P-127

A Distributed Program Composition System

Robert L. Brown

February 1989

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 89.6

NASA Cooperative Agreement Number NCC 2-387

RIACS

Research Institute for Advanced Computer Science

A Distributed Program Composition System

Robert L. Brown
Research Institute for Advanced Computer Science
Mail Stop 230-5
NASA Ames Research Center
Moffett Field, CA 94035-4000
rlb@riacs.edu

RIACS Technical Report 89.6
February, 1989

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 from NASA to the Universities Space Research Association (USRA).

ABSTRACT

This report investigates a graphical technique for creating distributed computer programs and describes a prototype implementation which serves as a testbed for the concepts herein. The type of programs under examination is restricted to those comprising relatively heavyweight parts that intercommunicate by passing messages of typed objects. Such programs are often presented visually as a directed graph with computer program parts as the nodes and communication channels as the edges. This class of programs, called parts-based programs, is not well supported by existing computer systems; much manual work is required to describe the program to the system, establish the communication paths, accommodate the heterogeneity of data types, and to locate the parts of the program on the various systems involved.

The work described in this report solves most of these problems by providing an interface for describing parts-based programs in this class in a way that closely models the way programmers think about them: using sketches of digraphs. Program parts, the computational nodes of the larger program system, are categorized in libraries and are accessed with browsers. The process of programming has the programmer draw the program graph interactively. Heterogeneity is automatically accommodated by the insertion of type translators where necessary between the parts. Many decisions are necessary in the creation of a comprehensive tool for interactive creation of programs in this class. This report explores the possibilities and elaborates on the issues behind such decisions.

This report describes an approach to program composition, not a carefully implemented programming environment. However, section 5 describes a prototype implementation that can demonstrate the ideas described ehrein. Readers solely interested in the description of the composition system *per se* should read sections 1, 4, 5, 6, and 7, and may comfortably omit sections 2 (past work) and 3 (underlying virtual machine).

TABLE OF CONTENTS

1. PROBLEM STATEMENT AND BACKGROUND

1.1. Problem Description

The problem under examination here is the construction of very large computer programs whose components employ multiple computers of diverse types. While not new, this problem has become of interest given the large number of types of computers now connected to common, high speed networks. There are numerous types of computers, *e.g.*, von Neumann, symbolic, vector, massively parallel, and database, in common use today. No one type of machine is sufficiently rich in expression, general-purpose, or fast enough to satisfy all aspects of modern computational needs. The most sophisticated of computations need the resources of several of these machines simultaneously. Yet, the technology and methodology for creating such programs has always been clumsy, hard to use, and almost solely available only to those with high expertise in the way computer systems work.

The purpose of this study is to examine a new approach to creating multimachine computations. Our approach is to define a composition system that is capable of generating very large programs, which we call applications, that are the combination of computations developed and tested on the individual computers constituting the larger system of computers. Each type of computer is specialized for a particular class of tasks, and by combining programs on each into a single, cooperating computation, a leap in the sophistication of the type of problem solvable by computers is possible. In this report, we describe one approach that offers that leap.

The type of environment to which this work applies is described by four characteristics:

- Computers interconnected by communications networks.
- A computing environment with a wealth of existing software.
- A community of knowledgeable computer users using this environment to perform their computations.
- A wide diversity in the type of computing machines available.

The problem is to formulate a way in which the users of the systems can construct computations that take advantage of the wide diversity of resources and, at the same time, allow them to use much or all of the existing software. The solution must be of a form that requires little special knowledge of the network, the operating environments of the individual resources, or of distributed programming languages and techniques. We do not propose our approach as the ultimate solution, but as an approach that makes significant progress.

We divide the problem and our approach into three broad categories: inherent problems, our choices, and artificial problems resulting from those choices. The three inherent problems are as follows:

1) **The model of computation problem.** What model of computation best matches the architecture of diverse networked computers we have outlined? We seek a model that is a natural fit to the architecture.
2) **The language problem.** What programming language best suits the model of computation we have chosen? We seek a language that matches the model and is highly usable.
3) **The mapping problem.** How do we best map the language back onto the architecture?

Corresponding to the three inherent problems are three choices made in the course of this research. We acknowledge that these are not the only choices possible, but are the one we chose to study. This work seeks to support the claim that these choices are good ones. The choices, corresponding to the problems, are as follows:

1) **Representation-transformation graphs.** Representation-transformation (RT) graphs are a common way of describing a computation in terms of how the data is represented and what transformations take place to convert that data from one representation to another. In RT graphs, there are two types of nodes: representation nodes and transformation nodes. Representation nodes can be made an implicit part of the arcs connecting transformation nodes without any loss of generality. Data flow graphs [Davi82] are RT graphs with implicit representation nodes.
2) **Visual language.** We choose a visual programming language for creating RT graphs because RT graphs are inherently two-dimensional and only a visual language can display that structure naturally. This motivation is elaborated on later.
3) **Node-to-program association.** We choose completely self-contained programs to correspond to the transformation nodes of the RT graphs and make the representation nodes implicit. These choices allow the language to map directly onto the networked computers environment. Nodes (programs) run on virtual machines and the edges of the graph represent communication channels, implemented by the network, between the machines. Benefits of this approach are elaborated upon later.

Whenever choices for the solutions to inherent problems are made, a new set of problems, called *artificial problems*, arise. Our choices generate a set of artificial problems, mostly concerning the implementation of the solution choices to the inherent problems. These new problems include, but are not limited to, the following:

1) **Decomposition of the language into atoms.** Though the only basic atoms of the language are nodes and edges, others may be necessary to make the language more usable and understandable.
2) **Human factors.** How can a visual language be constructed in a way that makes it easy to use and the visual representation match the user's concept of how the graph looks?

3) **Structures for implementing the language.** What data structures are needed to represent the RT graph yet can also be used to represent the visualization of the program?

4) **Accommodating heterogeneity.** Because the language must accommodate a diverse set of computing resources, what mechanisms are necessary to describe and accommodate that diversity? In particular, diversity appears as different data representations, resource naming schemes, and methods for invoking computations.

More artificial problems will arise when additional requirements are placed on the design and implementation of this system for programming networked resources. These four problems, however, are addressed in the course of this study.

We believe that the choices made for the solution of the inherent problems are good ones because they provide a good match to the problems, thus introducing a sense of coherence into our system. Other examples of architectures, languages, and mappings exist that exhibit a similarly high degree of coherence. For example, von Neumann architectures, where the procedural languages for them, Fortran, Algol, Pascal, and so on, are a good abstraction of the underlying architecture and hence map cleanly onto the architectures. Likewise, data flow architectures and applicative languages [Acke79, Acke82] are a good match for each other. This concept of matching the computing environment with its inherent problems to the choices for solution is a fundamental one necessary for the success of a programming system.

Having mapped out the research area and our approach, we now present a statement of the goal of this work:

> *To explore a parts-based data flow paradigm as a way of programming a distributed heterogeneous computer system whose nodes are a part of a high-speed communication system.*

Our approach is experimental:

> *We designed and built a prototype composition system to validate the concepts, discover new ideas and principles, and explore the limitations of our approach.*

1.2. Contributions of this Work

The primary contributions and results of this work are as follows:

1) An understanding of the parts-based model of computing. Sections 4 and 6 elaborate this result. Our model of programming can express computations not possible using other approaches.

2) Characterization of a visual parts-based programming language. Section 5 elaborates this result. The visual interface distinguishes this work from other work in networked programming.

3) A working prototype that demonstrates a high-level language for program composition and validates the parts-based approach. The prototype is an important piece of this work and is a usable and self-contained programming environment that will serve as the basis for applying parts-based programming to specific and real problem domains.

1.3. Motivation and Context

1.3.1. A Scenario of Computation

A collection of computing resources connected by a network offers possibilities to those who rely upon computing technology, such as practicing computational scientists in physics (*e.g.* computational fluid dynamics) and chemistry (*e.g.* materials sciences), that far exceed those offered by a single isolated computer. For example, an advanced application may involve several distinct tasks, each responsible for a specific part of the overall computation. Using a computational fluid dynamics example, one portion may be responsible for assisting the user in the design and modification of the airplane structure to be tested. Another portion may be responsible for the generation of the simulation grid surrounding that computation, with assistance from the user but using symbolic mathematics, possibly on a specialized processor. A third portion may be responsible for the bulk of the fluid flow simulation, calling on a fourth portion to obtain flight configuration information (angle of attack, Mach number, *etc.*) from the user or from a flight regime database. A fifth portion may be responsible for monitoring the progress of the simulation and displaying results or partial results on a color graphics screen.

The hypothetical application described above demonstrates that a variety of computing resources may be needed for applications that could be in common use today if the software technology were available. A combination of symbolic processors, user interface graphics machines, supercomputers, and general purpose computers are necessary for advanced computational solutions. These resources differ not only in their architecture and data representation formats, but also in the style and languages used to program them. Some may use the traditional imperative programming style, yet others may use functional, data flow, or data structures oriented styles. The combination of language, style, and interface used to program a machine is called its *paradigm*.

In the future, scientific applications will become even more complicated as parallel architectures become commonplace. Parallel computers are an inevitability (a compelling argument for this appears in an article by Denning [Denn86]); several exist and are in common use today. Many of these machines offer a low degree of parallelism which is most often used to enhance multiprogramming. Others, such as hypercubes [Seit85], the Connection Machine [Hill85], and the MPP [Batc85] offer a high degree of parallelism but are not suited for all portions of a complicated application. As more advanced and specialized architectures become available, the need to take advantage of them, not as stand-alone resources, but in cooperation

with computations running on more general purpose machines, will become more acute.

Such a computing environment is not novel, and with the proliferation of supercomputers it is becoming more commonplace. At the Ames Research Center of the National Aeronautics and Space Administration (NASA), the computing environment consists of vector supercomputers, parallel and massively parallel computers, symbolic computers, and graphics engines, as well as traditional uniprocessor machines. Applications are written in several dialects of Fortran and LISP, in C, in assembly languages, in specialized vector languages, and in a variety of object-oriented languages.

Hence, we base our decision to investigate composition of distributed programs from existing programs, as opposed to a more tightly integrated programming approach based on remote procedure calls or a new language such as SR [Andr88], on the requirement to accommodate the wealth of existing software developed in multiple paradigms in the computational sciences domain. Existing numerical computation applications are often written in a highly optimized style for a particular machine architecture, such as a vector processor, or are optimized for a particular machine, such as a vector processors, or perhaps even a particular vendor's vector processor. Restructuring and rewriting these codes will not only entail a vast amount of work, but will likely result in a significant loss of efficiency on the machine for which they were optimized.

The need outlined above indicates a problem. There is no existing technology that allows a programmer to compose together, in a straightforward way, program parts developed under two or more paradigms into a single coherent and complete application.

1.3.2. Requirements on Solutions

Any solution to the need described above must account for the realities of the environment in which it is used. Some *ad hoc* requirements on such a technology include the following:

1) **Existing software requirement.** In the scientific computing domains, there is an enormous wealth of software, the rewriting of which would require an enormous expenditure. Many scientific computing laboratories continue to routinely use codes that have continually evolved since being generated a decade or more ago. Any system that seeks to advance the state of scientific computing by allowing the creation of large multipart applications must allow existing software to be incorporated.
2) **Multiparadigm requirement.** A single application must be able to integrate program modules created in different programming paradigms.
3) **Heterogeneity requirement.** Although two computers cooperating in a single application may have different data formats, any solution must allow them to

share results. The data generated by one computer must be usable without special consideration by any other computer.

4) **Immutable program requirement.** Some of the existing software in use in scientific computing is "immutable." That is, it cannot be changed because it is licensed or proprietary and the source code for that software is not available. Hence, composition systems for scientific computing must have a mechanism for incorporating unchangeable codes.

5) **Ease of use requirement.** Such a system as the one studied in this work will be used by practicing computational scientists: physicists, chemists, biologists, and engineers, as well as, but not solely, computing experts. The system must not add undue complexity to the programming environment.

6) **Ease of transportability requirement.** Any system proposing to offer the ability to compose together programs on several different types of machines must not require a large amount of work to add a new system or type of system into the set available to its user.

These new requirements add a new set of artificial problems to be solved in order to assure that they are satisfied.

1.4. Existing Partial Solutions

Some partial solutions to the distributed program composition problem do exist, in particular, remote procedure call and integrated distributed languages. These two are discussed and compared to the list of requirements stated above.

1.4.1. Remote Procedure Call

One way to create distributed applications is with remote procedure calls (RPC), as described in Nelson's thesis [Nels81]. RPC allows the programmer to compose larger programs from smaller procedures, and to assign procedures to various machines that support the RPC servers. A proper implementation of an RPC system includes stub and server generators, semi–automatic programs that facilitate the creation of "stub" or proxy procedures that are linked into the user's main program and translate the procedure call and arguments into a message passed to the RPC server on the designated machine. Moreover, complete RPC systems support the automatic generation of the application-specific RPC server on the remote machine, embodying the actual procedures written by the programmer and interface code that translates incoming RPC requests from the subroutines into procedure calls. The server must also handle return values from procedures, or at least synchronization when the procedure call completes.

RPC satisfies the existing software requirement reasonably well given adequate stub generators, since existing subroutines and procedures may be executed remotely more-or-less transparently. Existing software, however, is not always packaged in the form of procedures in the language supported by the RPC system; often it is packaged as complete programs. RPC accommodates the immutable program

requirement, but at the subroutine level since immutable subroutines may be "wrapped" into a server for remote execution. Whether RPC systems satisfy the ease of use requirement depends almost entirely on the implementation of the support software surrounding the stub and server generators. RPC satisfies the heterogeneity requirement because it deals with programs at the procedure level in the same manner as compilers. Hence, it is able to determine the data types of the arguments to remote procedures (by looking them up in a symbol table) and generate the proper code to translate from the local data types to the remote data types (in reality, this is most often done by locally translating into a common network form and then remotely translating from that into the target representation).

RPC does not meet the multiparadigm requirement in that the procedure call mechanism itself does not. Procedure calls work best between subprograms written in the same language and, though some manufacturers support cross-language calls, many do not and the mechanism is ill-suited for accessing programs written in paradigms other than imperative languages. This is incompatible with our goal to accommodate diverse resources not available within a single machine. RPC works best when the program using it starts as a single-machine program and then portions are moved to some other machine.

RPC further suffers from a lack of properly implemented support systems, that is, automatic stub generators and partitioning aids, often resulting in only custom RPC facilities being used for a specific application, such as to support applications that run on a supercomputer and use the graphics input and output devices available on high–performance workstations.

A third difficulty with RPC in the scientific community is the lack of support for global data. It is very common for numerical codes in use at the national research centers to be coded in Fortran and make extensive use of global, or common, memory to store large matrices of coefficients, partial results, and so on. To pass these arrays from machine to machine as the application's program counter migrates from machine to machine would place a heavy load on the networks. A modified approach would be to cluster those routines sharing a section of global memory into a single large module and run that module on the machine best suited to it. The use of global data in scientific codes is partly a result of the use of procedure calls as the program composition technique because both procedure calls and global data are a part of the language, typically Fortran.

1.4.2. Distributed Languages

Another way to create distributed applications is by use of a special language for describing them, such as in SR [Andr88], CSP [Hoar78], and other languages of the type described by Andrews and Schneider [Andr83]. These languages suffer from most of the same problems as RPC and may not meet the "multiparadigm requirement" or the "existing software requirement" if they do not allow the inclusion of existing program modules written in other languages. In fact, this class of languages represents a new paradigm for programming, and does not provide a direct

means of accommodating existing paradigms. Any solution must allow programs written in these languages to be incorporated into an application.

It is feasible that a distributed programming language may, however, be used solely to describe those aspects of the computation that are involved in the communication and synchronization necessary for managing a distributed computation, and not be used to perform much or any of the actual computation taking place. A similar technique is used for the control of a parallel and distributed computation in a language called Concurrent C [Brow88a]; the Concurrent C portions manage the rendezvous between servers and clients while modules written in C and Fortran implement the computation performed by the program. A similar approach is taken in the scheduling of parallel programs in a system named SCHED developed by Jack Dongarra at the Argonne National Laboratory [Dong86]. In general, any distributed programming language based on a non-distributed or existing language can be used in this way.

1.4.3. Distributed Operating Systems

Distributed operating systems bear mention because they offer the possibility of supporting distributed programming. However, because they are not programming systems, they cannot be completely compared to the requirements. Notable examples of such systems are the V kernel at Stanford [Cher84], the Mach operating system at Carnegie-Mellon [Acce86], MOS at the Hebrew University of Jerusalem [Baro86], and LOCUS at UCLA [Walk83]. These systems have contributed much to the knowledge of how distributed programs can be created, but do not implement a complete programming environment because they do not specify the language in which the computations are encoded. The client-server model used by systems such as these represent another programming paradigm that can be used within a composition system satisfying the requirements.

1.5. Problem Statement

Given the diversity of modern computing environments, how is it possible to take advantage of the diversity and yet give it an integrated appearance without rewriting existing application parts? Some applications can be developed now in this environment, but doing so requires specialized knowledge, often known only to systems programmers, and the techniques result in a loss of productivity for the practicing computational scientist.

The objective is to devise a new paradigm for program construction that accommodates all existing paradigms. Figure 1.1 shows an abstract representation of the solution. The top level, labelled "Composition System," takes advantage of programs written in several paradigms (displayed at the second level). In turn these paradigms can be implemented in different machines (at the bottom level). Notice that paradigms may exist (*e.g.* Paradigm #3) that are only implemented on a single machine.

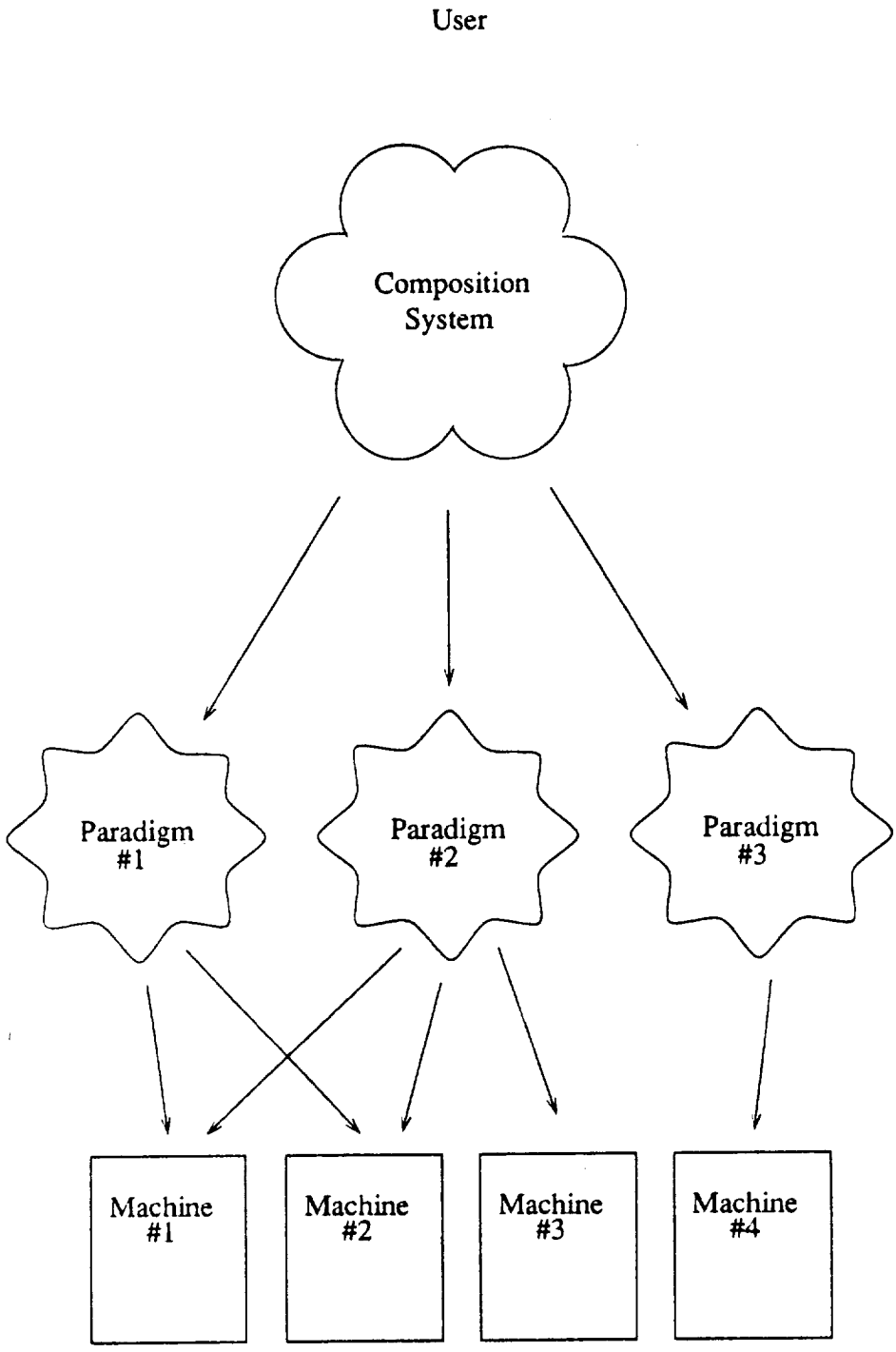


Figure 1.1: Multiparadigm Composition System

The goal of the research in this work is to provide a uniform way of connecting together parts of applications into larger applications. The methodology described herein provides a step towards network transparency, lessening the clumsiness of using a network of computers without removing the benefits of having a diversity of resources available. The heterogeneity made available to the practitioner is important in that it is at the core of the diversity offered by the network of computers.

We assert that important properties of the composition system are as follows:

1) **Parts orientation.** The abstraction of programs seen by the user while in the design environment is one of parts of programs stored in a database. This is analogous to the concept of modules as described by Parnas [Parn72] but is more generalized. Parts are not restricted to be program subroutines.
2) **Hierarchical composition.** Programs constructed with the composition system can themselves be placed into the parts database for use in other composed programs. Hence, the parts database will contain primitive programs as well as composed ones, and the difference will be transparent to the user.
3) **Graphical interface.** It is our goal that the constructed program should resemble an engineering drawing. Engineering disciplines use diagrams and drawings routinely for designs in domains other then computing. Figure 1.2 shows an abstract example of how a drawing may represent a computation. The boxes represent parts and the lines represent a relation between parts.

There are many options for implementation of a system with these properties that satisfies the requirements listed. Procedure call-based composition does not satisfy

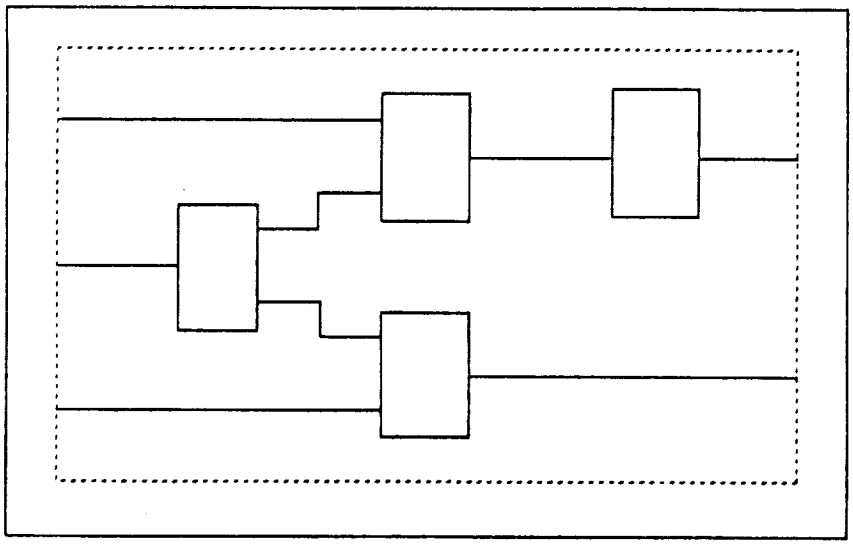


Figure 1.2: Abstract Composed Program Drawing

the multiparadigm requirement, though fares well elsewhere. The particular choices used in herein are outlined in the following sections.

1.6. A New Approach

Because of the deficiencies of existing techniques for writing distributed applications in scientific domains, a new approach is pursued. It is inspired by the way the UNIX operating system [Ritc74] uses pipes to interconnect programs. The power of programming by connecting program parts with pipes has been described by Hailpern:

> *As operating systems go, the UNIX operating system provides for reasonable dynamic linkage editing through its pipes. Programs running under UNIX are supposed to take their input from standard input and send their output to standard output. The pipe facility allows the standard output of one program to be tied to the standard input of another program, forming one large program out of a chain of small programs. If a program follows this convention, it can be written in any available language. Of course, this is a restricted form of multiparadigm system, because the composition is restricted to a linear chain of programs and the interface between programs is restricted to a sequence of ASCII characters [Hail86].*

The technique of program composition described above is commonplace. It has the advantage that programs developed by separate authors in separate languages can be interconnected, and, Hailpern's statement notwithstanding, even binary data can be passed if the programs on either end agree on the representation. The procedure call mechanism is not as flexible; few systems exist that allow linking together procedures and subroutines from vastly different languages.

The one attribute that all programs have in common is that they (usually) require input and always generate output. It should be possible, therefore, to devise a system whereby the inputs are produced by programs as well as by users, and the output may be interpreted by programs. Such a system allows the creation of applications based on existing programs and thus may result in extremely large new applications.

The technique of interconnecting programs into larger programs is not new; linkage editors were designed for just that purpose, but using subroutines instead of complete programs, and advanced linkage editors allowing for simple multiparadigm environments consisting of more than one language have existed for decades [McCa63]. Compile, assemble, and load programming systems are among the earliest examples of the use of multiple independent programs being combined into a larger application, in this case for program development. Traditionally, however, job control languages (JCL) have been used to describe applications based on multiple programs, and little, if any, allowance was made for multiple programs to execute at the same time. These applications were multi–step "jobs" that communicated via

scratch files. An early example was IBM's OS/360 [IBM67]. The limitations of such systems were inherent. They did not allow for simultaneous execution of programs and they did not allow for the distribution of parts of the application over multiple machines. They did, however, in a clumsy way, allow for more than linear composition. If you construct a graph that describes the relationship of program steps in a multi–step job, where the nodes are the programs and where an arc from program A to program B means that B cannot begin execution until A has completed and generated its output, then any program may have outward arcs to more than one other program, meaning that the results generated by that program are used by more than one other. Likewise any program may have inward arcs from more than one other program, meaning that it requires the results generated by more than one program in order to accomplish its computation. There may not, however be any cycles in the graph, as that would result in a situation where no program in the cycle may be begun until, ultimately, it has completed.

1.6.1. Graphical Pipe-based Composition

The problems discussed above regarding job control languages do not usually exist in pipe–connected languages such as supported by the UNIX shell [Bour78]. Because programs are connected by a communication channel, the pipe, a program may begin processing once it has received less than the complete set of data generated by the program generating the data it reads. As stated by Hailpern, however, the UNIX shell only allows linear composition, a restriction that does not exist in most job control languages. The solution to the linear composition problem in pipe–connected languages is evident when one reflects back on the graph constructed (as described above) that specifies the data dependencies among all the programs in the application. The arcs become pipes through which one program passes data to the next. These pipes are unidirectional, just as the arcs are directed. The data flows in the direction of the arc, from producer to consumer. Hence, the solution to the linear composition problem is to devise a language whereby arbitrary combinations of interconnections are allowed among the steps of a job, or "parts" of an application. The concept of separately compiled and linked whole programs remains as it was in the multi–step job and in the linear pipe–connected pipeline, but the linear composition restriction disappears.

A graphical approach is under investigation instead of a new textual language because it offers the possibility of having all the advantages of the pipe–connected languages without the linear composition restriction. Using such an approach, the programmer would, using a graphical editor, sketch a diagram of his network of programs, using an pictorial representation for the programs (at the nodes of the graphs) and lines to represent the pipes. We intend to explore the applicability of such techniques to the task of composing distributed programs. Some of the disadvantages of purely textual techniques for non–linear composition are as follows:

- Textual descriptions are difficult to construct because each program involved in the distributed computation may have several inputs and outputs. Text is

principally one–dimensional, that is, a flow of words extends on one direction: left–to–right. The interconnection taxonomy of a distributed program can extend to more dimensions. Examples of multiconnection textual command shells do exist for the UNIX operating system, but they are all limited to a fixed number of connections and a particular interconnection scheme, and suffer from using only linear text to describe a non–linear network.

- Textual descriptions are difficult to understand by reading. Looking at a written description of how programs with several inputs and outputs are interconnected, it is difficult to grasp abstractly a picture of the interconnection graph. Linear composition systems do not suffer from this difficulty since the linear nature of the text nicely reflects the linear nature of the composed program.

Graphical descriptions have the potential for being easier to construct. In a simple case, the interconnection graph may directly reflect the topology of the underlying network of computers and pictures of such topologies have been in use for many years. Some specific advantages are as follows:

- Graphical representations show, in a two dimensional drawing, the interconnection of the program. Since text is only one dimensional and the program network graph is potentially two– or higher–dimensional, a graphical representation will in all cases more closely match the network picture when more than linear composition is used. Complicated networks with dimension higher than two can, in many cases, be mapped into two–space.
- The system will run on a workstation capable of rendering objects graphically, rather than purely textually. In 1988, examples of such machines are the Sun workstation [Sun86] or the Apple Macintosh [Appl85]. The editor and local program graph storage will be managed by the workstation, potentially providing a modern sophisticated user interface that extend beyond purely keyboard–oriented systems.

