive type-checking. All operations accept a specific type of input. If a token arrives that does not have the correct type, an error value of the appropriate type is produced. For example, if the *add* operation receives a data value whose type is not integer or real, then an error value of the appropriate type is produced. Also, all formal and actual parameters are checked for type equivalence. If the types of the parameters are not equivalent, an error value is produced by the function.

Graphically, the arcs of a program graph represent identifiers. The scope of an identifier is the node from which the arc emanates and all nodes to which it enters. Each arc is labeled with the type of the token that the arc carries. Unique labels for each type are given in Table 4.1. When a programmer defines a new data type, a unique label is associated with arcs which carry tokens of that type.

4.2.3. Expressions and Operators

All instructions, language constructs, and function definitions in PDL are expressions. The primitive expressions are the arithmetic and boolean expressions which consist of one or more

Data type	*Label*
integer	IN
real	RE
character	CH
boolean	BL
array	AR
record	RC
union	UN
function	FN

Table 4.1 - arc labels for data types

operations acting upon several operands. These expressions are expressed in prefix notation.[7] The operations actually perform pointwise computation upon streams of operand values. The use of pointwise operations is explained in Section 4.3.1. Operations may fire or execute only when each of its input arcs carry a token and its output arcs are empty. The primitive operations can be utilized in user-defined functions and program constructs. The primitive operations of each data type are presented in Appendix E.

Each primitive operation is represented graphically by one of the function icons shown in Figure 4.3. The icon is labeled with the name of the operation or a symbol denoting the operation. Each icon has either one, two, or three input arcs and an output arc. If the output of an operation, represented by a function icon, is needed by more than one instruction, the output token is transmitted through a fork which produces the appropriate number of replicated tokens.

Operands are either constants or identifiers which are bound to some value. A PDL constant can be: true, false, nil, integer numbers, real numbers, character constants, character string constants, and error[type-spec] which specifies an error value of some type.

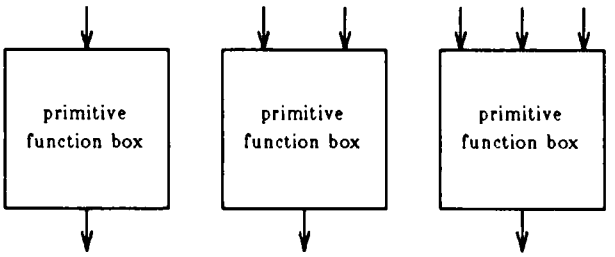


Figure 4.3 - graphical representation of primitive functions

[7] Val uses an infix notation for its operations. A prefix notation was used for PDL to exploit the easy correlation between a displayed graph and its corresponding textual representation. It is straightforward to translate the prefix notation into a infix notation.

Operands are represented graphically by the arcs of the data flow program graph. Each arc is labeled with the token type and optionally with the name of the identifier the arc denotes. Constants are represented by the icon shown in Figure 4.4. The value of the constant is the label of the icon. The icon has one input arc which is used as a trigger. When a token of any type is present on the input arc, the icon will produce the constant. A trigger was used to ease the realization of program graphs.

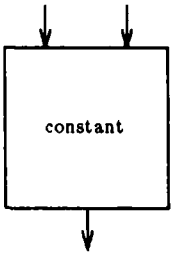


Figure 4.4 - graphical representation of constants

4.2.4. Language Constructs

The program constructs of PDL include the *begin*, *if-then-else*, *case* (Val uses the tagcase construct), *for-iter*, and *forall* constructs. Each of the constructs are expressions, called multi-expressions, which produce a tuple of result values. The syntax of each language construct is given in Appendix B.

Each construct is represented by an *abstraction* icon, where appropriate. Many of these icons are incorporated from the symbols of Nassi-Shneiderman diagrams [NASS73]. The icons are an abstract representation of the lower-level dataflow diagrams of the constructs (if a diagram is supported). The lower-level diagrams are represented using data-dependent and user-defined function icons incorporated from the graphical base languages discussed in Section 2.3. The program constructs, the abstraction icon used to represent the construct, and icons used to define the

lower level diagrams of each construct are now discussed.

4.2.4.1. If-Then-Else Construct

The *if-then-else* construct permits the selection of two alternative expressions which return result values depending upon a boolean test expression. This construct is similar to those used in conventional languages except that all input values used within the two alternative expressions must be present for the construct to fire. Each alternative expression receives every input value. The construct produces an error value if the test expression or an alternative expression returns an error value.

The *abstraction* icon used to represent the *if-then-else* construct is shown in Figure 4.5. Five icons are used to represent the lower-level dataflow diagram of the construct; one icon for the test expression, one for each of the two alternative expressions, and two data-dependent nodes. The two data-dependent nodes, merge and switch, are used to pass input values to the selected alternative and the correct result values out of the construct. Both icons are controlled by the same control token produced by the boolean test icon. Both of these icons were incorporated from Kosinski's graphical base language mentioned in Section 2.3.2. The lower level diagram is shown in Figure 4.6.

4.2.4.2. Begin Construct

The *begin* construct is used to compute several sub-expressions. The result from the sub-expressions are used to compute a final result which is returned using the return expression. The return expression is Val's result expression. The construct introduces new identifiers and bind values to the identifiers using the sub-expressions. These identifiers can be used in the return expression.

The *begin* construct is graphically represented by the abstraction icon shown in Figure 4.7. The icon is identical to Nassi-Shneiderman's Begin-End symbol. The lower-level dataflow diagram of the construct is defined by the sub-graph composed of the construct's constituent

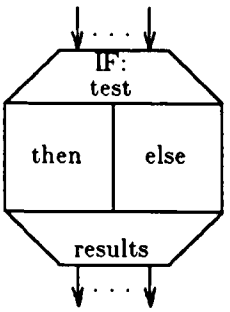


Figure 4.5 - graphical representation of the *If construct*

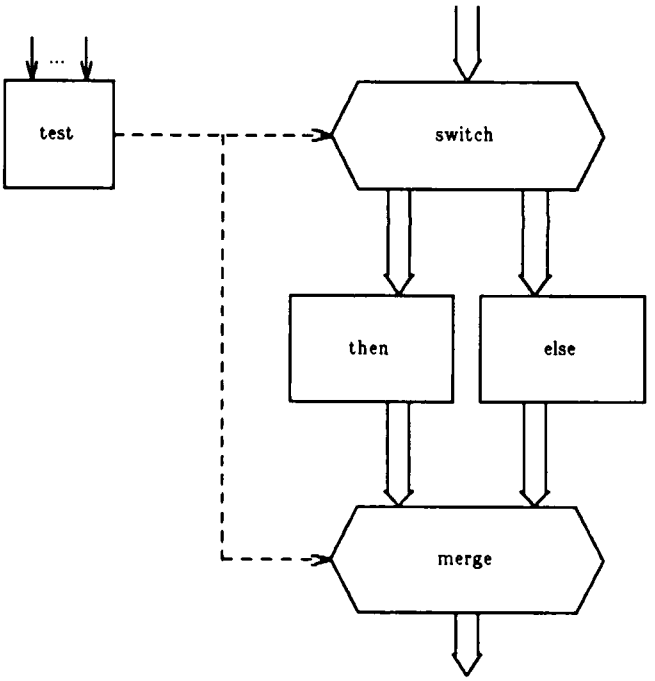


Figure 4.6 - lower level dataflow diagram of the *If construct*

expressions (icons). Any icons can be used within the sub-graph.

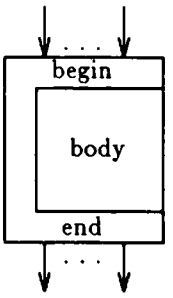


Figure 4.7 - graphical representation of the *Begin construct*

4.2.4.3. Case Construct

The *case* construct accesses values bound to identifiers of type union. An example of the construct is shown in Figure 4.8. The result of a *case* construct is the value of the expression or arm whose tag name matches the value of the test expression of the construct. If no match occurs, then the result is the construct's default expression. The default expression of PDL is Val's *otherwise* expression. The expression following the word **case** (i.e., the test expression) must be of type union and the tag names in the expressions (arms) of the construct must be the tags of that union

Let X be of type: **union**[A:**int**; B:**array**[**int**]; C:**real**]

If X has tag A and constituent value 3,

```
case P:=X
  tag A:    P + 4
  tag B:    P[6]
  tag C:    P + 2.35
end
```

the value produced by the construct will be 7. The union type has three tags, one of type integer, one of type array, and one of type real.

Figure 4.8 - example of the *case* construct

type. If the tag names of the construct comprise every tag in the union type, then the default expression is not needed. The identifier which appears after the word **case** is introduced into each expression of the construct excluding the default arm. The type of the identifier is the type indicated by the tag of a certain expression. When an expression is executed (i.e., the tag of the test expression matches the tagname of the executing expression), the value name is bound to the value from the test expression.

No abstraction icon is utilized for the *case* construct. The *case* construct is graphically represented by the lower-level dataflow diagram shown in Figure 4.9. The *route tokens* icon passes any input values and control to the correct expression depending upon the value of the input tag. Once the input values have passed to the correct expression, that expression executes and passes its result values to the *funnel tokens* icon. The *funnel tokens* icon collects the input

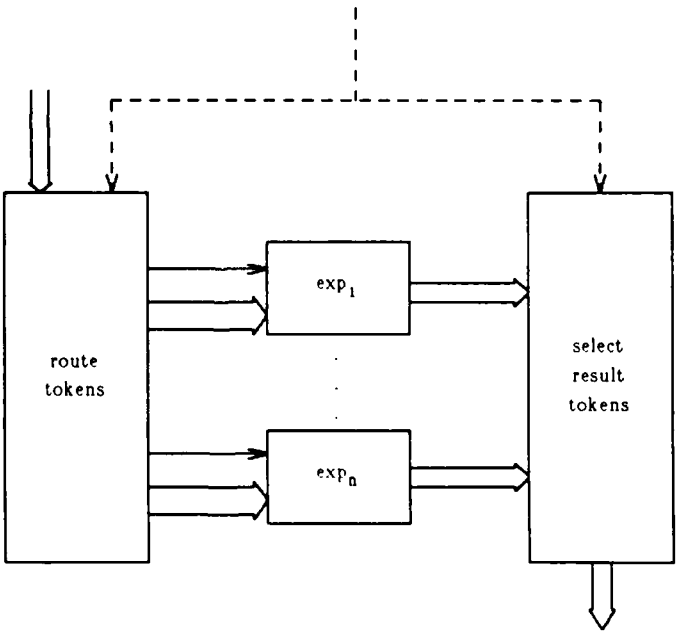


Figure 4.9 - graphical representation of the *Case construct*

tokens from the group of arcs specified by the token on the first input arc. The icon places the tokens on its output arcs in the order that the tokens were collected. The *route tokens* and *funnel tokens* are abstraction icons.

4.2.4.4. For-Iter Construct

The *for-iter* construct is used for iteration where the result of one iteration of the loop is dependent upon the results of previous iterations. By using the *iter* expression within the construct, value names are rebound to new values just prior to the next iteration of the loop. The construct consists of four parts: the initialization of variables, a test for loop termination, the redefinition of the loop identifiers, and the production of result values.

The *for-iter* construct is graphically represented by the abstraction icons shown in Figure 4.10. A lower-level diagram is not supported. One icon, the loop control icon, is used to accept the initial values of the loop identifiers,test for loop termination, and produce result values. There is one input arc for each initial value and one output arc emanating from the icon for each value produced by the loop. On successive iterations, the control icon checks the termination test for a true result. If the result is true, the icon outputs the final values of the loop variables. Otherwise, the variables are re-iterated (passed to the loop body icon). The second icon, which is an abstract representation of the body of the loop, is used to update the loop identifiers after each iteration of the loop. Any number of input and output arcs are allowed to enter and leave the body of the loop.

4.2.4.5. Forall Construct

The *forall* construct is used to explicitly specify concurrency in PDL. Using the *forall* construct, all cycles of a loop are performed simultaneously. The construct generates a set of values, and either produces the set as an array or produces the result of performing some operation on the set. The *forall* construct consists of three parts: the range specification which defines the

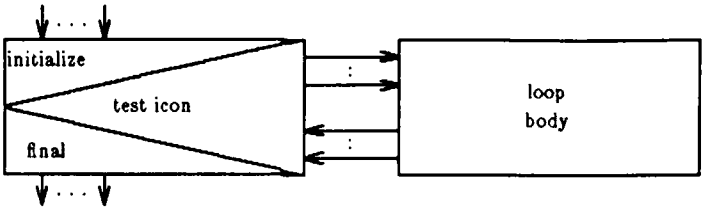


Figure 4.10 - graphical representation of the *for-iter* construct

parallelism within the construct, the body which contains the expressions to be evaluated, and the result accumulator which collects the results of the expressions and produces an array or a single result. The range specifies the number of separate and independent instantiations of the body of the *forall*. Each instantiation of the body proceeds asynchronously. Two types of result accumulators are used: the *construct* accumulator and the *eval* accumulator. The *construct* accumulator produces a one dimensional array where each element in the array is one of the values produced by an instantiation of the *forall* body. The *eval* accumulator produces a single result which is computed by performing an associative and commutative operation (i.e., addition or multiplication) upon the result values of the instantiations. Only one result accumulator can be used in a *forall* loop.

The *forall* construct is graphically represented by the abstraction icon shown in Figure 4.1. The icons used to represent the lower level diagram of the construct are shown in Figure 4.2. The *forall* body is instantiated once for each element in the range of the construct. The input arcs of the *distribute* icon carry the lower and upper bounds of the range and any input values used within the *forall* body (represented by the fat arc). The *distribute* icon passes one element of the range to each instance of the body of the construct. One of two accumulator icons may be used; the *construct* accumulator or the *eval* accumulator. The op_name label of the *eval* icon is the operation performed on the results.

4.2.5. User-defined Functions

PDL incorporates Val's function definitions and calling. Functions are allowed to have as many parameters as needed. The types of all parameters and the returned values must be specified in the function header. Formal and actual parameters must have equivalent types or an error is produced by the function. All parameters are values and cannot be rebound in the function body. The function body contains several expressions which can produce multiple results including simple expressions, language constructs, and other function calls.

User-defined functions are graphically represented by the abstraction icon shown in Figure 4.11. The icon is an abstract representation of the lower-level diagram or sub-graph of the component expressions of the function. The number of input, output arcs is equivalent to the number of input, output parameters defined in a function header. The label of the icon is the name of the function it represents. Programmers can define unique icons to denote user-defined functions.

4.3. Features Taken From Lucid

This section presents the language features of Lucid incorporated into PDL. These features include the use of sequences instead of single values for all identifiers and Lucid's special filters. Since the identifiers denote sequences of data tokens (values), the operations of the language per-

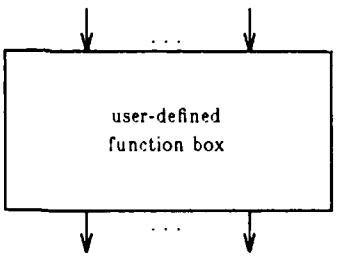


Figure 4.11 - graphical representation of a *user-defined function*

form pointwise computation on streams. These operations are called filters.

4.3.1. Infinite Sequences

The identifiers of PDL denote an arbitrarily long sequence (or stream) of values, not just a single value. Each sequence has an associated data type; all elements of the sequence are of that type. Associated with each sequence are the special BOS and EOS control tokens. The BOS and EOS tokens denote the beginning and end of a sequence, respectively. A Stream can contain a single value by having the single value enclosed by BOS and EOS tokens. Sequences are graphically represented by the input and output arcs of a program graph.

4.3.2. Operators are Filters

All operations in the language accept sequences of data and produce a sequence of result values. The operations act as *pointwise* filters, continuously producing a set of output values for each set of input values. The semantics of most operations direct an operation to scan for the BOS token. When the BOS token is present on each of the input arcs, a BOS token is placed on each of the output arcs. As each set of input values is absorbed, the operation performs its computation upon the values, producing a set of output values which are placed on the output arcs. Each input arc must carry a token and the output arcs must be empty before a filter can fire. After the filter has fired, the input arcs are empty and the output arcs will carry result tokens. When an EOS token is present on each of the input arcs, an EOS token is placed on the output arcs. The operation is now considered to be finished firing. The firing of the *merge* is different. When a BOS, EOS token is present on the control arc, the node places the BOS, EOS on the output arc. All filters operate in parallel and asynchronously.

4.3.3. Special Filters

PDL incorporates Lucid's special filters to isolate certain values of a sequence or to create new sequences. Two sub-classes of special filters are used. One sub-class extracts certain elements from a stream and produces new streams from other streams. This sub-class accepts input

stream(s) of any, but equivalent type. Included in this sub-class are the first, rest, and concatenate filters[8]. The second sub-class produces a resultant data stream depending upon the values of a corresponding control stream. This sub-class accepts two input streams; one of which consists of data tokens of any type, and the other of which consists of boolean control tokens. Included in this sub-class are the whenever, advance_upon, and as_soon_as filters. All special filters can perform either pointwise or *non-pointwise* computation. Non-pointwise filters are an extension of pointwise filters because these filters can have some internal memory associated with them. Non-pointwise filters can maintain their identity between each token of the input stream and/or need not produce an output token set for each input token set. Thus, the special filters are *history sensitive* functions. A brief explanation of each filter follows. The syntax of the filters is given in Appendix B.

first: This filter produces a stream of tokens, each having the value of the first token in the input stream.

rest: The output of this filter depends upon the current step in the execution of a program. At each step of the execution, the output token produced is the next token to arrive on the input arc (i.e., the next token in the stream). The filter discards the first value of the input stream. The stream produced is the input stream less the first token.

concatenate: This filter produces a resultant stream which is formed by concatenating the second input stream to the first input stream.

[8] Some of the names of the filters incorporated from Lucid have been changed to clarify their usage.

whenever: For each input token set, if the value of the boolean control token is true, then the data token is placed on the output arc. Otherwise, the filter discards the data token. The filter produces a stream of all values of the data stream for which the corresponding control stream values were true.

as_soon_as: This filter repeatedly inputs token sets until the control token value is true. If the control token value is never true, the filter produces nothing. The filter produces a stream of values, each token in the stream having the value equivalent to the corresponding data value for the first true control value.

advance_upon: This filter accepts and places the first data value on its output arc. Then, as successive token sets are input, if the value of the control stream is true, a new data value is input and placed on the output arc. Otherwise, the old data value is *repeated* and placed on the output arc again. Thus, the data stream is *stretched* by repeating some of its values.

The graphical icon used to represent the first and rest filters is shown in Figure 4.12a. The icon has one input arc and one output arc. The graphical icon used to represent the other filters is shown in Figure 4.12b. This icon has two input arcs and a single output arc.

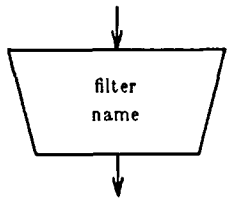


Figure 4.12a - icon used for the first and rest filters

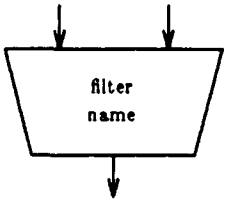


Figure 4.12b - icon used for the concatenate, whenever, as_soon_as, and advance_upon filters

4.3.4. Is Current Declaration

The use of Lucid's special filters and streams necessitates the use of Lucid's *is current* declaration. The declaration allows programmers to write programs with nested iteration. When a programmer uses nested iteration in PDL, the values of outer loop identifiers must remain constant while the values of inner loop identifiers take on each value in the stream. The declaration is used to *freeze* a certain value of an outer loop identifier which can then be utilized in the inner loop. Figure 4.13 presents an example of the use of the *is current* declaration. The effect of the example is to set up an outer loop in which x and n are only updated between executions of the inner loop (lines 3-4). The values of X and N are the *frozen* values of the tokens from streams x and n. For example, if stream x contains the elements 1, 2, and 3, and stream n contains the ele-

```
1  t = asa(p,=(index,N))
2    where
3      X is current x;
4      N is current n;
5      p = concatenate(1,*(p,X));
6    end
```

Figure 4.13 - example of the *is current* declaration

ments 4, 5, and 6; then as identifier p is updated (line 5), X and N remain constant. On the first iteration of the inner loop, X and N have 1 and 4 as their values, respectively. If the *is current* declaration was not used, the values of x and n would be updated (i.e., be bound to each value in the stream) for each iteration of the inner loop, not the outer loop, which is desired. The graphical icon used to represent the *is current* declaration is shown in Figure 4.14.

4.3.5. User-defined Functions

When a programmer defines a function in PDL the programmer is really defining a new filter which behaves like the special filters of the language. The function continuously accepts input, performs an operation upon the input, and produces a new stream of result values. Functions are allowed to be either pointwise or non-pointwise in their operation. Functions are defined using the basic expressions and program constructs described in Section 4.2 along with the special filters.

4.4. Features Taken from Id

As stated in Section 4.2, Val and Id are similar in content and have many of the same language features. For this reason, only one language feature was taken exclusively from Id, the *apply* operator. The operator allows run-time bindings of procedure definitions to its parameters to be performed. The reason for the *apply* operator's inclusion in PDL is that the operator allows

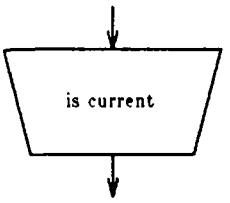


Figure 4.14 - graphical representation of the *is current declaration*

run-time identification of procedures to be applied on argument lists. Thus, several run-time bindings of a procedure definition are possible.

The graphical representation of the Apply operator is shown in Figure 4.15. The apply icon utilizes two input arcs, one of which is a fat arc, and one output arc, which is also a fat arc. The first input arc carries a function definition while the second (fat) arc carries the input parameters. The output arc carries the result values which are computed by "applying" the function definition to the input tuple. One function definition can be applied to several sets of input parameters by using one apply function box for each set of input values.

4.5. Motivations

PDL was designed for experimentation with writing parallel code in the form of dataflow programs. The language serves as a prototype language from which future designers can formulate new languages by adding new features or deleting existing ones. Since the language supports

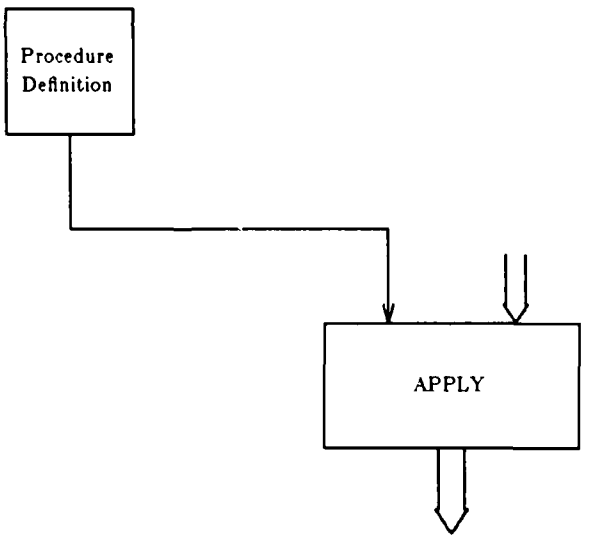


Figure 4.15 - graphical representation of the *Apply Operator*

both graphical and textual representations for programs, it is expected that the language will have only positive effects for programmers.

Several design goals were followed in the formulation of the language. These were discussed in Section 4.1. The features incorporated from Val form a *core language* which could be extended with new language features. Using Val as the core language has helped PDL achieve two of its design goals: abstraction and extensibility. Many of Val's language features which were incorporated into PDL's design facilitate the use of abstraction. The functionality in Val allows PDL to use abstraction icons, to represent each language feature. Programmers need not be concerned about the execution details of a function or language construct after they have been defined, only what the function or language construct does. The use of Val as a core language also helped PDL to achieve the design goal of extensibility. Several extensions have been made to the features incorporated from Val. These include Lucid's filters and streams, and Id's *apply* operator.

Lucid's filters and streams were incorporated into the design of PDL for two reasons. First, the use of streams and the special filters allows PDL to exhibit *history sensitivity*. Because PDL exhibits history sensitivity, the language can be used for real time programming, if desired. The second reason for the use of Lucid's filters and streams is that their use potentially increases the amount of parallel computation in a PDL program. All filters can act in parallel and asynchronously. Filters which require input from another filter do not have to wait for the filter to process the whole stream before it can begin executing. A programmer does not have to be concerned about the rate at which two filters process and produce data items. Also, all tokens in a stream may be processed simultaneously if there are no data dependencies between the tokens.

Id's *apply* operator was incorporated into the design of PDL to allow the run-time identification and binding of a procedure to its parameters. Several run-time bindings of a procedure are possible. Also, run-time decisions can be made to determine which procedure to apply to an argument list.

Some experimentation must be performed on the language before an assessment of its full potential for application can be made. Since the language serves as a prototype language for future development, the language is expected to evolve until an optimal language is found. A formal specification of PDL has been developed to aid future users and designers in their utilization of the language. The next chapter presents the specification technique which was used to formally describe PDL.

CHAPTER 5

Specification of PDL

5.1. Introduction

In this chapter, the formal specification of PDL is described. The specification incorporates the three specification techniques introduced in Chapter 3 to describe the graphical representation, textual syntax, and semantics of the language constructs. These three descriptions are combined in a data structure as described in Section 5.2. Mallgren's algebraic specification technique is used to formally describe the graphical icons used in the language. In Section 5.3, the graphical data type used to describe the function nodes of PDL is described. In Section 5.4, the BNF grammar used to describe the textual syntax of PDL is discussed. In Section 5.5, the specification of the semantics of each language construct is discussed.

5.2. Specification Model

Three specification techniques are utilized to define the semantics, graphical representation, and textual syntax of each of PDL's syntactic constructs. A data structure, the form of which is shown in Figure 5.1, is used to describe the complete specification of each of PDL's syntactic constructs. A brief discussion of each component of the data structure follows.

The *names* component defines which language constructs (language defined operations or program constructs) are described by the data structure. A data structure may define more than one of PDL's constructs. The name of each construct defined by the data structure must be given in the *names* component of the data structure. A language construct name cannot appear in more than one data structure.

```
data structure <name>
 {
   names: construct names;
   semantics: semantic description of construct
   graphics: reference to graphical description
   textual: reference to BNF description of construct
 }
```

Figure 5.1 - data structure for defining an operation

The *semantics* component defines the behavior of the language construct in terms of an axiomatic description. The *semantics* component defines the number of input values used and output values produced by the constructs described. The component also defines the transformations performed on the input values to produce the output values.

The *graphics* component makes a reference to a procedural description which defines how to draw the icons representing the language construct. A graphical data type (tree structured node) and its operations are used to create and manipulate program graphs.

The *textual* component gives a reference to the grammar rule in the BNF grammar which defines the syntax of the language constructs defined by the data structure. The BNF grammar gives the textual symbol that is used for an operation.[9]

Each component of the specification will now be described in detail. A complete specification of the graphical representation, syntax, and semantics of the language is presented in Appendix B, C, and D, respectively. An example of a complete specification of a primitive operation and compound program construct will be given at the end of this chapter.

[9] For primitive operations, an icon can be labeled with either the name of the operation it denotes, or the symbol used for that operation.

5.3. Graphical Specification

The graphical specification utilizes the algebraic specification of graphical data types to describe the components of the graphical dataflow programming language. A new graphical data type, Tree-Structured Node (tnode), is defined to describe the nodes and arcs of a dataflow program graph.

Tnodes are ordered tree structured program graphs which contain functional nodes (primitive or user-defined) and arcs. Tnodes are constructed with the help of data types *point*, *string* and *name* incorporated from Mallgren's work[MALL82]. Data type *point* corresponds to a point on the screen. Data type *name* is a collection of constant names. Data type *string* is a sequence of characters.

Simple tnodes contain the primitive function nodes of the language and arcs connecting the nodes. All nodes and arcs have a name or label that is supplied by the programmer. Tnodes also have programmer supplied names. More complex tnodes are constructed by allowing tnodes to contain other tnodes (user-defined sub-graphs), called *progeny*, as well as primitive nodes and arcs. An inserted progeny is viewed as a single node in the tnode. The extend operation is used to view the progeny's corresponding sub-graph. Each primitive function node in a tnode has a unique name, but two distinct tnodes may contain primitive function nodes with the same name.

The data type *Tnode* and its operations are shown in Figure 5.2. Each operation, and the mapping of its domain to its range, is defined below. The operations are split into the basic generators, generators, and inquiry operations (as described in Section 3.1). The basic generators include the *nullnode*, *moveto*, *put_node*, *arc_to*, and *text* operations. The operations preceded by a • are the basic generators. The operation preceded by a * is a hidden operation which the user cannot use when creating program graphs.

Tree-Structured Node (tnode,tp)

•	nullnode	name ⇒ tnode
•	moveto	tnode X point ⇒ tnode
•	put_node	tnode X ds_name X name ⇒ tnode
•	arc_to	tnode X name.int X name.int X name ⇒ tnode
•	text	tnode X string ⇒ tnode
	curpos	tnode ⇒ point
	replace_node	tnode X name X name ⇒ tnode
	remove_node	tnode X name ⇒ tnode
	remove_arc	tnode X name ⇒ tnode
	expand	tnode X name ⇒ tnode
*	display	tnode ⇒ picture

Figure 5.2 - data type *Tree-Structured Node*

function *nullnode*(n:name): tnode

Empty user-defined function nodes are created by the *nullnode* operation.

function *moveto*(N:tnode, p:point): tnode

The current position becomes point p.

function *put_node*(N:tnode, d:ds_name, n:name): tnode

Put_node inserts a node with name n at the current position in dataflow graph N. Parameter *d* makes a reference to the specific data structure that the icon being inserted denotes. The data structure utilizes another reference to a procedural description which defines how the icon is drawn. The *ds_name* of all language operations is the name of the data structure which defines the operation. The *ds_name* of all program constructs is the name of the data structure which defines the construct. The *ds_name* of a user-defined function is the name of the function.

function *arc_to*(N:tnode,n1:name.x,n2:name.y,n:name):tnode

Arc_to draws an arc from the xth output connector of node n1 to the yth input connector of node n2. The new arc is given name n. The current position is not changed.

function *text*(N:tnode, s:string): tnode

Text inserts the pictorial representation of string s starting at the current position. The current position does not change.

function *replace_node*(N:tnode,n1:name,n2:name,d:ds_name):tnode

Replace_node replaces node n1 with node n2. The data structure used to describe node n2 is specified by d.

function *remove_node*(N:tnode, n:name): tnode

Remove_node removes all nodes named n from tnode N. If there are no nodes named n, the value of N is returned unchanged.

function *remove_arc*(N:tnode, n:name): tnode

Remove_arc removes all arcs named n from tnode N. If there are no arcs named n, the value of N is returned unchanged.

function *expand*(n:name): tnode

Expand takes a single abstraction icon (node) n and expands the node into its corresponding sub-graph. If node n is not a abstraction icon, no expansion will occur.

function *curpos*(N:tnode): point

Curpos returns the current position of N.