1.6.2. Graphical Composition Considerations

A graphical interface may allow for non–linear composition of program parts into a larger application. As a part of this research, a prototype composition system has been constructed in order to verify some of the otherwise unsupported statements about the benefits of such a system. Though much of this work centers around the abstract composition system, many references are made to the working system, indeed, an entire section is devoted to its description. The working system is named the "Distributed Program Composition System" (DPCS). As so well stated by Lantz [Lant80], "Any attempt to discuss both an abstraction and an implementation can lead to difficulty..." Lantz quoted Cheriton on the same topic:

> *It is advantageous to remain faithful to the current design and implementation of [DPCS] in our discussion so that remarks are supported by implementation, testing, and experience. It is also advantageous to include how we believe the system should have been*

> *done, drawing on the benefit of hindsight and experience. It is equally advantageous to abstract the discussion with a particular system to provide wider applicability of our conclusions. All three of these competing goals govern this report; we trust the reader will recognize the different tacks in the course of the discussion [Cher78].*

An underlying operating system is needed to make pipe-connected distributed programming possible, more so than for procedure call-based composition, which typically only requires a compiler, a supporting machine instruction, and a linkage editor. Section 3 contains a short model for an operating system that is capable of supporting this form of composition. The essential parts of the system and its requirements on an underlying machine architecture are described there.

1.6.3. Terminology

This sections presents intuitive descriptions for the terms used in the graphical composition system used throughout the remainder of the study. More rigorous definitions are given later. Since some of these words have different meanings in related areas of study, the reader should occasionally refer back to this section.

Primitive program. A primitive program, often called just a "primitive," is a traditional computer program that can be invoked, or executed, outside the context of the composition system being described here. One constraint added is the ability to describe the number and type (input or output) of external communication paths used by the program. Support from the underlying operating system is required to invoke a primitive.

Composite program. A composite program is one that was constructed by a program composition system, and differ from primitive programs in that they require the support of the program composition system loader phase to start.

Program part. Program parts are the unit of composition, corresponding to a unit of computation on the underlying virtual machine. A program part is either a primitive program or a composite program. Program parts have unique names by which they can be identified. Associated with both types of program parts is the body, comprising a composite program or a primitive program, and a description of its ports.

Socket. A socket is the point through which a program part receives input or sends output. Looking at it from inside the part, it is a language dependent item, typically a handle of some sort, referenced by all I/O statements in the program. Examples include Fortran logical unit numbers and UNIX file descriptors. Viewed externally, it is place from where output data appears and to where input data can be sent. Typically, operating system support is required in order to capture the output from ports or direct input to ports of other programs. In UNIX, the mechanisms for accomplishing I/O through ports are pipes, sockets, files, and devices.

Communication link. A communication link is a construct whereby one part communicates with another through their ports. These paths are typically operating system constructs such as UNIX pipes or other interprocess communication (IPC) channels.

Program network. A program network consists of a graph and a boundary. The graph consists of nodes and edges, the nodes being program parts and the edges being communication paths between program ports or from program ports to the boundary. The boundary is an object that may hold the ends of edges but is not a program part and performs no computation per se. A fully–connected program graph is one that has all ports on all parts connected to either other ports or the boundary.

Composition system. The composition system is an interactive editor that allows its user to construct program networks by selecting program parts and then connecting the ports to other ports or the boundary. Traditional editing functions such as saving existing work and reading previously saved work are supported as well. Another name for the composition system is the editor or graphical editor.

1.7. System Decomposition

This section presents an overview of the structure and implementation of the prototype program composition system. A more complete description is provided later.

The system decomposes into several distinct high-level modules, described here. These are the graphical editor, the underlying description language, the part selection browser, the program linker and loader, and the execution monitor. Figure 1.3 presents a block diagram. Different versions of this system may or may not contain all pieces or may contain only minimally functional pieces of the full system.

The program composition system places demands on the underlying operating system for functions it does not directly implement, such as program primitive invocation. A model for a support system is described in Section 3.

1.7.1. Graphical Editor

The graphical editor is a purely interactive program that runs on the user's workstation. As defined, the editor cannot be used by other computer programs because the input for it comes almost exclusively from devices accessible only to the human user. A separate non-interactive editor allows computer programs to construct program graphs.

The components of the editor are the sketch pad where the user draws the program network and the control panel where operations on the network, such as saving work, starting execution, or other global operations, are invoked. Depending on the degree of sophistication of the implementation, some of the features described below may or may not be included in the editor.

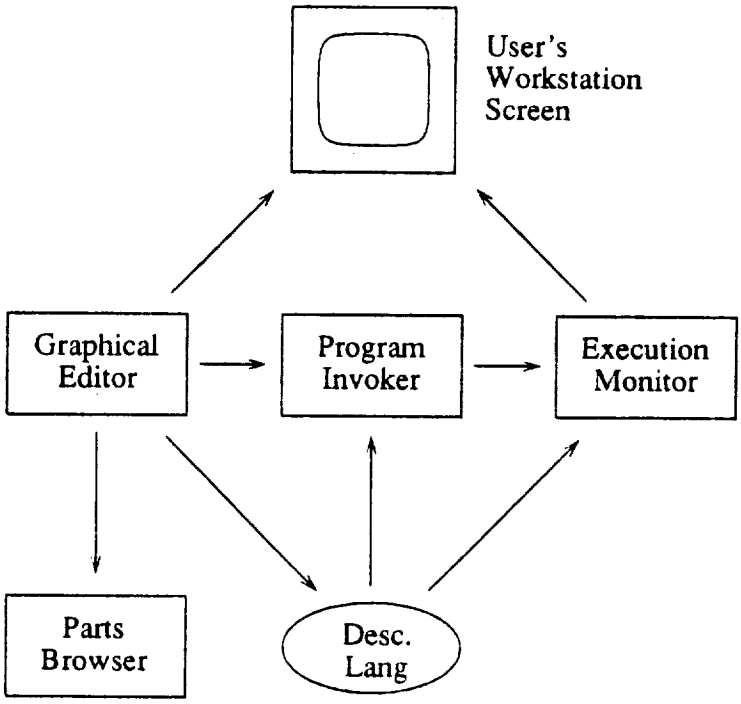


Figure 1.3: Block Diagram of DPCS

The sketch pad is the area where the current state of the program network is displayed. Each program part is represented by an icon with some textual information attached stating its name. Program ports on a part are represented by small icons attached to the periphery of the part. Communication paths are represented by lines connecting ports to other ports or the boundary. The boundary is represented by a bounding box around the entire network.

Figure 1.4 is an example of how the sketch pad might appear; it is an extension of the abstract figure presented earlier. The figure does not show a concrete representation but rather a rough sketch of how parts, represented as icons, and communication links, represented as lines, might appear. The dashed line represents the boundary, which itself has input and output connections.

Minimal features of the sketch pad include the following:

- Parts placement. The user has the ability to select a new program part from a library (whose nature is yet unspecified) and place it on the sketch pad.
- Port connection. The user has the ability to connect a program port to another program port or to the boundary.

More advanced features may be supported by the editor to enhance its usability and the productivity of the user. Some of these features include the following:

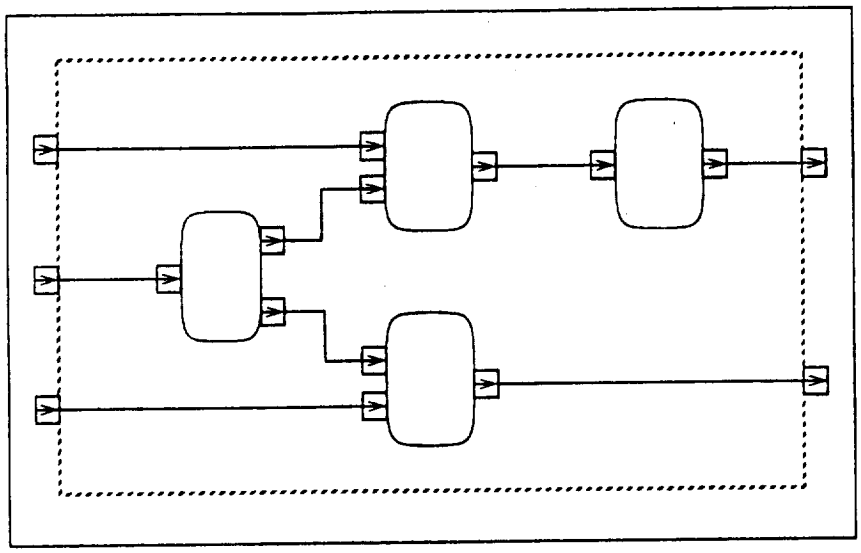


Figure 1.4: Sample Sketch Pad Layout

- Zooming. Because the program network may conceivably contain very many program parts, the user needs the ability to focus attention on a particular part of the graph, magnifying it to a higher degree to emphasize the section under consideration; this is called "zooming." Zooming is independent of the ability to define composite parts as it does not allow the user to examine the internal network of a composite part. Analogous to zooming, and potentially as useful, is scrolling - allowing the user to selectively look at different parts of the network.
- Part placement assistance. Just as program beautifiers exist for traditional programming languages, a sketch pad may have some built in support for placing program part icons in visually pleasing places. There are several levels for this capability ranging from simple icon placement grid enforcement to automatic network restructuring to minimize some criterion function as ports are connected. Graph visualization is a research topic of its own and work is ongoing in that area [Newb88].
- Annotation. Traditional textual languages have a convenient method of adding comments to programs. Because the program is text and the comments are text, they can be merged in with notation indicating the separation of program from comment. In the graphical language being described here, very little, if any, text need appear on the sketch pad in order to display the interconnection structure. Therefore, the editor needs to use a different mechanism for supporting program comments. One method would be to allow a text window to be associated with each item (part, port, communication path, boundary connection) on the pad. The editor then allows the user to request that information, presumably with a minimum amount of effort. Some information in the form of comments could be provided by the editor itself, rather than supplied by the user. Examples include

revision numbers of parts, data types generated by ports, and complexity metrics on the entire network.

- Communication path routing. The task of connecting two ports, or a port to the boundary, need not include manual routing of the path line through the picture of the network. The editor can provide an automatic routing mechanism, such as variations on those described for robotics [Loza79].
- Expansion of composite parts. It may be desirable to expand in place composite program parts. Advanced support for redesigning the layout of the program graph will be necessary for in-place expansion to be feasible.

The control panel of the editor contains operators for global operations, that is, operations that affect the entire program graph, rather than a single piece on the sketch pad. The minimal functions provided by the control panel are the following:

- Workspace saving. Once a program network has been constructed, or during any point in its construction, the state of the work needs to be save in long-term storage for later use.
- Leaving the editor. A capability to leave the editor and return to the system control program is a necessary function of the control panel.

Extensions to the operations allowed on the control panel include the following:

- Program invocation. Invoking the program network is not a function of the editor. However, requiring the user to leave the editor in order to run the network is an unnecessary restriction, since the program network is essentially an interpreted program (with compiled primitive programs embedded in it). Interactive debugging may be a part of the editor, and so the editor and debugger may be combined into a single piece of the system.
- Sketch pad control functions. Additional global operations are managed by features on the control panel. Examples include clearing the sketch pad, loading a new workspace, documenting the entire workspace, and printing its contents.

1.7.2. Network Description Language

The output of the editor includes a specification in a language called the "network description language" (NDL) that describes completely the abstract network. NDL does not describe the way the editor displays the network on its sketch pad; this information is stored separately, though in an identical format.

An NDL program is a semi–readable ASCII file. Though NDL programs can be constructed manually using a conventional text editor, the language is not designed to support it and so it is not an easy task. Because NDL programs describe the entire graph, they name program parts. Composite parts mentioned are not necessarily expanded in line, but instead are mentioned by reference to the names of their NDL programs. The expansion into a graph containing only primitive parts is handled elsewhere.

1.7.3. Program Invoker

The program invoker component of the system takes as its input a program in NDL, expands all the nested networks associated with composite parts, and initiates the computations described by the primitive parts, linking them up as appropriate with communication paths.

The task of the program invoker is particularly complicated when the system is one that allows the composition of distributed programs on a network of heterogeneous machines. It must have knowledge of how to start a program part on a remote machine and how to connect the ports of that program to other parts on either the same machine or other machines. To support heterogeneity, the invoker must have the ability to attach data translation modules to parts so that a part running on a machine with one data format (byte ordering, floating point format, character string formats, *etc.*) can receive data generated by a program running on a machine with a different format.

1.7.4. Execution Monitor

Normally, once a composed program begins execution, it runs to completion without intervention from the composition system. Program parts perform their computation, pass data among themselves, and run to completion with no interference from any monitoring system. This execution scenario, however, does not allow for network debugging, monitoring, or collection of performance data.

Part of the composition system is an execution monitor which, when linked in by the program invoker, can be used to collect information about the data passing through the communication paths and, where possible, collect information about the progress of the parts at the nodes. Collected information may be displayed visually as performance charts and link traces. Such information could be presented visually and represented as annotations to the original program network as displayed by the graphical editor.

1.8. Structure of the Report

The remainder of this report elaborates on the design and prototype implementation of a program composition system with a graphical interface, and its extension to a heterogeneous distributed system. Below is an annotated summary of the report.

Section 2 - Related work in visualization. One of the novel aspects of this work is its use of visualization of the computation. In fact, the composition technique itself is visual, and the mechanism for connecting programs is interactive with graphical feedback. This section presents a model for visualization in computation and discusses others' work and how it fits the model.

Section 3 - The virtual machine model. The composition system is not a complete computing environment. It relies heavily upon functionality exported by the

underlying operating system. This section presents an idealized model for that operating system.

Section 4 - The basic composition system. This section presents a description of the semantics of composed programs, the basic building blocks, and the invocation semantics of the programs.

Section 5 - The program development environment. Development of composed programs is accomplished using a visual interface. This section presents that interface, the abstractions and representations used, and describes the structure of the prototype.

Section 6 - Extensions to a distributed system. New abstractions are necessary to extend the basic system to a distributed system, principally in the areas of remote execution and accommodating heterogeneity. This section elaborates those distinctions.

Section 7 - Conclusions and future work. This section draws conclusions about the work on DPCS and presents pointers for future research in this area.

2. MODELS FOR PROGRAM VISUALIZATION

2.1. Introduction

The work described in this report relies heavily on a visual interface for a particular style of distributed programming, as described in the previous section. This section describes different models for program visualization and systems using visualization that relate to this work. The models aid the understanding of visually based programming systems by describing a framework into which other existing and proposed systems may be fit, thus facilitating comparisons of the features and goals of each.

Many researchers have worked in the area of program visualization. Some seek to make computer programming more accessible to the naive user. Others seek to simplify program debugging and understanding. Others attempt to make the output of computations understandable. All deal with the visual representations of programs and data.

In understanding program visualization, it is useful to distinguish between display and construction. Display refers to the graphical display of data and programs. Visual construction, however, implies a form of "direct manipulation" [Shne83], whereby the user constructs data directly rather than by specification. Display alone seeks to answer the question, "What might it look like?" Visual construction asks the question, "How do I form it?" and involves interaction with the user. Hence it is a form of interactive programming, whether it be programs or data being constructed; there is no distinction. The form of the interaction, however, is spatial and visual, typically in at least two spatial dimensions. Interactive programming involving only text is one dimensional and does not qualify as visual construction.

Related to program visualization itself is the topic of visual languages, a term that has several different interpretations. To some, "visual languages" refer to languages that support visualization, that is, graphical animation and rendering languages such as LOGO and Twixt [Gome85] (an animation system allowing an animator to produce a movie generated entirely by the computer). To others, as in this report, "visual language" means a language for producing a computation and having a sketch pad or direct manipulation interface. Sketch pad interfaces to languages may be of at least two types, programming by example or programming by declaration. With the former, the program itself is represented by "graphical" actions performed by the user, as in Rehearsal [Finz84]. In the latter, though the interaction may be either graphical or textual, the principal attribute is that the program is being declared by the programmer; the actions that the programmer takes to construct the computation have no direct relationship to the computation itself.

A further distinction is made by some among "visual," "graphical," and "pictorial." In this report, these terms have the following meanings:

- **Visual.** Visual systems have a spatial quality, relying on two (or more) dimensions to display data or programs. Though visual systems could use only

text (by spatially orienting that text) they typically also include non-textual pictures and graphics. Non-visual systems can always be displayed in one dimension without any loss of meaning. Visual systems can be either graphical or pictorial and can use aspects of both.

- **Graphical.** Graphical systems are visual systems that use more than just text, typically including traditional graphics objects (lines, boxes, circles) to display some inherent relation over the data. The distinction of graphical systems is that the graphics is used to display a relation among objects, not just objects themselves.
- **Pictorial.** Pictorial systems are those that use abstract images, called icons, to denote a part of the data. The term "icon," defined in Merriam-Webster's dictionary [Merr81] means "pictorial representation." Pictorial representations differ from graphical representations in that the pictures can be unstructured, not necessarily denoting a relation or a structure, but are suggestive of an object or action. Korfhage and Korfhage [Korf86] categorize icons as either "object" or "process." Object icons are usually a concrete, simplified pictorial representation of the object they identify. Process icons, on the other hand, are more abstract, often utilizing arrows and object icons to denote action applied to an object.

2.2. IPO Model for Visualization

We present a simple model for categorizing visualization in computation based on a simple model of computation and then describe how visualization relates to its parts. The model is based on representations of the following form:

$$y = f(x)$$

In this notation, x represents the input to the computation, f represents the computation itself, and y represents the output of the computation. This model is not restricted to the traditional definition of "function"; nondeterminism is allowed. All that is implied is that computations have input, they operate on them, and produce output. This is also generally known as the "IPO" model for computation, meaning "input, process, output." We use the term "program" in place of "process;" process has a different meaning in this report.

The model is useful because it provides a first means of categorizing types and uses of visualization. There are three parts to the IPO model, and so there are three parts to "IPO visualization."

One more distinction is necessary, between static and dynamic visualization of data. Static visualization is well understood; it refers to a rendering of a data object, where "object" means a bundle of related, structured information referenced by a single name or handle. Dynamic visualization incorporates a time dimension and can be thought of as a sequence of renderings of a static object, where the change from one rendering to the next is described by a relation associated with the object. In the case of composite objects, there may be one or more relations for each component.

The important point is that dynamic visualization varies with time, and therefore involves continuing computation whereas static visualization only requires a single computation of graphical output.

With the IPO model, we can now form a matrix of areas where visualization and construction techniques apply. This matrix has as its rows the labels "input," "output," and "program." The columns of the matrix are labelled "visualization" and "construction." The cells in the matrix list examples of systems that apply to the pair, row and column, denoted by the cell. This matrix is presented in table format in Table 2.1. What follows examines cells of this matrix and how they relate to the visual program construction technique in this report.

2.2.1. Output and Input Visualization

Output and input data visualization in the IPO model are sufficiently similar in treatment that they are combined in this discussion. Data is abstract, not having any inherent representation. In order to use and manipulate data, we impose a representation on it, and then manipulate that representation. Any piece of data may have several representations, some of which are amenable to manipulation by computer programs and some amenable to being understood by humans. We call the former "internal" representations of the data and the latter "external" representations. Internal representations are typically in a binary format specified by the data architecture of the computer operating on them. External representations may be purely textual, purely graphical, or a combination of the two. Data

Table 2.1: Visualization/Construction Matrix

	Visualization	Construction
Input	General data visualization systems	Drawing programs, dialog boxes
Output	Graphical rendering packages	Interactive animation, scientific visualization
Program	Flow charts, Nassi-Shneiderman charts	Visual programming languages

visualization refers to converting data from any representation into a graphical representation.

The traditional field of computer graphics concerns itself with the visualization of the resultant data of computations. Many books and journals exist describing the research in algorithms and program systems for this purpose. The work in this area addresses the previously stated question, "What might the data look like?" It is often the case that the computation is solely concerned with the generation of the picture, as with graphical rendering programs such as the BRL CAD and ray-tracing package [Muus87] whose input is only data describing the desired picture. In other instances, the computation is not concerned *per se* with generating visual data (such as a numerical simulation) and produces results which, in the past, were printed as tables of numbers by a line printer. In the present, however, it is commonplace to not rely solely on the tables of numbers to represent the results. Instead users often use sophisticated systems for the purpose of converting those results into graphical form.

Output visualization is not limited to static pictures. Often the computation performed has too many parameters to represent meaningfully in the inherent two-dimensions of a graphical output device. Hence, time animation is necessary to visualize additional parameters. The Twixt animation system has been used to convert the results of numerical computations in computational chemistry into an animation of a trimer atomic system potential energy surface. Similarly, animation systems such as GAS [Banc88] exist for the sole purpose of generating animations of results from numerical simulations on supercomputers. Many other systems are described in the graphics literature that perform similar functions. The point is that output visualization can be static or dynamic. The complexities of output visualization are not accounted for in the simple IPO model of program visualization. If we add a time parameter to the functional form, however, to indicate that the input may vary with time and therefore the output may vary with time, what results is a time-variant IPO model that better describes dynamic visualization. Hence, the function form becomes:

$$y(t) = f(x(t))$$

Program input can also have visualization techniques applied to it, and can therefore be subjected to the same forms of visualization as output data, answering the previously stated question, "What might it look like?"

Program input can be represented visually using graphical input objects first commonly seen on the Xerox Star system [Purv83] and subsequently made popular on the Apple Macintosh personal computer [Appl85]. This style of input has become popular and commonplace; many commercial implementations exist. Dialog boxes are a form of visual input, where the data is represented using real-world analogies such as knobs, sliders, switches, and push buttons. Frequently such input systems allow the application programmer to create and display graphical representations of data in the program and display them in a way that allows the user to modify them, often resulting in direct modification of the data corresponding data. For example, an

enumerated data type in the C language can be represented as a labelled "cycle" which, when displayed, allows the user to point to a picture reminiscent of a knob and click (analogous to turning a knob) on it to cycle through the constant values in the enumerated type. Hence, the input parsing takes the form of translating human activities (typing, pointing, dragging a pointer, *etc.*) into data values or functions on data values such as a successor function on an enumerated data type.

2.2.2. Visual Data Construction

Data can be subject to the second question, "How do I construct it?" The solutions to this question include data visualization, as described above, but also include manipulation techniques that can have a spatial as well as textual quality. Visual data manipulation refers specifically to the technique of modifying data by modifying a graphical external representation of it.

An example of direct manipulation of data is in a program named ConcertWare for the Apple Macintosh [Mitc85]. A part of this program allows the user to design a new simulated musical instrument by drawing the waveform generated by the instrument. The waveform is stored internally as a time-indexed array of amplitudes and is displayed externally as a waveform. By manipulating the waveform picture, the user changes the data stored in the internal representation. A picture of this is given in Figure 2.1. Another example, from the computational sciences, is a system that allows the user to graphically design a simulation grid around a given shape, such as an airfoil, and then use that grid as the space over which a fluid dynamics computation is performed.

Direct manipulation techniques can be applied to output visualization as well as input visualization. For example, scientific visualization systems operate on the data

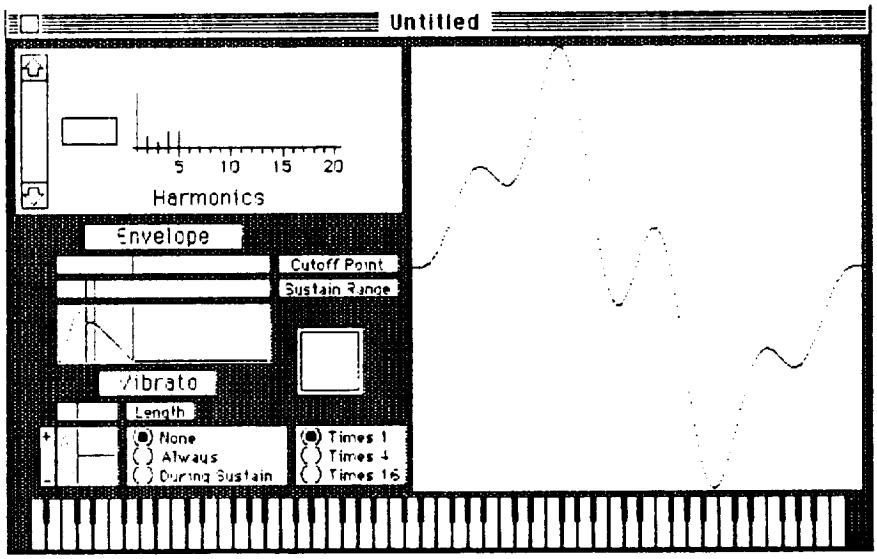


Figure 2.1: ConcertWare's Waveform Display and Input

produced by other programs, producing graphical renderings. Typically the amount of data is large and the best way to represent it is not known to the visualization system. Hence, with the proper interface, a user could guide the system, using a direct manipulation technique, into producing the desired image.

2.2.3. Program Visualization

Visualization of computer programs typically only uses static graphics, which is useful for presenting a graphical view of a static program for the purposes of documentation or aiding understanding of the program. Programs, represented as data, do not change in time except for a class of dynamic list-oriented languages. Many techniques have been devised to display one aspect or another of a program, its control flow, data flow, data dependencies, *etc.* Examples are traditional flow charts and Nassi-Shneiderman charts [Nass73]. These techniques can be somewhat useful for program documentation but typically represent programs at too fine a granularity to be any more useful for construction than textual representation. It is possible to generate these graphical forms of programs from the source code of the program itself. Such is an example of converting one external representation, the source code, into another, the chart or diagram.

In the literature, the term "program visualization" refers to visualizing the state of the computation implemented by the program [Baec75, Brow85a, Brow85b, Myer83, Petr87, Reis85]. Such displays may be static, taking "snapshots" of data structures inside the program, or dynamic, using animation techniques to time animate data structures. The primary goal of program visualization systems is to aid in the understanding of the operation of a program, either for teaching programming or for debugging a program. These systems convert directly from internal representations to graphical output representations.

2.2.4. Program Construction

The next question to ask about visualization and computer programs is, "How do I construct it?" This is the realm in which the work of this report exists. Programming systems do exist that have visual interfaces, these typically display control flow, data flow, or data dependencies. Figure 2.2 shows one example, again drawn from the Macintosh environment, a program named V.I.P. [Main86], which allows the user to create a program by drawing its control flow chart using traditional symbols. We consider this approach to be at too low a level to be practical because segments of programs that have long runs of sequential statements tend to be very hard to read, and complicated control structures are too big to conceptualize. Additionally, no provision exists for augmenting the program flow chart with annotation or program comments. The same amount of code written in Pascal takes about one third the screen space. One redeeming feature of V.I.P., however, is that procedure call boxes contain a field that, when activated with the mouse, expands to a table giving the formal parameter names, their types, and the actual parameters, which can be modified by the user. This provides a convenient abridged on-line manual for the procedure.

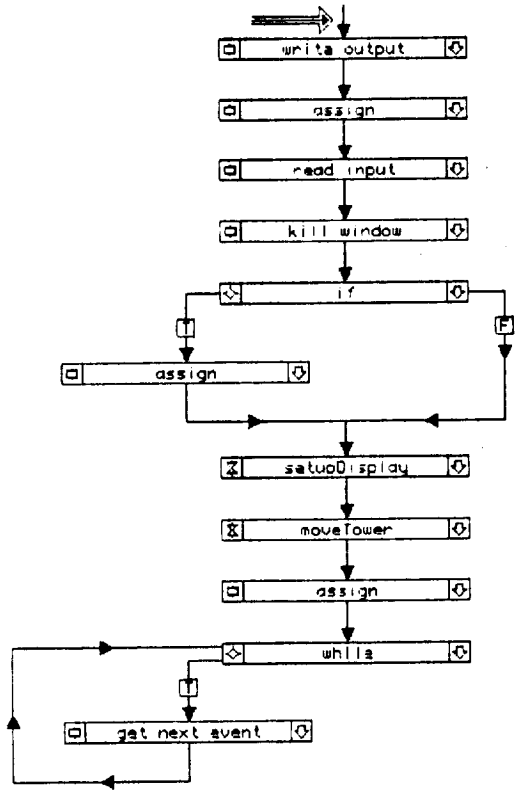


Figure 2.2: V.I.P. Control Flow Chart Example

An area where visual languages are more useful is in the construction of programs or structures that have inherent visual qualities, such as those described by data flow graphs and Petri nets. An example from the Macintosh environment is Stella [Rich87], a signal flow-based simulation language in which the nodes are operators that continuously work on their inputs and provide outputs. An example of a Stella program for simulating the flow of water from a tank is given in Figure 2.3. Such languages are gaining popularity because they take advantage of the two-dimensional screen space to display a structure that does not map well into a flow of text. Another example of a visual construction system is GreatSPN [Chio87], which allows the user to interactively construct Petri nets and analyze them.

2.3. Past Work

The application visualization techniques to computing and computer usage can be traced back to the early work of Sutherland [Suth63] and Englebart [Engl68]. Smalltalk [Gold84] makes extensive use of visualization techniques in what is otherwise principally a textual programming environment.

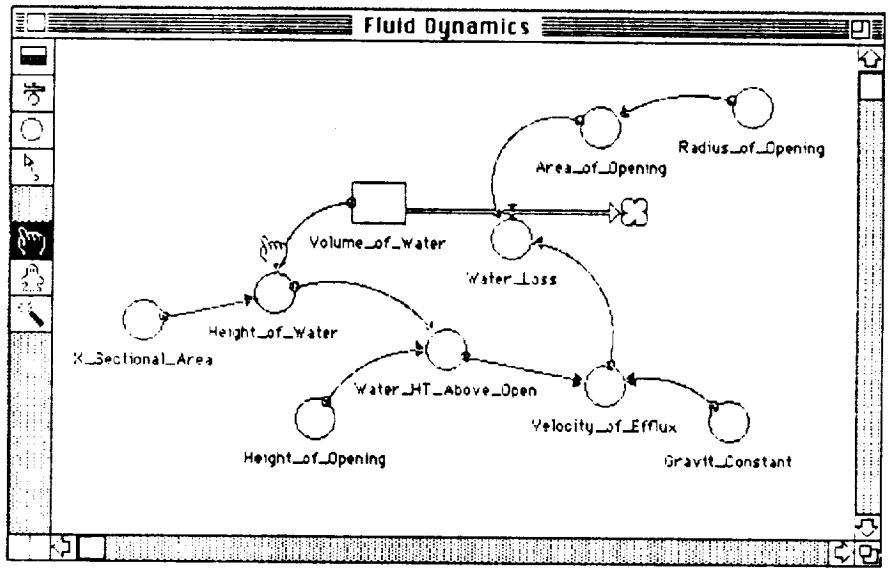


Figure 2.3: Example Visual Program from Stella

This section surveys past work in visualization techniques and is divided into three sections: surveys of visualization models, surveys of important specific works, and description of other work by category.