This new data type is combined with several of the types presented by Mallgren [MALL82] to form the complete graphical specification of PDL. The types incorporated from Mallgren's work include ***point***, ***name***, ***string***, ***region***, ***user interaction***, and ***continuous picture***. An explanation of the uses of these data types can be found in [MALL82 and MAL82b] and is not given here.

5.4. Textual Specification

Since most of the syntax of the language is extracted from Val, the BNF grammar presented in [ACKE79] is adapted for PDL. One modification of Val's syntax was converting the notation used for all operations from infix to prefix. Some additional syntactic categories were added to the BNF grammar to describe the language features incorporated from Lucid and Id. The main syntactic category used in the grammar is the multi-expression (multi-exp). Since program constructs are translated into a graphical base language representation which utilize run-time decision nodes (i.e., merge and switch), grammar rules for these nodes are not needed and are not given.

5.5. Semantic Specification

The specification of the semantics of the language utilizes a form of axiomatic specification to describe each language feature supported by the language. Axioms are used to describe the primitive operations, special filters, and run-time, data-dependent operations of PDL. Rules of inference are used to describe PDL's program constructs. The notation used for the axioms and rules of inference will now be described.

5.5.1. Notation for Axioms

Many of PDL's operations act on several data types. Since the semantic description is always identical for all data types the operation acts upon, one semantic description is used to

specify each operation. The semantic description states which data types are acceptable as inputs to the operation, the name of the operation (for a certain data type), and the axiomatic description of the operation. The semantic description is of the form:

```
FOR_TUPLE          <tuple of variables>
BOUND_TO_TUPLES    <tuple list>
IN                  <axiomatic description>
```

where the tuple list consists of several tuples of the form (data type, ... ,data type,operation name). The operation name is the name that a programmer can use to label a node in the dataflow program graph. The data types define the type of the tokens that the operation acts upon.

The description states that the variables in the tuple of variables are bound to the data type and operation values in one tuple of the tuple list. The tuple of variables are then utilized in the axiomatic description which specifies the semantics of the operation. An example of a tuple list is given in Figure 5.3. The variables VT and OPNAME are bound to the values in one of the tuples in the tuple list. The tuple list has two tuples, one for data type integer and one for type real. Each tuple of bound variables is used in the axiomatic description of the operation to form a specific axiomatic description for the data type of the tuple. The axiomatic description utilizes the same form as that used for Hoare's axiom which was introduced in Chapter 3.

```
FOR_EACH_TUPLE   (int,int_add), (real,real_add)
BOUND_TO_TUPLE   (VT,OPNAME)
IN                (axiomatic description)
```

Figure 5.3 - example of a tuple list for *add* operations

A dataflow operation can be performed when each arc (or a combination of arcs) has a token with a valid type and when no arc carries a token with the error value. Some operations have other conditions which must be true for the operation to fire. If all conditions have been satisfied, the operation executes by performing some transformation upon the input tokens and producing output tokens. When an operation (node) has finished firing, the input arcs are empty and a token with a valid type is carried on the output arc(s).

The axioms which are used to describe operations have the form:

$$<\text{PREC}>\{\text{OPNAME(inputs)}\}<\text{POSC}>.$$

where <PREC> are the pre-conditions which must be satisfied and <POSC> are the post-conditions which will be true after an operation has executed. The pre-conditions of the axiomatic description test for the conditions that must be satisfied for the operation to fire. Pre-conditions also test for a valid arc configuration. Arc configurations consist of the types of the tokens that must be present on an arc before and after the operation has fired. Each input arc is denoted by $I_{\#}$ where '#' is the input arc number. Arc configurations have the form:

$$<I_1\text{:type}, \ldots, I_n\text{:type};O_1\text{:type}, \ldots, O_n\text{:type}>$$

where the types of the input and output arcs are listed in order (i.e., input arc one is listed first, input arc two is listed second, and input arc *n* is listed *nth*). The input arc types are separated from the output arc types by a semicolon. Each arc type in the arc type lists are separated by a comma. For example, $<I_1\text{:VT},I_2\text{:VT;E}>$, where VT denotes an integer, states that input arcs one and two must carry a token of type integer and the output arc is empty.

The execution description gives the name of the operation and the inputs used by the operation. The OPNAME is the name of the operation as specified by the language definition.

The post-condition describes the arc configuration and any other conditions that are true after the operation has fired. The arc configuration uses the same notation as the arc configuration in the pre-condition. The post-condition arc configuration also defines what values will be present on the output arcs after an operation has fired. These values may be expressed in terms of the values that were on the input arcs.

An example of a axiom for the addition operation is shown in Figure 5.4. The axiom states that if each arc carries a token of type VT and the output arc is empty, then the output of the operation will be the addition of the two input values.

Two global axioms are used to describe the behavior of operations when an error value or an invalid input appears on an arc. The first axiom states that if an error value is present on any input arc of an operation, the operation produces an error value of the appropriate type. The second axiom states that if a value of an input arc is invalid, then the operation will produce an error value of the appropriate type.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_add), (real,real_add)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1 + I_2:VT>$$

Figure 5.4 - example of an axiom for *add* operations

5.5.2. Notation for Rules of Inference

Rules of inference are used to specify the language constructs of PDL. Each rule of inference has the form:

FOR_EACH (list of variables)
BOUND_TO_ONE_OF (list of types)
IN

$$\frac{H_1, \cdots, H_n}{H}$$

where the H_1, ... ,H_n are previously proven axioms and H is the axiom which is being proven correct. Axioms H gives the syntactical statement, pre-conditions, and post-conditions of the program construct. Each axiom, H_1, ... ,H_n, uses the same notation discussed in Section 5.5.1. The pre and post conditions of axiom H give the valid arc configurations of the construct along with any other conditions which are true before or after the construct fires. An example of a rule of inference for the *forall* construct is shown in Figure 5.5.

FOR_EACH(VT_1,VT_2,VT_n)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{c}<i{:}int,I_1{:}VT_1,\ldots,I_n{:}VT_n;E_1>\{S_1\}<E_1,\ldots,E_n;O_i{:}VT_2> \wedge (L \le i \le U),\\ F \in \{plus,mult,and,or,min,max\}\end{array}}{<I_1{:}VT_1,\ldots,I_n{:}VT_n;E>\{\textbf{forall i in } |L,U| \textbf{ do } S_1 \textbf{ eval:} F\}<E_1,\ldots,E_n;F(O_1,F(O_2,\ldots,F(O_{U-L}O_{U-L+1})\ldots)):VT_2>}$$

Figure 5.5 - rule of inference for the *forall* construct

5.6. An Example

An example of a complete specification of an operation and a language construct will be presented in this section. The data structure which is used to specify the *add* operations of PDL is shown in Figure 5.6. The name of the operations which the structure is used to specify include the ***int_add*** and ***real_add*** operations. The *semantics* component of the structure gives the axiom

which is used to specify the operations. The *graphics* component gives a reference to the draw_add_node procedure which draws the correct icon. Finally, the *textual* component gives the correct textual syntax used for the operation.

The data structure used to specify the *forall* construct of PDL is shown in Figure 5.7. The *semantics* component of the structure gives the rule of inference used to specify the construct. The *graphics* component gives a reference to a executable procedure which draws the abstraction icon used to represent the construct. Finally, the *textual* component gives a reference to the grammar rule in the BNF grammar which describes the *forall* construct.

Appendix B gives the complete specification of the semantics, syntax, and graphical representation of PDL.

```
data structure add_node
{
 names: int_add, real_add;
 semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_add), (real,real_add)
  IN
                    <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;I1 + I2:VT>
 graphics: draw_add_node;
 textual:  +(I1,I2);
}
```

Figure 5.6 - example of data structure for *add* operation

forall_eval_node
{
 names: forall_eval
 semantics:
  FOR_EACH(VT_1,VT_2,VT_n)
  BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
  IN

$$\frac{\begin{array}{c}<i{:}\mathbf{int},I_1{:}VT_1,\ldots,I_n{:}VT_n;E_1>\{S_1\}<E_1,\ldots,E_n;O_1{:}VT_2> \wedge (L\le i\le U),\\ F \in \{\mathbf{plus},\mathbf{mult},\mathbf{and},\mathbf{or},\mathbf{min},\mathbf{max}\}\end{array}}{<I_1{:}VT_1,\ldots,I_n{:}VT_n;E>\{\mathbf{forall}\ i\ \mathbf{in}\ [L,U]\ \mathbf{do}\ S_1\ \mathbf{eval}{:}F\}<E_1,\ldots,E_n;F(O_1,F(O_2,\ldots,F(O_{U-L},O_{U-L+1})\ldots)){:}VT_2>}$$

 graphics: draw_forall_node;
 textual: forall_expr
}

Figure 5.7 - example of a data structure for *forall* construct

CHAPTER 6

Summary, Conclusions, and Suggestions for Future Work

6.1. Summary and Conclusions

In this thesis, the formal specification of the prototype dataflow programming language, PDL, that supports both graphical and textual representations of programs has been presented. PDL serves as a core language which is expected to evolve. Future designers can add new or delete existing features to obtain an optimal language. The specification technique developed allows designers to add new features by defining the semantics, textual representation, and graphical representation for the added features. The language supports an applicative programming style (incorporated from Val and Id) which is augmented by Lucid's streams and filters.

Several conclusions are made concerning PDL and its formal specification. First, we are convinced that PDL meets its design goals. PDL supports the simultaneous existence of graphical and textual representations for *all* language constructs. The use of both representations is expected to have a positive effect on the programming process; the only disadvantage will be the time required to learn how to merge the representations into a useful programming tool. At each stage of program development, a programmer could write a program using the type of representation that is most convenient for writing that part of the program. The use of graphics in programming can only have positive effects, not just for dataflow programming, but for programming in general.

The design of PDL places no barriers on extensibility. Since PDL adopts a graphical base language, new language constructs can be incorporated into the language by assuring the constructs can be translated into graphical base language equivalents. A new data structure which

defines the textual syntax, graphical representation, and semantics of the new construct would need to be introduced. Since PDL serves as a prototype, the language is expected to evolve. The extensibility of PDL can be found in many of its language features. Several examples of how the language can be extended include the relaxation of the strong typing used in PDL and allowing the eval and construct expressions to appear in a *forall* loop together.

As another result of using a graphical base language, the idea of abstraction can be adopted naturally in PDL's design. Each language construct can be represented by a single icon where appropriate. Once the lower level base language program graph has been defined, the programmer can utilize an abstraction icon to represent the graph. A programmer may also utilize programmer-defined abstraction icons.

The second conclusion concerns the formal specification of PDL. Since the technique incorporates three formal techniques to define the language, the technique is also formal and can be used for formal reasoning. The main goal of developing the specification was to provide language implementors with an unambiguous and precise description of the language so that the language could be implemented on a graphical workstation. Since Hoare's axiomatic specification technique was used to define the semantics of the language, the specification can also be used for formal reasoning about programs written in the language.

The use of an axiomatic specification technique to describe the semantics of PDL is interesting because axiomatic definitions describe th state of program variables before and after the execution of a program statement. PDL describes the state of arcs before and after the execution of program expressions. The states of the arcs are expressed as the pre and post conditions of axioms.

Because of the rigorous manner in which the semantics of PDL were specified, the final semantics of PDL evolved in a positive manner through numerous interactions with committee members. As the semantics were defined, we were forced to discover the unambiguous and precise semantics of all language constructs.

6.2. Suggestions for Future Work

PDL was designed for experimentation with writing parallel code in the form of dataflow programs. The proof of the work performed in the thesis work will only come with the implementation of PDL on a graphical workstation environment and the development of translation mechanisms to translate PDL programs into code that can run on parallel host machines. Once PDL has been implemented, an evaluation of the language should be performed. Thus, three areas of future work are noted.

The first area of future work is the implementation of the language and the development of a graphical workstation environment. Work has begun at the University of Southwestern Louisiana to develop such a programming environment. To date, no implementation of the language or a graphical work station environment has been finished. This implementation would include developing a mapping mechanism between the graphical and textual representations of the language.

After the development of a programming environment and the implementation of the language on the environment, an evaluation of the language by programmers should be performed. The evaluation would include the identification of which features should and should not be in the language, and the development of several test programs on the programming environment. Although some programs have been developed (on paper) using the language, a complete evaluation of PDL is not possible until a graphical programming environment is implemented.

Another area of future work is the development of a translation mechanism which would translate PDL programs into code which could be run on parallel host machines. This translation may not be easy and would depend upon the primitives supported by each individual parallel host machine.

A final area of future work is the development of a translation mechanism which would translate PDL programs into code to be executed on the *Dataflow Simulator (DFSS)* which is currently on the Multics system at the University of Southwestern Louisiana. This translation

would be easy because PDL has many of the same constructs supported by the base language of *DFSS*.

REFERENCES

[ACKE79] Ackerman, W. B., and Dennis, J. B., *Val -- A Value-oriented Algorithm Language, Preliminary Reference Manual*, Laboratory of Computer Science, MIT, Cambridge, MA, February 8, 1979.

[ACKE82] Ackerman, W. B., "Data Flow Languages," *Computer*, Vol. 15, No. 2, February 1982, pp. 15-25.

[ADAM68] Adams, D., *A Computation Model with Data Flow Sequencing*, Technical Report CS-117, Computer Science Dept., Stanford University, Palo Alto, CA, 1968.

[AHUJ86] Ahuja, S., Carriero, N., and Gelernter, D., "Linda and Friends," *Computer*, Vol. 19, No. 8, August 1986, pp. 26-34.

[APT81] Apt, K. R., "Ten Years of Hoare's Logic: A Survey - Part I," *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 431-483.

[ARVI78] Arvind, Gostelow, K. P. and Plouffe, W., *An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science Technical Report #114a, University of California, Irvine, Irvine, CA, December 8, 1978.