2.3.1. Models and Surveys

This sections describes past work in developing models for visualization of programs. Existing systems developed that fit these models are described in the next section.

2.3.1.1. Chang's Model for Visual Languages

In his survey, Chang [Chan87] attempts to clarify the meaning of the term "visual language" by presenting two broad categories of programming languages that fall under that term. The first is *visual information processing languages*, which deal with data that have some inherent visual representation, such as digitized images, but the languages themselves have no inherent or imposed visual representation. The second is *visual programming languages* which are programming languages that either impose visual representation on inherently nonvisual objects (such as data structures) or are themselves presented visually. Because "visual programming language" is a broad category, Chang further distinguishes four subcategories, based on the matrix formed by dealing with either inherently visual or inherently nonvisual objects on one axis, and the languages themselves being visually presented or not on the other axis.

Chang's model describes the objects dealt with by programming languages as generalized icons having two parts, written as (X_m, X_i) where X_m is the meaning

(semantics) of the object, and X_i is the visual representation (syntax) of the object, and "e" is used for X_m or X_i to denote a null object. Using this notation, visual programming languages transform objects with no inherent visual representation into those that do have visual representation, that is, $(X_m, e) \Rightarrow (X_m, X'_i)$. Visual information processing languages, those that deal with visual images, are denoted $(e, X_i) \Rightarrow (X'_m, X_i)$.

Chang's model for the data used in visualization is similar to ours, but is broader to include languages the manipulate inherently visual data. Where our model differentiates between the semantics of the data and its internal and external representations, Chang's emphasizes the semantics and inherent or fabricated visual representations.

The definition of visual programming languages in Chang's model applies to the work in this report. The program composition system, in the abstract sense, is a construction of program parts interconnected by communication paths but in implementation appears as pictures, or icons, connected with lines. There is nothing inherently visual about the program parts; they are abstract functions that compute outputs based on inputs. The program composition system, specifically the graphical editor (as described in Section 1) imposes a visual representation on parts where previously none existed.

2.3.1.2. Raeder's Summary

Raeder presents a survey of visual and interactive programming [Raed85], offering a general discussion on its virtues and a survey of past work. This work concerns visual program construction, and establishes a framework into which such systems can be placed.

Raeder identifies four traditional roles of pictures in programming. The first is to depict control flow, as with flow charts, structured charts, and state transition diagrams. These diagrams are low level, and do nothing to display the structure of the data. In the cases of flow charts and state diagrams, however, they are useful for displaying algorithm animation.

The second type of picture is for data flow depiction. Data flow diagrams are almost always associated with programs written in a data flow language. They show control flow and data flow in one graph in that the control of a data flow program follows the data. By typically only incorporating low level operators and using only trivial data structures, they become messy and overly complicated when scaled up to a practical size.

The third type of picture is for data structure depiction. Raeder asserts that "data structures account for a major part of the illustrations (and doodles) we make during program development" but that "there has not been much work on standardized (or even formal) ways to display data structures." Additionally, because we think of data

in programs at different levels of abstraction, a good display technique would require depiction at these different levels. How to accomplish this display is an open question.

The final type of picture Raeder calls "topology," used by programmers "to describe the overall structure of their systems." Such diagrams fail, Raeder says, because they do not support descriptions of the low level aspects of a program.

The composition system of this report is an example of the fourth type of picture. Raeder's remark about the inadequacy of this technique to describe the statement-level part of programming is correct, but our system makes no claims to support programming at the statement level. Our composition system exists so that a programmer can take programs written in other languages and combine them together into a larger system of programs. Construction of the program parts is a separate concern not addressed here.

One interesting aspect of pictures used in programming he calls "metaphorically rich," and in discussing such pictures, he states some important points.

> *We can reason and make judgements in terms of these pictures by giving meaning to the graphical relationships concerning shape, size, distance, and so on. For example, it seems reasonable to require that programs that are similar in function look similar and that different types of programs be easy to distinguish. This is definitely not the case with conventional program text. Further, "good" features (such as time/space efficiency, ease of use, ease of understanding and maintenance, and so on) should make a program "pretty," whereas "bad" features should make it unpleasant to look at. An error should make the program look imbalanced. These goals are very hard to achieve, but ultimately we will have to build mechanisms for meeting them into our programming systems if we want to make fuller use of human reasoning power. This can be done only by incorporating pictures into the systems. [Raed84]*

Specifically, we consider the point about well-written programs looking good and poorly written programs looking bad to be an important one. However, as with textual languages, "pretty-printers" may be devised that marshal a program into a standard style.

2.3.1.3. Shu's Dimensional Analysis

Shu presents a classification for visual programming that is a dimensional analysis technique that can be used to compare one visual programming system to others. The classification places work into one of two categories: visual environment or visual language. Though Shu's categorization of visualization systems is not fundamentally different from others, the evaluation criteria presented is a new contribution.

Visual environment includes systems that incorporate graphics into an otherwise traditional programming environment. Also in this category are visualization of a program and its execution (static and dynamic), visualization of data, and visualization of system design. In this report we call this category *program visualization* (PV) because it adds graphics on top of an existing development environment. Typically, in PV systems the programmer starts with an existing source program written with a text editor, and then uses the tools in the PV system to view it (*e.g.*, control structure, data flow, data dependency) graphically and to monitor the progress of its execution. PV tools are aimed at program understanding, documentation, and debugging.

Shu decomposed visual languages, the second top level category, into three lower ones, languages for processing visual information, for supporting visual interaction, and for programming with visual expression. The first, languages for processing visual information, is concerned primarily with database languages that report the results of their queries graphically. This is not to say that the databases must store visual information, only that they generate images as results. The second category, languages for visual interaction, contains those that allow the programmer to manipulate graphical images. Animation languages and graphical rendering languages, such as Twixt fall into this category. The languages themselves are textual but they manipulate pictures.

The third subcategory in the visual languages category is visual programming languages, which contains languages where the constructs for expressing the computation are visual. In this report, we also refer to this category as "visual programming languages" (VPL). Shu has developed a scale, Figure 2.4, with three important aspects of VPLs that can be used for comparative analysis. The first two of these apply to traditional, one-dimensional languages as well.

The first aspect of visual programming languages is the *level of the language*, which is an "inverse measure of the amount of details that a user has to give to the computer in order to achieve the desired results." A language that tells the computer "what" to do instead of "how" to do it is considered very high level.

The second aspect of VPLs is the *scope of the language*, referring to its breadth of applicability. Domain-specific languages and problem-oriented languages are small in scope, being only applicable to the problem or domain for which they were designed. General-purpose languages, such as C and Ada, are higher or broader in scope.

The third aspect of VPLs, *visual expression*, refers to the degree to which the language depends on visualization for the construction of its programs. A visual language that adds visual constructs to an existing textual-based language might be low in visual expression if the construction graphically maps directly into statements in the textual language. On the other hand, a language that has no useful representation in text might be high in visual expression because the only meaningful way to represent an algorithm or program in the language is with images.

The composition system in this report can be rated on Shu's three-dimensional scale. We assert that it is high in all categories. The level of the language is

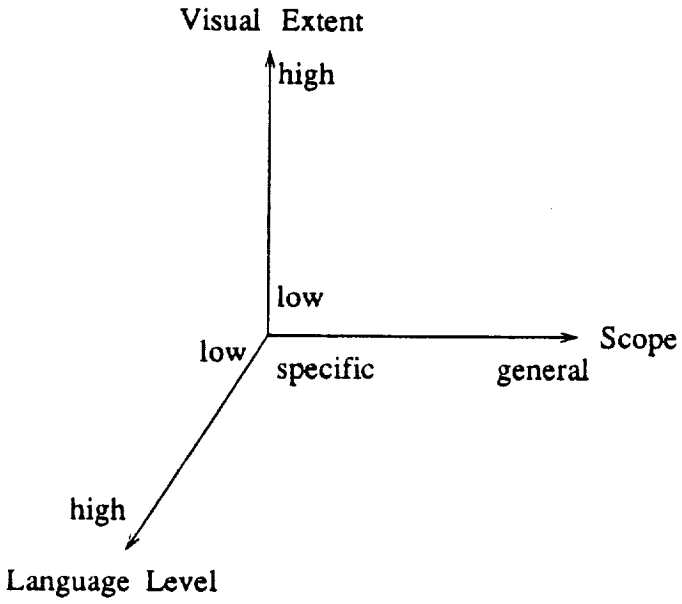


Figure 2.4: Shu's Scale for Visual Programming Languages

extremely high, that being one of its primary goals. The operators that are composed with our composition system can be arbitrarily complex and are defined in languages outside the composition system. The scope of the language is high, although not maximal. The scope of composed programs is as broad as the scope of the parts used within it. The visual expression is also high because, other than for elucidation and external representation, there is no useful textual representation of composed programs. Indeed, there is information that can be gleaned from looking at the picture of the program that cannot be easily gleaned from any one-dimensional text representation.

2.3.2. Other Specific Works in Visual Languages

This section describes in more detail selected past work in visual programming languages. The selection is not meant to be exhaustive but rather to represent work that relates to the visual interface of the program composition system described in this report. Existing systems are described here, each because it is considered a fundamental work in visual programming, because it relates in some specific way to the interface chosen for our composition system, or because it provides an interesting contrast to our system. More works than these were surveyed for this research [Bela84, Catt86, Haeb86, Mill84, Mori85, Tani82].

2.3.2.1. Jacob's State Transition Visual Language

Jacob devised and implemented a visual language for describing finite state automata (FSA) applied to user interface design [Jaco85]. The work is based in part

on work by Parnas in specifying user interfaces using transition diagrams that are common in depicting FSA [Parn69]. Jacob differentiates between two types of visual programming: That which deals with objects that have an inherent visual representation, such as an engineering drawing, a typeset report, and fonts, and that which deals with a visual representation of an inherently abstract non-visual object, such as a computer program or a finite state automaton. Jacob elaborates on the difficulty of devising suitable visual representations of abstractions:

> *A more difficult problem arises in the second category of visual programming language, representing something abstract - time sequence, hierarchy, conditional statements, frame-based knowledge. To provide visual programming languages for these objects, we must first devise suitable graphical representations or visual metaphors for them. The powerful what-you-see-is-what-you-get principle is not much help, since these objects are abstract, but the successful application of the visual programming language paradigm to these situations still depends critically on choosing a good representation. Graphical representation of abstract ideas is a powerful form of communication, but a difficult one. In the absence of an applicable theory of graphical communication, proper use of such representations often requires extensive experimentation. [Jaco85]*

Jacob uses the traditional diagrammatic notation for finite state automata, with circles representing state and lines representing state transitions, as his visual representation. Each automaton describes a part of a user interface system, and so, associated with each state transition may be semantics, implemented in a standard procedural language, that describe the actions of the user interface to be taken when that transition occurs.

The key points made by Jacob are as follows:

1) The availability of graphical workstations provides the opportunity for describing algorithms and processes visually rather than with linear text.
2) Careful choice of the proper graphical representation is critical. The representation "must leave no doubt as to the behavior of the system."
3) Hierarchy is important. That is, one FSA must have the ability to call on another. Jacob likens this hierarchy to nonterminal symbol hierarchy in BNF.

Jacob's system is similar to the one in this report in that it deals with visualizing a structure that does not have an inherent visual representation. We have described it here because it incorporates many of the same ideas as our system: visualizing a known diagram form, direct manipulation of that form, and hierarchy.

2.3.2.2. Programming by Rehearsal

Providing a contrast to the representation manipulation method of Jacob is an example of "programming by manipulation" in the "programming by rehearsal"

system developed at Xerox PARC [Finz84]. This system allows educational curriculum designers who are not otherwise programmers to design educational software by, essentially, manually stepping the computer through the algorithm implementing the product, and then having the computer step through the algorithm automatically. The analogy used by this system is that of performers acting out a play, but rather than reading from a script, they learn their lines and actions by memorizing what the director (the programmer) tells them to say and do. Hence, the Rehearsal system is designed for non-programmers who may know what they want the computer to do for them but don't know how to express it in a traditional programming language.

Programming by Rehearsal is an example of a direct manipulation demonstration-based language, where the actions of the programmer are learned and become encoded in an algorithm. It is based on Smalltalk 80 [Gold84], and each of the performers in the play composed by the programmers maps into a message to a Smalltalk type-manager.

The relationship between the Rehearsal system and the program composition system of this report is that they both are programming systems with visual interfaces, but the former allows the programmers to create a program by stepping through it manually, essentially teaching the computer the program. The composition system, on the other hand, is not a demonstration-based manipulation language. Using the visual interface, the programmer describes the computation by designing it, as opposed to the Rehearsal system where the programmer uses exemplary methods to describe the computation.

2.3.2.3. Pascal/HSD

The Pascal/HSD system [Diaz80] uses graphical notation to convey a structure of Pascal programs that is lost when the program is coded in the final form: the decomposition steps that lead to the program. The goal of this system is to allow the use of structured programming and stepwise refinement together. Structured programming alone cannot support the concepts of top-down programming; these two ideas are separate concerns. Whether structured programming is used or not, the chain of refinement used in creating the final program is lost in the coding. Stepwise refinement can be thought of as a context-free grammar describing the semantics of the algorithm; each left-hand side stating an abstraction, and the right-hand side stating its refinement. The final program is a sequence of only these right-hand sides. At best, the abstractions from which they were derived exist only in program comments.

The HSD system supports the control structures of Pascal, and hence supports structured programming. It also supports a notation for stepwise refinement. The diagrams in Pascal/HSD resemble flowcharts. Elements of the diagrams are the usual control structures of Pascal as well as "action" items, which are abstractions. The control flow in the diagram is top-to-bottom, but abstracted action items branch to

the right, and the expansion of the abstraction is a separate control flow proceeding downwards from there.

Pascal/HSD is an example of an early control-flow graphical programming language. One of its most important contributions is the abstraction capability, also an important feature of the composition system of this report. However, it describes computation at a lower level than the composition system. Programs constructed using Pascal/HSD could be incorporated as primitive parts into applications developed under the composition system in this report.

On the other hand, the notion of encoding and retaining the decomposition process is one that can be applied to our composition system. The hierarchy of parts in DPCS, formed by the composite parts, retains some of the top-down decomposition process used by the programmer. In DPCS, the programmer can create a high-level description of the program, using abstract composite parts. Next, each composite part can be successively refined until a program graph of all primitive parts results. If the programmer leaves the composite parts in the program graph, and expands each individually, the structure will be retained.

2.3.2.4. PIGS and Pigsty

PIGS [Pong83] is another visual programming system built on Pascal. It uses Nassi-Shneiderman diagrams (NSD) [Nass73] to represent the program structure. The system has a graphical interface that allows the user to interactively create the NSDs and insert Pascal-like program statements into the slots in the diagrams. At runtime, the language is executed interpretively and the user has the opportunity to insert breakpoints that will cause the program to halt execution and become available for examination or change. Additionally, the system can be told to bypass the execution of external routines (thus supporting stubs) and present a trace of the program.

PIGS is very much like Pascal/HSD in that it relies on an existing language and adds a graphical interactive front-end to that language. A novel aspect of PIGS, however, is that it was used for a more interesting visual language, Pigsty [Pong86]. Pigsty combines the Pascal-like procedural programming of PIGS (actually an extension of PIGS) with the message-passing style of Hoare's Communicating Sequential Processes (CSP) [Hoar78] for communicating between modules. Program modules, written in PIGS, have explicit *ports*, which are the points at which the program directs input and output operations. The two types of ports are called *InPort* and *OutPort*, used for input and output respectively. The ports are not type-checked at compile time, so any data type may be passed through them; type checking is performed at run time. Because the system is designed as a closed environment, there is no support for executing the program modules on different machines and, hence, no data conversion between data types on different machines is supported.

The language, essentially Pascal, is extended to have CSP-like input and output statements. A distinction between these input and output statement semantics and those of CSP is that in CSP the statements name a process to which or from which

data is sent or received while in Pigsty the statements name ports associated with the current process. Input and output statements are in CSP notation, as follows:

InPort ? variable

OutPort ! expression

Pigsty allows the programmer to create an array of processes, that is, several instances of the same program module replicated in the program in a regular way. Additionally, a module may declare and use an array of InPorts or OutPorts to communicate with an array of processes.

Program modules are interconnected by linking the output ports of modules to the input ports of other modules. In this sense, Pigsty and the program composition system of this report are very similar. Pigsty provides a hierarchy mechanism, whereby a group of interconnected modules may be formed into a single, higher-level module, and then used in another program. Pong points out that this hierarchy provides two features. First, the ability to design a program using abstract modules and then later decompose those abstract modules into existing modules. Second, the hierarchy supports a bottom-up approach to programming, and as the programs become sufficiently large, groups of modules can be clustered into a single module, simplifying the picture.

Like our composition system, Pigsty has an interactive graphical interface that allows the programmer to "draw" the interconnection of the modules. Pong likens this approach to a syntax-directed editor, stating, "the duty of assuring that a program is syntactically correct is shifted from the parser to the editor." This aspect of the graphical editor holds in the program composition system of this report, that is, the programmer is unable to create a syntactically incorrect program. A sample of a Pigsty diagram is in Figure 2.5, showing a solution of the classical "Dining Philosophers" problem.

2.3.2.5. HI-VISUAL Interactive Iconic Programming

The HI-VISUAL ("Hiroshima Visual") iconic programming language [Mond84, Hira86, Yosh86] developed at Hiroshima University uses small pictures to represent the operations that can be used in the language. For example, a picture of a television camera might be used to denote an operator that takes data from such a camera. In HI-VISUAL, these small images, called "icons," are used to denote operations, data, and other items such as control and data types. A program in this language is a two-dimensional layout of icons with short arrows between them. Typically, every other icon is an operator icon and the others are data icons. When an operator icon appears with an arrow from it to a data icon, it means that that program produces a set of data whose type is represented by the icon.

HI-VISUAL supports hierarchy, as does Pigsty and the program composition system of this report. Unlike those other systems though, it also supports interactive

programming. As the programmer creates the iconic picture of the program, any operators that have all of their inputs satisfied may start execution. The interactive feature is only possible at the top level of the hierarchy, however, because as the programmer develops the lower level functions, the inputs (and outputs) are abstracted, that is, not yet connected to any real source of data. Hence, there are both data and data-type icons; data icons are real repositories of data in a top level program that is being created interactively; data-type icons are place holders in lower level programs.

Because the data and data-type icons are explicit in HI-VISUAL, the type of data that is generated by and required by operators is also explicit. Though this may seem an unnecessary restriction, it has the potential for simplifying the programming process as described by the authors:

> *In addition ..., the system provides the facility to navigate the program development process in the following way: Assume that the user knows the input and output data, and does not know how to get the output data from the input data. The system will display a list of all candidate icons to be applied under the condition of input data type or input and output data types. The scrolling facility is provided for cases in which there are many candidate icons.*

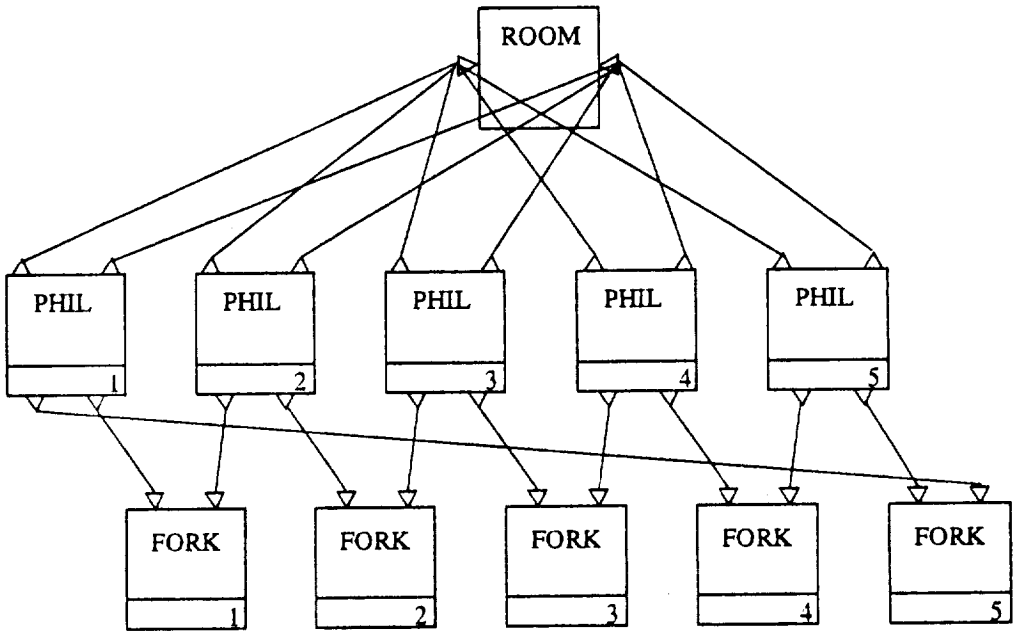


Figure 2.5: Example Pigsty Program

In summary, HI-VISUAL is an interactive, iconic programming language that is based on a data flow model, that is, the graph describing the program is a data flow graph, with the nodes representing operators and the arcs (which also contain icons) denoting the flow of data from one operator to another. The language supports hierarchy, interactive program development, and assistance in operator (icon) selection. The scope of the HI-VISUAL language is determined by the scope of the operators in its icon database.

HI-VISUAL is similar to our composition system in that it is a representation-transformation graphs, but with explicit representation nodes. No accommodation is made, however, for creating networked programs or for accommodating data type heterogeneity, that is, the case where the representation for the same data differs between two interconnected operators.

2.3.2.6. The VERDI Language for Distributed Programming

Graf describes a graphical language for distributed programming [Graf87a, Graf87b]. This work is relevant to the composition system of this report because both claim to allow the construction of programs over multiple processors. Graf's system is named VERDI, for "Visual Environment for Raddle Design and Investigation" and is based on a distributed system specification named Raddle [Form86]. Raddle is a textual language, like CSP, for specifying distributed systems. However, such languages have shortcomings, as described by Graf:

> *[These languages] suffer from a mismatch between the use of a linear technology (text) to describe non-liner phenomenon (distributed or concurrent computation). Thus, practicing designers must largely maintain their conceptual understanding of distribution and concurrency outside the formal expression of the design.*

The idea of using graphical representation to describe an important aspect of a program not otherwise described in the text has been previously described in the Pascal/HSD system. There, the hierarchical design was described in the graphical representation of the program. Here, the relationship among the parts of the program implementing distribution or concurrency is being described graphically.

2.3.2.7. Pict/D

Glinert and Tanimoto's Pict/D programming system [Glin84] is roughly based on Pascal and uses pictures, graphics, color, and sound. The language is very low level, allowing only four unique variables and having limited control structures. Icons represent program operations, and lines represent control structures. Program modules can be encapsulated and placed in a library for use in other programs.

The programming process consists of selecting operations from a library of icons and then placing them on a sketch pad. Selecting a prewritten routine implies a call operation on that procedure.

Pict/D provides an incremental execution environment. The programmer can have the system execute the program as it is being developed. When the runtime system reaches a yet-unprogrammed portion, it stops and allows the programmer to fill in what is missing. As a Pict/D program is running, if selected to do so by the programmer, the system will animate its execution, showing which part is currently executing.

The authors of Pict/D conducted an experiment using 55 undergraduate and 10 graduate computer science students. In general, the responses were positive. The experimenters found that subjects less experienced with programming liked it better than those with more programming experience, confirming that Pict/D may be useful as a tool for teaching programming. One question asked was, "If it were possible to do so (in the future), how would you rate the chances that you would prefer on a regular basis to use an expanded, Pict-like system to write your computer programs rather than a language like Pascal?" Answers were on a scale of one to five, five being most favorable. Of the undergraduates, 90.7 percent responded with an answer of three or higher, yet only 20 percent of the graduate students answered in that range. Both sets of students, however, felt the use of color was important; 90 percent in both categories gave answers of four or higher.

Pict/D suffers low compactness of representation, and hence a lack of scalability. In a sample program given by the authors, a Pascal code segment with nine statement (not including declarations and begin/end statement) occupied the entire sketch pad region of the editor.

2.3.3. Other Graphical and Visual Programming Languages

The previous sections highlighted the work of several other researchers in graphical and visual languages. This section is organized differently, presenting brief remarks about a larger set of works, organized not by the specific works themselves, but by coarse categories into which they fit.

2.3.3.1. Languages for Novices

A large number of visual languages are oriented towards making programming easier for those unfamiliar with the art. The domain of programming for novices is not solely occupied by visual languages; textual languages exist in this realm, too, such as BASIC and LOGO. However, some believe that there is an inherent difficulty in learning programming when presented with a one-dimensional, textual language that uses many of the same words as natural language but with different and more rigorous meanings. A simple example is the use of the word "or" in formal (Boolean or computer) language and in natural language. In both cases, "or" applies to two statements but, in the former case, it results in truth if either *or both* statements are true, while in the latter case it typically means that either one or the other statement is true. It is subtle shifts in semantics like this that can cause continual confusion. Glinert and Tanimoto state it thus:

> *Those who wish to progress beyond the canned software state, however, discover that programming is painstaking work. Worse yet, learning to program is, for many, even more forbidding; indeed, the attempt is often eventually abandoned in frustration. ... Why do programmers — especially novices — often encounter difficulties when they attempt to transform the human mind's multidimensional, visual, and often dynamic conception of a problem's solution into the one-dimensional, textual, and static representation required by traditional programming languages? ... we believe it is time to take advantage of the human brain's ability to process pictures more efficiently than text. [Glin84]*

Visual languages for novices are related to visual languages that are aimed at increasing programmer productivity, whether that programmer is a novice or experienced. Distinguishing this category suggests another parameter in a multidimensional models, such as Shu's, for describing visual programming languages: scalability. Many visual languages are good tools for teaching programming in the small, for example, for understanding the subtleties in a solution to the Producer-Consumer problem or the Dining Philosophers problem. However, they often lack the ability to be usable for creating large, practical, general applications. Part of the problem causing low scalability in visual languages is their lack of compactness of representation; little is displayed in much space, though that which is displayed may be very clear. Pict, as described above, is one such language low in scalability. Another is BLOX [Glin86], a visual "building block" language based on Pascal. VIPS [Chen86], a "Visual Programming Synthesizer," is aimed at giving non-programmers a tool they can use to develop their own applications. PLAY ("Pictorial Language for Animation by Young People") [Tani86] is similar to Rehearsal in that it presents a stage and players metaphor to the programmer. It's orientation, creating simple animations, is similar to LOGO.

2.3.3.2. Systems for Displaying Concurrency

Concurrent and parallel programming systems are prime candidates for visual interfaces because of an inherent two-dimensional display potential. On one axis the sequential aspects of a computation can be displayed, and on the other the parallel aspects can be displayed. Diagrams of this sort are commonplace in descriptions of timing signals inside a clocked digital circuit, such as a computer, where time advances from left to right and different timing signals stacked vertically display concurrent activities. Synchronization in such diagrams is usually indicated by vertical dashed lines relating two or more signals.

Parallel programming languages are not new, though the application of visual display and construction techniques to them is relatively new. The literature contains many reports on parallel languages that do not use, but potentially could benefit from, a visual interface [Prat85, Jord85, Schw86]. Likewise, as described below, there do

exist parallel programming systems that use visual techniques. Once such system is Pigsty, previously described in this section. Other parallel languages and approaches (*e.g.* Babb's HEP data flow programming [Babb85]) use graphical representations but do not have a graphical interface.

An important aspect of parallel programming not present in sequential programming is the need to specify data dependencies among parts of the computation to state which pieces can be executed concurrently. The dependencies can be displayed graphically using a directed graph structure, where the nodes of the graph are the pieces of computation and the edges reflect the data dependencies. Given a natural representation such as this, visual programming languages can be devised that enable the user to state these dependencies visually. Once such system is CODE from the University of Texas [Sobe88]. CODE is based on a data dependency model of parallel computation, visually using an arrow from A to B to denote DEPENDS(A,B). Additionally, CODE uses dashed lines to represent "exclusion dependencies" which are predicates that place additional constraints on the invocation of a module. In CODE, the specification of the dependency relations is separate from the specification of the units of computation.

A similar system, SCHEDULE, from the Argonne National Laboratory [Dong86], also allows the user to interactively and graphically specify the data dependencies among a set of modules, this time, Fortran subroutines. The programming system generates a Fortran main program that is then compiled and run on a parallel machine. SCHEDULE adds post-execution display of an event trace of the program, showing how subroutines are dynamically scheduled. In SCHEDULE, an additional graphical notation is used to show that a particular scheduled subroutine may be further decomposed into parallel units.

2.3.3.3. Database Query Languages

Visually-oriented languages for accessing data stored in a database are similar to languages for novices in that they are principally oriented towards making a difficult computer-based task easier to use and faster to learn. In their study, Rockart and Flannery [Rock83] call for more support for "end-user programming," that is, providing the ability to allow non-programming computer users to write their own programs. They report that over half of the applications used by end users in their study involved the extraction and subsequent analysis of data from sources, though an amazing 34 percent was keyed in from other reports. Hence, there exists a special need for languages that allow end users to write their own retrieval and analysis programs for data extracted from databases. Visual languages oriented towards this task may be one answer. Two styles of visualization in database systems are common: forms and entity-relationship (E-R) graphs. Forms are common for queries and E-R graphs are good for visualizing the structure of data.

At times the term *visual query language* is applied to languages for creating queries into databases holding visual information, such as digitized photographs or maps (*e.g.* [Rous84]). Because the issues in those languages concern the way that a

query into visual information is best stated and not with visual queries themselves, those languages are not surveyed here.