[ARVI80] Arvind, Kathail, V. and Pingali, K., *A Dataflow Architecture with Tagged Tokens: Preliminary Report*, Technical Report #174, Laboratory for Computer Science, MIT, Cambridge, Mass., September 1980.

[ARVI82] Arvind, and Gostelow, K. P., "The U-Interpreter," *Computer*, Vol. 15, No. 2, February 1982, pp. 42-49.

[ASHC76] Ashcroft, E. A. and Wadge, W. W., "Lucid - A Formal System for Writing and Proving Programs," *SIAM Journal of Computing*, Vol. 5, No. 3, September 1976, pp. 336-354.

[ASHC77] Ashcroft, E. A. and Wadge, W. W., "Lucid, a Nonprocedural Language with Iteration," *CACM*, Vol. 20, No. 7, July 1977, pp. 519-526.

[BACK78] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, Vol. 21, No. 8, August 1978, pp. 613-640.

[BIC84] Bic, L., "Execution of Logic Programs on a Dataflow Architecture," *Proceedings of 11th International Symposium on Computer Architecture*, Ann Arbor, MI, June 1984.

[BROC78] Brock, J. D., "Operational Semantics of a Data Flow Language," Laboratory for Computer Science TM-120, MIT, December 1978.

[BROO87] Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.

[CARL85] Carlson, W. W. and Hwang, K., "Algorithmic Performance of Dataflow Multiprocessors," *Computer*, Vol. 18, No. 12, December 1985, pp. 30-40.

[DAVI78] Davis, A. L., *Data Driven Nets: a Maximally Concurrent, Proceudral, Parallel Process Representation for Distributed Control Systems*, Technical Report UUCS-78-108, Computer Science Department, University of Utah, July 1978.

[DENN74] Dennis, J. B., "First Version of a Dataflow Procedure Language," *Lecture Notes in Computer Science*, 19, (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York, 1974, pp. 362-376; also Computation Structures Group Memo 93-1, Project MAC, MIT, Cambridge, Mass., August 1974.

[DENN80] Dennis, J. B., "Data Flow Supercomputers," *Computer*, Vol. 13, No. 11, November 1980, pp. 48-56.

[DENN84] Dennis, J. B., et. al., "Modeling the Weather with a Data Flow Computer," *IEEE Transactions on Computers*, Vol. C-33, No. 7, July 1984, pp. 592-603.

[FAUS86] Faustini, A. A. and Lewis E. B., "Toward a Real-Time Dataflow Language," *IEEE Software*, Vol. 3, No. 1, January 1986, pp. 29-35.

[GAUD86] Gaudiot, J., "Structure Handling in Data-Flow Systems," *IEEE Transactions on Computers*, Vol. C-35, No. 6, June 1986, pp. 489-500.

[GRAF85] Grafton, R. B., and Ichikawa, T., "Visual Programming - Guest Editors' Introduction," *Computer*, Vol. 18, No. 8, August 1985, pp. 6-9.

[GURD85] Gurd, J. R., Kirkham, C. C., and Watson I., "The Manchester Prototype Dataflow Computer," *CACM*, Vol. 28, No. 1, January 1985, pp. 34-52.

[GUTT78] Guttag, J. V., and Horning, J. J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica* 10 (1978), pp. 27-52.

[HAN85] Han, S. Y., *A Language for the Specification and Representation of Programs in a Data Flow Model of Computation*, University Microfilms International, Ann Arbor, MI, 1985.

[HANK81] Hankin, C. L., and Glaser, H. W., "The Dataflow Programming Language CAJOLE - An Informal Introduction," *SIGPLAN Notices*, Vol. 16, No. 7, July 1981, pp. 35-44

[HOAR69] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *CACM*, Vol. 12, No. 10, October 1969, pp.576-580.

[HOAR73] Hoare, C. A. R., "An Axiomatic Definition of the Programming Language Pascal," *Acta Informatica* 2, (1973) pp.335-355.

[KARP69] Karp, R. M., and Miller, R. E., "Parallel Program Schemata," *Journal Computer Systems Sciences*, Vol. 3, No. 2, May 1969, pp.147-195.

[KOSI73] Kosinski, P. R., *A Dataflow Programming Language*, IBM T. J. Watson Research Center, Report RC 4264 (#19076), March 1973.

[KOS73b] Kosinski, P. R., "A Dataflow Language for Operating Systems Programming", *ACM SIGPLAN Notices*, Vol. 8, September 1973, pp. 89-94.

[LAND78] Landry, S. P. and Shriver, B. D., *A User's Guide to the Data Flow Simulator*, Computer Science Department, University of Southwestern Louisiana, Lafayette, LA, 1978.

[LAND81] Landry, S. P., *System Oriented Extensions to Dataflow*, PhD Dissertation, Computer Science Department, University of Southwestern Louisiana, Lafayette, LA, November 1978.

[LOND78] London, R. L., et. al., "Proff Rules for the Programming Language Euclid," *Acta Informatica* 10, (1978) pp. 1-26.

[MALL82] Mallgren, W. R., *Formal Specification of Interactive Graphics Programming Languages*, The MIT Press, Cambridge/London, 1982.

[MAL82b] Mallgren, W. R., "Formal Specification of Graphic Data Types," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, October 1982, pp. 687-710.

[MATW85] Matwin, S. and Pietrzykowski, T., "PROGRAPH: A Preliminary Report," *Computer Languages*, Vol. 10, No. 2, (1985), pp. 91-126.

[MCGR82] McGraw, J. R., "The VAL Language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, January 1982, pp. 44-82.

[MCGR83] McGraw, J. R., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauret, J., Dobes, I., and Hohensee, P., *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual*, Version 1.1, M-146, Lawrence Livermore National Laboratory, Livermore, CA, July 1983.

[NASS73] Nassi, I. and Shneiderman, B., "Flowchart Techniques for Structured Programming," *SIGPLAN Notices*, Vol. 8, No. 8, August 1973, pp. 12-26.

[PAGA81] Pagan, F. G., *Formal Specification of Programming Languages: A Panoramic Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[PAGA87] Pagan, F. G., "A Graphical FP Language," *ACM SIGPLAN Notices*, Vol. 22, No. 3, March 1987, pp. 21-39.

[PATN84] Patnaik, L. M., Bhattacharya, P., and Ganesh, R., "DFL: A Data Flow Language," *Computer Languages*, Vol. 9, No. 2, February 1984, pp. 97-106.

[RAED85] Raeder, G., "A Survey of Current Graphical Programming Techniques," *Computer*, Vol. 18, No. 8, August 1985, pp. 11-24.

[RODR69] Rodriguez, J. D., *A Graph Model for Parallel Computations*, Technical Report TR-64, Project MAC, MIT, Cambridge, MA, 1969.

[RUMB77] Rumbaugh, J., "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, Vol. C-26, No. 2, February 1977, pp. 138-146.

[SHAR85] Sharp, J. A., *Data Flow Computing*, Ellis Horwood Limited, New York, 1985.

[SHRI80] Shriver, B. D., Landry, S. P., and Srini, V. P., "Dataflow," *Encyclopedia of Computer Science and Engineering*, (A. Ralston and E. Reilly, editors), Van Nostrand Reinhold Company, pp.471-474.

[SRIN86] Srini, V. P., "An Architectural Comparison of Dataflow Systems," *Computer*, Vol. 19, No. 3, March 1986, pp. 68-88.

[TREL82] Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P., "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys*, Vol. 14, No. 1, March 1982, pp. 93-143.

[WADG85] Wadge, W. W. and Ashcroft, E. A., *Lucid, the Dataflow Programming Language*, Academic Press, New York/London, 1985.

APPENDIX A

PROGRAM EXAMPLES

This program is used to perform mergesort on a stream of integer values. Since streams are used, an array is not needed to hold the elements to be sorted. The main function of the program is *msort*. Function *msort* calls function *merge* to merge the elements of two streams. The number of elements to be sorted is arbitrary.

```
function msort(a:int; returns int)
 if =(first(next(a)),eod) then a
 else
  begin
    b:bool := fby(false,not(b));
    t1:int := whenever(a,b);
    t2:int := whenever(a,not(b));
    return(merge(msort(t1),msort(t2)));
  end;
end


 function merge(x,y:int; returns int)
  if takexx then return(xx) else return(yy) end;
   where
    xx:int       := upon(x,takexx);
    yy:int       := upon(y,not(takexx));
    takexx:bool := if =(yy,eod) then true
                        else
                          if =(xx,eod) then false
                            else
                              if <(x,y) then true
                                 else false;
  end;
end
```

This function is used to perform matrix multiplication on matrices of arbitrary size. (NOTE: the number of rows in the matrix must be equal to the number of elements in each row.) The function uses three *forall* loops to perform the multiplication. An array of arrays is used to implement the matrices.

```
function matrix_mult(A,B:array[array[int]], n:int; returns array[array[int]])

 forall i in [1,n]
       construct(forall j in [1,n]
                construct( forall k in [1,n]
                          eval plus *(A[i][k],B[k][j]);
                          end;)
                end;)
      end;
 end
```

This function is used to compute the factorial of *n*.

```
function fact(n:int; returns int)
  begin
     b:bool1 := if +(y,1) then true
                else false
     x:int  := fby(1,*(x,y));
     y:int  := fby(n,-(y,1));
     return(fby(upon(x,bool1),eod))
  end;
end
```

APPENDIX B

TEXTUAL SPECIFICATION OF PDL

In the following BNF grammar, $\{$text$\}$ means that zero or more occurrences of the enclosed text are allowed. $[$text$]$ means that zero or one occurrences of the enclosed text is allowed.

Type Specifications

type-spec	::=	basic-type-spec
	|	compound-type-spec
	|	type-name
basic-type-spec	::=	boolean
	|	character
	|	integer
	|	null
	|	real
	|	function
compound-type-spec	::=	**array**[type-spec]
	|	**record**[field-spec $\{$;field-spec$\}$]
	|	**union**[tag-spec $\{$;tag-spec$\}$]
field-spec	::=	field-name $\{$,field-name$\}$:type-spec
tag-spec	::=	tag-name $\{$,tag-name$\}$:type-spec
field-name	::=	name
tag-name	::=	name
type-name	::=	name

Type definitions

type-def-part ::= {type–def;} | {type–def;} type-def

type-def ::= **type** type-name = type-spec

type-name ::= name

Constants

constant ::= nil
| false
| true
| integer-number
| real-number
| character-string
| **error**[type-spec]
| **empty**[type-spec]

Expressions

expression ::= simple-expression
| relational-op(expression,simple-expression)

simple-expression ::= term
| adding-op(simple-expression,term)

term ::= factor
| multiplying-op(term,factor)

factor ::= primary
| unary-op(primary)

```
primary           ::=  constant
                     |  value-name
                     |  invocation
                     |  array-ref
                     |  array-generator
                     |  record-ref
                     |  record-generator
                     |  union-test
                     |  union-generator
                     |  error-test
                     |  prefix-operation
                     |  (multi-exp)

unary-op          ::=  + | - | !

relational-op     ::=  < | <= | > |
                     |  >= | = | !=

adding-op         ::=  + | - | || | cat

multiplying-op    ::=  * | / | mod | &&

value-name        ::=  name

invocation        ::=  function-name(multi-exp)

function-name     ::=  name

array-ref         ::=  primary(multi-exp)

array-generator   ::=  [ [multi-exp: ] multi-exp {;[multi-exp:]multi-exp} ]

                  |    primary [ [multi-exp: ] multi-exp {;[multi-exp:]multi-exp} ]
```

record-ref ::= primary.field

record-generator ::= **record**[field-def {;field–def}]

| **replace** primary [field:multi-exp {;field:multi–exp}]

field-def ::= field-name:multi-exp

field ::= field-name{.field–name}

field-name ::= name

union-test ::= **is** tag-name(multi-exp)

union-generator ::= **make** type-spec [tag-name:multi-exp]

tag-name ::= name

error-test ::= **is-error**(multi-exp)

prefix-operation ::= **char**(multi-exp)
| **int**(multi-exp)
| **int_c**(multi-exp)
| **floor**(multi-exp)
| **trunc**(multi-exp)
| **float**(multi-exp)
| **abs**(multi-exp)
| **exp**(multi-exp)
| **max**(multi-exp)
| **min**(multi-exp)
| **array**(multi-exp)
| **array_low**(multi-exp)
| **array_high**(multi-exp)
| **array_size**(multi-exp)
| **array_cat**(multi-exp)
| **set_bounds**(multi-exp)
| **first**(multi-exp)
| **rest**(multi-exp)
| **advance_upon**(multi-exp)
| **asa**(multi-exp)
| **whenever**(multi-exp)
| **concatenate**(multi-exp)
| **fby**(multi-exp)
| **is_current**(multi-exp)
| **return**(multi-exp)

basic-multi-exp ::= expression
| basic-multi-exp,basic-multi-exp
| invocation
| (multi-exp)

invocation ::= function-name(multi-exp)

multi-exp ::= basic-multi-exp
| conditional-exp
| begin-exp
| case-exp
| iteration-exp
| forall-exp
| apply-exp

Program Structures

conditional-exp ::= **if** multi-exp **then** multi-exp
{**elseif** multi-exp **then** multi-exp}
else multi-exp
end

begin-exp ::= **begin** decldef-part
return multi-exp
end

decldef-part ::= {decldef;}
| {decldef;} decldef

decldef ::= decl
| def
| decl {,decl} := multi-exp

decl ::= value-name {,value-name} :type-spec

def ::= value-name {,value-name} := multi-exp

```
case-exp          ::=   case test-expression
                          {tag-list:multi-exp}
                          [default:multi-exp]
                        end

test-expression   ::=   value-name := multi-exp

tag-list          ::=   tag tag-name{,tag-name}

tag-name          ::=   name

iteration-exp     ::=   for decldef-part do
                          if multi-exp then
                              iter-end
                            else iter-end
                        end

iter-end          ::=   if expression then iter-end
                          {elseif expression then iter-end}
                          else iter-end
                        end

                  |     case test-expression
                          {tag-list:iter-end}
                          [default:iter-end]
                        end

                  |     begin
                          decldef-part
                          return iter-end
                        end

                  |     multi-exp

                  |     iter {def;}

                  |     iter {def;} def
```

forall-exp ::= **forall** value-name **in** [multi-exp]
decldef-part
construct expression | **eval** forall-op multi-exp
end

forall-op ::= **plus** | **times** | **min** | **max** | **or** | **and**

apply-exp ::= **apply**(function-def,apply-op)

function-def ::= function-module

apply-op ::= < primary {,primary} >

Function Modules

function-module ::= **function** function-name({decl;} decl [;]
returns type-spec {,type–spec})
type-def-part
multi-exp
end

APPENDIX C

GRAPHICAL SPECIFICATION OF PDL

Tree-Structured Node (tnode,tn)