FORMAL [Shu84, Shu85] is a forms-oriented language that stores all its data in two-dimensional hierarchical forms. The language primarily uses text laid out in two dimensions on a screen, and allows the user to design new presentations (forms) of data extracted from other forms (databases). FORMAL is similar to QBE [Zloo81] and FORMMANAGER [Yao84] in its visual aspects; the user creates a two-dimensional expression of a query into one or more databases.

Larson surveyed visual database languages [Lars86] and identifies four areas where visualization is used: (1) forms for displaying data, (2) request formulation by direct manipulation of the E-R graph, (3) query interfaces for novices, and (4) design interfaces for administrators. Larson describes a previous work of his own in category 3 — query interfaces for novices [Lars84]. In this system, syntax charts are used to describe the syntax of queries on the database system. The query interface consists of stepping the user through the query language by displaying the relevant syntax chart and allowing the user to point to the desired next-step in formulating the query. The syntax involves nested charts, so often additional charts may pop up in windows. Whenever a database object is needed in a query, the system pops up an E-R chart for the database, allowing the user to point to a particular entity or entity attribute.

2.4. Summary

Visual interfaces to programming languages and systems are becoming more and more commonplace. Visual languages have found application in describing parallel computations, in describing database operations, in teaching novices ways to program computers, in proving an intuitive interface to animation, and in numerous other areas. This section has skimmed the surface of the field, concentrating on important works and overviewing general areas of application.

The composition system of this report shares attributes with many of the systems described herein. In a sense, it provides an interface for novices in that almost all of us are novices at composing sophisticated networked applications. On the other hand, the composition system provides a capability not reasonably present in textual languages: non-linear composition, or the ability to create pipe-connected programs where each part has more than a single input and output.

Of all the extant systems surveyed, Pigsty is the most like our composition system, in that it allows the construction of concurrent or distributed programs that communicate and synchronize by passing messages. However, our composition system explicitly claims to accommodate heterogeneity, a claim not presented by Pigsty. The particular ability of our composition system to accommodate a network of heterogeneous machines is not an attribute of its visual interface *per se*, but it is an important attribute nonetheless.

The issue of scalability and compactness of representation is an important one in visual languages, perhaps the most important. Whether or not visual programming

languages can encode a sufficiently complex algorithm or interconnection structure in a cognitively manageable picture is an open question. We believe that our composition system provides a step in that direction, by composing strictly at a very high level, where each node of the program graph is potentially a highly complex operation.

3. MODEL FOR THE VIRTUAL MACHINE

3.1. Introduction

This section describes the operating system underlying the composition system. This operating system implements a *virtual machine* interface to programs above it. The virtual machine described here is an abstraction, described in terms of what operators it presents to those who use it.

High-level languages offer a way of programming a computer that is more powerful than programming the underlying machine. The job of a compiler is to translate programs from a high-level language into a language that matches the level of operations offered by the computer's hardware. This hardware usually implements a set of very low level operators, designed to move data among a set of registers and to perform arithmetic and Boolean operations. Often, though, the underlying machine is insufficiently powerful to implement all the features of the language, and it must be augmented by adding new, more powerful, operators. These extensions are what is called the run time library. Hence, the virtual machine that is the target of compilers is the hardware plus the run time library.

Whereas compilers target a low level machine, the composition system described in this report targets a high level machine. Many operators used by the composition system, such as network control and file management are not implemented directly by the underlying machine targeted by compilers. The composition system programs the operating system, just as the compilers program the hardware and run-time library. This distinction between compilers and the composition system is extremely important in this work.

3.2. Overview of the Virtual Machine Model

This section presents a model for a complete virtual machine and examines in detail portions of that model. The primary purposes of the model are to demonstrate that a computing system can be viewed as a coherent and hierarchical set of virtual machines and to clearly specify the nature of the virtual machine underlying the composition system.

It is important to realize that the model presented here does not reflect solely the software parts of a computing system. Rather, it incorporates and logically binds together all parts of the system, whether they are implemented in hardware or software. Some parts, such as arithmetic units, will by necessity be of hardware origin. Other parts, such as maintenance of the global naming directory, will be implemented in software. The model does not require that a sharp distinction between hardware and software be maintained.

3.3. Aspects of The Model

The virtual machine model describes the functions performed by that machine without giving details of the implementation. The model is useful because it frees the designer of the composition system from the constraints imposed by any existing machine, and calls attention to the functions required to support a composition system.

The model presented in this report employs the following principles:

- Hierarchical structure. The system is a series of interfaces. The uppermost interface (below the composition system) describes how programs and programmers use the system to perform work. The operation of the functions of this interface are described in terms of functions defined in lower interfaces. Continuing this description of interfaces downwards results in a hierarchical structure where each interface, or level, relies only on the operations of levels defined below it. We call this "functional hierarchy" because it describes the functional, or control, structure of the system.
- Object based. This model describes an object based system, meaning that data manipulated by programs running on the system are either atomic or are compositions of other data and that these data objects, regardless of type, can be named in a uniform manner in a high level language. These objects are aggregations of the data objects implemented by the underlying machine, and have a well-defined structure known only to those modules that allocate and manipulate them. This is an important aspect, that the structure of the objects is definable, that is, there is a representation of the structure, and this representation is only known to the code in an application that manages it. This is known as "information hiding."
- Type managers. The levels in the system are described as abstract data type managers: the data types are abstractions and managed by a module in the system. Each level manages just a few related object types. Type managers are packages of functions that implement all the operators on a particular data type. Traditionally, the term "abstract data type manager" further implies that the internal representation of the objects managed is not available to routines outside the type manager: it enforces information hiding.
- Capabilities. Objects in this system are described and manipulated by using capabilities, a concept originally proposed by Dennis and Van Horn [Denn66]. Capabilities are pointers that refer to objects; they encode the data type of the object, the operations (of the type manager) that can be performed on it, and an identifier that can be mapped uniquely to the storage location of the data object. In a distributed computing system, capabilities must be interpretable on any node of the network by virtue of encoding the machine identifier responsible for managing the resource identified by the capability. Our model distinguishes between two types of capabilities: global and local. Global capabilities uniquely identify an object in the distributed system, encoding its location. Local, or open capabilities, are only interpretable on a single machine. The create operations in the type

managers return global capabilities and only global capabilities can be stored in directories. The open operations in type managers convert global capabilities into open capabilities.

3.4. Summary of Past Work

Past work relevant to this model includes studies of hierarchically designed systems, which will be summarized briefly. This is not an attempt to provide a thorough review of the literature as was attempted in Section 2. Dijkstra's THE system, Liskov's Venus, and SRI's PSOS inspired this design. The layering in Comer's XINU operating system [Come84] was partially inspired by this model

3.4.1. Dijkstra's THE System

One of the first published reports of a hierarchically designed system is on Dijkstra's "THE" system [Dijk68]. This description is included here because it is an example of a simple and elegant level-based operating system. An understanding of this system will help the reader make the leap to the much larger model presented in this report. Each level in Dijkstra's system is a type manager for a particular object or resource. His levels, depicted in Figure 3.1, are as follows:

- Processor allocation. This level implements the processor scheduling algorithm, services interrupts, and handles process synchronization. Above this level, processes deal only with virtual processors.
- Memory allocation. This level handles primary and secondary memory pages and assigns them to program pages, which Dijkstra calls "segments." Above this level, processes deal only with segments and need not be concerned with their placement in primary or secondary memory.
- Console management. This level handles the passing of messages from processes to the system console and from the system operator to a process. Because this level is above the memory manager, it can reside in "virtual memory" and hence be moved to and from secondary store upon need. Above this level, processes can behave as though they had their own private operator's console.
- Device management. This level has the device driving processes that provide buffered input and output to higher level processes. Dijkstra justifies placing this function above the console layer because these processes must be able to communicate with the operator in the event of equipment failures.
- User processes.
- The operator. Dijkstra does not elaborate on the operator's view of the system nor the language used to interact with it.

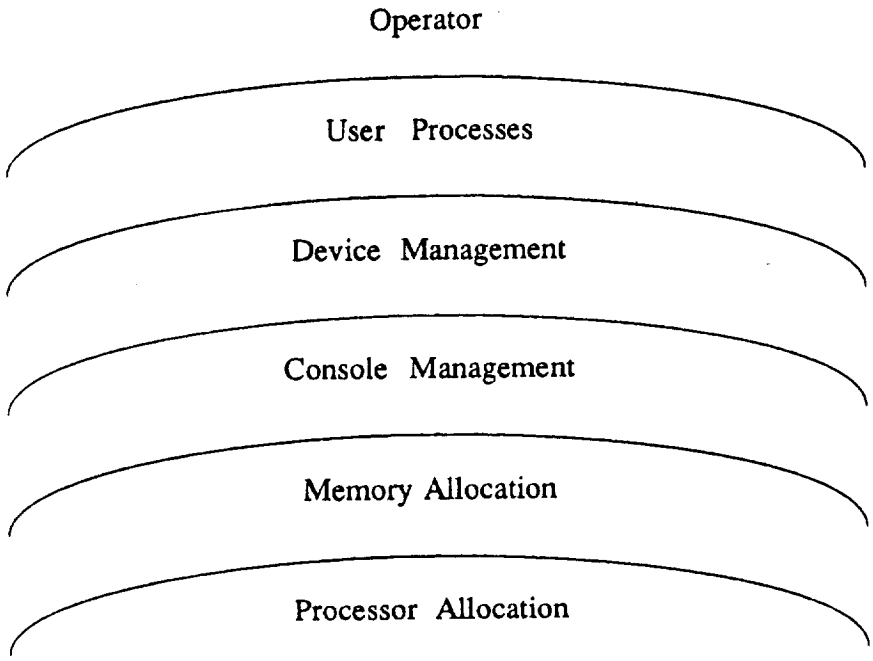


Figure 3.1: Levels in the THE Operating System

Dijkstra claims that one of the primary benefits of using this design methodology was the ease of testing. Because the system is layered, he was able to implement level zero, test it thoroughly, add level one, test it thoroughly, and so on. He asserts:

> *It seems to be the designer's responsibility to construct his mechanisms in a way — i.e. so effectively structured-that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation.*

He is so confident that his technique is sound that he states:

> *At the time this was written, the testing had not yet been completed, but the resulting system is guaranteed to be flawless.*

The THE operating system is important because it was the first system designed and built as a set of type managers.

3.4.2. The Venus Operating System

The Venus system [Lisk72] is another example of an early operating system utilizing hierarchical structuring techniques. The system description, however, does not show it to be strictly hierarchically structured. One of the purposes of the Venus project was to study the effect of the underlying machine architecture on the design of the operating system. To this end, the Venus researchers started by extending the

basic machine, an Interdata 3, by adding micro code to implement at least three type managers, one for segments, one for virtual devices, and one for virtual machines. What differentiates Venus from THE is that the type managers are not only described by the operations they perform, but by the objects or resources they manage as well.

The segment level in the microcode of the Venus machine maintained a central mapping table to convert memory references into physical addresses. Memory on Venus was both segmented (64 kilobyte maximum segment size) and paged (256 byte pages). When a page fault occurs, the microcode ran a software routine to resolve the reference. When that software routine completed, it executed a special instruction, EL1, which returned control to the microcode routine where the fault was originally detected.

The primary object maintained by the virtual machine level was a primitive process, sixteen of which could exist on the machine. Each process had its own registers and program counter. The instructions implemented in this level include the semaphore P and V operations, defined by Dijkstra [Dijk68]. Hence, the system scheduler was implemented in this microprogrammed level and the semaphores were used for synchronization and process switching.

The higher levels of Venus, above the microcode, managed dictionaries, queues, and system resources. Dictionaries mapped external names (used by the programmer) to internal names (used by the segment handler). Because the dictionary handler existed above the segment manager in the system, dictionaries could be stored in data segments; in fact, they were. Associated with each dictionary was a semaphore used for managing concurrent updates. Hence, the dictionary manager had all the essential parts of a monitor.

Queues in Venus provided a mechanism whereby one process could send information to another. Though the exact nature of Venus queues is not important here, what is significant is that the queue manager used a dictionary to hold the head pointer for all existing queues. Hence, the operations on queues were composed in part of operations on dictionaries. Likewise, one of the objects managed by the queue level was the map from queue names to their head pointers and this object was composed in part of a dictionary. This is an example of both functional and data composition and hierarchy.

The major contribution made by the Venus system is that in a hierarchically structured system, whether a function is implemented in hardware, microcode, or software is of little consequence. What is important is that each level manages a specific object, referred to as a *resource* in Venus. Unlike many later systems, and the model presented in this report, Venus was designed by first extending the abilities of the basic machine and then building an operating system on top of that. Another major contribution of Venus is that the objects managed by a specific level may be compositions of other objects defined at lower levels. Dictionaries are stored in and refer to segments. The segments themselves are managed at a lower level then the dictionary manager and are composed of elementary machine bytes.

3.4.3. The Provably Secure Operating System

The Provably Secure Operating System (PSOS) [Neum80] developed at SRI International is a 17 level hierarchically designed operating system based on capabilities for referencing user objects. PSOS is much more extensive than our design presented in this section, and seeks to specify all aspects of the operating system in a way that allows for the proof of a set of multilevel security properties.

The prime contribution of PSOS is that it is a thorough operating system, formally specified and hierarchically designed. The design methodology, named HDM, uses four stages for the development of a system: definition of the interface, hierarchical decomposition, module specification, and implementation. The specification stages use a formal specification language named SPECIAL.

3.5. Details of the Virtual Machine Model

Our virtual machine model, like most models, can be thought of as specifying or describing a family of systems, each member resulting from completing the design in a slightly different way from the others. The primary feature of the model is its hierarchies of type managers.

The advantage of structuring a system in this way are as follows:

1) The hierarchical design allows the interfaces to the type managers to be specified separately. Guidelines on how to decompose a high level interface specification into modules and lower level specifications is described by Parnas [Parn72] and Yourdon [Your75].
2) During implementation, the system can be built, one level at a time, from the lowest level upwards. Each level can be tested individually with less concern over the previously tested levels and no concern for the unimplemented levels.
3) If formal verification techniques are used, the effort required to prove the correct operation of a large system only grows linearly with its size [Spit78].

In THE, Venus, and PSOS, the hierarchy is dictated by functional composition of the system. This means that the description of the modules and interfaces between them is the primary hierarchy of interest. In our model, a similar hierarchy exists, along with another resulting from the fact that each level in the system is implemented as an object manager. This is the object hierarchy and it demonstrates how the objects managed by the system form a hierarchy of composition. The functional composition further reveals a higher-order classification of objects: the metatypes. Each of the three metatypes refers to a group of object-types addressed in a similar fashion. The metatypes in our model, represented in Table 3.1, each have different purposes in the system. They also have different ways of being handled and described by user processes. Elements of the metatype set correspond to the frequency of access of objects in a computer system. The elementary types, because of their nature, are accessed frequently. Hence, the method of accessing them must be fast and easy. The system types are less frequently accessed, but because they are application independent, are accessed frequently enough to warrant special treatment.

The user-defined types, being application dependent, may be accessed least frequently of all.

The simplest metatype is called *elementary*. Elementary types are strongly related to the primitive types available in most modern computers: numerical scalars (binary or decimal integer, fixed point, and floating point), address values (pointers), and capabilities. Processes handle several elementary types by value as they are small enough to be held in machine registers (if the machine has registers) and be passed as units between procedures. Other elementary types are referenced by virtual memory address. Capabilities are elementary types in our model because they must be handled with some of the same operations as the other elementary types. This includes the ability to copy them between memory and registers, pass them as parameters to procedures, and store them in long term files. One design [Denn80] goes so far as to specify special machine registers to hold capabilities. Additionally, all the objects in the next category, the system types, are referenced by capabilities. Therefore, capabilities cannot exist in that category themselves.

The next metatype is called *system* types because they are special enough to warrant special treatment in this model. This class of metatypes is the most important one with regards to the composition system, because it is through objects with types in this class that program parts communicate. System types are application independent and pertain to all user processes. For example, the model has a dictionary or directory for converting external, or user names for objects into internal, or system names used for locating the objects. The use of such a directory is so pervasive in modern systems that it is included as a basic part of our model. Another such system type is the input/output device. Because devices exist on all computing systems, including the modules that manage them as part of the operating system is compelling.

In our model, system types are managed by type managers, similar to Ada packages [Luck80], that form the main body of the upper machine levels of the model. System types are invariably referenced by capabilities and their internal representation is not available to user processes. Additionally, in a distributed system based on our model, the system types may be transparently distributed throughout the network.

Table 3.1: The Types Hierarchy

Metatype	Referenced by	Examples
elementary	value or address	scalars, pointers, capabilities
system	capability	files, channels, extended types
user	extended capability	user defined

The most sophisticated element of the metatypes is the user-defined types. Our model includes support for allowing user processes to define and manage new object types not handled by the system, but does not elaborate on the mechanism used to implement this. These user types are typically application dependent and so are not part of the basic model. An example of a user-defined type is a graphical object display list, in which case the type manager might include operations to extend the display list, alter transformation matrices within the list, or cause the display list to be visualized.

User processes reference instances of extended-type objects by a special extended-type capability. The extended-type managers themselves are referenced by standard capabilities. This allows for a type manager defined within the virtual machine for allocating values from the space of extended-type types. The mapping from extended capability to object, however, is left to the user-defined type manager and is not specified in this report.

3.5.1. Object Hierarchy

The object-type hierarchy typically exists in all systems, though not always as a strict total-ordered one. The hierarchy-forming predicate R(a, b) for object types is *contains*. The relation *contains(a, b)* is true if and only if the description of object type *a* includes an instance of an object of type *b* and the consistency assertions about the state of *a* are only correct if the consistency assertions concerning the state of *b* hold. Notice that this definition closely parallels the "USES" predicate defined by Parnas [Parn74].

An example of an element in the object type hierarchy is a process object. Though the details are explained later, in our model, a process consists of many parts, one of which is a set of input and output streams. These streams are in fact defined by capabilities as part of the process object and they may refer to files, devices, or communications ports. One of the assertions about a process object is that its input capability references an object that allows reading. Hence, if the assertions about the input stream (device, file, or communications port) fail, the assertions about the process state fail, too. It is not coincidental that this example hierarchy reflects a similar hierarchy described in Section 1 — that of a program part consisting of part-specific data and sockets. This analogy results from program parts being static instances of what are later dynamically instantiated as processes, and the sockets become capabilities for I/O objects.

3.6. The Model

The primary description of our model centers about the functional hierarchy, presented top-down in Table 3.2. Each level in Table 3.2 is the manager for a set of objects of given type; each level provides operations for creating, deleting, and changing the states of objects. Levels 1-8 implement the managers for the elementary objects on each machine. Levels 9-14 implement the principal system objects provided by the model; most are sharable among all machines of a distributed

Table 3.2: The Full Functional Hierarchy

Level	Name	Objects	Example Operations
15	Composition System (shell)	User Programs	Creation of composed programs
14	Extended Types	Extended type objects (from programming language)	create typeMark, register server
13	User Processes	Processes	create, kill, suspend, resume
12	Directories (name service)	Directories	create, destroy, attach, detach, search, list
11	Devices	I/O devices: printer, keyboard, display, *etc.*	create, destroy, open, close, read, write
10	Long-term storage	Files	create, destroy, open, close, read, write
9	Communications	Links	create, destroy, open, close, read, write
8	Capabilities	Capabilities	create, validate, attenuate
7	Virtual memory	Segments	read, write, fetch
6	Local secondary storage	Blocks of data, device channels	read, write, allocate, free
5	Primitive processes	Primitive process, semaphores, ready list	suspend, resume, wait, signal
4	Interrupts	Fault handler programs	invoke, mask, unmask, retry
3	Procedures	Procedure segments, call stack	markStack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalars, arrays	load, store, unaryOp, binaryOp, branch, arrayRef, *etc.*
1	Electronic circuits	Registers, gates, busses, *etc.*	clear, transfer, complement, activate, *etc.*

version. The horizontal lines in the table indicate the division points between sections that manage different metatypes. This section does not elaborate on levels 1-8; they are documented elsewhere [Brow84, Denn84].

The levels must conform to two general rules:

- Hierarchy. Each level adds new operations to the machine and hides more primitive, lower-level operations. The set of operations visible at a given level form the instruction set of an abstract machine that can be used to program operations at that level. Hence, programs can invoke visible operations of lower levels but no operations of higher levels.
- Information Hiding. The details of how an object of given type is represented or where it is stored are hidden inside the level responsible for that type. Hence, no information can be changed in an object except by applying an authorized operation to it.

The description that follows is an overview of the upper levels of the model, based on Table 4.2. The model resulted from starting with a knowledge of what the highest level should look like and then successively decomposing that description into levels of less and less abstraction.

3.7. The Multi-machine Levels: 9-15

Levels 9-15 are called *multi-machine levels* because they manage objects that can be shared among the machines. Operations familiar from single-machine operating systems must be carefully evaluated for extension to multi-machine systems.

Hiding the maps between names and objects is a central principle in many third generation operating systems. It is responsible for the machine independence of the user environment. To extend the principle for multi-machine systems, the design must hide the locations of all sharable objects (*i.e.*, channels, directories, files, devices, user processes, and extended types). This requires the solution of three problems: reliable exchange of information between processes on different machines, global naming of objects, and efficient access to objects. The first problem is solved by the *communications level*, the second by the *directories level*, and the third by *distributing the interpretation of capabilities*.

Level 9 provides a single mechanism, the channel, for reliably exchanging information between processes, independent of whether they are on the same or different machines. A channel appears to its users as a queue of memory segments. The create operation returns a global capability for the channel, which can be subsequently opened for reading or writing, but only once for each. Once opened, processes use an open-channel capability. The level 9 type manager enforces a rendezvous between opens for reading and writing on a single channel because each open may come from a different machine and each, the reader and writer, needs to know where the other end resides. Channels have properties similar to pipes in UNIX

[Ritc74] and ports in iMAX [Kahn81]. Level 9 is the lowest level at which the communications level has access to the functions of the host machine needed to meet its reliability requirements [Denn83, Come84].

Level 10 is a file system extended so that it can open files that may be stored remotely. Level 11 provides access to other devices such as printers, plotters, terminal keyboards, and terminal displays; it is extended to use channels to connect to devices on other machines. A standard interface for opening, reading, writing, and closing all files, devices, and channels exists, implementing device independence, or more correctly, media dependence.

Level 12 provides a global directory tree structure and a mechanism for ensuring that portions cached at each machine are consistent. Each entry in a directory contains a name, access list, and a capability. This level can find a directory given a directory capability, but is not responsible for locating any other object.

Level 13 implements user processes, which are virtual machines containing programs in execution. A user process includes a primitive process, a virtual memory, a current directory pointer, and parameters passed on invocation. User processes should not be confused with primitive processes (level 5). Process invocation semantics include the ability to preallocate capabilities for the input and output channels used by the program. It is a fundamental attribute that when the shell, level 15, invokes a program in a new process, that it have the ability to tailor the set of I/O capabilities for that program. The precise mechanism for passing these capabilities into the programs, however is not specified and may be language dependent. Some languages may use capability lists, in which case, the programs can reference the I/O capabilities as small integers serving as indices into the list. In other cases, the I/O capabilities may be placed in a well-known location when the program is started, such as in a procedure activation record expected by the main program.

Level 14 is an extended types manager. It creates protected type-marks and instances of objects of each type. Because it utilizes the same capability validation mechanism as in level 6, a procedure call is no more expensive for extended type objects than for other system operations. Unlike Hydra [Wulf81] or PSOS [Neum80], our model places type extension close to the user level. This supports the efficiency principle: since we do not have to deal with the most general case deep inside the operating system, we can use standard implementations for common system objects.

Level 15 is the shell, the interpreter of a high level language making up the user interface. The composition system in this report is an example of an implementation of a level 15 shell. Other shells may exists as well. Typical user-written application programs reside at level 15, also, unless the applications are built above user-defined type managers. Pictorially, the top of the virtual machine is as in Figure 3.2, showing that a text-oriented shell, the composition system, and user applications all reside at level 15.

3.8. General Comments on Level Structure

The level structure is a hierarchy of functional specifications. The purpose is to impose a high degree of modularity and enable incremental verification, installation, and testing of the software.

A program at a given level may directly call any visible operation of a lower level; no information flows through any intermediate level. The level structure can be completely enforced by a compiler, which inserts procedure calls or expands functions in-line [Habe76]. It has been used in, among others, an efficient operating system, XINU, for a small distributed system based on LSI 11/02 machines [Come84].

The level structure discussed here should not be confused with the layer structure of network protocols [Tane81]. In network protocols, information is passed down through all the layers on the sending machine and back up through all the layers on the receiving machine. Each layer adds overhead to a data transmission, whether or not that overhead is required. Models for long-haul network protocol structure may not be efficient in a local network [Pope81]. All these ideas are embodied in Level 9.

No one level is capable of efficiently hiding the locations of all objects in a distributed system. For example, the file system must know whether a given file is local or not to perform the most efficient read and write operations. The device level must know the machine to which a given device is connected. The user process level must know whether a given process is to be spawned on the current machine or another. A single central mechanism cannot do all this efficiently. Accordingly, we are led to the principle that each level is responsible for hiding the location of the objects it manages.

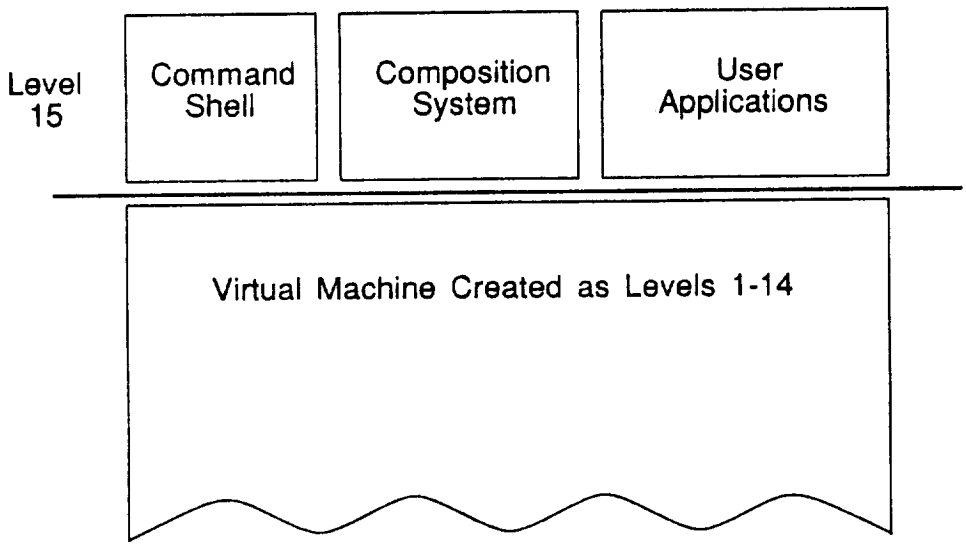


Figure 3.2: Upper Levels of the Machine

4. FUNDAMENTALS OF PROGRAM COMPOSITION

4.1. Introduction

This section describes the important attributes of composed programs and the program composition system (PCS). PCS is a subset of the larger system, the distributed PCS (DPCS) and concentrates on those components that are only concerned with program composition. Many of the aspects of PCS described in this section reflect decisions made during the creation of a prototype composition system. Those decisions, for the most part, were not made with ease of implementation in mind, but with clarity of abstraction as the prime consideration. However, because a real PCS prototype does exist, and the best abstractions often lose their purity once implemented, the material in this section tends to track a middle ground between the ideal and the practical, wavering closer to the abstract than the concrete. Herein is described what composed programs are, what their components are, and what considerations go into constructing them.

4.1.1. Review of Pipe-connected Programs

The program composition system allows the construction of programs that consist of computational parts and communication paths among them. We call the abstract form of such programs *representation-transformation* (RT) graphs because they exhibit a graph structure with two types of nodes: data representations and data transformers. Edges in RT graphs associate data transformation nodes with the data representations they produce and consume. A simplification of RT graphs eliminates the representation nodes and uses directed edges to represent data passing from one transformation node to another. RT graphs reduced in this way are often called *data flow graphs*.

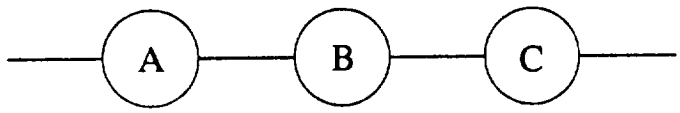
PCS generalizes on the linear program graphs that can be constructed using the UNIX shell by allowing graphs with nodes having an arbitrary number of in-edges and out-edges. Figure 4.1 shows examples of linear and nonlinear program graphs. Nonlinear composition is useful for a variety of applications. For example, the UNIX programs *join* and *comm* take two inputs and produce one output. Using these programs from the UNIX shell requires the use of temporary files, the very thing that pipe-constructed programs seek to avoid. Another example is the program *comp* in the Utah Raster Toolkit [Pete86]. It combines two images into a third. A typical image compositing script has several *comp* steps in it, each requiring a temporary file to hold one of the inputs. In Figure 4.1, the program C in the nonlinear composition example may be one of these programs requiring two inputs. Additionally, linear composition does not allow two programs to communicate with each other, as is the case in traditional client/server and coroutine models.

Likewise, programs exist that have more than one output. An example from UNIX is *tee*, a simple program that splits its input into two identical outputs, but only one of them can be a pipe. Another example is the popular program *awk*, a string oriented

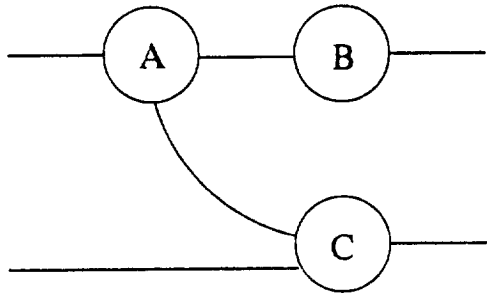
language with syntax very much like the C programming language. Awk allows multiple outputs but, again, only one can be a pipe. If awk would allow the programmer to direct outputs to multiple pipes, and if it could be used in a non-linear composition language, then it could be used as a general-purpose router of textual data. In Figure 4.1, program A in the nonlinear example could be this multiple-output awk, directing output matching one pattern one way and that matching a second pattern another.

This attribute of programs, having more than one input and output, is not an artificial artifact, but an essentiality that extends the capabilities of the programs and provides a strong motivation for the data flow graph representation of a network of programs. Only very simple networks can be constructed from programs or functions that have at most one input and output, whereas by extending that number from one to two, arbitrarily complex program networks can be constructed. Extending the number of inputs and outputs beyond two does not add any significant functionality, but it can, in many cases, simplify the structure of the network and allow it to more graphically reflect the nature of the computation.

This section describes the structure of the type of pipe-constructed programs used herein and elaborates on the primary abstractions used in the program composition system, namely parts, sockets, links, and the boundary. These abstractions are not independent, because sockets do not exist without being associated with either a part or the boundary. Within the PCS, a program is described fully by the triple formed of the parts (with their associated sockets), the boundary



Linear Program Composition



Nonlinear Program Composition

Figure 4.1: Linearly and Nonlinearly Composed Programs

(also with sockets) and the links. Hence, a composed program can be described as follows:

Program = (Parts, Links, Boundary)

This definition is recursive because parts may be composed programs. In brief, a part is a program with sockets and descriptive information, links connect socket, and the boundary is a set of external sockets. These abstractions are elaborated upon in this section.

4.2. Semantics of Composed Programs

Program parts composed together in the way just described intercommunicate and cooperate toward the solution of a computational problem. The means by which they communicate and the way they cooperate form the semantics of these programs. The internal semantics of the parts themselves is a separate concern, not directly addressed here.

Each part in a composed program is an autonomous unit of computation whose internal structure is not strongly influenced by being involved in a larger PCS application. When the state of a part becomes such that it needs the services of another part, or its services are required by another, it performs a communication operation through the operating system to pass data to or receive data from another part. The method of communication and scheduling is not specified by the composition system; that is entirely an attribute of the individual parts.

An early work describing a similar application of communicating processes for specification of parallel computation is presented by Kahn and MacQueen [Kahn77]. Much of their concern, however, dealt with the scheduling of composed programs of this sort on a uniprocessor, an important aspect of their system because they support dynamic creation of new program nodes. In PCS, no assumption is made about the nature of the processor(s) that run the code in the primitive parts other than it offers the virtual machine interface presented in Section 3. In one implementation, the virtual machine may be a time multiplexed real processor, as in contemporary uniprocessor timesharing systems. In another implementation, each virtual machine may use a real processor in a multiprocessor system where each processor shares a common physical memory. And yet in another implementation, each virtual machine may be a distinct processor only connected to the others by way of a communications network. Of these three possible implementations, the last both matches the model of composed programs most closely and presents the most complicated set of problems. If, in the networked processors case, the machines differ in program and data type representation, the composition system is obliged, by the ease of use requirement, to handle the primitive part location and data conversion management issues.

Two important aspects of the semantics of composed programs are the form of the data passing through the links and a set of interconnection primitives that form the basis for constructing large applications. Even though PCS does not enforce any

particular communication and scheduling abstraction, some common ones do exist that are useful in different circumstances.

When a composed program is executing, each primitive part runs as a process on a distinct virtual machine having, in our virtual machine model, a single thread of control. The state of a process running a part may be such that it is either waiting on a message from one of its input sockets or it is computing and not yet ready to receive a message.

4.3. Message Formats

Parts communicate by passing data among themselves across paths called *links*. The unit of communication is called a *message*. All data moving into and out of a program part passes through a socket object on the part. Associated with every socket is a data type, a static description of the type of data expected by the socket. The designer of the program part determines the data type; it is associated with the part description in the parts database. The representation of the typed data is a run-time attribute, and the alternatives are described here.

Data passed through message links can be either structured or unstructured. With unstructured data, only the sending and receiving processes places any interpretation on the contents of messages. One advantage of unstructured messages is simplicity; each part can consider the data it receives as simply a block of data words copied from the memory space of one process to the memory space of another and the interpretation depends only on the coding and state of the receiving process. Another advantage of unstructured streams is efficiency; no extra software is required to manage the structuring information that must accompany structured messages.

Unstructured data streams have a disadvantage in that the content of a message is placed in the memory space of the receiver and then the receiving process uses that data in a way that is entirely dependent on its own internal state rather than on the content of the message itself. Hence, the validation of messages is left to the programmer; no support is provided by the underlying system. Problems can occur when a receiver is in a state where it expect a message of one particular type, and one of a different type arrives.

Unstructured messages have a further disadvantage when used to pass messages between processes running on separate computers having different native data formats. For example, consider a message containing an integer followed by zero or more floating-point numbers, the integer stating how many floating point numbers follow. If such a message is passed between processes having different byte-ordering for integers and different floating-point formats, the receiving process will be unable to use the message unless is knows how to interpret the data representation formats of the sender.

Associated with structured messages is information that states the types and representations of the data. This additional information may be encoded in the message itself, in which case the message data is said to be *self-identifying*, or it

may be explicitly associated with the programs that send and receive the data. When the data description is not in the message stream, there is usually a *data description language* (DDL) or *interface description language* associated with the socket. The DDL states, in a formal way, the structure of the message and the data types of its components. DDL specifications must be supplied by the programmer, thus adding to the programming effort, and are static at run-time.

Self-identifying message formats encode the structure and data types of the messages in the messages themselves. This information can be added at run-time, without special assistance from the programmer, if the compiler generates the structure and type information and places it in the language's input and output primitives.

Both predesribed (via a DDL) and self-identifying message streams have the advantage that, when passed between machines with different data representations, there is sufficient information available to the communication system to translate from the sender's representation to the receiver's. This is an important consideration in DPCS. With predefined message types, the type is associated with the socket and that information can be used by the DPCS program development environment to assure that the programmer does not try to link together sockets that have different abstract data types. Self-identifying types are more flexible, though, because the data types placed on the link are not determined until run-time.

For DPCS, we chose a hybrid approach. Sockets have associated with them a data type name that allows the development environment to perform type checking when the programmer creates a link. Also, we use self-identifying data streams so that automatic representation conversion can be performed at run-time.

4.4. Communication Structures

This section describes communication structures that represent possibilities as fundamental "building blocks" in composed programs.

4.4.1. Procedure Call Emulation

Procedure calls are a common technique for program composition in high level languages. The semantics of a procedure call are (1) the calling procedure makes a request of another procedure, passing it data it needs to satisfy the request and the caller stops execution. Next, (2) the called procedure incorporates the input data into its namespace and (3) computes on behalf of the caller. When the callee completes, it (4) returns a result to the caller, who (5) then resumes execution. The called procedure's execution is temporally bound between the time it is called and the time it returns a result, and it restarts its execution each time it is called.

The behavior of procedures can be emulated with messages passed over a communications channel. Using the graph notation of previous figures, Figure 4.2 depicts the interconnection structure. Part A runs autonomously, and occasionally requires the services of part B, using a procedure call-style interface. Meanwhile,

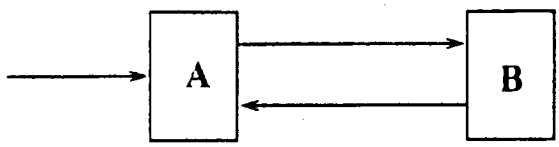


Figure 4.2: Procedure Call Emulation

when B starts execution, it waits on an input from its input port. Now, A requires service from B, so it sends a message encoding the request parameters to B and immediately waits for a reply. B, upon receiving the request message, computes based on the parameters, returns a result and then cycles back to its beginning to wait for another request. At this point, A consumes the result and proceeds. A distinction between procedure calls and our emulation is that procedures usually do not retain state between calls, even though the mechanism to do so exists in several languages. When emulating pure, non-state-retaining procedures, the called-on part must explicitly reset itself to an initial state each time it returns a value.

Remote procedure call uses a similar protocol, since the caller does not (necessarily) share an address space with the called procedure in which to store an activation record, and so must pass the activation record to the called procedure in a message.

4.4.2. Coroutine Emulation

Coroutines, originated by Conway [Conw63], are similar to procedure calls except that the called procedure is not restarted from its beginning each time it is called. Instead, once it returns a result, it stops and waits, retaining its state, until another request arrives. This behavior can also be easily emulated with messages using the same scenario as with procedures, except once the called routine (B) returns a result, it stops and waits for another request without restarting itself. Here, the state-retaining property of parts is necessary. The interconnection structure for coroutines is the same as for procedures (Figure 4.2). The popular server/client model of computing is an example of coroutine interconnection.

Coroutines offer the possibility of concurrent execution of the routines involved. For example, once the called routine returns a result, it need not immediately stop and wait for the next request; it can continue execution. In the cases where the virtual machine is implemented on multiple processors, this scenario offers true concurrent execution of pieces of a program. When coroutines do not stop and wait on each other synchronously, their communication behavior mimics the rendezvous mechanism of Ada.

4.4.3. Pipelining

Pipelining is a more general case of concurrent coroutines than the example given in the last subsection. When two routines, A and B, are pipelined together, the sense

of who is the caller and who is called almost disappears. The situation is like a traditional producer-consumer model, where one routine, A, produces data required by B. Pipelined routines can run asynchronously and the semantics of communication are as follows. When A has completed the production of a piece of data required by B, it sends it to B in a message and proceeds. Meanwhile, once B reaches a state where it needs another piece of data in order to proceed, it waits on a message from A. Upon receiving data, B proceeds. Hence, B can be working on its latest input at the same time that A is working on the next one. Figure 4.3 shows the interconnection structure of a pipelined computation of two routines.

Pipelining can be extended to an arbitrary number of routines, where each routine is working on the piece of data most recently completed by the routine before it in the pipeline. In such a case, each routine except the first and the last is both a producer and a consumer. This style of computation is analogous to a production line in an assembly plant, where each worker performs a specialized operation on a piece of equipment (computation) and then passes the partial product on to the next worker (routine) in the line.

As in production lines, pipelined computations can suffer from bottlenecks, where one worker (routine) takes much more time to complete its work than the others around it. If the slowest routine is continuously slower than the others, work will build up in the pipeline before it, and routines after it in the pipeline will spend time waiting on input, producing results only at the rate of that slowest routine. On the other hand, if the time it takes a routine to complete its work on an input is highly variable and is only sometimes longer than those around it, then speed matching can be achieved by placing buffers between the routines so that the communication link itself can hold data in transition from one routine to the next.

4.4.4. Classification

Nonlinear construction allows demultiplexing, or classification parts to be included in program networks. Classification parts read from a single input port, classify each input object, and then route the objects to one of several outputs depending on the value of some part of the data. Figure 4.4 shows the interconnection structure of a three-way classifier, program A, that splits its input into streams to programs B, C, and D. Classifier schemes can become complicated when the demultiplexed stream of objects must be reassembled in proper order later in the program.

Figure 4.3: Pipelining Interconnection Structure

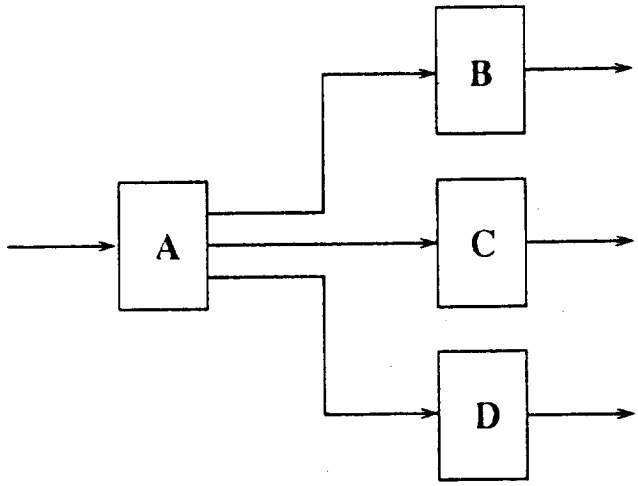


Figure 4.4: Classifier Interconnection

4.5. Structure of Composed Programs

The product of the program composition system is called a *composed program*, and the concern of the composition system is providing the ability to create composed programs by combining existing computer programs into a larger system of programs that communicate with each other in ways defined at the time the composed program was written. Composed programs have components named *parts*, *links*, and the *boundary*. Additionally, every program has a name and annotation, identical to that described later in the description of program parts. Program parts and the boundary consist of descriptive information and *sockets* which contain the descriptions of the interconnections of these elements to other elements. A composed program is a set of parts and a single boundary plus a set of links that describe the interconnections of the sockets belonging to parts and the boundary. Each of these components are elaborated upon individually.

There are three types of attributes associated with components of composed program: static, display, and dynamic. The concern of this section is principally the static attributes. The others are left for subsequent sections and sections. The static attributes are those that exist as a part of the definition of the program; dynamic attributes are those that exist only when the component is included in a composed program or when that program is in execution. Display attributes are those the concern the visualization of the program in the graphical editor.

4.5.1. Parts

The unit of computation in the composition system is the program part, usually called just *part*, corresponding to a node on a program graph. Normally, the part is the finest level of detail at which the PCS user deals with the description of how a

computation takes place. There are two types of parts, primitive and composite. Parts have numerous attributes and associated substructures, some of which are concerned with the semantics of the part, others of which are concerned with its view or syntax, and some have meaning in both domains.

Another way of describing parts takes into consideration the analogy between program composition in PCS and a design process. Parts are the units of design, without which no composed program could exist. They are similar to process components in SDMS [Hagi84] though in PCS the term *process* is not used in the static description of a composed program. Constructing a new program using the composition system entails selecting program parts and then interconnecting them with links. Such a composed program must contain at least one part.

Program parts in PCS are either *primitive* or *composite*. Primitive programs are *atomic*, that is, they cannot be further decomposed into other parts within the composition system. Such parts correspond to the traditional sense of computer programs and may be derived from a variety of sources, thus satisfying the existing software requirement described in Section 1.

Within PCS, parts comprise numerous pieces of static, dynamic, and display information. The static information, describing what the part is and how it can be fit in with other parts in a composed program, is as follows:

its name,
its module name,
the set of its sockets,
its initial argument list.

Parts have names that describe their function. There may be several namespaces for part names; corresponding to different computational domains. For example, if the domain of application is graphics, part names may be such things as "Animate," "Render," "Ray Trace," and "Digitize." In fluid dynamics, on the other hand, names may be such as "Generate Grid," "Convert Airframe," and "Boundary Conditions." There are no rigid rules for the selection of part names, only an informal rule that they be descriptive and unique within a specialized domain namespace.

Because the composition system interconnects parts by attaching communication links between them, it must know *a priori* what input and output connections are possible on a part. We call the information about a single point of connection on a part a socket. The set of sockets associated with a part uniquely describe its input and output connections. Sockets are more fully described in the next subsection.

The initial argument list for a part is passed to the createProcess routine in level 13 of the model operating system. The precise semantics of the argument list are system and part dependent.

The module name is used when the part is invoked, and is described in the invocation semantics section later in the section.

The characteristics of primitive parts necessary to allow them to join into a system of programs composed with PCS are few. Principally, because the relation that joins programs in a PCS-composed system is communication, primitive programs must be constructed so that they receive their input data by reading using a capability that supports reading, and they write output data by writing to a capability that supports writing. What type of object these capabilities reference is immaterial; they can refer to files, links, or devices as managed by the upper levels of the model operating system. Primitive parts are not restricted to perform all of their input and output by way of sockets; they may use whatever other facilities are available to them for acquiring input and routing output. For example, they may utilize dialog boxes on the workstation screen (if appropriate) for input and output. Not being constrained to accessing input and output resources made available by the composition system is an important attribute: The specification of the interconnection of program parts is independent from the construction of the parts themselves. This doctrine was also observed by Sobek, *et al.*, but for constructing parallel programs [Sobe88]:

> *A unit of computation at one level of abstraction may itself have an arbitrarily complex sequential or parallel structure without this structure impacting the relationships between this module and the balance of the computation structure at the higher level of abstraction. The computation modules (units of computation) may thus be arbitrary programs in any high level language supported in an execution environment or may themselves be complex parallel computation structures.*

Composite parts, as they appear within the composition system, are indistinguishable from primitive parts. They, however, are not atomic, but are other program networks created with PCS. The implication of composite parts is that once a program has been composed using PCS, it can be reduced into a single program part and then included in another program network, which in turn can be reduced to a single part, and so on. This mechanism allows a higher level of program abstraction than provided if only primitive parts can be used.

Figure 4.5 shows a simple program network of three parts, one of which is composite. The figure shows the original network, comprising parts A, B, and C, the expansion of the composite part C, and the result of expanding the original network to contain only primitive parts. The expanded network is one that can be invoked using only virtual machine operations for creating the interprocess communication channels and invoking the primitive parts.

The notion of having composed parts introduces a hierarchy into the system, not unlike the hierarchy in procedure-based composition systems. The "uses" relation introduced by Parnas [Parn79] forms a hierarchy of programs or procedures. The relation for composed programs in use here is "comprises." A composed part comprises one or more other parts, some or all or which may be composed. This is a

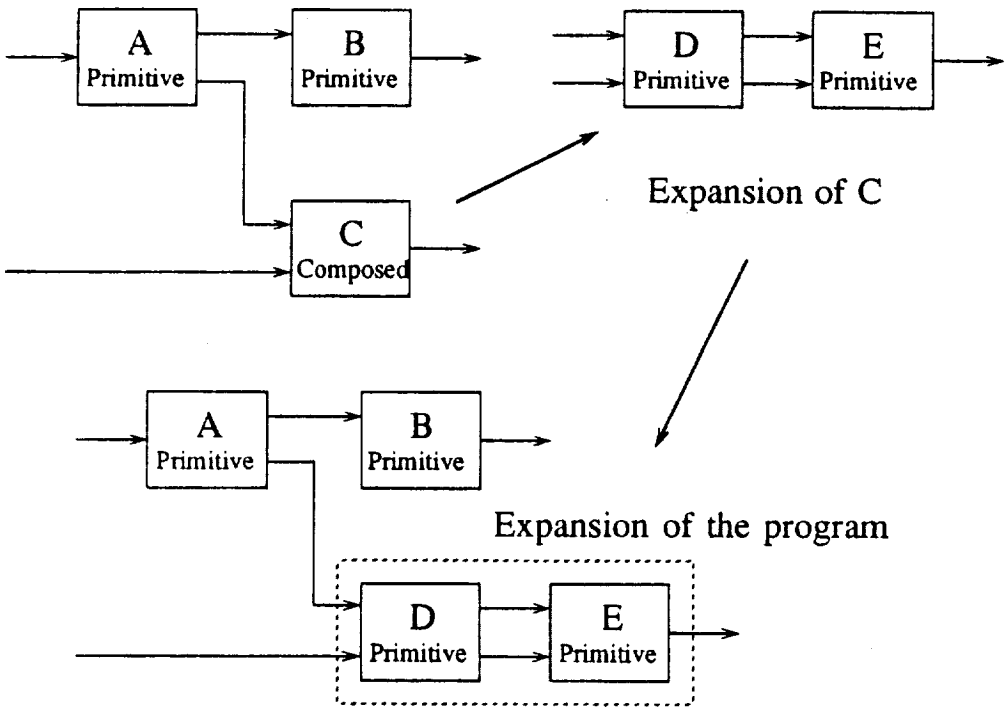


Figure 4.5: Linearly and Nonlinearly Composed Programs

static hierarchy; once a network of programs is expanded, whether parts are composed of others or not is immaterial.

Sockets for composed parts are extensions of those sockets inside part that are connected to the boundary (explained later). So, in Figure 4.5, the connections for the composed part C are in actuality the connections to the internal parts D and E. Thus, external connections ultimately attach to only primitive parts after all composite parts have been expanded.

4.5.2. Sockets

The next abstraction of concern is "sockets." Sockets do not exist in isolation. Rather, they are an element of parts or the boundary, as explained later. From the point of view of a part, a socket is the object to which it writes output data and from which it reads input data. Within a primitive part, sockets are referenced by way of capabilities, or some variation of capabilities, that reference files, devices, or interprocess communication objects.

The static attributes of a socket are as follows:

its name,
its port,

its direction (input or output), and
its data type.

Sockets in this composition system are named, there being one namespace per object (part or boundary) that contains sockets. In UNIX, by convention, there are three default names for every program: stdin, stdout, and stderr; more may be defined at run-time. Parts in PCS do not inherently have default sockets; each program must define its own, and these socket names are part of the static description of the socket. Names cannot be used in PCS for dynamic (run-time) binding to I/O ports of parts.

The direction of a socket, input or output, determines which way data passes through it. For sockets associated with a part, the part reads from input sockets and writes to output sockets. For those sockets associated with the boundary, input sockets pass data from outside the program network to the inside, and output sockets pass data from within to the outside.

The port number associated with a socket is an artifact of the way some systems preallocate I/O channels for programs. In terms of the virtual machine model of Section 3, the port number can be thought of as an index into a capability list storing the initial I/O capabilities of the part. Though seemingly an artifact of contemporary operating system technology, the port mechanism is included in this design to help satisfy the "existing software" and "immutable program" requirements stated in Section 1. A new mechanism allowing primitive parts to access their initial I/O capabilities is presented later in this section.

The data type associated with a socket describes the type of the data that passes through it. In general, only sockets that have the same or compatible data types may be connected with a link. The data typing scheme may be complex, using a hierarchy of data types with inherited attributes such as in Smalltalk. In PCS, we chose, for the sake of simplicity, to draw type names from a flat namespace. Data types on sockets constitute a static attribute, stored in the parts database.

Programs intercommunicate by way of sockets. Sockets map into the particular programming language in which the part associated with the socket is programmed. For example, if the part is programmed in Fortran, then the "logical unit number" used in WRITE and PRINT statements maps into a particular socket on the part. Hence, the internal coding of a part is entirely defined within the language in which it is programmed, with the additional abstraction of socket that is used in the program statements that generate output and read input.

4.5.3. Links

The link is the abstraction for the operating system object that passes data from one socket to another. Links are unidirectional in this system, just as sockets are. There are no unique static attributes associated with links; they inherit their attributes from the sockets and parts that they interconnect. In particular, links do not have names in the composition system; during program editing, the user names them by pointing to them. The two sockets connected by a link may both be on program

parts, or one on a part and the other on the boundary. In this system it is not allowed to have a link connect two sockets, each on the boundary. This restriction is arbitrary, but no loss of generality derives from it, because, at any point in a link, an identity part (that is, one that passes its input to its output unmodified) may be inserted.

Links are managed by level nine in the virtual machine model, where they are called communication channels. The term "links" is in the vocabulary of the composition system, which comprises level 15 of the model.

Links between parts can only connect output sockets to input sockets, since data can pass only in one direction. Links between a part and the boundary connect sockets of the same direction, because, in essence, the boundary socket is an extension of the part socket to which it is connected.

Links are a critical abstraction of the composition system because they form the implementation of the composition relation. All data passing between two program parts must go through a link, hence, making it a place where program debugging and monitoring facilities can be concentrated.

4.5.4. The Boundary

The boundary of a composed program contains all the sockets that describe all of its external connections. Hence, the boundary consists of a set of sockets. These sockets have the same attributes as sockets associated with parts. Input sockets on the boundary are the points where, abstractly, data passes into the program. Output sockets on the boundary define the points at which data passes out of composed programs.

In one sense, boundary sockets are abstractions having little or no concrete realization because they do not reflect operating system objects; connections from outside a program to a boundary socket actually are made to the part socket on the other end of the link attached to the boundary socket. Boundary sockets do, however, have names, and a single name space exists for the entire boundary. These names are used when the program containing the boundary is collapsed into a composite part; they become the names of the external sockets on the composite part.

4.5.5. Synopsis

In summary, composed programs, sometimes called program networks, comprise program parts, sockets, links, and a boundary. Figure 4.6 shows these components in a visual representation, using the same program network as in Figure 1.2. Program parts can be primitive, managed by the operating system, or composite. Parts comprise some descriptive information and a set of sockets. Sockets comprise names, data types, and directions. Links have no static information. The boundary comprises a set of sockets. The hierarchy for these components is displayed in Figure 4.7.

There are several types of names and namespaces used in the composition system. Sockets have names, but these only have meaning within the context of a part, when programming it and when it is invoked. Parts have names, too, and these

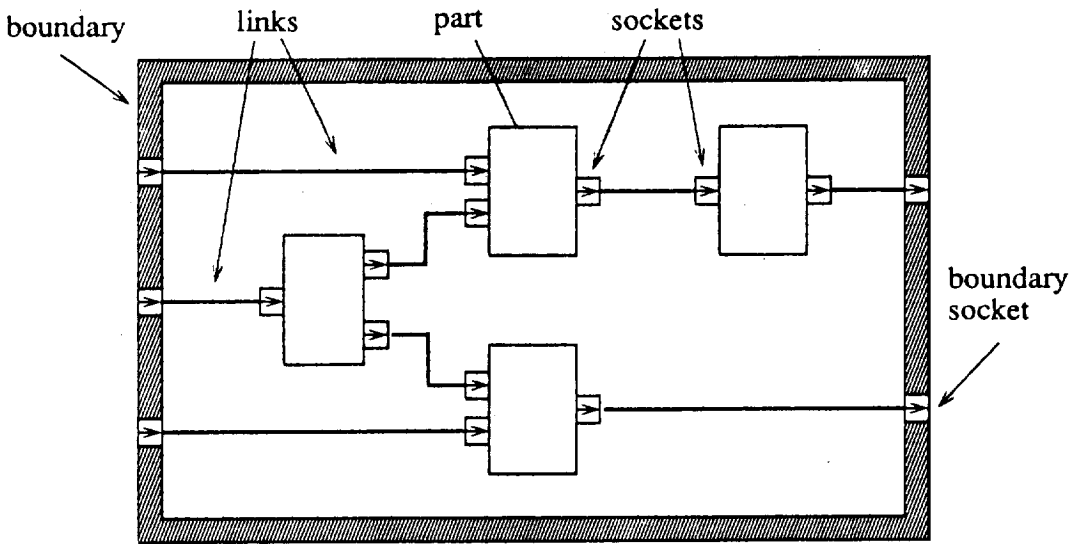


Figure 4.6: Components of Composed Programs

are only relevant during the part selection process during program development. Embedded in parts are names that are only interpreted by the operating system. Except for the part names which are used for part selection, none of these names are used during the composed program development process, and even the part names could be replaced with icons. Generally, during development, the user names items by pointing. This distinction is elaborated on in the next section.

4.6. Parts Semantics

This section concerns the internal construction of primitive program parts, and how the programmer, knowing that the part will be used within PCS, can take advantage of that feature. Though the composition system strives to satisfy the existing software requirement stated in Section 1, optimization of primitive parts is possible in the cases where the parts are constructed from scratch or altered with the composition system in mind. For purposes of explanation, examples of how the programmer can use PCS-provided facilities is given here in an abstract programming language similar to C [Kern78]. In providing these examples, we assume the existence of a data type named *cap*, a capability.

4.6.1. Port Mapping

One of the first areas where consideration of the composition system can help in the coding of primitive parts is in the mapping from socket names to the capabilities for those sockets. Normally, the programmer can rely on the port field of a socket and access the capability for the socket through a capability list established by the program loader. This mechanism can work in a number of extant languages and systems. In Fortran, the port number can become, with support from the loader, the

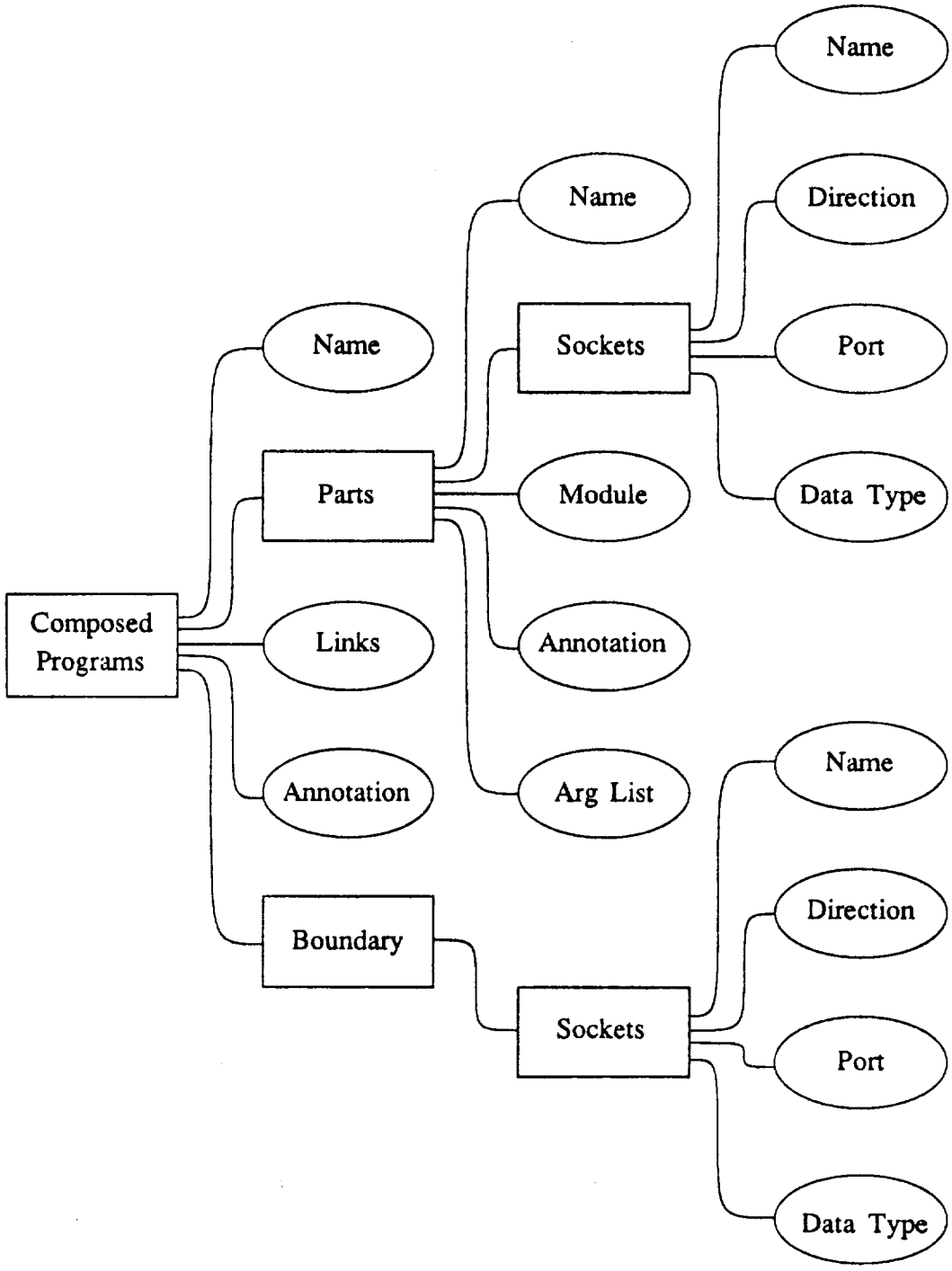


Figure 4.7: Hierarchical Structure of Composed Programs

Fortran "logical unit number." In UNIX systems programmed in C, the port number can be used as the "file descriptor" number normally returned by the system when a file, pipe or device is opened. This is one of the mechanisms, in fact, implemented in the PCS prototype.