•	nullnode	name ⇒ tnode
•	moveto	tnode X point ⇒ tnode
•	put_node	tnode X type X name ⇒ tnode
•	arc_to	tnode X name.int X name.int X name ⇒ tnode
•	text	tnode X string ⇒ tnode
	curpos	tnode ⇒ point
	replace_node	tnode X name X name ⇒ tnode
	remove_node	tnode X name ⇒ tnode
	remove_arc	tnode X name ⇒ tnode
	expand	tnode X name ⇒ tnode
*	display	tnode ⇒ picture

axioms

curpos

tn1	curpos(nullnode) = origin
tn2	curpos(moveto(P,x)) = x

replace_node

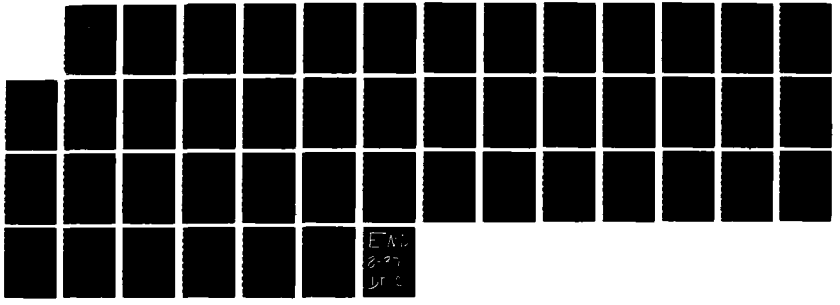
tn3	replace_node(nullnode,
tn4	replace_node(moveto(P,
tn5	replace_node(put_node(P,
tn6	replace_node(arc_to(P,

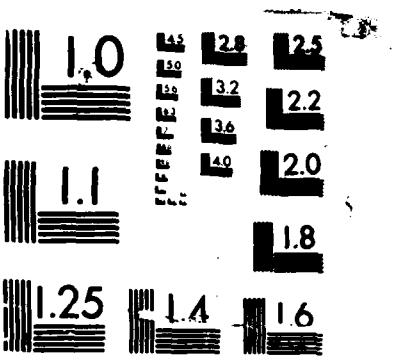
remove_node

tn7	
tn8	
tn9	
tn10	
tn11	

AD-A181 950 THE DESIGN AND SPECIFICATION OF PDL: THE PROTOTYPE DATAFLOW LANGUAGE(U) ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA D J WOLFE MAY 87 2/2

UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART

remove_arc

tn12 remove_arc(nullnode,n) = nullnode

tn13 remove_arc(moveto(P,x),n) = remove_arc(P,n)

tn14 remove_arc(put_node(P,t,n_1),n) = remove_arc(P,n)

tn15 remove_arc(arc_to(P,n_1.i,n_2.j,n_3),n) = **if** (n = n_3) **then** P
else remove_arc(P,n)

tn16 remove_arc(text(P,s),n) = remove_arc(P,n)

expand

tn22 expand(P,nullnode) = nullnode

tn23 expand(moveto(P,x),n) = expand(P,n)

tn24 expand(put_node(P,t,n_1),n) = **if** (n = n_1) $\wedge$ (t = udf)**then**
draw_subgraph(n)
else expand(P,n)

tn25 expand(arc_to(P,n_1.i,n_2.j,n_3),n) = expand(P,n)

tn26 expand(text(P,s),n) = expand(P,n)

display

tn27 display(nullnode) = p.nullnode

tn28 display(moveto(P,x)) = display(P)

tn29 display(put_node(P,t,n)) = p.sum(display(P),p.node(curpos(P),t))

tn30 display(arc_to(P,n_1.i,n_2.j,n_3)) =
p.sum(display(P),p.arc(pos(n_1.i),pos(n_2.j)))

tn31 display(text(P,s)) = p.sum(display(P),p.text(curpos(P),s))

Continuous Picture (picture,p)

* •	nullpict		⇒	picture
* •	inspatch	picture X region	⇒	picture
*	domain	picture	⇒	region
*	sum	picture X picture	⇒	picture
*	addpatch	picture X region	⇒	picture
*	restriction	picture X region	⇒	picture

axioms

p1	domain(nullpict) = nullregion
p2	domain(inspatch(P,R)) = R ∪ domain(P)
p3	sum(P,nullpict) = P
p4	sum(P_1,inspatch(P_2,R)) = sum(adddpatch(P_1,R),restriction(P_2,∼R))
p5	addpatch(nullpict,R) = inspatch(nullpict,R)
p6	addpatch(inspatch(P,R_1),R_2) = inspatch(inspatch(addpatch(P,R_2–R_1),R_1–R_2),R_1 ∩ R_2)
p7	restriction(nullpict,R) = nullpict
p8	restriction(inspatch(P,R_1),R_2) = inspatch(restriction(P,R_2),R_1 ∩ R_2)
p9	inspatch(P,nullregion) = P
p10	inspatch(inspatch(P,R_1),R_2) = inspatch(P,R_1 ∪ R_2)

User Interaction (ip)

program operations

prompt	string	⇒	
getkey		⇒	char
getposn		⇒	point
getpick		⇒	name
post	tnode X name	⇒	
unpost	name	⇒	

user actions

getprompt		⇒	string
keystroke	char	⇒	
position	point	⇒	
pick	point	⇒	boolean

auxiliary functions

*	postpict	state	⇒	tnode
*	pickname	state X point	⇒	name
*	image	state	⇒	picture

axioms

getkey

ip1 wait($init) = true

ip2 wait(getkey$return(S)) = true

ip3 wait(keystroke$return(S,c)) = false

ip4 value($init) = undefined

ip5 value(keystroke$return(S,c)) = c

getposn

ip6 wait($init) = true

ip7 wait(getposn$return(S)) = true

ip8 wait(position$return(S,p)) = false

ip9 value($init) = undefined

ip10 value(position$return(S,p)) = p

getpick

ip11 wait($init) = true

ip12 wait(getpick$return(S)) = true

ip13 wait(pick$return(S,p)) = ¬p.visible(image(S),rg.pickregion(p))

ip14 value($init) = undefined

ip15 value(pick$return(S,p)) = pickname(S,p)

getprompt

ip16 wait($init) = true

ip17 wait(prompt$return(S,s)) = false

ip18 wait(getprompt$return(S)) = true

ip19 value($init) = nullstring

ip20 value(prompt$return(S,s)) = s

pick

ip21 value(S,p) = ¬p.visible(image(S),rg.pickregion(p))

ip22 postpict($init) = tp.nullpict

ip23 pickname(S,p) = **if** s.empty(tp.picknames(postpict(S),rg.pickregion(p))
then n.null
else s.first(tp.picknames(postpict(S),rg.pickregion(p)))

ip24 image(S) = tp.display(postpict(S))

Region (region,rg)

	null	region	$\Rightarrow$	region
	universe		$\Rightarrow$	region
	lineseg	point X point	$\Rightarrow$	region
	box	point X point	$\Rightarrow$	region
	complement	region	$\Rightarrow$	region
	intersect	region X region	$\Rightarrow$	region
	union	region X region	$\Rightarrow$	region
	difference	region X region	$\Rightarrow$	region
	empty	region	$\Rightarrow$	boolean
	contains	region X region	$\Rightarrow$	boolean
	contains	ptregion X point	$\Rightarrow$	boolean
	equal	region X region	$\Rightarrow$	boolean
*	pickregion	point	$\Rightarrow$	region

axioms

object mapping

rg1	nullregion $\approx \{\ \}$
rg2	universe $\approx \{q \mid q \text{ is in the universe}\}$
rg3	lineseg$(q_1,q_2) \approx \{(x,y) \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2,$ $(y_2-y_1)*(x_2-x_1) \times y = x_1 \times y_2 - x_2 \times y_1\}$
rg4	box$(q_1,q_2) \approx \{(x,y) \mid x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$
rg5	complement$(R) \approx \sim (R')$
rg6	intersect$(R_1,R_2) \approx R_1' \cap R_2'$
rg7	union$(R_1,R_2) \approx R_1' \cup R_2'$
rg8	difference$(R_1,R_2) \approx R_1' - R_2'$
rg9	empty$(R) = (R' = \{\ \})$
rg10	contains$(R_1,R_2) = R_2' \subseteq R_1'$
rg11	containspt$(R,q) = q \in R'$
rg12	equal$(R_1,R_2) = (R_1' = R_2')$
rg13	pickregion$(q) \approx \{q\}$

Point (point,pt)

•	point	real X real	⇒	point
	polarpt	real X real	⇒	point
	origin		⇒	point
	absissa	point	⇒	real
	ordinate	point	⇒	real
	radius	point	⇒	real
	angle	point	⇒	real
	sum	point X point	⇒	point
	difference	point X point	⇒	point

axioms

pt1 polarpt(x_1,x_2) = point(x_1*cos(x_2),x_1×sin(x_2))

pt2 origin = point(0,0)

pt3 abscissa(point(x_1,x_2)) = x_1

pt4 ordinate(point(x_1,x_2)) = x_2

pt5 radius(point(x_1,x_2)) = $(x_1^2 + x_2^2)^{1/2}$

pt6 angle(p) = **if** radius(p) = 0 **then** 0
elseif ordinate(p)≥0
then arccos(abscissa(p) ÷ radius(p))
else 360 − arccos(abscissa(p)÷radius(p))

pt7 sum(point(x_1,x_2),point(x_3,x_4)) = point(x_1+x_3,x_2+x_4)

pt8 difference(point(x_1,x_2),point(x_3,x_4)) = point(x_1−x_3,x_2−x_4)

Name(name,n)

	$name_1$		⇒	name
	. . .			
	$name_m$		⇒	name
	equals	name X name	⇒	boolean
*	lessthan	name X name	⇒	boolean

String (string,st)

•	nullstr		⇒	string
•	inschar	string X char	⇒	string
	concat	string X string	⇒	string

axioms

st1	concat(s,nullstr) = s
st2	concat(s_1,inschar(s_2,a)) = inschar(concat(s_1,s_2),a)

theorems

st1T	concat(nullstr,s) = s

APPENDIX D

SEMANTIC SPECIFICATION OF PDL

With each of the axioms given below, it is assumed that the inputs are not the error value and that they are valid. Two global axioms are given to specify the behavior of all operations when an error value or an invalid input is present on one of the operation's input arcs. The two axioms (error and valid) are given first.

All operations act on streams of data values. The manipulation of the BOS and EOS tokens is not shown in the specifications. It is assumed that the programming environment would manipulate these control tokens. The specification of the special filters utilizes a different notation than the other operations to refer to each element in the stream. I.i refers to the ith element in stream I. Finally, the actions that operations perform are machine and implementation dependent (i.e., the addition of integers and real numbers is different on different computers).

error: The error axiom states that if any of an operation's input arcs carry a token with the error value, the operation will produce an error value with an appropriate type (the type the operation ordinarily produces).

$$<I_1:VT,...,I_n:VT;E_1,...,E_m> \wedge I_i = \text{error}(VT_i),(1 \leq i \leq n)\{OPNAME(I_1,...,I_n)\}<E,...,E;\text{error}[VT]:VT,...,\text{error}[VT]:VT>$$

valid: The valid axiom states that if any of an operation's input arcs carry an invalid token (i.e., the type of the token is not the type expected by the operation), the operation will produce an error value with an appropriate type (the type the operation ordinarily produces).

$$<I_1:VT,...,I_n:VT;E_1,...,E_m> \wedge \text{invalid}(I_i),(1 \leq i \leq n)\{OPNAME(I_1,...,I_n)\}<E,...,E;\text{error}[VT]:VT,...,\text{error}[VT]:VT>$$

add: This operation accepts and consumes 2 input tokens of type integer or real and produces a single output token of type integer or real which is the sum of the input tokens. The subtract, multiply, and divide operations are similar to the add operation in their specification (except that a different operation is performed on the input values).

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_add), (real,real_add)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1+I_2:VT>$$

subtract:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_subtract), (real,real_subtract)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1-I_2:VT>$$

multiply:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_multiply), (real,real_multiply)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1\times I_2:VT>$$

divide:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_divide), (real,real_divide)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1\div I_2:VT>$$

exponentiation:

This operation accepts and consumes 2 input tokens of type integer or real and produces a single output token of type integer or real which is the result of raising the first token to the power of the second token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_exp), (real,real_exp)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1^{I_2}:VT>$$

modulus:

This operation accepts and consumes 2 input tokens of type integer and produces a single output token of type integer which is the remainder of an integer division of the inputs. The first input is the dividend of the division and the second is the divisor.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_modulus)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1 \bmod I_2:VT>$$

negation:

This operation accepts and consumes one input token of type integer or real and produces a single output token of type integer or real which is the result of the unary negation of the input token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_negation), (real,real_negation)
IN

$$<I_1:VT;E>\{OPNAME(I_1)\}<E;0 - I_1:VT>$$

absolute_value:

This operation accepts and consumes one input token of type integer or real and produces a single output token of type integer or real which is the absolute value of the input token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_absolute), (real,real_absolute)
IN

$$<I_1:VT;E>\{OPNAME(I_1)\}<E;\|I_1\|:VT>$$

maximum:

This operation accepts and consumes two input tokens of type integer or real and produces a single output token of type integer or real which is the largest of the two input tokens. The minimum is similar to the maximum operation except that it produces the smallest of the two inputs.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_max), (real,real_max)
IN

$$(I_i \geq I_j, 1 \leq i,j \leq 2) \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; I_i{:}VT \rangle$$

minimum:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_min), (real,real_min)
IN

$$(I_i \leq I_j, 1 \leq i,j \leq 2) \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; I_i{:}VT \rangle$$

equal: This operation accepts and consumes two input tokens and produces a single output token. The input tokens are either of type integer, real, character, or boolean. Both input tokens must be of the same type. The output token is of type boolean. The value of the output token is True if the input tokens are equal, False otherwise. The specification of the not_equal, less_than, greater_than, greater_than_or_equal, and less_than_or_equal are similar to the equal operation except that different comparisons are made.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_equal), (real,real_equal),
(bool,bool_equal), (char,char_equal)
IN

$$(I_1 = I_2) \Longleftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool \rangle$$

not_equal:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_not_equal), (real,real_not_equal),
(bool,bool_not_equal), (char,char_not_equal)
IN

$$(I_1 \neq I_2) \Longleftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool \rangle$$

greater_than:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_greater_than), (real,real_greater_than),
(bool,greater_than), (char,char_greater_than)
IN

$(I_1 > I_2) \Leftrightarrow P \wedge <I_1:VT, I_2:VT; E> \{OPNAME(I_1, I_2)\} <E, E; P:Bool>$

less_than:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_less_than), (real,real_less_than),
(bool,bool_less_than), (char,char_less_than)
IN

$(I_1 < I_2) \Leftrightarrow P \wedge <I_1:VT, I_2:VT; E> \{OPNAME(I_1, I_2)\} <E, E; P:Bool>$

greater_than_or_equal:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_greater_equal), (real,real_greater_equal),
(bool,bool_greater_equal), (char,char_greater_equal)
IN

$(I_1 \geq I_2) \Leftrightarrow P \wedge <I_1:VT, I_2:VT; E> \{OPNAME(I_1, I_2)\} <E, E; P:Bool>$

less_than_or_equal:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_less_equal), (real,real_less_equal),
(bool,bool_not_equal), (char,char_not_equal)
IN

$(I_1 \leq I_2) \Leftrightarrow P \wedge <I_1:VT, I_2:VT; E> \{OPNAME(I_1, I_2)\} <E, E; P:Bool>$

and: This operation accepts and consumes two input tokens of type boolean and produces a single output token of type boolean. The value of the output token is the logical AND of the two input tokens. The specification of the *or* operation is similar to the *and* operation except that the logical OR of the two inputs is the result.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (bool,bool_and)
IN

$$(I_1 \wedge I_2) \Longleftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool \rangle$$

or:

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (bool,bool_or)
IN

$$(I_1 \mid I_2) \Longleftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E \rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool \rangle$$

not: This operation accepts and consumes a single input token of type boolean and produces a single output token of type boolean. The value of the output token is the logical negation of the input token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (bool,bool_not)
IN

$$\langle I_1{:}VT; E \rangle \{OPNAME(I_1)\} \langle E; \tilde{} I_1{:}VT \rangle$$

array_create:
This operation accepts and consumes a single input token which is the type specification for the array to be created and produces an empty array of the indicated type.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,int,array_create)
IN

$$\langle I_1{:}VT_1{:}E \rangle \{OPNAME(I_1)\} \langle E; empty[VT_1]{:}VT_2 \rightarrow VT_1 \rangle$$

array_select:

This operation accepts and consumes two input tokens. The first input token is of type array and the second is of type integer. Input token two serves as the index of input token one. The operation selects the element of the array at the index and produces that value.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_select)
IN

$$<I_1:VT_1 \rightarrow VT_2, I_2:VT_1; E> \{OPNAME(I_1,I_2)\} <E,E;I_1(I_2):VT_2>$$

array_replace:

This operation accepts and consumes three input tokens. The first input token is of type array, the second is of type integer, and the third has the same type as the elements of the array. Input token two serves as the index of input token one. Input token two is the value which is to replace the value at the index of the array. The operation produces an array with identical elements to input token one except that the element at the index is replaced by the token on input arc three. The notation M[x/v] reads "M with x for v."