Kahn and MacQueen solve the problem of naming sockets, what they call ports, in the parts implementation language by building it into the language, POP-2. Each part, what they call a process, has a process header of the following form:

Process *<name>* *<parameter list>* ;

The parameter list portion has two parts, ordinary and ports. The ports parameters name the process-specific variables that refer to input and output ports in I/O statements. In the following example, QO refers to an output port and QI an input port:

Process *QUEUE* out *QO* in *QI* ;

Such a mechanism cannot be used by itself in PCS, since PCS adheres to the multiparadigm requirement. However, if such a language as is shown above is used to write primitive parts, they could be incorporated into PCS. The POP-2 style of naming ports is a part of the specification of the program, though, and does not address the technique in the virtual machine for making the port capabilities available to the program at run-time.

Another mechanism takes advantage of the abstraction of the socket name, and allows the program to directly map this name into a capability. This mapping operation is strongly analogous to the mapping from names to capabilities provided by the directory manager, but is implemented instead by the shell.

The programming interface to PCS is provided logically in a run-time library, which can be thought of as a way of making calls directly to the shell (level 15) of the operating system. In this regard, it is safe to think of the user's program as running at level 16 of the virtual machine — the applications level. In implementation, however, the shell (PCS) is not involved in the management of the execution of a part, except when monitoring and debugging facilities are evoked.

The interface, as a procedure call, is as follows:

declare *socketCap* of type *cap;*
socketCap = socketCapFromName (NAME);

In this case, *NAME* is a text string containing the name of a socket.

This mapping need not be done explicitly by the program part. A prelude to the part can, given information provided by the programmer as to the names of the sockets, perform the mapping transparently.

4.6.2. Multiparadigm Parts

The composition system does not specify the language in which primitive parts are constructed. All that is required is that the parts communicate through the I/O facilities provided by the underlying virtual machine. The language exists at a level of abstraction lower than the composed program. All usable languages have an abstraction for input and output. This aspect transcends language and allows any to be used in PCS.

4.7. Network Description Language

This section describes the representation language called the *Network Description Language* (NDL), for composed programs. Because, thus far, only the static attributes for programs have been presented, this description only contains the static attributes. Later, as more components of composed programs are described, more parts of the description language will be presented. NDL is introduced here to serve as a foundation for explaining other attributes precisely in the next section, which concerns the visual attributes.

The syntax of the network description language is a simple name-value S-expression notation, where the form

(name value)

is used consistently. The names of data items are all fixed and defined by the language. Values may be integer scalars, string scalars, integer arrays, string arrays, or composite. Composite elements add a level of nesting of notation, in a style similar to LISP. The notation was chosen for clarity and ease of parsing. Each of the components of composed programs are described below by example.

Sockets have a simple representation, in that they do not have any composite components. An example of a socket represented in NDL is as follows:

```
(socket
        (name "stdin")
        (port 0)
        (direction 0)
        (type " ")
)
```

The elements of this description should be self-explanatory. The specification of the meaning of the data type values is deferred.

The description of program parts is similar in structure to sockets, except that they contain sockets as elements. An example part NDL notation is as follows:

```
(prog
        (name "sort lines")
        (socket
                (name "stdin")
```

```
            (port 0)
            (direction 0)
            (type " ")
        )
        (socket
            (name "stdout")
            (port 1)
            (direction 1)
            (type " ")
        )
        (module "/usr/bin/sort")
        (args
            "sort"
            "-d"
        )
)
```

Again, this LISP-like notation should be self explanatory. Because parts contain sockets, their NDL representations contain NDL description of those sockets. As described earlier, the module name is a string that is passed to the directory manager of the virtual machine so that it can be mapped into a capability. In the NDL example above, the directory manager is implemented by a UNIX-style operating system.

Boundary NDL descriptions are almost identical to part NDL descriptions, except that all that they contain are socket descriptions. Likewise, since there is no static information associated with links, there is nothing in their static NDL descriptions. There is, however, information in the visual part of their NDL descriptions, described in the next section. Prototype boundary and link description look like the following:

```
(boundary
    (socket
        ...
    )
)

(link
    ...
)
```

These descriptions contain both net-specific information, such as where links attach, and visual information, such as how they are routed on the sketch pad.

Program networks have an NDL notation. Networks, or composed programs, comprise part descriptions, a boundary description, link descriptions, and annotation. A prototypical net description is as follows:

```
(net
        (annotation
                " ... "
        )

        (part
                ...
        )

        (boundary
                ...
        )

        (link
                ...
        )

)
```

There may be, and usually are, several part and link elements in net descriptions.

4.8. Invocation

This section describes the issues and choices surrounding converting a static description of a program graph into a running set of communicating programs.

The module name attribute of parts is in the namespace maintained by the directory manager. When the composition system begins the execution of a composed program, it invokes primitive parts by using the operating system operations in the process manager level and composite parts recursively through itself. This process is analogous to initially expanding the program graph so that only primitive parts exist, and then invoking them all using the process manager routines.

The createProcess operation in the process manager requires (at least) a capability that references the long-term storage object that stores the code for the primitive part. This capability is not stored as a static attribute of the part itself in order to avoid the problems concerning long-term storage of capabilities in other storage objects (the only long-term storage of capabilities in the virtual machine is in directories, managed by level 12). The design of this composition system stores a module name in the part, and that name must reside in some namespace. For our system, we chose the name space supported by the underlying virtual machine: the directory system. This directory can be thought of as a name server, mapping human-readable names into handles that can be interpreted by any of the upper levels of the operating system. The composition system does not add its own namespace for naming primitive parts; because it programs the virtual machine of Section 3, it uses the facility that machine provides.

The algorithm used by PCS to convert a static program graph into a running program on a single virtual machine traverses the list of parts creating the links to other parts and outside objects as necessary. Once all links for a part are created, it starts the part. Abstractly, the invocation algorithm is as follows:

```
for each P in Parts[*] do
begin
    for each S in P.sockets[*] do
    begin
        if ¬ S.link.exists then
        begin
            S.link.cap := linkCreate ( );
            S.link.exists := TRUE;
        end
        P.portSet := P.portSet ∪ S.link.cap;
    end
    P.cap = directorySearch ( P.moduleName );
    processCreate ( P.cap, P.portSet );
end
```

The *linkCreate* operation returns a capability for a link. The abstract algorithm above does not show how only one user of a link subsequently calls *linkOpen* for reading and one calls *linkOpen* for writing.

The algorithm presented invokes all parts in the program graph whether they contribute to the overall computation or not, an *oblivious* algorithm. An alternative is to only invoke parts if they have a contribution to make to the computation. We define three predicates describing the behavior of parts in a parts-based computation:

CONTRIBUTES(p) ⇔ removing p from the computation changes the result of the computation.

WRITES(p) ⇔ p ever writes to one of its output sockets.

READS(p) ⇔ p ever reads from one of its input sockets and there is data to read.

The results produced by a computation comprise the union of all its outputs, hence the only way for a program part to contribute to the computation is if it generates output data (parts can, as a side effect, impact the computation by failing to read their inputs, causing producers up-pipe to block; we do not further consider this case). For example, if a classifier part never routes data to one of its outputs, then there may be parts further down the pipeline that never receive any input data. Hence, CONTRIBUTES and WRITES are equivalent for side-effectless parts-based programs. However, it is in general undecidable whether a part having one or more

outputs will generate any output and hence, in general, CONTRIBUTES is also undecidable. If, however, we constrain parts such that they may only generate output if they receive input,

WRITES(p) ⇒ READS(p), and hence
CONTRIBUTES(p) ⇒ READS(p),

and they do not otherwise cause side effects, then a *lazy data-driven* invocation policy can be used: only invoke parts when they receive input. When these constraints cannot be met, the oblivious algorithm must be used.

Lazy invocation is advantageous to oblivious invocation in those cases where the program has parts for which CONTRIBUTES is false. The advantage stems from two sources: conserving resources and, in some instances, enabling invocation. Conserving resources occurs when invocation of unused parts results in computing resources being allocated and subsequently released with no benefit to the computation. Enabling invocation occurs when parts bound to unavailable resources need never be started. For example, a classifier may route data through one of multiple outputs based on knowledge of what resources are available and which are used further down the pipelines.

Oblivious invocation can result in lower run times for program graphs. Invoking a program incurs overhead, generated by the composition system, the operating system, and by the part itself; it may not immediately start reading input. If the composition system does not start parts until they have available input, then the invocation overhead can directly increase the total running time of the program, and will propagate this delay down the pipe.

A second type of lazy invocation, *demand-driven*, only invokes parts when the parts to which they write try to read data from them. Demand driven evaluation does not suffer from the undecidability problem. Parts with outputs attached to the border are immediately invoked. As they try to read from their inputs, the parts that supply those inputs are then invoked. The disadvantage of this scheme is that more parts will be started than necessary because of the second clause of the READS predicate, that of data being present, cannot be determined. Hence, READ implying CONTRIBUTES no longer holds.

4.9. Summary

This section has presented a description of what composed programs are, what their components are, and how they are represented within the composition system. Little or no reference has been made to the visual representation of composed program, an important aspect. We have concentrated on the composition mechanism, not the implementation of a particular algorithm using this system.

From the experience of designing and implementing parts-based programs, as described in this section, come the following principles:

1) Static parts-based program graphs describe a useful set of applications.
2) Parts-based programs having parts with more than a single input and output allow a larger class of programs to be created than when parts are constrained to have a single input and output, as is the case with the UNIX shell.
3) Basic building blocks, such as procedure call emulation, coroutines, rendezvous, pipelining, and classifiers have been identified as useful in creating parts-based programs.
4) Allowing a program graph to be reduced to a single program part and placed in the library of parts enhances the level of abstraction achievable in PCS and enables the construction of larger programs.
5) The boundary abstraction reinforces the notion that a composed program, from the outside, structurally similar to a program part.
6) A textual network description language provides a straightforward way of saving the state of a program graph on external storage, and provides a representation that can be used outside the context of a program development environment. NDL proved useful in debugging the prototype.

An oblivious invocation algorithm provides a safer means of starting a program graph when no constraints are placed on the behavior of program parts. Data driven lazy invocation can conserve resources and enable invocation if constraints are placed on the input/output relationships of parts. Demand-driven lazy invocation is less effective in conserving resources.

5. THE PROGRAM DEVELOPMENT ENVIRONMENT

5.1. Introduction

This section deals with the visual aspect of the program composition system, describing how composed parts-based programs appear and how they are created. Our research included the construction of a usable prototype for experimenting with pipe-based program construction. This section examines that prototype, the design goals that underlie it, and the conclusions we draw from it.

The design goals of the visual programming environment for PCS programs are as follows:

- Provide a language well matched to the class of problems for which PCS programs are best suited.
- Be easy to use.
- Present an environment in which the programmer can develop, modify, execute, and debug PCS programs.

Most of the topics in this section center on the graphical editor and its prototype implementation, called *protoPCS*. This prototype is further described in a User's Manual [Brow88b].

The graphical interface to our composition system is motivated by the inability of linear text to describe a nonlinear interconnection structure among parts. The network description language, NDL, is a textual representation of non-linear programs that can precisely state what the parts are and how they interconnect. It is inadequate, however, for practical use; the NDL text gives no sense of the network of parts. The interface depicts a network as a directed graph whose nodes denote operators and edges data flow paths. We call these graphs representation-transformation (RT) diagrams.

In theory, several interfaces and editors could exist for creating and modifying composed programs, just as multiple editors exist for textual data representations. We describe only one such interface here, that one which has been implemented. The description, however, is sufficiently abstract as to allow a family of interfaces of which all members implement similar functionality. This approach, to describe the abstract interface and functionality, and then create multiple implementations that satisfy that specification, was used by the author and colleagues in the implementation of a concurrent language debugger [Brow88a].

5.2. Designing a Prototype

The paradigm of programming within PCS is as follows: The user first develops a mental image of the parts and their interconnections. Within the PCS programming environment he or she lays out this diagram on a sketch pad by selecting parts and pointing to places on the workstation screen. A built-in browser helps the selection of

specific parts from an inventory. Next, the user connects the sockets of the parts with links by pointing to the beginning and ending sockets of each link. The user switches back and forth between adding parts and adding links until the graph is complete, connecting some sockets to the boundary of the sketch pad, denoting external connections. When the graph has been completed, the user asks the environment manager to invoke the program.

Based on this scenario, we designed an implementation to incorporate the features we desired. Several early decisions directed this design, as follows:

- All the components of PCS programs have visual representations.
- The programmer creates a PCS program by directly and spatially manipulating its components on a sketch pad.
- The program development environment supports top-down programming.
- Textual information can be attached where appropriate and natural, e.g. in naming a part or in annotating.
- There is an inventory of parts including both primitive and composed parts. The user selects parts from this inventory.
- The development environment provides syntax checking and does not allow the programmer to create a syntactically incorrect program.
- Primitive parts cannot be edited within PCS. New primitive parts built from existing programs can, however, be added to the inventory without leaving the programming environment.
- Each major component has annotation associated with it. The annotation is visually suppressed until requested by the programmer.
- The environment allows the creation of abstract parts that have no implementation, allowing for top-down program design.
- The environment allows files and devices to be attached to parts.
- Debugging and monitoring are an integral part of the environment and have a visual, direct manipulation interface.

The remainder of this section examines in detail how these design principles are met, the visual representations of the components of composed programs, the elements and concerns of the environment manager, and some discussion of the implementation of the prototype.

5.3. Visual Representations

This section describes the visual representation of the components of composed programs, namely: parts, links, sockets, and the boundary. Each component of a composed program has a visual representation, and the combination of the individual components creates a visualization of the entire part. Figures in Section 1 and Section

4 show an increasingly developed abstract portrayal of the visualization of a composed program. Each component of composed programs is described in a subsection.

5.3.1. Part Representation

Parts are the unit of design and the prime focus of attention in composing programs. The visual representation of a part must contain sufficient information to enable the reader of the program to immediately identify the function of the part. Toward this goal, a pictorial component, an icon, is added to the part description. Work exists studying the efficacy of small diagrams and pictures in presenting information [Rohr84, Bere86, Mont86] which generally agrees that icons denoting actions are more difficult to design than icons denoting objects. Conveying the operation of a part in a picture is even more difficult to achieve for heavyweight, complicated parts. Icons are better suited to lightweight operations such as arithmetic operators (arithmetic symbols can be used) or as representations of objects.

Parts represent actions, or transformations, and so their icons should reflect the action. One scheme that works for some parts is to use a data plot roughly showing the function computed by the data. Another scheme avoids the issue of creating meaningful transformation icons by displaying, in the same icon, both the input and output representations with an arrow, denoting transformation, between them. Figure 5.1 shows two example icons of this type, one for centering lines of text and the other for a Fourier transform.

The existing static components of program parts that have visualizations are the name, the sockets, the argument list, and the annotation. The module name does not need to be immediately available for visualization because it is only a piece of information that aids in the invocation of a composed program. PCS makes distinction between information that is continually visible and that which is quickly available on demand. Because of the potentially large amount of text that may be associated with the annotation and argument components of the part, they are not continuously visible but can be made visible with a single "point and select" operation. Hence, the part visualization contains a virtual button that, when pressed, displays the text of the annotation or argument list.

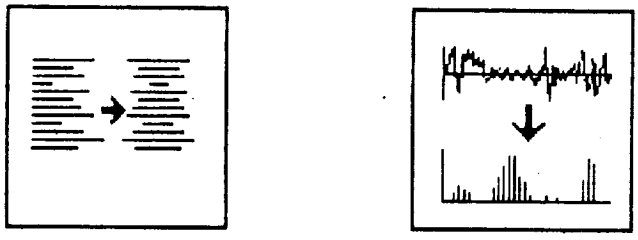


Figure 5.1: Representation-transformation Icons

Figure 5.2 shows an example of a part in the prototype PCS. The static components are labeled in the figure. The example part is for sorting, and the icon is reminiscent of the sorting graphs seen in Baeker's sorting movie [Baek81]. Notice that in this icon, text is relied upon to complete the picture of a sorting part, but that the body of the icon shows a graph of a function. We consider using function graphs in an icon a useful device.

File and document objects are represented in much the same way as part objects. Documents are read-only, however, and are constrained to having a single output socket. Files may be read and/or write, and can have any combination of sockets. Files do not have an arguments button and documents replace the arguments button with a contents button.

5.3.2. Boundary Representation

The boundary of a composed program holds the sockets that can later be used in splicing this program, as a composite part, into other composed program. Visually, the boundary is a border around the sketch pad in which the programmer creates the program and is used as an attachment point when the programmer creates a link between a socket on a part and the outside. Conceptually, the border around the program delineates what is outside the part from what is inside the part. In the prototype, the parts representation is rectangular and the border is rectangular, thereby suggesting each other. In the prototype, the boundary is a thick gray border around the entire sketch pad.

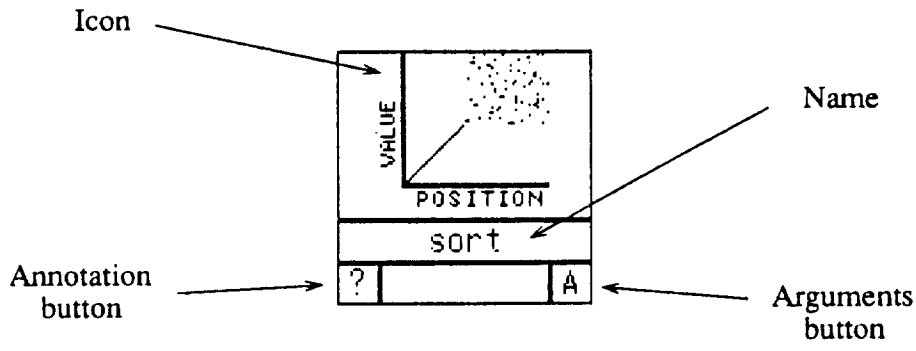


Figure 5.2: A PCS Part Visualized

5.3.3. Links Representation

Links connect sockets which are either both on a part or one on a part and one on the boundary. Conceptually, links are lines drawings from the source socket to the destination socket. Ideally, the routing of the link through the sketch pad, from one socket to the other, should be natural and easy to follow. There should be no ambiguities at link crossings, and few or no crossings.

5.3.4. Socket Representation

Parts and the boundary contain sockets, and so their representations contain socket representations. Sockets are static objects on parts; the programmer must know which socket picture refers to which socket and be able to point to them when creating links.

Sockets appear as small boxes on the edges of parts and embedded in the boundary. Socket representations contain small arrows denoting the direction of data flow. Figure 5.3 shows a part with two sockets.

5.4. Development of Composed Programs

Development of composed programs involves a graphical editor, a language-specific editor oriented towards the visual syntax of PCS. The program development cycle in PCS is similar to that used for textual languages: the programmer develops a program in an editor and then invokes it. Hierarchical programs can be constructed top-down or bottom-up; each has its application. A bottom-up approach can be used by a programmer who is interested in extending the inventory of parts; the programmer creates new composite parts and then stores them in the inventory for others to use. The top-down approach is useful for the user developing applications. When constructing a program, the user may opt to insert an abstract part rather than deal with the details of implementing that component immediately. The abstract part becomes a placeholder for subsequent expansion. This allows the programmer to concentrate on the overall structure of the application and only be concerned with finer

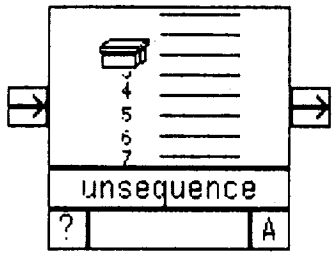


Figure 5.3: A PCS Part with Sockets

details of implementation at a later time. In its current state, the prototype development system does not implement abstract parts.

The graphical editor runs on a computer with a moderately large graphics screen. A specific requirement of the host for the editor is that it have a large graphics screen capable of displaying a composed program of several parts and a device that allows the user to designate specific places on the screen. The editor presents a modal interface; the user enters a mode, performs an action, and then enters another mode. The default mode allows the user to select components of the program under construction and perform editing operations on them, such as deleting or moving them. The user changes modes by pressing a button on a tools palette. The button denoting the current mode remains highlighted. The major modes are selecting/editing the graph, adding parts, and adding links.

Figure 5.4 shows the layout of the protoPCS editor. The editor display comprises five main parts or panels. The largest panel is a sketch pad in which the user composes the PCS program. The other sections are a control panel containing commands and options available to the programmer, the tools palette containing

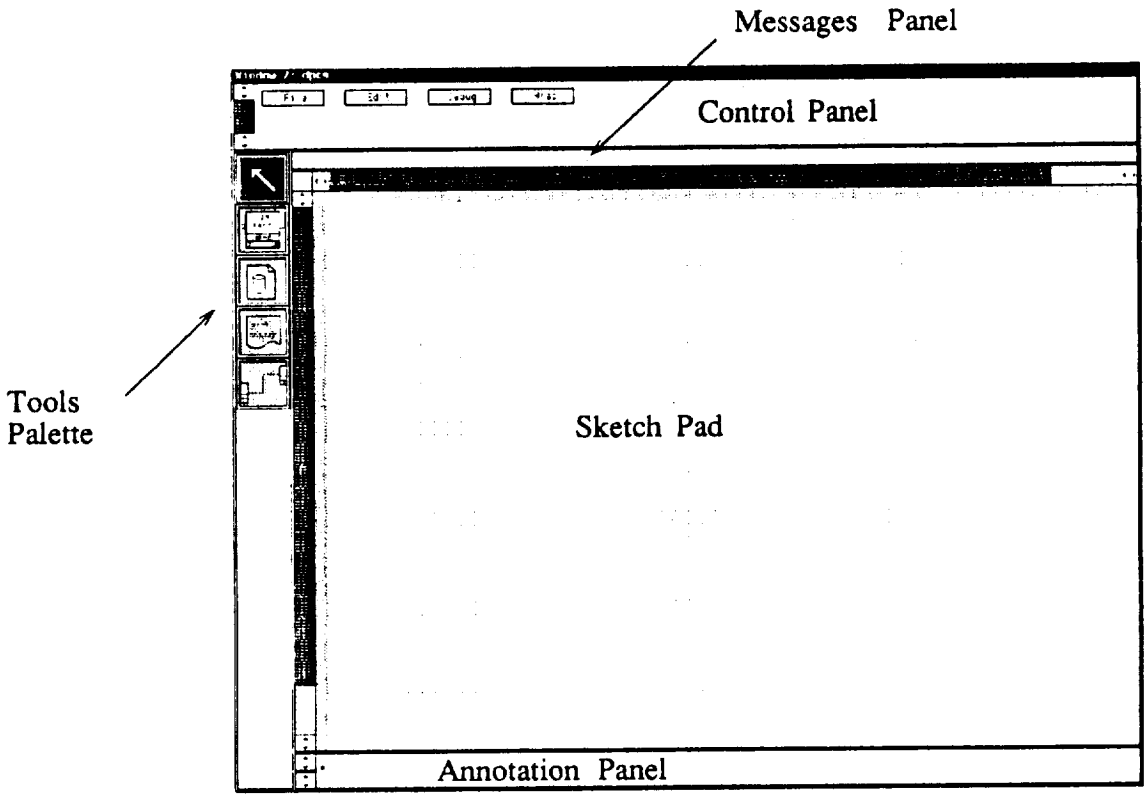


Figure 5.4: Prototype PCS Layout

virtual buttons to invoke composition tools, an annotation window in which the user can view and edit the annotation for the program being composed, and a small messages panel used to pass instructions to the programmer.

Within the control panel at the top of the screen are the headers for menus that contain commands to the editor. The "file" menu contains commands for saving and restoring the program graph, for erasing the sketch pad, and for invoking the program. The "edit" menu contains commands for deleting and moving objects on the sketch pad. The "debug" menu contains commands for viewing the internal data structures of the editor and for setting flags within the editor to cause debugging information to be placed in a logging file. The only command in the "misc" menu is for creating new parts.

The host that runs the editor, the development machine, need not be the same as the one that runs the composed program, the target machine. The target machine must, however, offer a virtual machine interface such as the one specified in Section 3.

Parts selection is a key element in the design of the composition system. Ideally, program parts are clustered into groups such that all the parts within a particular group are related by what they do or the type of data on which they operate. This type of organization can lead to problem solving environments that, by proper clustering of parts, present versions of the composition system that are tailored for a particular domain. The clusters can be hierarchically structured, say, at the top level by problem domain and then below by the type of data used by the parts. If the type definition system is hierarchical, as in Smalltalk, then the part-group organization will be hierarchical to reflect the data-type organization. The Smalltalk browser is a good example of a mechanism for traversing a tree of data types, allowing the user to examine more and more refined types.

The virtual machine model in Section 3 has a createProcess operation that allows its user to pass arguments to the newly created process, much as one would pass arguments to a called procedure. The arguments exist so that programs can be tailored at run-time to a particular application. This mechanism needs to have some analog in the composition system. Users of PCS have access to the arguments list for each part. Like annotation, arguments are normally not visible on the sketch pad but may be displayed and changed on demand by selecting a piece of any of the part visual representations. When a shell supports a programming mechanism, as PCS does with composite parts, it also needs to extend the argument mechanism to the outer interface. PCS does this by defining a macro mechanism for program arguments. Arguments are numbered, starting at one, and are text strings. When the network is invoked, the programmer has the option of specifying a list of arguments. PCS scans the argument lists to the component parts of the network and replaces the abstract argument variables with the values specified.

The prototype PCS does not impose a rigid inventory structure on the programmer or user. PCS uses the UNIX directory structure to store part descriptions. Hence, hierarchy is built in. The browser allows the user to move down and up a directory hierarchy.

As a principle, annotation is an essential part of programs. Textual languages handle annotation very well — it is encoded in-line with the program text. Visual languages, being inherently non-textual, do not as nicely accommodate annotation. In one sense, the perfect programming language does not need annotation; the program itself tells the story at all levels best. This ideal of "self documenting code" however, has never been met. Programs are implementations of algorithms and invariably take into consideration the target architecture and therefore digress from the pure abstract algorithm. Hence, supplemental annotation can help the reader of the program understand the algorithm, the decisions that went into the implementation, or things to be considered when using the program. Annotation can also describe the inputs and expected outputs of the program, as is the case with program user manuals.

Catalogs of program parts can be rigorously constructed under a rigid set of constraints concerning correctness, accuracy, speed, and usage documentation of the programs in the catalog. In such a case, the programmer needs to have easy access to information concerning the use of the program. Such information is highly textual, though it may contain graphs and charts. The TR diagram shows the reader the overall interconnection structure of the program graph. The documentation should remain available so that the reader can dig deeper into the operation of the program. Though the name and icon of a part serve to convey some of the function of the part, only more detailed information can reveal its true function.

The ability to create new program parts from existing programs is an important one in PCS. We chose to make the operation of introducing new parts into the inventory built in to the composition system and take advantage of the visual interface. To that end, the PCS prototype contains a mechanism, invoked by command, that allows the user to describe a new part by supplying the static information, such as name, icon, annotation, and sockets. Once the part has been described, it can be placed in the parts inventory.

5.4.1. Development of a Program

This section describes the process used by the programmer to develop a simple PCS program and follows the description with a running example. This example will result in a complete, though fairly simple, executable program.

The program we will write is a simple one. Its input is a refer-style database [Lesk78] consisting of bibliographic references. The output is that same database, but with the names of the authors in all uppercase format. This is a very difficult task using the UNIX shell; there is a program to convert input to uppercase, but no application to convert just a portion of an input. This example demonstrates the capability of PCS to handle simple programs that cannot be directly written with UNIX shell. The description below alternates between how we accomplish a subtask in the prototype system and the general principles behind the task.

Having a rough sketch of the desired program graph in mind or on paper, the user focuses first on the representation of the data as it enters the program. For this problem, the input is lines of text. The user looks ahead into the problem and realizes

that he or she will have to split the text into two streams, one that becomes capitalized and another that is not. In order to reassemble these streams, the user will tag each line with its sequence number. To start, the user selects the parts tool from the tools palette, and is presented with a dialog box showing some of the parts available to us (Figure 5.5). The user know these tools work on text streams because they are all in the "TextTools" directory, and he or she quickly locates a part named "sequence" which will add sequence numbers to lines of text. The user selects this part and then presses the "Open" button. Double clicking the part name would have had the same effect.

Once the user has selected a part, the browser window disappears and the editor prompts, in the messages panel, the user to place the part on the sketch pad. The programmer moves the pointing device to the desired place on the sketch pad and depresses a button. As he or she moves the mouse around; the part moves with it, and upon releasing the button the part becomes fixed in the location of the pointer, as in Figure 5.6. The user notices that the icon for the "sequence" part shows an abstract picture of the input and output representations with an arrow between. The user chose to place the part in the upper left corner of the sketch pad because the mental picture of the desired graph has this as the first transformation in the sequence, and the user prefers to read RT diagrams left-to-right.