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any_type,array_replace)
IN

$$<I_1:VT_1 \rightarrow VT_2, I_2:VT_1, I_3:VT_2; E> \{OPNAME(I_1,I_2,I_3)\} <E,E,E;I_1[I_3/I_1(I_2)]:VT_1 \rightarrow VT_2>$$

array_size:

This operation accepts and consumes one input token which is of type array and produces a single output token of type integer which is the size of the array.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_size)
IN

$$<I_1:VT_1 \rightarrow VT_2; E> \{OPNAME(I_1)\} <E;O_1:VT_1>$$

array_low:

This operation accepts and consumes one input token which is of type array and produces a single output token of type integer which is the lower bound of the array.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_low)
IN

$$<I_1:VT_1 \rightarrow VT_2;E> \{OPNAME(I_1)\} <E;O_1:VT_1>$$

array_high:

This operation accepts and consumes one input token which is of type array and produces a single output token of type integer which is the upper bound of the array.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_high)
IN

$$<I_1:VT_1 \rightarrow VT_2;E> \{OPNAME(I_1)\} <E;O_1:VT_1>$$

array_set_bounds:

This operation accepts and consumes three input tokens. The first input token is of type array, the second and third are type integer. The operation produces an array which identical to the first input token except that the high index has the value of input token two and the low index has the value of input token three.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_set_bounds)
IN

$$<I_1:VT_1 \rightarrow VT_2,I_2:VT_1,I_3:VT_1;E> \{OPNAME(I_1,I_2,I_3)\} <E,E,E;O_1:VT_1 \rightarrow VT_2> \wedge \text{lower}(O_1)=I_2 \wedge \text{upper}(O_1)=I_3$$

array_concatenate:

This operation accepts and consumes two input tokens of type array and produces a single output token of type array. The array produces has a size equivalent to the sum of the sizes of the input arrays, a low index equal to input token one's index, and a high index equal to input token two's high index. Input token two is concatenated with input token one to form the new array.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_concatenate)
IN

$$<I_1:VT_1 \rightarrow VT_2,I_2:VT_1 \rightarrow VT_2;E> \{OPNAME(I_1,I_2)\} <E,E;O_1:VT_1 \rightarrow VT_2>$$

record_create:

This operation accepts and consumes one input token which is the type specification for the record to be created and produces an record of the indicated type.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,any,record_create)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;Record[VT_1]:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2>$$

record_select:

This operation accepts and consumes two input tokens. The first input token is of type record and the second is a field of the record. The operation produces the value of the named field.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (field,any,record_select)
IN

$$FN_i=I_2 \wedge <I_1:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2,I_2:VT_1;E>\{OPNAME(I_1,I_2)\}<E,E;I_1.I_2:VT_2>$$

record_replace:

This operation accepts and consumes three input tokens. The first input token is of type record, the second is a field of the record, and the third is a value which is to replace the current value of the named field. The operation produces a record similar to the record on input arc one except that the value of the named field is replaced by the token on input arc three.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (field,any,record_replace)
IN

$$FN_i=I_2,(1\leq i\leq n) \wedge <I_1:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2,I_2:VT_1,I_3:VT_2;E>$$
$$\{OPNAME(I_1,I_2,I_3)\}<E,E,E;I_1|I_2:I_3|:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2> \wedge (1\leq i\leq n)$$

union_create:

This operation accepts and consumes one input token which is the type specification for the union to be created and produces a union of the indicated type.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,any,union_create)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:T_i \rightarrow VT_2> \wedge (1\leq i\leq n)$$

union_tag_test:
This operation accepts and consumes two input tokens and produces a single output token of type boolean. The result of this operation is true if the union on input arc one was created with a tag of input arc two. Otherwise, a value of False is produced.

FOR_TUPLE (VT_1,VT_2,VT_3,OPNAME)
BOUND_TO_TUPLES (tag,any,boolean,union_tag_test)
IN

$$(\text{tagname}(I_1,I_3) \Longleftrightarrow P) \wedge <I_1:T_1 \rightarrow VT_2,I_2:VT_1;E>\{\text{OPNAME}(I_1,I_2)\}<E,E;P:VT_3>$$

real_to_int:
This operation accepts and consumes a single input token of type real and produces a single output token of type integer. The input token is converted to an integer using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (real,int,real_to_int)
IN

$$<I_1:VT_1;E>\{\text{OPNAME}(I_1)\}<E;O_1:VT_2>$$

char_to_int:
This operation accepts and consumes a single input token of type character and produces a single output token of type integer. The input token is converted to an integer using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (char,int,char_to_int)
IN

$$<I_1:VT_1;E>\{\text{OPNAME}(I_1)\}<E;O_1:VT_2>$$

float: This operation accepts and consumes a single input token of type integer and produces a single output token of type real. The input token is converted to a real number using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,real,float)
IN

$$<I_1:VT_1;E>\{\text{OPNAME}(I_1)\}<E;O_1:VT_2>$$

int_to_char:

This operation accepts and consumes a single input token of type integer and produces a single output token of type character. The input token is converted to a character using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,char,int_to_char)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$$

floor: This operation accepts and consumes a single input token of type real and produces a single output token of type integer. The input token is converted to an integer using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (real,int,floor)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$$

trunc: This operation accepts and consumes a single input token of type real and produces a single output token of type integer. The input token is converted to an integer using ***machine dependent*** conversion rules.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (real,int,trunc)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$$

first: This operation accepts and consumes a stream of input tokens and produces a single stream of output tokens each of which have the value of the first token in the input stream. The notation $I_{\#}.i$ refers to the ith element of the stream on input arc #. For example, $I_1.1$ refers to the first element of input stream number one.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,first)
IN

$$<I_1.i:VT;E>\{OPNAME(I_1)\}<E;I_1.1:VT>$$

rest: This operation accepts and consumes a stream of input tokens and produces a single stream of output tokens. At each step of the execution, the output token produced is the next token to arrive on the input arc (i.e., the next token in the input stream). The stream produced by the filter is the input stream less the first token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,rest)
IN

$$<I_1.i:VT;E>\{OPNAME(I_1)\}<E;I_1.i+1:VT>$$

whenever:

This operation accepts and consumes two streams of input tokens and produces a single stream of output tokens. For each token set, if the value of the token on the second input arc is True, then the token on the first input arc (data token) is placed on the output arc. Otherwise, the filter discards the data token. The filter produces a stream of all values of the data stream whose corresponding control stream value was true. Two axioms are used to define the semantics of this operation.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,boolean,whenever)
IN

$$<I_1:VT_1,True:VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E.E;I_1.iVT_1>,$$

$$<I_1:VT_1,False:VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E,E;E>$$

advance_upon:

This operation accepts and consumes two streams of input tokens and produces a single stream of output tokens. This filter places the first token in input stream one on the output arc. Then for each successive token set, if the value of the token on the second input arc is True, then the token on the first input arc (data token) is placed on the output arc. If the value is False, the value that was last placed on the output arc is placed on the output arc again and the token on input arc one is left on the arc. The filter produces a stream of values of the data stream, some of which are repeated. Three axioms are used to define the semantics of the operation.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,boolean,advance_upon)
IN

$$<I_1.1:VT_1,I_2.1:VT_2;E>\{OPNAME(I_1.1,I_2.1)\}<E,E;I_1.1:VT_1>,$$

$$<I_1:VT_1,True:VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E,E;I_1.i:VT_1>,$$

$$<I_1:VT_1,False:VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<I_1.i,E;O_1.i-1>$$

concatenate:

This operation accepts and consumes two streams of input tokens and produces a single stream of output tokens. The resultant stream is formed by concatenating the second input stream to the first input stream.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,concatenate)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;O_1:VT>$$

is_current:

The is_current accepts and consumes a single stream of input tokens and produces a single stream of output tokens. The declaration is used to *freeze* certain the value of an identifier. The declaration allows programmers to write programs with nested iteration.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,is_current)
IN

$$<I_1.i:VT;E>\{OPNAME(I_1.i)\}<E;I_1.i:VT>$$

merge: This operation has three input arcs and a single output arc. Input arc one serves as the control input for the operation. Input arc one carries tokens of type boolean and input arcs two and three carry tokens of any type. The operations consumes the input tokens on the control arc and the input data arc pointed to by the control token. A copy of the data token on the specified input data arc is is placed on the output arc. Two axioms are used to define his operation.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (bool,any,merge)
IN

$$<\text{True}:VT_1,I_2:VT_2,E;E>\{\text{OPNAME}(I_1,I_2,I_3)\}<E,E,E;I_2:VT_2>,$$

$$<\text{False}:VT_1,E,I_3:VT_3;E>\{\text{OPNAME}(I_1,I_2,I_3)\}<E,E,E;I_3:VT_2>$$

outbound_switch:
This operation has two input arcs and two output arcs. Input arc one serves as the control input for the operation and input arc two serves as the data input. Input arc one carries tokens of type boolean and input arc two carries tokens of any type. The operations consumes the input tokens and the value of the control token is used to select the output arc on which a copy of the data token will be placed. If the value of the control token is True, then the data token is placed on output arc one. Otherwise, the data token is placed on output arc two. Two axioms are used to define this operation.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (bool,any,outbound_switch)
IN

$$<\text{True}:VT_1,I_2:VT_2;E,E>\{\text{OPNAME}(I_1,I_2)\}<E,E;I_2:VT_2,E>,$$

$$<\text{False}:VT_1,I_2:VT_2;E,E>\{\text{OPNAME}(I_1,I_2)\}<E,E;E,I_2:VT_2>$$

identity:
This operation accepts and consumes one input token of any type and produces an output token of any type which has the value of the input token. Thus, a copy of the input token is placed on the output arc.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,identity)
IN

$$<I_1:VT;E>\{\text{OPNAME}(I_1)\}<E;I_1:VT>$$

route_tokens:

This operation accepts and consumes two input tokens. The token on input arc one is the control token and is of type integer. The token on input arc two is of any type. The operation places the data token on the output arc specified by the control token.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,int,route_tokens)
IN

$$(1 \leq I_2 \leq n) \wedge \langle I_1{:}VT_1, I_2{:}VT_2 E_1, \cdots, E_n \rangle \{OPNAME(I_1, I_2)\} \langle E, E; E_1, \cdots, E_{I_2-1}, I_1, E_{I_2+1}, \cdots, E_n \rangle$$

funnel_tokens:

This operation accepts and consumes N input tokens. The token on input arc one is the control token and is of type integer. The tokens on input arc two to N are of any type. The operation places the data token on the output arc specified by the control token. The operation consumes the control token and the data token from the input arc specified by the control token. The data token is placed on the output arc.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,int,funnel_tokens)
IN

$$\langle I_1{:}VT_2, \ldots E_{I_1-1}, I_{I_1}{:}VT_1, E_{I_1+1}, \ldots; E \rangle \{OPNAME(I_1, \cdots, I_n)\} \langle E_1 \cdots E_n; I_{I_1}{:}VT_1 \rangle$$

apply: This operation accepts and consumes two input tokens and produces N output tokens. The token on input arc one is a procedure definition which is to be ***applied*** to the argument list on input arc two. The operation is used to make a run-time binding of the procedure definition to the argument list. The operation produces N output tokens which are the results of ***applying*** the procedure definition to the argument list.

FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (function,any,apply)
IN

$$\langle I_1{:}VT_1, I_2{:}VT_2, \ldots I_n{:}VT_2; E_1, \ldots, E_m \rangle \{OPNAME(I_1, \ldots, I_n)\} \langle E, E; O_1{:}VT_2, \ldots, O_m{:}VT_2 \rangle$$

input: This operation is used to receive input from the programming environment. It accepts input from a programmer's terminal and produces a token which has the value that was input.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,input)
IN

$$<;E>\{OPNAME\}<;O_1:VT>$$

output:

This operation is used to output values to the programming environment. The operation accepts and consumes a single token of any type. It then outputs the token to the programming environment.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,output)
IN

$$<VT;>\{OPNAME(I_1)\}<E;>$$

constant:

This operation accepts and consumes two input tokens and produces a single output token. Input token one is used to trigger the firing of the node. Input arc two carries the value of the constant to be produced by the node. The output token has the same type and value as the *data constant* token.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,constant)
IN

$$legal\_constant(I_2) \wedge <I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,I_2:VT;I_2:VT>$$

fork: This operation accepts and consumes a single input token and produces n copies of the input token on N output arcs. Tokens of any type are accepted and any number of output arcs can be specified.

FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,fork)
IN

$$<I_1:VT;E_1...E_n>\{OPNAME(I_1)\}<E;I_1:VT,...,I_1:VT>$$

if-then-else:

The if-then-else construct accepts one control input and n data inputs. The data inputs are sent to either the *then* or the *else* expression depending upon the value of the control input. If the control input is true, then the data inputs are sent to the *then* expression. Otherwise, the data inputs are sent to the *else* expression. Once an expression receives the data inputs, the expression executes and produces m output values which are the values produced by the construct.

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{l}<I_1:\text{bool},\cdots,I_2:VT,\ldots,I_n:VT;E_1,\cdots,E_m> \wedge I_0 = \text{true}\ \{S_1\}<E_1,\cdots,E_n;O_1:VT,\ldots,O_m:VT>,\\ <I_1:\text{bool},\cdots,I_2:VT,\ldots,I_n:VT;E_1,\cdots,E_m> \wedge I_0 = \text{false}\ \{S_2\}<E_1,\cdots,E_n;O_1:VT,\ldots,O_m:VT>,\end{array}}{<I_1:\text{bool},I_2:VT,\ldots,I_n;E_1,\cdots,E_m>\{\textbf{if } I_0 \textbf{ then } S_1 \textbf{ else } S_2\}<E_1,\cdots,E_n;O_1:VT,\cdots,O_m:VT>}$$

begin: The begin construct accepts n input values and produces m output values. The construct uses several sub-expressions (S_i) which receive input values from the input values received by the construct and/or the values produced by other sub-expressions. The values produced by each sub-expression can be used by the other sub-expressions or can be values produced by the construct.

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{l}<A_1:VT,\ldots,A_p:VT;E_1,\ldots,E_m> \wedge A_p \in \{\{I_1,\ldots,I_n\} \cup \{O_k\}\}, 1 \leq n \text{ an ki}\\ \{S_i\}<E_1,\ldots,E_p;O_i:VT> \wedge 1 \leq \leq n\end{array}}{<I_1:VT,\ldots,I_n:VT;E_1,\ldots,E_m>\{\textbf{begin } S_1;S_2;\ldots;S_k \textbf{ end}\}<E_1,\ldots,E_n;O_1:VT,\ldots,O_m> \wedge O_iO_i, 1 \leq i \leq n}$$

case: The case construct accepts n input data values and one value of type union. The union-typed value determines which sub-expression of the construct will be executed (i.e., the expression whose tag matches the tag of the union-typed value). The selected expression receives all input values and the constituent value of the union, executes and produces m output values. These values are the output values produced by the construct.

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{l}<I_1:VT,\ldots I_n:VT;E_1,\ldots,E_m>\{S_{k+1}\}<E_1,\ldots E_n;O_1:VT,\ldots,O_m:VT>,\\ \textbf{Is } T_i\ x \wedge <x \mid T_i:VT,I_1:VT,\ldots I_n:VT;E_1,\ldots,E_m><E_1,\ldots E_n;O_1:VT,\ldots,O_m:VT>\end{array}}{<I_1:VT,I_n:VT;E_1,\ldots,E_m>\{\textbf{case } p:=x:\text{union } \textbf{of } T_1:S_1;\ldots;T_k:S_n;\textbf{default}:S_{k+1}\textbf{end}\}<E_1,\ldots E_n;O_1:VT,\ldots,O_m:VT>}$$

forall_eval:

The forall_eval construct accepts *n* input data values and the lower and upper bounds of the range of the construct. The construct creates upper-lower+1 unique instantiations of the body of the construct (S_i) which are executed concurrently. Each instantiation produces a single value. The construct produces a single value which is the result of applying an operation (op_name) on all the values produced by the unique instantiations of the body.

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{c}<i{:}int,I_1{:}VT,\ldots,I_n{:}VT;E_1>\{S_i\}<E_1,\ldots,E_n;O_i{:}VT> \wedge (L\le i\le U),\\ F \in \{plus,mult,and,or,min,max\}\end{array}}{<In_1{:}VT,\ldots,In_n{:}VT;E>\{\textbf{forall } i \textbf{ in } [L,U] \textbf{ do } S_i \textbf{ eval:}F\}<E_1,\ldots,E_n;F(O_1,F(O_2,\ldots,F(O_{U-L},O_{U-L+1})\ldots))>}$$

forall_construct:

The forall_construct construct acts in the same manner as the *forall_eval* construct except that an array is created. The elements of the array are the values produced by the unique instantiations of the body.

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{<i{:}int,I_1{:}VT,\ldots,I_n{:}VT;E_1>\{S_i\}<E_1,\ldots,E_n;O_i{:}VT> \wedge (L\le i\le U),}{<In_1{:}VT,\ldots,In_n{:}VT;E>\{\textbf{forall } i \textbf{ in } [L,U] \textbf{ do } S_i \textbf{ construct}\}<E_1,\ldots,E_n;O{:}int\rightarrow VT>}$$

for_iter:

The for_iter construct accepts *n* input values which are the initial values of the loop variables of the construct. On each iteration of the loop, an *if-then-else* construct is used to test for loop termination. If the loop is to terminate, an expression (S_1 is evaluated. The results of the expression are the results produced by the construct. If the loop does not terminate, another expression (S_2) is evaluated. In this expression, some updating of the variables of the loop is performed (i.e., the next iteration of the loop is performed).

FOR_EACH(VT)
BOUND_TO_ONE_OF(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{l}<In_1{:}VT_1,\ldots,In_k{:}VT_k;E_1,\ldots,E_j> \wedge (1\le k,j\le n)\{\textbf{initialize}\}<E_1,\cdots,E_k;O_1{:}VT_1,\cdots,O_j{:}VT_j>\\ \neg B \wedge <In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m>\{S_1\}<In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m>\\ B \wedge <In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m> \wedge (1\le m\le n)\{S_2\}<E_1,\cdots,E_n;O_1{:}VT_1,\cdots,O_m{:}VT_m>\end{array}}{<In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\ldots,E_m>\{\textbf{for initialize do if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}\}<E_1,\cdots,E_n;O_1{:}VT_1,\cdots,O_m{:}VT_m>}$$

APPENDIX E

DATA STRUCTURES OF PDL

This appendix gives the data structures used to completely describe all operations and program constructs of PDL. The *names* component gives the names of the language constructs described by the data structure. The *semantics* component gives the semantical description of the language construct. The *graphics* component gives a reference to an executable procedure that draws the icons representing the language construct. The *textual* component gives a reference into the BNF grammar to the grammar rule which describes the language construct. (The actual syntax used within a program may also be given in the *textual* component).

add_node
{
names: int_add, real_add;
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (int,int_add), (real,real_add)
IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E.E;I_1 + I_2:VT>$$

graphics: draw_add_node;
textual: $+(I_1,I_2)$; (simple-expression)
}

```
subtract_node
{
 names: int_subtract, real_subtract;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_subtract), (real,real_subtract)
   IN
                  <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;I1-I2:VT>
 graphics: draw_subtract_node;
 textual:  -(I1,I2);   (simple-expression)
}



multiply_node
{
 names: int_multiply, real_multiply;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_multiply), (real,real_multiply)
   IN
                  <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;I1×I2:VT>
 graphics: draw_multiply_node;
 textual:  *(I1,I2);   (term)
}



divide_node
{
 names: int_divide, real_divide;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_divide), (real,real_divide)
   IN
                  <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;I1÷I2:VT>
 graphics: draw_divide_node;
 textual:  /(I1,I2);   (term)
}
```

exponentiation_node
{
names: int_exp, real_exp;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_exp), (real,real_exp)
  IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1^{I_2}:VT>$$

graphics: draw_exp_node;
textual: exp(I_1,I_2); (prefix-operation)
}

modulus_node
{
names: int_mod;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_modulus)
  IN

$$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_1 \bmod I_2:VT>$$

graphics: draw_mod_node;
textual: mod(I_1,I_2); (term)
}

negation_node
{
names: int_negation, real_negation;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_negation), (real,real_negation)
  IN

$$<I_1:VT;E>\{OPNAME(I_1)\}<E;0 - I_1:VT>$$

graphics: draw_negation_node;
textual: -(I_1); (factor)
}

absolute_value_node
{
names: int_absolute, real_absolute;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_absolute), (real,real_absolute)
  IN

$$<I_1:VT;E>\{OPNAME(I_1)\}<E;I_1:VT>$$

graphics: draw_abs_node;
textual: abs(I_1); (prefix-operation)
}

maximum_node
{
names: int_max, real_max;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_max), (real,real_max)
  IN

$$(I_i \geq I_j, 1 \leq i,j \leq 2) \wedge <I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_i:VT>$$

graphics: draw_max_node;
textual: max(I_1,I_2); (prefix-operation)
}

minimum_node
{
names: int_min, real_min;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_min), (real,real_min)
  IN

$$(I_i \leq I_j, 1 \leq i,j \leq 2) \wedge <I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;I_i:VT>$$

graphics: draw_min_node;
textual: min(I_1,I_2); (prefix-operation)
}

equal_node
{
names: int_equal, real_equal, bool_equal, char_equal;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_equal), (real,real_equal),
                  (bool,bool_equal), (char,char_equal)
  IN

$$(I_1 = I_2) \Leftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E\rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool\rangle$$

graphics: draw_equal_node;
textual: $=(I_1,I_2)$; (expression)
}

not_equal_node
{
names: int_not_equal, real_not_equal, bool_not_equal, char_not_equal;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_not_equal), (real,real_not_equal),
                  (bool,bool_not_equal), (char,char_not_equal)
  IN

$$(I_1 \neq I_2) \Leftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E\rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool\rangle$$

graphics: draw_not_equal_node;
textual: $!=(I_1,I_2)$; (expression)
}

greater_than_node
{
names: int_greater_than, real_greater_than, bool_greater_than, char_greater_than;
semantics:
  FOR_TUPLE (VT,OPNAME)
  BOUND_TO_TUPLES (int,int_greater_than), (real,real_greater_than),
                  (bool,greater_than), (char,char_greater_than)
  IN

$$(I_1 > I_2) \Leftrightarrow P \wedge \langle I_1{:}VT, I_2{:}VT; E\rangle \{OPNAME(I_1, I_2)\} \langle E, E; P{:}Bool\rangle$$

graphics: draw_greater_than_node;
textual: $>(I_1,I_2)$; (expression)
}

```
less_than_node
{
 names: int_less_than, real_less_than, bool_less_than, char_less_than;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_less_than), (real,real_less_than),
                   (bool,bool_less_than), (char,char_less_than)
   IN
          (I1 < I2)⇐⇒P ∧ <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;P:Bool>
 graphics: draw_less_than_node;
 textual:  <(I1,I2);   (expression)
}


greater_than_equal_node
{
 names: int_greater_equal_than, real_greater_equal_than,
          bool_greater_equal_than, char_greater_equal_than;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_greater_equal), (real,real_greater_equal),
                   (bool,bool_greater_equal), (char,char_greater_equal)
   IN
          (I1 ≥ I2)⇐⇒P ∧ <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;P:Bool>
 graphics: draw_greater_than_equal_node;
 textual:  >=(I1,I2);   (expression)
}


less_than_equal_node
{
 names: int_less_equal_than, real_less_equal_than,
           bool_less_equal_than, char_less_equal_than;
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (int,int_less_equal), (real,real_less_equal),
                   (bool,bool_not_equal), (char,char_not_equal)
   IN
          (I1 ≤ I2)⇐⇒P ∧ <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;P:Bool>
 graphics: draw_less_than_equal_node;
 textual:  <=(I1,I2);      (expression)
}
```

```
and_node
{
 names: bool_and
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (bool,bool_and)
   IN
            (I1 ∧ I2)⟺P ∧ <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;P:Bool>
 graphics: draw_and_node;
 textual:  &&(I1,I2);   (term)
}


or_node
{
 names: bool_or
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (bool,bool_or)
   IN
            (I1 | I2)⟺P ∧ <I1:VT,I2:VT;E>{OPNAME(I1,I2)}<E,E;P:Bool>
 graphics: draw_or_node;
 textual:  |(I1,I2);   (simple-expression)
}


not_node
{
 names: bool_not
 semantics:
   FOR_TUPLE (VT,OPNAME)
   BOUND_TO_TUPLES (bool,bool_not)
   IN
                        <I1:VT;E>{OPNAME(I1)}<E;~ I1:VT>
 graphics: draw_not_node;
 textual:  !(I1);   (factor)
}
```

array_create_node
{
names: array_create
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,int,array_create)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;empty[VT_1]:VT_2 \rightarrow VT_1>$$

graphics: draw_array_create_node;
textual: empty[type-spec]; (constant or array-generator)
}

array_select_node
{
names: array_select
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_select)
IN

$$<I_1:VT_1 \rightarrow VT_2,I_2:VT_1;E>\{OPNAME(I_1,I_2)\}<E,E;I_1(I_2):VT_2>$$

graphics: draw_array_select_node;
textual: (array-ref)
}

array_replace_node
{
names: array_replace
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any_type,array_replace)
IN

$$<I_1:VT_1 \rightarrow VT_2,I_2:VT_1,I_3:VT_2;E>\{OPNAME(I_1,I_2,I_3)\}<E,E,E;I_1[I_3/I_1(I_2)]:VT_1 \rightarrow VT_2>$$

graphics: draw_array_replace_node;
textual: (array-generator)
}

```
array_size_node
{
 names: array_size
 semantics:
  FOR_TUPLE (VT1,VT2,OPNAME)
  BOUND_TO_TUPLES (int,any,array_size)
  IN
                    <I1:VT1→VT2;E>{OPNAME(I1)}<E;O1:VT1>
 graphics: draw_array_size_node;
 textual:  array_size(I1)   (prefix-operation)
}
```

```
array_low_node
{
 names: array_low
 semantics:
  FOR_TUPLE (VT1,VT2,OPNAME)
  BOUND_TO_TUPLES (int,any,array_low)
  IN
                    <I1:VT1→VT2;E>{OPNAME(I1)}<E;O1:VT1>
 graphics: draw_array_low_node;
 textual:  array_low(I1);     (prefix-operation)
}
```

```
array_high_node
{
 names: array_high
 semantics:
  FOR_TUPLE (VT1,VT2,OPNAME)
  BOUND_TO_TUPLES (int,any,array_high)
  IN
                    <I1:VT1→VT2;E>{OPNAME(I1)}<E;Oi:VTi>
 graphics: draw_array_high_node;
 textual:  array_high(I1);   (prefix-operation)
}
```

array_set_bounds_node
{
names: array_set_bounds
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_set_bounds)
IN

$$<I_1{:}VT_1 \rightarrow VT_2, I_2{:}VT_1, I_3{:}VT_1; E> \{OPNAME(I_1,I_2,I_3)\} <E,E,E;O_1{:}VT_1 \rightarrow VT_2> \wedge lower(O_1)=I_2 \wedge upper(O_1)=I_3$$

graphics: draw_array_set_bounds_node;
textual: set_bounds(I_1,I_2,I_3); (prefix-operation)
}

array_concatenate_node
{
names: array_concatenate
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,any,array_concatenate)
IN

$$<I_1{:}VT_1 \rightarrow VT_2, I_2{:}VT_1 \rightarrow VT_2; E> \{OPNAME(I_1,I_2)\} <E,E;O_1{:}VT_1 \rightarrow VT_2>$$

graphics: draw_array_concatenate_node;
textual: array_cat(I_1,I_2); (prefix-operation)
}

record_create_node
{
names: record_create
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,any,record_create)
IN

$$<I_1{:}VT_1; E> \{OPNAME(I_1)\} <E; Record[VT_1]{:}FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2>$$

graphics: draw_record_create_node;
textual: (record-generator);
}

record_select_node
{
names: record_select
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (field,any,record_select)
IN