The next step in our example, in Figure 5.7, is to continue in the same vein until all program parts are placed on the sketch pad. It is not necessary to first place all the

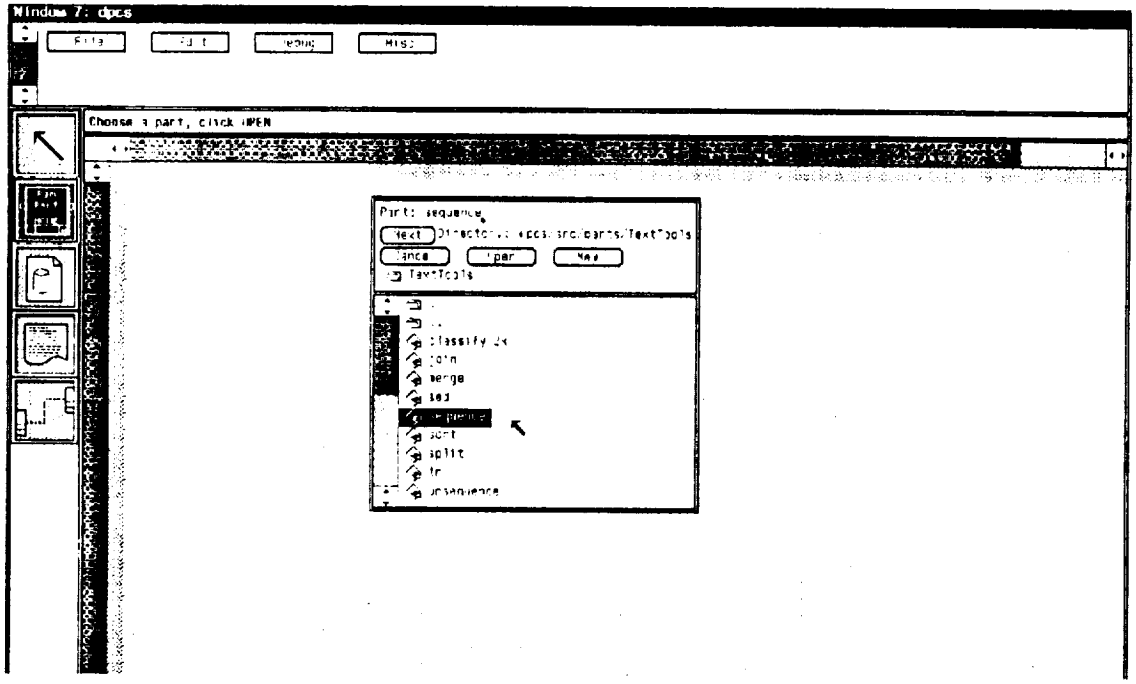


Figure 5.5: The Parts Browser

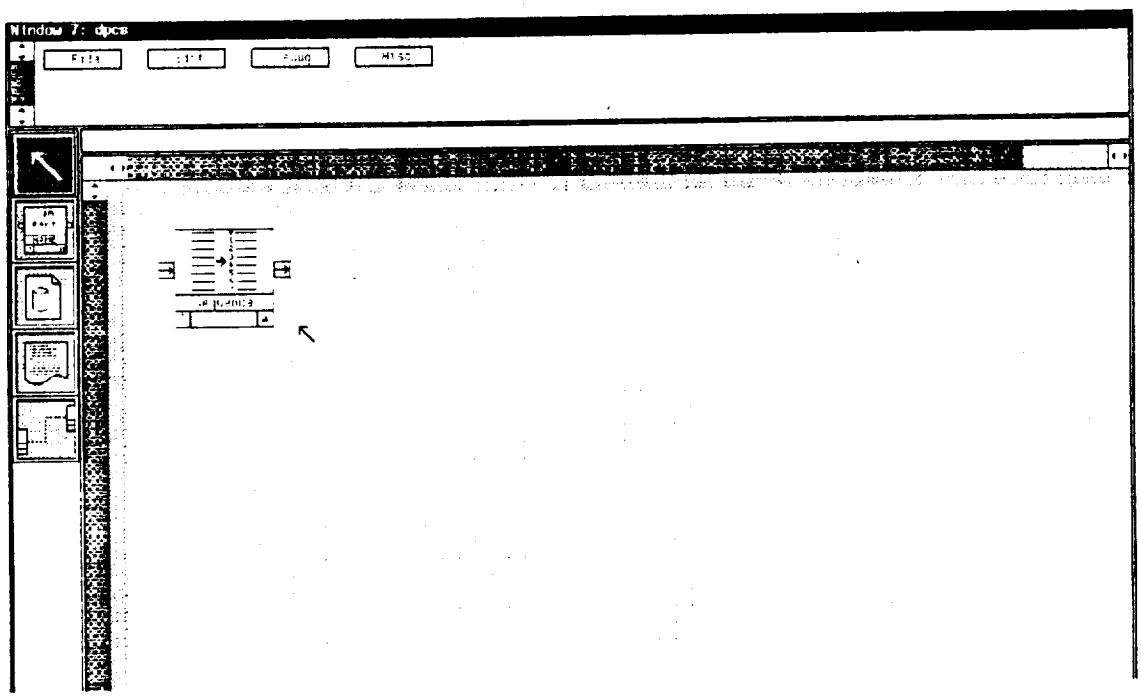


Figure 5.6: Placement of the First Part

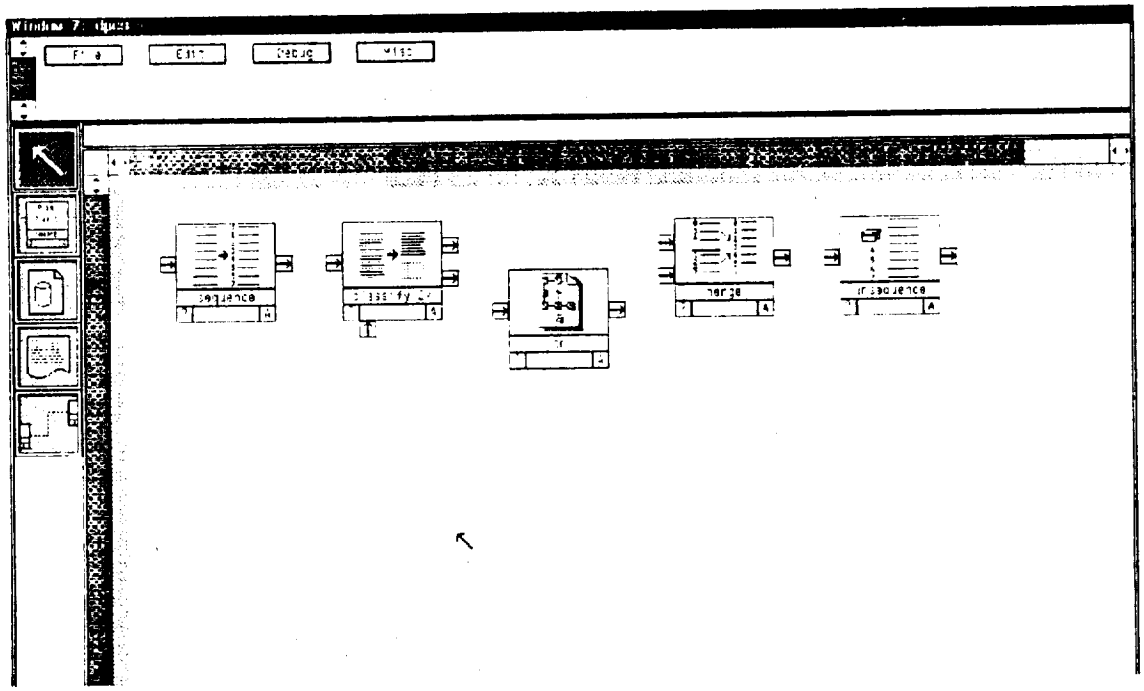


Figure 5.7: All Parts in Place

parts down; operations in PCS can take place in any order. The second part chosen by the programmer was "classify 2x," meaning "classify two ways." This is a general-purpose classifier for text as described in Section 4. It has one data input, two data outputs, and a fourth socket which is used for a classification script. This particular classifier is based on the UNIX program "awk" and so the classifier program is an awk script. The third part is the standard UNIX program "tr" which can translate characters in a text stream from elements in one set to corresponding elements in a second set. The user positioned this part so that the links, when drawn, do not take excessively long routes. The icon for "tr" is the generic icon for protoPCS, that is, the one used when no specific icon is designed for the part. The fourth part, "merge," performs the function suggested by the icon — merging lines of text. The fifth part, as the icon suggests, removes the sequence numbers from the text.

In general, the layout of successive parts in a RT diagram comes from knowledge of the structure of the graph. Though careful consideration of layout is not necessary, it is generally good to place parts that share a link near to each other. An ideal editor of this type could restructure the diagram after a few parts have been placed, but in PCS, the interconnections are not available to the editor yet. If the editor were to require that, before a part is place, the programmer state where it is connected, then the editor could choose a good place to put it on the sketch pad. This is not the case with protoPCS.

Returning to our example, the user next selects the link tool from the tools palette. This tool allows selected pairs of sockets to be connected with links. Once all links are draw, the sketch pad looks like as shown in Figure 5.8. If, in the process, the user tries to connect two input or two output sockets together, the editor disallows it. When the user wants to connect a socket to the boundary, as with the input socket for "sequence," he or she first selects the socket and then select a place on the boundary; the editor draws the link.

In general, link routing on the sketch pad is one of the most difficult aspects of any graph editor and is still an open question how to do it best. ProtoPCS uses a simple algorithm that extends the links from both sockets simultaneously, each seeking the other end, backtracking when deadlocked. Aside from routing, links provide a graph editor with the interconnection information for the network it needs to redesign the graph on the sketch pad.

With all links connected in our example, the user next tailors the general-purpose parts of the graph to suit the needs of the program. In particular, the "tr" program needs to be supplied with program arguments that tell it what translation to perform. The user clicks the "A" button on the "tr" part and is presented with the dialog box shown in Figure 5.9. The design of PCS program arguments depend on the host system that runs the part. In this case, the host is a UNIX machine and so the arguments dialog box allows us to enter UNIX-style arguments. The figure shows that this invocation is to translate all lower case letters to upper case. The user clicks "Okay" and the dialog box disappears.

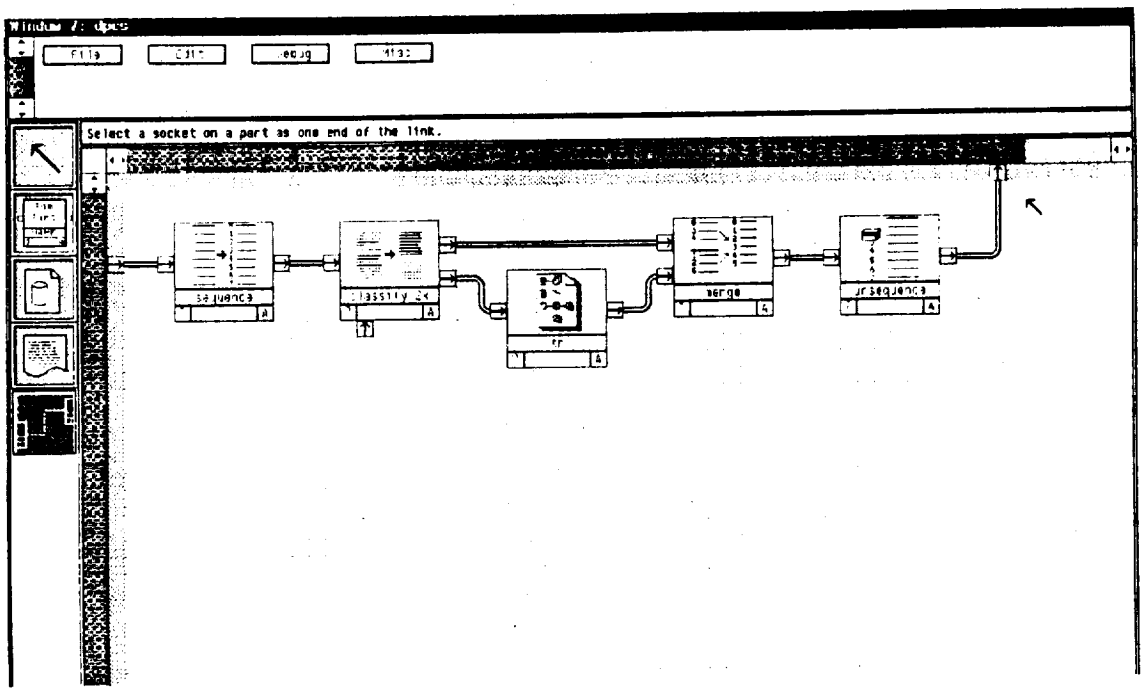


Figure 5.8: Links Drawn

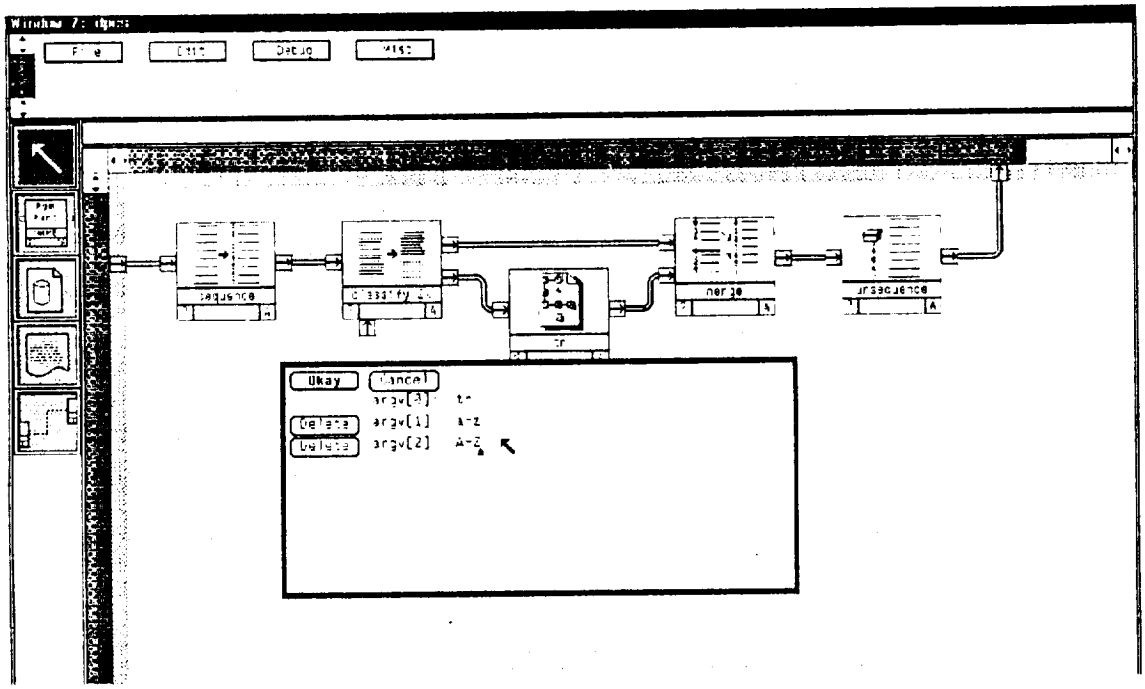


Figure 5.9: The Arguments Dialog Box

The next step in the construction of our example is to create the script that tailors the classifier step. The classifier is a complicated program, sufficiently so that the user has forgotten how to use it. He or she clicks on the annotation button, labelled with a question mark on the part, and is presented with the window of information shown in Figure 5.10. The user is reminded by this of the names of the data output ports and how to construct a program to drive the classifier. The user closes the annotation window and prepares to create the script as a document object for the classifier.

Once reminded how to use the classifier, the user sets out to create a new document object to hold the script, using the "Create Part" option on the "Misc" menu. The user is presented with the dialog box shown in Figure 5.11. Using the mouse to manipulate the items, he or she constructs the image of the document object and then creates the contents of the document by pressing the "Edit Text" button. The user is presented with a standard text editor window in which we type the classifier script. Knowing that the classifier is based on awk, the text the user types is as follows:

```
$2 == "%A"  { print | "output-2"
              next }
            { print | "output-1" }
```

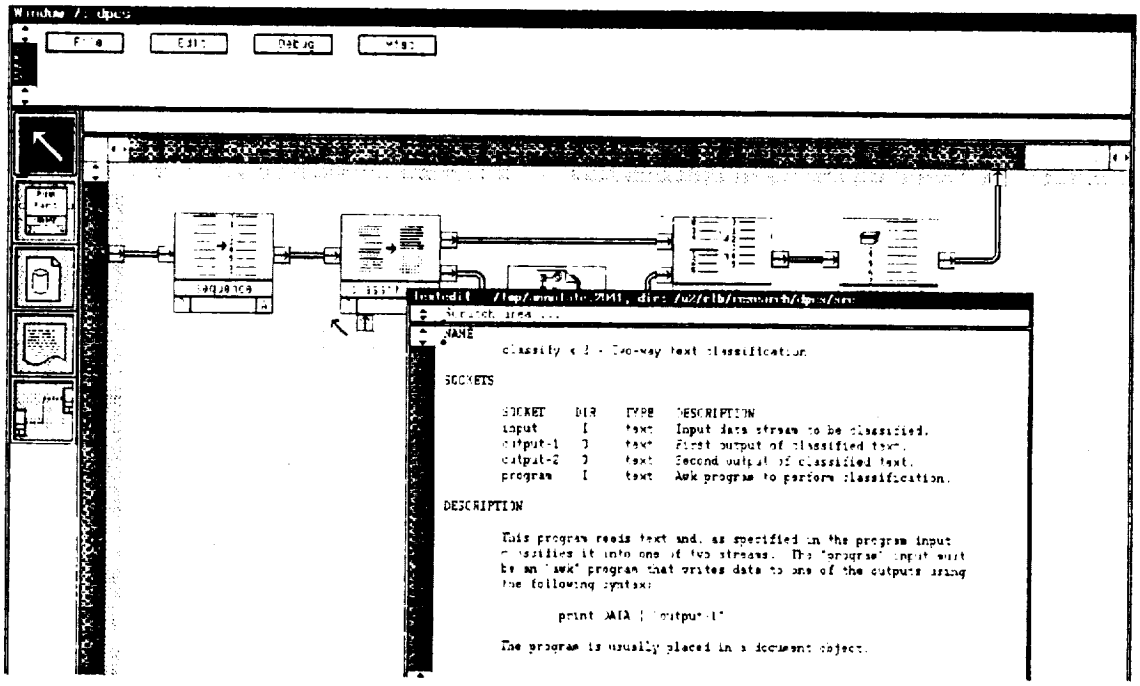


Figure 5.10: The Annotation Window

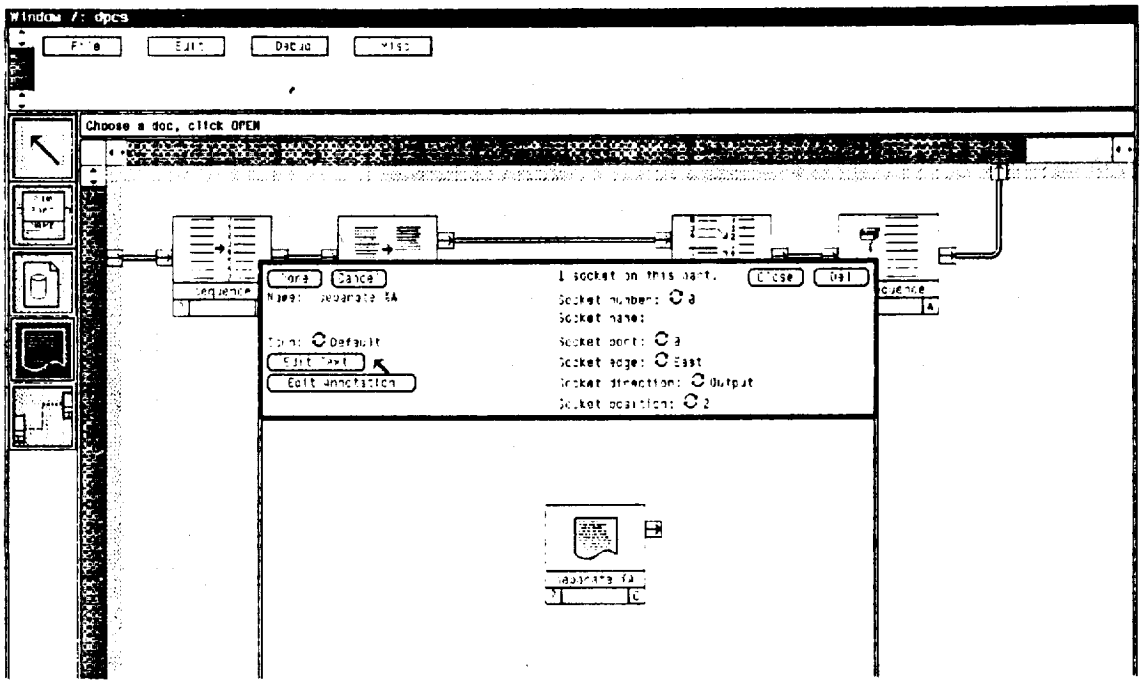


Figure 5.11: The Dialog for Making a New Document

The syntax of the classifier script derives from the syntax of the UNIX program "awk," but has been augmented to allow output statements to name a port to which they direct data.

When finished creating the script, the user links it into the program network as show in Figure 5.12. The program is now a complete network and the user is ready to try it out. He or she selects the "Run" command from the "File" menu and is presented with the dialog box show in Figure 5.13. Each boundary socket is listed in this dialog box, and the user has the option of connecting them to windows, files, or nothing. In the current case, the user chooses to connect the input socket to a file containing bibliographic references and attach the output socket to a window.

Program invocation brings up several issues. Because program graphs in PCS have explicit external connections, they must be resolved when the user invokes the program. In UNIX, where programs inherit the external connections of their parents, this is not as critical. The prototype PCS accommodates UNIX programs in that if port number two, called "stderr" in UNIX, is not otherwise used, it is opened to the log file that is used to record the process of the execution of the network. Another issue that arises is debugging and monitoring. Ideally, the programmer should be able to monitor the progress of the program, and tap into the pipes of the computation to make sure that the right transformation is being performed by the parts.

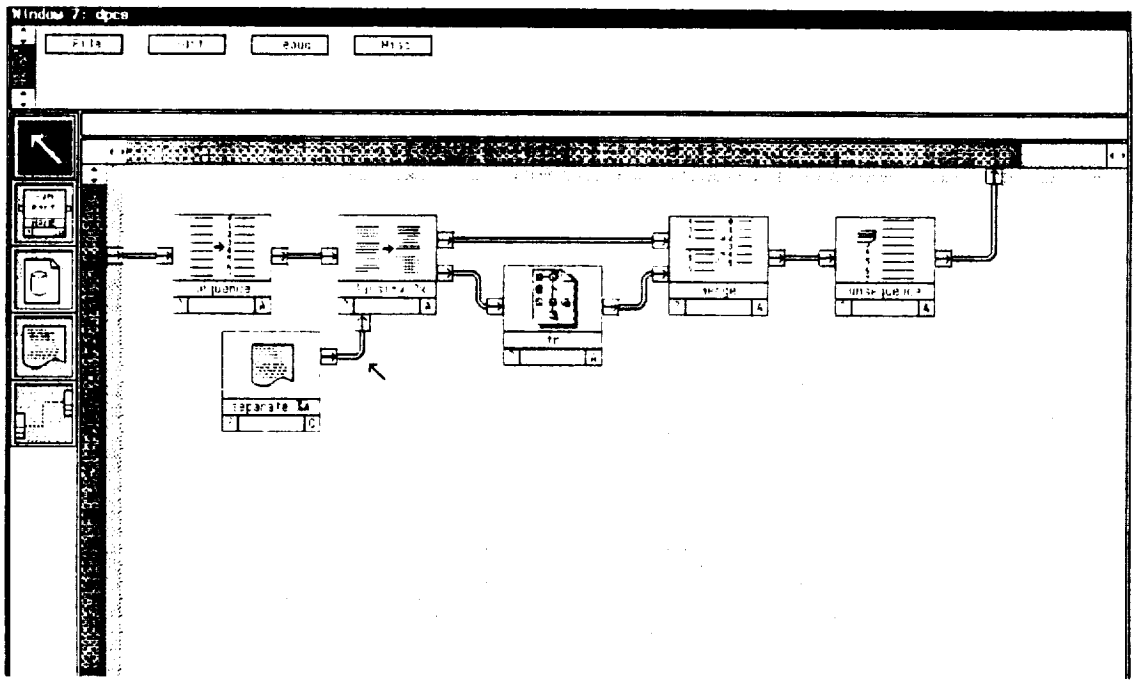


Figure 5.12: The Completed PCS Program

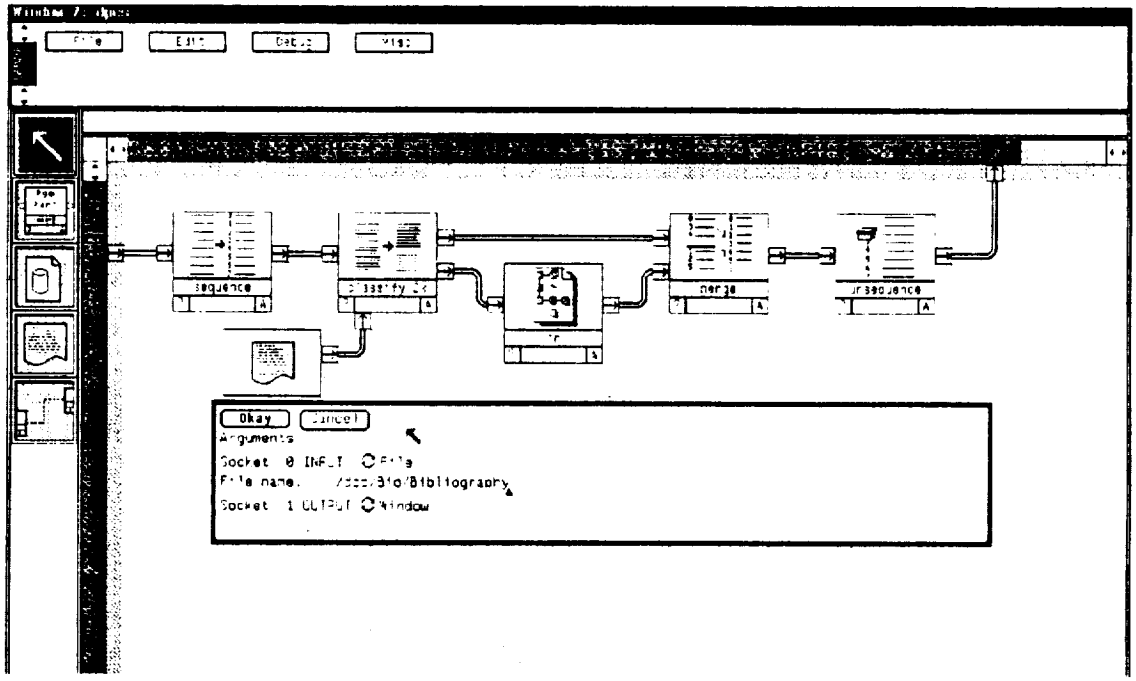


Figure 5.13: The Invocation Dialog Box

Back to the example, PCS has now invoked the program and popped up an output window, shown in Figure 5.14, showing the final output. Subsequently, PCS shows a status log of the execution of the program, containing any error messages, the completion status of the parts, and confirmation that the entire network has completed.

5.5. Environment Manager Internals

This section briefly discusses the issues concerning the implementation and data structures of the graphical editor that forms the front-end for the program development environment. Though the program graph appears to the user as a picture, it is stored within the composition system as a linked data structure from which the picture is derived. The composition system maintains two major data structures, one to describe the abstract RT graph and the other to describe how it is displayed on the workstation screen.

Separating the graph and display structures simplifies the construction of the prototype composition system by separating the modules that operate on the two structures. In general, modules in protoPCS are oriented towards either managing the program graph or managing the display. The structures are interlinked; the display structure references objects in the graph structure.

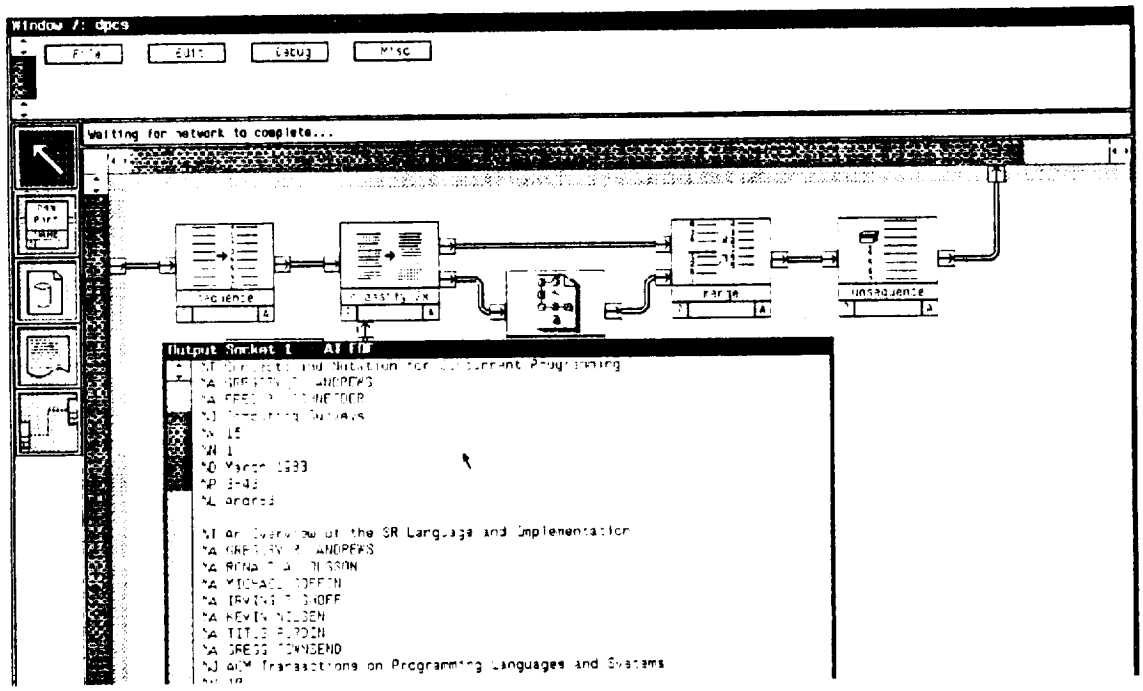


Figure 5.14: The PCS Program Output Window

Each set of modules, graph and display, implement a set of operations that implement the editor. Graph-maintaining modules are further subdivided into those concerned with the parts, the links, the sockets, and the boundary. Each type manager (TM) implements at least routines to create, read (from an NDL description), write, and destroy the object it manages. The RT graph structure is an interlinked dynamically allocated data structure composed of those objects. Other operations necessary in the TMs are as follows:

- Parts: add to graph, delete from graph, associate with link, display annotation, edit arguments, traverse all parts.
- Links: add to graph, delete from graph, associate with socket.
- Boundary: add socket, delete socket.