$$FN_i=I_2 \wedge <I_1:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2, I_2:VT_1;E>\{OPNAME(I_1,I_2)\}<E,E;I_1.I_2:VT_2>$$

graphics: draw_record_select_node;
textual: (record-ref);
}

record_replace_node
{
names: record_select
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (field,any,record_replace)
IN

$$<I_1:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2, I_2:VT_1, I_3:VT_2;E>\{OPNAME(I_1,I_2,I_3)\}<E,E,E;I_1|I_2:I_3|:FN_1 \rightarrow VT_2 \text{ X } \cdots \text{ X } FN_n \rightarrow VT_2> \quad FN_i=I_2,(1\leq i\leq n) \wedge (1\leq i\leq n)$$

graphics: draw_record_replace_node;
textual: (record-generator);
}

union_create_node
{
names: union_create
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (type_spec,any,union_create)
IN

$$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:T_i \rightarrow VT_2> \wedge (1\leq i\leq n)$$

graphics: draw_union_create_node;
textual: (union-generator);
}

union_tag_test_node
{
names: union_tag_test
semantics:
  FOR_TUPLE (VT_1,VT_2,VT_3,OPNAME)
  BOUND_TO_TUPLES (tag,any,boolean,union_tag_test)
  IN

$$(\text{tagname}(I_1,I_3) \Longleftrightarrow P) \wedge \langle I_1{:}T_1 \rightarrow VT_2, I_2{:}VT_1; E\rangle \{\text{OPNAME}(I_1,I_2)\} \langle E,E;P{:}VT_3\rangle$$

graphics: draw_union_tag_test_node;
textual: (union-test);
}

real_to_int_node
{
names: real_to_int
semantics:
  FOR_TUPLE (VT_1,VT_2,OPNAME)
  BOUND_TO_TUPLES (real,int,real_to_int)
  IN

$$\langle I_1{:}VT_1;E\rangle \{\text{OPNAME}(I_1)\} \langle E;O_1{:}VT_2\rangle$$

graphics: draw_real_to_int_node;
textual: int(I_1); (prefix-operation)
}

char_to_int_node
{
names: char_to_int
semantics:
  FOR_TUPLE (VT_1,VT_2,OPNAME)
  BOUND_TO_TUPLES (char,int,char_to_int)
  IN

$$\langle I_1{:}VT_1;E\rangle \{\text{OPNAME}(I_1)\} \langle E;O_1{:}VT_2\rangle$$

graphics: draw_char_to_int_node;
textual: int_c(I_1); (prefix-operation)
}

float_node
{
names: float
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,real,float)
IN

$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$

graphics: draw_float_node;
textual: float(I_1); (prefix-operation)
}

int_to_char_node
{
names: int_to_char
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (int,char,int_to_char)
IN

$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$

graphics: draw_int_to_char_node;
textual: char(I_1); (prefix-operation)
}

floor_node
{
names: floor
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (real,int,floor)
IN

$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$

graphics: draw_floor_node;
textual: floor(I_1); (prefix-operation)
}

trunc_node
{
names: trunc
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (real,int,trunc)
IN

$<I_1:VT_1;E>\{OPNAME(I_1)\}<E;O_1:VT_2>$

graphics: draw_trunc_node;
textual: trunc(I_1); (prefix-operation)
}

first_node
{
names: first
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,first)
IN

$<I_1.i:VT;E>\{OPNAME(I_1)\}<E;I_1.1:VT>$

graphics: draw_first_node;
textual: first(I_1); (prefix-operation)
}

rest_node
{
names: rest
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,rest)
IN

$<I_1.i:VT;E>\{OPNAME(I_1)\}<E;I_1.i+1:VT>$

graphics: draw_rest_node;
textual: rest(I_1); (prefix-operation)
}

whenever_node
{
names: whenever
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,boolean,whenever)
IN

$<I_1{:}VT_1,True{:}VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E,E;I_1.iVT_1>$,

$<I_1{:}VT_1,False{:}VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E,E;E>$

graphics: draw_whenever_node;
textual: whenever(I_1,I_2); (prefix-operation)
}

advance_upon_node
{
names: advance_upon
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,boolean,advance_upon)
IN

$<I_1.1{:}VT_1,I_2.1{:}VT_2;E>\{OPNAME(I_1.1,I_2.1)\}<E,E;I_1.1{:}VT_1>$,

$<I_1{:}VT_1,True{:}VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<E,E;I_1.i{:}VT_1>$,

$<I_1{:}VT_1,False{:}VT_2;E>\{OPNAME(I_1.i,I_2.i)\}<I_1.i,E;O_1.i-1>$

graphics: draw_advance_upon_node;
textual: advance_upon(I_1,I_2); (prefix-operation)
}

concatenate_node
{
names: concatenate
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,concatenate)
IN

$<I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,E;O_1:VT>$

graphics: draw_concatenate_node;
textual: concatenate(I_1,I_2); (prefix-operation)
}

is_current_node
{
names: is_current
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,is_current)
IN

$<I_1.i:VT;E>\{OPNAME(I_1.i)\}<E;I_1.i:VT>$

graphics: draw_is_current_node;
textual: is_current(I_1); (prefix-operation)
}

merge_node
{
names: merge
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (bool,any,merge)
IN

$<True:VT_1,I_2:VT_2,E;E>\{OPNAME(I_1,I_2,I_3)\}<E,E,E;I_2:VT_2>$,

$<False:VT_1,E,I_3:VT_3;E>\{OPNAME(I_1,I_2,I_3)\}<E,E,E:I_3:VT_2>$

graphics: draw_merge_node;
textual: NONE
}

outbound_switch_node
{
names: outbound_switch
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (bool,any,outbound_switch)
IN

$<True:VT_1,I_2:VT_2;E,E>\{OPNAME(I_1,I_2)\}<E,E;I_2:VT_2,E>$,

$<False:VT_1,I_2:VT_2;E,E>\{OPNAME(I_1,I_2)\}<E,E;E,I_2:VT_2>$

graphics: draw_outbound_switch_node;
textual: NONE
}

identity_node
{
names: identity
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,identity)
IN

$<I_1:VT;E>\{OPNAME(I_1)\}<E;I_1:VT>$

graphics: draw_identity_node;
textual: NONE
}

route_tokens_node
{
names: route_tokens
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,int,route_tokens)
IN

$(1 \leq I_2 \leq n) \wedge <I_1:VT_1,I_2:VT_2;E_1, \cdots ,E_n>\{OPNAME(I_1,I_2)\}<E,E;E_1, \cdots ,E_{I_2-1},I_1,E_{I_2+1}, \cdots ,E_n>$

graphics: draw_route_tokens_node;
textual: NONE
}

funnel_tokens_node
{
names: funnel_tokens
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (any,int,funnel_tokens)
IN

$<I_1:VT_2,...E_{I_1-1}, I_{I_1}:VT_1, E_{I_1+1},...;E>\{OPNAME(I_1, \cdots, I_n)\}<E_1 \cdots E_n;I_{I_1}:VT_1>$

graphics: draw_funnel_tokens_node;
textual: NONE
}

apply_node
{
names: apply
semantics:
FOR_TUPLE (VT_1,VT_2,OPNAME)
BOUND_TO_TUPLES (function,any,apply)
IN

$<I_1:VT_1,I_2:VT_2,...,I_n:VT_2;E_1,...,E_m>\{OPNAME(I_1,...,I_n)\}<E,E;O_1:VT_2,...,O_m:VT_2>$

graphics: draw_apply_node;
textual: apply(I_1,I_2,...,I_n) (apply-exp)
}

input_node
{
names: input
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,input)
IN

$<;E>\{OPNAME\}<;O_1:VT>$

graphics: draw_input_node;
textual: NONE
}

output_node
{
names: output
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,output)
IN

$$<VT;>\{OPNAME(I_1)\}<E;>$$

graphics: draw_output_node;
textual: NONE
}

constant_node
{
names: constant
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,constant)
IN

$$legal\_constant(I_2) \wedge <I_1:VT,I_2:VT;E>\{OPNAME(I_1,I_2)\}<E,I_2:VT;I_2:VT>$$

graphics: draw_constant_node;
textual: NONE
}

fork_node
{
names: fork
semantics:
FOR_TUPLE (VT,OPNAME)
BOUND_TO_TUPLES (any,fork)
IN

$$<I_1:VT;E_1...E_n>\{OPNAME(I_1)\}<E;I_1:VT,...,I_1:VT>$$

graphics: draw_fork
textual: NONE
}

if_then_else_node
{
names: if_then_else
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\langle I_1{:}\text{bool}, \cdots, I_2{:}VT,\ldots,I_n{:}VT;E_1, \cdots, E_m\rangle \wedge I_0 = \text{true}\ \{S_1\}\langle E_1, \cdots, E_n;O_1{:}VT,\ldots,O_m{:}VT\rangle,\quad \langle I_1{:}\text{bool}, \cdots, I_2{:}VT,\ldots,I_n{:}VT;E_1, \cdots, E_m\rangle \wedge I_0 = \text{false}\ \{S_2\}\langle E_1, \cdots, E_n;O_1{:}VT,\ldots,O_m{:}VT\rangle,}{\langle I_1{:}\text{bool},I_2{:}VT,\ldots,I_n;E_1, \cdots, E_m\rangle\{\textbf{if } I_0 \textbf{ then } S_1 \textbf{ else } S_2\}\langle E_1, \cdots, E_n;O_1{:}VT, \cdots, O_m{:}VT\rangle}$$

graphics: draw_if_node;
textual: (conditional_expr)
}

begin_node
{
names: begin
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\langle A_1{:}VT,\ldots,A_p{:}VT;E_1,\ldots,E_m\rangle \wedge A_p \in \{\{I_1,\ldots,I_n\} \cup \{O_k\}\},1\leq n\ \alpha n\ ki \quad \{S_i\}\langle E_1,\ldots,E_p;O_i{:}VT\rangle \wedge 1\leq\leq n}{\langle I_1{:}VT,\ldots,I_n{:}VT;E_1,\ldots,E_m\rangle\{\textbf{begin } S_1;S_2;\ldots;S_k \textbf{ end}\}\langle E_1,\ldots,E_n;O_1{:}VT,\ldots,O_m\rangle \wedge O_iO_i\ ,1\leq 1\leq n}$$

graphics: draw_begin_node;
textual: (begin_expr)
}

case_node
{
names: case
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\langle I_1{:}VT,\ldots I_n{:}VT;E_1,\ldots,E_m\rangle\{S_{k+1}\}\langle E_1,\ldots E_n;O_1{:}VT,\ldots,O_m{:}VT\rangle, \quad \textbf{is } T_i\ x \wedge \langle x \mid T_1{:}VT,I_1{:}VT,\ldots I_n{:}VT;E_1,\ldots,E_m\rangle\langle E_1,\ldots E_n;O_1{:}VT,\ldots,O_m{:}VT\rangle}{\langle I_1{:}VT,I_n{:}VT;E_1,\ldots,E_m\rangle\{\textbf{case } p{:=}x{:}\text{union of } T_1{:}S_1;\ldots;T_k{:}S_n;\textbf{default}{:}S_{k+1}\textbf{end}\}\langle E_1,\ldots E_n;O_1{:}VT,\ldots,O_m{:}VT\rangle}$$

graphics: draw_case_node;
textual: (case_expr)
}

forall_eval_node
{
names: forall_eval
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\langle i{:}int,I_1{:}VT,\ldots,I_n{:}VT;E_1\rangle\{S_l\}\langle E_1,\ldots,E_n;O_l{:}VT\rangle \wedge (L\le i\le U),\quad F\in\{plus,mult,and,or,min,max\}}{\langle In_1{:}VT,\ldots,In_n{:}VT;E\rangle\{\textbf{forall i in [L,U] do } S_l \textbf{ eval:F}\}\langle E_1,\ldots,E_n;F(O_1,F(O_2,\ldots,F(O_{U-L},O_{U-L+1})\ldots))\rangle}$$

graphics: draw_forall_eval_node;
textual: (forall_expr)
}

forall__construct_node
{
names: forall_construct
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\langle i{:}int,I_1{:}VT,\ldots,I_n{:}VT;E_1\rangle\{S_l\}\langle E_1,\ldots,E_n;O_l{:}VT\rangle \wedge (L\le i\le U),}{\langle In_1{:}VT,\ldots,In_n{:}VT;E\rangle\{\textbf{forall i in [L,U] do } S_l \textbf{ construct}\}\langle E_1,\ldots,E_n;O{:}int\rightarrow VT\rangle}$$

graphics: draw_forall_construct_node;
textual: (forall_expr)
}

for_iter_node
{
names: for_iter
semantics:
FOR_EACH(VT)
BOUND_TO(int,real,bool,char,array,record,union,function)
IN

$$\frac{\begin{array}{l}\langle In_1{:}VT_1,\ldots,In_k{:}VT_k;E_1,\ldots,E_j\rangle \wedge (1\le k,j\le n)\{\textbf{initialize}\}\langle E_1,\cdots,E_k;O_1{:}VT_1,\cdots,O_j{:}VT_j\rangle\\ \neg B \wedge \langle In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m\rangle\{S_1\}\langle In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m\rangle\\ B \wedge \langle In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\cdots,E_m\rangle \wedge (1\le m\le n)\{S_2\}\langle E_1,\cdots,E_n;O_1{:}VT_1,\cdots,O_m{:}VT_m\rangle\end{array}}{\langle In_1{:}VT_1,\cdots,In_n{:}VT_n;E_1,\ldots,E_m\rangle\{\textbf{for initialize do if B then } S_1 \textbf{ else } S_2 \textbf{ end}\}\langle E_1,\cdots,E_n;O_1{:}VT_1,\cdots,O_m{:}VT_m\rangle}$$

graphics: draw_for_iter_node;
textual: (iteration_expr)
}

ABSTRACT

Several converging technologies have reached the point where they can be integrated and used to develop an advanced programming environment for writing parallel programs. These technologies include advanced graphics workstations, models of computation which can be used for parallel computation, and parallel architectures. Several manufacturers are providing commercially available parallel processing computers with potential for satisfying the performance requirements of many of today's computing problems. Unfortunately, these computers usually do not have adequate, user-friendly programming environments, and the programming primitives supported by each machine are different. Thus, there is a need for a more *general*, user-friendly programming tool.

The dataflow model of computation has been receiving increasing attention to discover the model's potential use in parallel programming. Several textual languages have been developed for experimentation with parallel programming. Graphical base language representations have been proposed, but not developed because of the absence of graphics technology needed to implement graphical languages. The potential to support a programming language which permits the simultaneous existence of graphical and textual representations for programs has become practical with recent advances in interactive graphics technology and graphics workstations.

This thesis is concerned with the design and formal specification of a dataflow programming language which supports the simultaneous existence of graphical and textual representations for programs. The features of the language are synthesized from existing dataflow languages. The specification combines three formal specification techniques to formally specify the textual syntax, graphical representation, and semantics of the language. The specification serves as a rigorous, unambiguous description of the language.

END

8-87

DTIC