Adding links to the graph only involves creating them and linking them to the sockets at their endpoints. Parts must be maintained in a separate table in order to support the "traverse" operation used by the invocation algorithm. Each of the operations with the exception of traversal is implemented in time that is constant as a function of the number of parts, links, and sockets. Additional structures, such as a table referencing all links and sockets, exist within the composition system to facilitate debugging.

The RT graph data is mostly doubly linked, thus does not exhibit the same purely hierarchical structure of the components of the abstract graph. The double linking reflects the nature of the operations on the graph. For example, when a user deletes a part, the links attached to it are deleted simultaneously. Hence, parts must reference (through their sockets) the links. Similarly, when the user deletes a link, the sockets on either end must be updated to indicate that they are not attached, and in the case of sockets on the boundary, they must be removed.

A sketch of the RT graph data structure is given in Figure 5.15. The three primary data structures are the parts table, the link table, and the boundary, corresponding to the three top-level elements of composed programs. There is only one boundary in a program, so it is a solitary structure. The structure representing a part contains all the information pertaining to that part, including a list of sockets. Likewise, the boundary contains a list of sockets. The link structure associates two sockets.

The next set of modules within the prototype manage the appearance of the RT graph on the sketch pad. The sketch contains representations of the objects of composed programs. The visual representation of these objects may have several visual components; parts have sockets and annotation buttons, the boundary has embedded sockets. The principle operations for the sketch pad are as follows:

- Place the visual representation of an object at a given position on the pad.
- Delete an object from the sketch pad.
- Convert a sketch pad coordinate to an underlying object and component of the object.

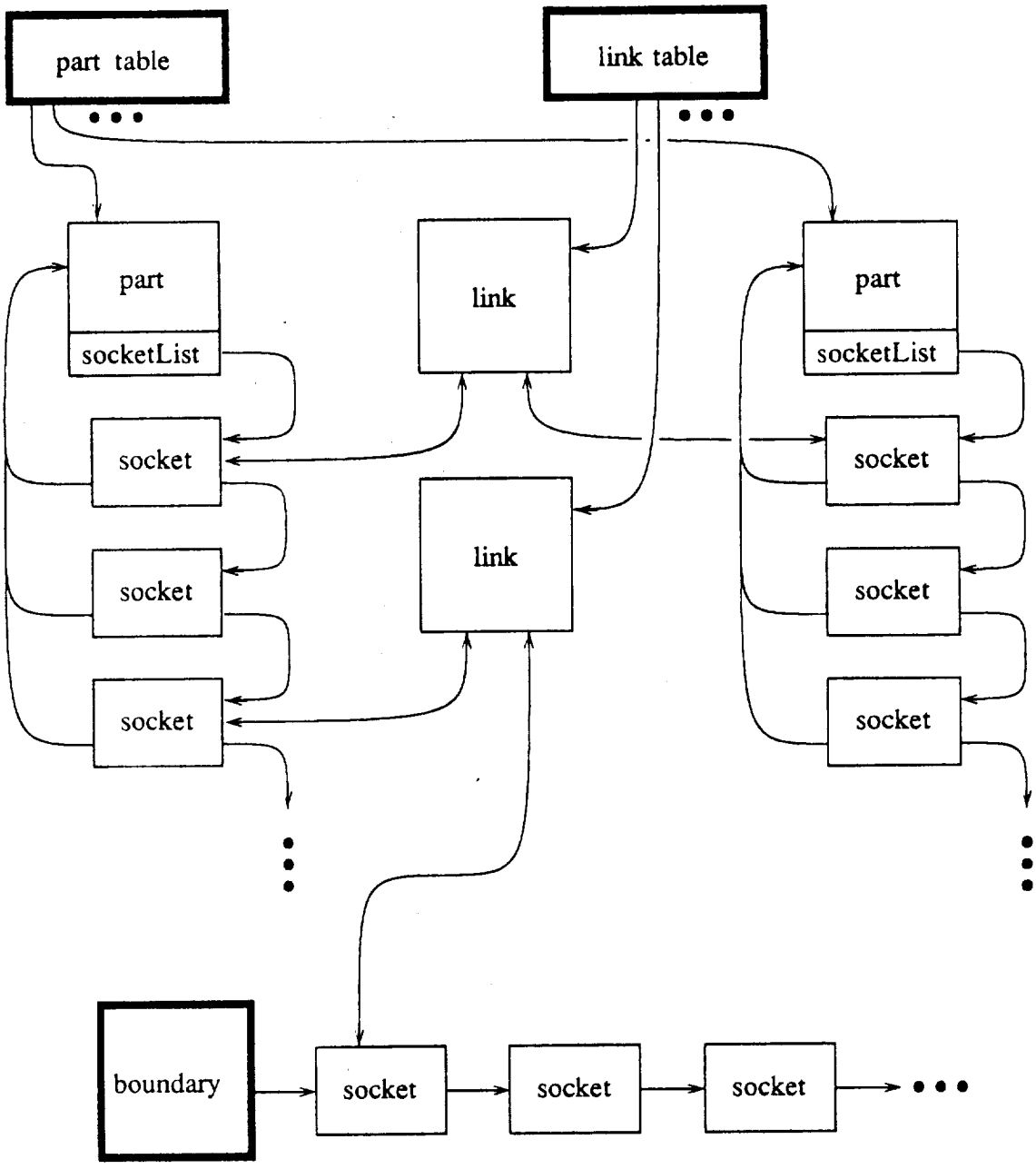


Figure 5.15: The RT Graph Data Structure

- Redisplay the entire graph.

Each object in the graph, parts, links, *etc.*, has a visual representation. The display list structure principally exists to encode the placement of objects on the sketch pad and provide a mapping from sketch pad position to underlying object. The last operation above, redisplaying the graph, establishes the requirement to encode the entire representation in a structure. A structure that describes all visible objects suffices for redisplaying the graph. Within the prototype, this structure is called the *display list*. The display list is a linked list of all objects displayed on the sketch pad. Each object comprises components, called *pieces*, that constitute the image of the object. Each piece occupies space on the sketch pad; this space is the union of a set of rectilinear regions named *minimum bounding boxes (MBB)*. As Items are added to the sketch pad, they are added to the display list.

Items on the display list correspond to the major components of the program: parts, links, and the border. Pieces within display list items correspond to the smallest selectable item on the sketch pad: sockets, annotation buttons, argument buttons, *etc.* Associated with each piece is an operations vector stating the entry points of each of the editing operations that can be applied to an item on the sketch pad: highlight, activate, select, delete, move, *etc.* The MBBs are the smallest unit of allocation of sketch pad space.

The operation of mapping a sketch pad location to a visual object and component (piece) of that object can be performed by traversing the display list until an object is located that contains a piece that contains a MBB that covers the given location. The time to perform a lookup by traversing the display list is proportional to the number of objects displayed, the number of pieces in them, and the number of MBBs that describe their locations. To optimize the lookup operation, the prototype also maintains a *display map* which maps directly from an (*x*, *y*) pair to the object that covers that location. With the display map, converting a sketch pad location to the object covering it can be done in constant time. However, adding an object to the display, and hence the map, now takes time proportional to its size. Mapping position to object occurs whenever the user points to an object on the sketch pad, requesting some action; the pointing operation is usually embedded within a longer sequence of commands. Adding an object to the sketch pad happens less frequently and is usually the last step of a sequence of operations. Hence, we chose to optimize the former at a slight cost of the latter.

Figure 5.16 graphically shows the display list structure and a portion of the display map. The boxes represent data objects, the names above the boxes denote type names, and the names inside the boxes denote field names.

5.6. Summary

This section has presented details of the program development environment and the prototype graphical editor used to verify the concepts underlying that environment. We have described the visual representation of the components of parts-based programs, how they relate on a sketch pad, how a user uses the

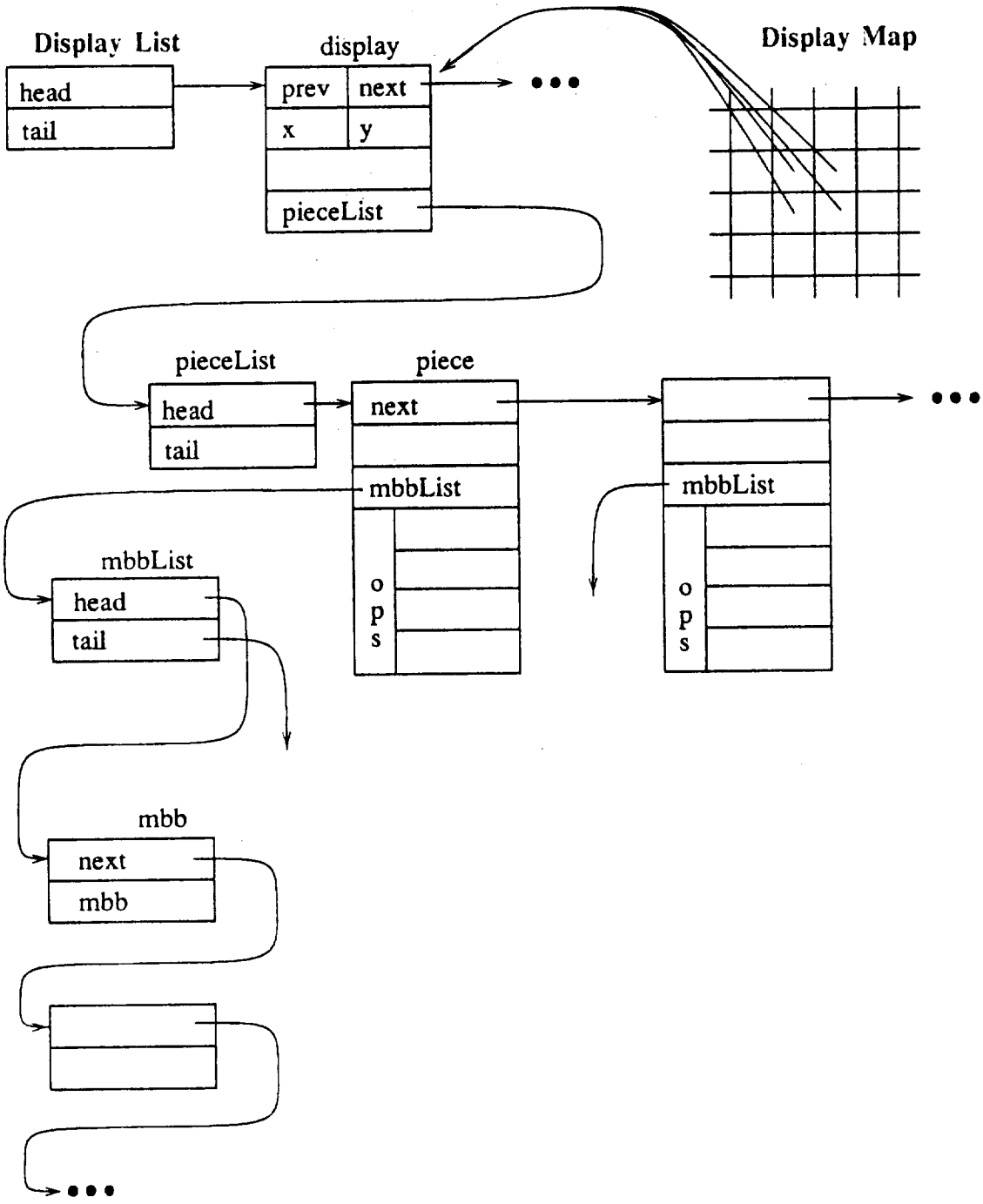


Figure 5.16: The Display List and Map Data Structures

graphical editor to construct a program, and how data structures can be designed to support such an editor.

The prototype development system is a large piece of software; we gained experience from designing it, implementing it, and using it. From the experience of constructing and using the prototype development system, we derive the following principles and conclusions:

1) It is possible to build a visual programming language for parts-based programming based on high-level static data flow graphs. Every top-level component in the language can have a visual representation.
2) Visual languages can retain textual annotation information without destroying the inherent visual quality of the language. By making the annotation invisible and allowing it to be quickly made visible, it does not clutter the visual program representation.
3) The visual language PCS is well matched to the parts-based programming paradigm described in Section 4. The diagrams produced by using the editor match the abstract diagrams we use to describe parts-based programming.
4) Common tools for constructing visual programming languages do not exist; there is nothing analogous to a compiler generator. As a result, researchers in visual programming languages face the task of implementing their compilers from primitive pieces, a difficult task.
5) Data structures that manage the program under construction and its display can be designed in a way that operations on them do not grow as the size of the program grows. The exception is invocation, where every part and every link must be created individually.

6. EXTENSIONS FOR A DISTRIBUTED SYSTEM

6.1. Introduction

The purpose of this section is to explore the possibilities and implementation of an extension to PCS that allows it to manage programs whose computational parts span multiple machines interconnected by networks. This extended system, DPCS, is functionally a superset of PCS. Its graphical interface is identical. Additional information must be associated with parts to implement the remote execution and hence, the invocation algorithm differs significantly.

Two motivations exist for extending PCS into DPCS. The first is to take advantage of the diversity of computing resources that are available on a network. For example, when PCS runs all of its program parts on the workstation that hosts the development environment, then PCS programs are limited to those computations and computational rates achievable by that workstation. By allowing program parts to run on any machine reachable by networks, programmers can use specialized resources or more powerful computers in their computations.

The second reason for extending PCS into DPCS is to take advantage of potential concurrency available when multiple processors are put to work on a computation. DPCS does not explicitly provide concurrent execution; that is a function of the implementation of a particular DPCS program. For example, if the procedure call emulation style of communication is used throughout a DPCS program, modeling a single thread of control, no opportunity for concurrency exists; only one part will be running at a time. However, if a pipelining or rendezvous model of communication is used, concurrency becomes inherent.

The goal of the extension is to allow a programmer of user to construct networked programs without the need to understand the intricacies of the underlying computer networks, their interfaces, or how to manage different types of resources explicitly. DPCS seeks to hide these details by virtualizing the network in the same way the operating system in Section 3 virtualizes the underlying computing machine.

6.2. New Problems

Extending PCS into DPCS introduces a new set of problems to be solved. These problems are as follows:

1) Binding parts to machines. If only one machine is used in the execution environment of PCS, no decision concerning which machine to use to run the parts was necessary. In DPCS, because each part can run on a different machine, there needs to exist a binding between part and machine.
2) Remote invocation. The virtual machine model specifies how to create processes on the local machine. This mechanism must be extended across the network.

3) Accommodating heterogeneity. Each machine in a DPCS computation can potentially have a different internal data representation and different scheme for naming resources.

We address these new problems individually, examine the alternatives, and make choices for how to solve them within DPCS. No absolute solutions exist for these problems; the alternatives must be weighed and the best choices for a given environment made.

6.2.1. Binding Parts to Machines

The invocation algorithm within DPCS needs to select, for each part, a machine on the network to use to execute the code of that part. This information takes the form of a mapping from part to machine and that mapping must be available when DPCS starts executing a part. There exists a spectrum of possibilities for the specification of this mapping, ranging from a very tightly prespecified mapping from part to machine identifier to a very loose description resolved at the latest possible moment. The result generated by the part-to-machine mapping is a unique identifier, such as a network address, used by the program invocation phase of DPCS to identify the machine on which that program part is started. DPCS uses a method that associates a named set of machine identifiers with each part. Collectively, the set of sets of machine identifiers is associated with the local program development environment, allowing the final binding to a specific machine to be a function of the local environment where the development system runs. The names of the sets of machines come from a flat namespace, though that namespace could be hierarchically structured. A hierarchical namespace should, for convenience, mirror some natural hierarchy of the machines available to a DPCS computation. A flat namespace presents a simpler approach, not depending on any inherent organization available computing resources. Hence, for simplicity, we chose a flat namespace for the names of machines sets. Thus, associated with each part is one of these names referencing a set of machines. The sets must be customized for each local environment that uses DPCS.

A problem resulting from associating a machine set with each part is that the invoker in DPCS does not have any information about how to select a member from the set on which to run the part. A trivial alternative is to always select the first element of the set, in which case all parts that name a particular set run on the machine named first in the set, an inefficient method. An extension to the flat namespace of set of machines is to allow expressions to be specified over those sets and then to associate those expressions with the parts. Operators in the expression language add the ability to select, for example, the most lightly loaded machine in the set, a machine not previously selected in this invocation, and a machine at random. The prototype, however, restricts the machine specification to a simple set, and the invocation algorithm selects the next host in the list, according to some ordering relation.

The form of the host specification for DPCS parts, in NDL notation, is as follows:

(machine "set-name")

The "set-name" corresponds to a name of a set of machine identifiers that is loaded as a part of the site-specific configuration information when the DPCS invocation algorithm begins.

By using abstract set names in parts, the programmer can constrain the possible target machines for any part. Because the mapping from abstract set name to specific machine identifiers is dependent on the local development environment, DPCS programs can be written in a way that allows them to move from one network environment to another without change. For each invocation environment, however, the programmer, or systems administrator, must providing the mappings from abstract machine set names into specific machines. By properly defining and using a standard set of abstract names, DPCS programs can become transparently portable if machines of all needed abstract sets are available in the new environment.

Alternatives exist for binding parts to machines. It may be the case that, for some parts, the choice of machine used to run its code is immaterial. Yet for other parts there may be only a single unique machine that can possibly run its code. In either case, there must exist static information associated with the part that states, in some formal way, what constraints exist on where that part runs. Such a constraint may take the form of naming a specific machine, in which case we say the mapping is *tightly bound*, or it may specify a set of machines or an algorithm for locating a machine, in which case we say the mapping is *loosely bound*. Similarly, the binding to machine may be associated with a part when it is created, which we call *early binding*, or determined when it is invoked, which we call *late binding*. Early/late binding and loose/tight binding are not distinct concepts. Early binding implies tight binding and late binding implies loose binding. The degree of looseness, that is, how the binding information is given, is the topic of primary concern.

Another possibility for loose binding is to have the available machines decide whether they can run a part and if so, with what degree of performance, a voting scheme. Voting requires that each part statically state their resource requirements. Wills describes schemes of this nature [Will88].

Loose binding provides more flexibility than tight binding. When a part is tightly bound to a specific machine identifier, that machine must be reachable from the development environment and must be available at the time of invocation. If a composed program with using early and tight binding is moved from one network environment to another, the programmer must manually change the binding on all internal parts before the program can be invoked. If a composed program uses loose binding, whether early or late, then, depending on the form of the loose binding, the composed program can be invoked in more network environments than the one in which it was created.

The binding information in a part takes the form of an expression which, when evaluated, results in a machine identifier on which the composition system will run that part. Components of that expression can include a specific machine identifier, a

set of machine identifiers, or a set of characteristics of machines capable of running that part. The characteristics may be static or dynamic. Static characteristics concern permanent attributes of machines such as data representations, instruction set, speed, memory size, and specialized resources attached. Dynamic attributes include load on the machines, available network bandwidth to it, or long-term storage currently available.

6.2.2. Remote Invocation

The second major change in DPCS to accommodate networks computations is the problem of invoking program parts remotely. The issues involved are naming of remote programs, issuing the remote analogy of the createProcess procedure, and establishing the communication paths that implement the links of a DPCS program.

First, we assume that we can communicate with resource managers, implemented as network servers, on any remote machine usable by DPCS programs. Network servers are programs that can be used by remote machines by establishing network connections to them and then sending requests and receiving responses. Our assumption relies on the availability of standard network protocols and standard techniques for connecting to network servers. Given this capability, we can create a server dedicated to invoking DPCS program parts on any remote machine. The server uses the virtual machine interface provided by the operating system to locate program parts, using the module name from the part specification, and to turn the pat module into a running process. Abstractly, the interface between DPCS and any remote invocation server mimics the interface to the createProcess procedure of the local operating system, but results in a remote procedure call to the corresponding procedure in level 13 of the remote machine.

6.2.3. Accommodating Heterogeneity

The most difficult issue in extending PCS to a networked system is accommodating the data representation heterogeneity inherent when varied computing resources are available on a network. This section examines the issues surrounding data representation heterogeneity and offers partial solutions that enable a large class of DPCS programs to run on a network offering such variety. Accommodating representation heterogeneity relies on a well-defined data typing scheme for sockets. The representations, a run-time attribute, derive from the typing scheme, a static attribute.

An example of the type of heterogeneity that may be encountered is in the representation of binary integers. One machine may represent a binary integer by ordering the bits in the value from high-order to low-order, while another machine may reverse that, placing the bits from low-order to high order.

Heterogeneity causes more problems than in the representations of common primitive data types. One machine may inherently store more information with a piece of data (*e.g.*, storing more precision in a floating-point number) than another machine. One machine may have a primitive data type not represented on another machine.

When aggregate data types are used by program parts and passed from one machine to another, the way in which the elements of the aggregate type are ordered or packed together into a unit may be different from machine to machine.

Accommodating data heterogeneity requires a translation from the representation of one machine into the representation of another. When there are N data representations offered by the union of machine types on the network, each machine must have access to algorithms to translate its data representations to each of $N-1$ other machines. Providing all possible translations results in $N \bullet (N-1)$ different data translation algorithms. Another approach is to adopt a "standard" data representation format, and, on each machine, provide an algorithm that translates from the native representation into the standard representation, and another that translates from the standard in the native representation, resulting in $2 \bullet N$ translation algorithms. The selection of the standard notation, however, is difficult.

When the range of a particular data type differs between communicating machines, and data flows from the machine of higher precision to the machine of lower precision, a data truncation must occur. When there are three or more machines communicating, and all three have different representation precisions, truncations may occur to differing degrees. A standard data representation must either be designed to handle the most precise (i.e. largest) data format for each particular type on the network, or must encode the length or precision of the data. Accommodating the longest representation results in storage, and hence transmission time, inefficiency and cannot accommodate all possible future representations. Encoding the length of a data item in the data stream, however, allows the representation to accommodate future machines using longer data representations unforeseen by the designer of the standard representation.

The standard representation must be compact and efficient to translate to and from. A possibility for standard representation is to use all decimal coded ASCII, using LISP-like S-expressions to encode structures. Functionally, ASCII encoding of data suffices, but it is an inefficient encoding because the cost of translating into and out of it is high. We performed a small experiment to verify the high cost of decimal encoding. Two programs were written, one a producer, one a consumer. The producer generated an array of N random floating-point numbers and sent a message containing N and the array to the consumer. When the consumer had read (and converted) the data, it sent a control signal back to the producer denoting completion. The producer timed how long it took to translate and send the message and wait for the response. The time to generate the random array was not counted. The same input/output buffering package was used in both cases. Table 6.1 shows the results when unconverted binary-format data was transmitted and when the data were converted to ASCII representation. For small values of N, the resolution of the computer's real-time clock and the overhead associated with just transmitting the data introduced errors. For large values of N, the ASCII encoding multiplied the time over one hundredfold.

Table 6.1: Transmission Times for ASCII and Binary

	ASCII Coded (ms)			Unencoded (ms)			
N	Mean	s	n	Mean	s	n	Mean Speed Up
250	40.0	0.00	12	425.0	25.76	12	10.6
500	41.7	5.77	12	810.0	49.36	12	19.4
1000	48.3	13.37	12	1585.0	86.60	12	32.8
2500	65.0	9.05	12	3895.0	198.01	12	59.9
5000	105.0	15.08	12	7656.7	51.76	12	72.9
10000	160.0	8.53	12	15303.3	11.55	12	95.6
25000	340.0	12.06	12	38145.0	191.29	12	112.2
50000	643.3	7.78	12	76271.7	393.63	12	118.6

Existing technology can be used to implement message passing between machines having differing data representations. Message systems comprise a producer and a consumer and an optional message queue between them, as shown in Figure 6.1. When the originator of a message needs to send a message, it makes a procedure call to the producer procedure which in turn places the message in the message queue and returns. When the receiver needs to read a message, it makes a procedure call to the consumer which returns the next message in the queue, or waits for one if the queue is empty. The parameters to the producer procedure contain the message to send; the parameters to the consumer module reference where to deposit the message in the receiver's memory space. Hence, the producer and consumer procedures only manage the message queue, hiding even its existence from the application program using them. The writer and reader in the application uses a straightforward procedure call interface to communication.

The producer-consumer example of communication can be extended to manage communication over a network by using a remote procedure call (RPC) between the writer and the producer, or between the reader and the consumer. Using the former case as an example, when the writer needs to transmit a message to the reader, it, as before, makes a procedure call to the producer procedure. However, now the procedure call is a remote procedure call managed by an RPC system that can translate the procedure parameters from one representation to another [Bers87].

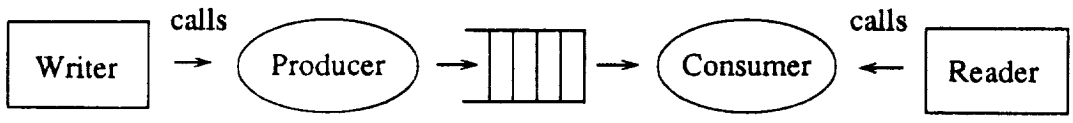


Figure 6.1: Producer-Consumer Communication

Because the content of the message is encoded in the parameters to the producer procedure, and RPC systems translate procedure parameters, the message is automatically translated. Figure 6.2 shows this modified message scheme.

6.3. Summary of Principles

From direct experience and the study of the requirements for extending PCS into DPCS, thus allowing it to invoke networked computations, come the following fundamental principles:

1) Early binding of parts to machines is too restrictive. Early binding results in program graphs that are specific to a particular development and invocation environment and are not transportable to other environments.
2) Late binding of parts to machines can be achieved by defining abstract sets of machines. Each part names an abstract machine set, and the definition of the members of the set is dependent on the local environment. Additional power can be added to abstract set selection by defining an expression syntax over the abstract sets.
3) Remote invocation of program parts can be accommodated by providing the createProcess primitive on all machines in a networked server accessible from all other machines on the network. This technique, however, relies on a standard network protocol that must be available on all machines participating in a DPCS computation.
4) Accommodating data representation heterogeneity efficiently requires the definition of a standard data representation, and then providing translation algorithms from and to each native data representation
5) The standard data representation must accommodate differing data lengths and precisions efficiently, by allowing data items to be tagged with the length of the data in standard representation.
6) Existing remote procedure call mechanisms, when augmented with queue management procedures, can be used to used to pass messages from one part to another, accommodating the data translations as necessary.

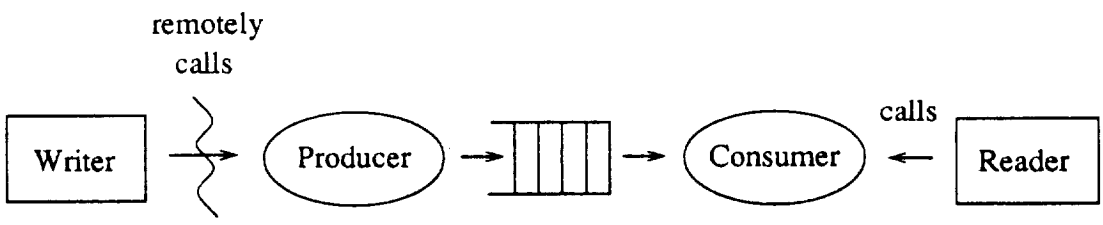


Figure 6.2: Message Passing Using RPC

7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusions

In this report we have introduced and elaborated upon a program development environment that offers a new, higher level, programming language, allowing creation of very large programs. The fundamental composition mechanism used herein is message passing, whereas traditional program composition techniques use procedure calls for composing large programs out of smaller units.

Additionally, programs composed with our system are a good match to a network of diverse computing resources. Network communication links are message passing devices and networked computing resources can execute programs that transform data. Hence, our representation-transformation (RT) graphs match the architecture offered by a set of computers interconnected by a network.

Just as the RT diagrams match networked computations well, our visual programming language matches the RT program graph structure, and the three components together construct a programming system exhibiting a high degree of coherence. The visual language allows the programmer to work directly from a RT diagram without the tedious task of translating it into an unnatural textual representation.

The strong points of visual parts-based composition, as derived from experience with our prototype, are as follows:

1) The model of computing using parts and message links is conceptually simple and closely matches the way we already think about networked computations. It can be used to create programs that automate procedures we already use as well as create programs that go far beyond what we might otherwise try.
2) Parts-based programs map cleanly onto the architecture of a network of computers.
3) The visual nature of the language allows users to write programs in a language that closely matches a way of thinking about them. There is no manual conversion from a conceptual structure diagram to a textual description language.

Parts-based programming with our prototype has its limitations, though. Some of these are the result of limitations in the prototype itself, others may be inherent in the approach. Some limitations are as follows:

1) At times, a low level operator, like one to split a data stream, is required in a DPCS program. Because all primitive program parts are defined outside the paradigm, creating such a low level part requires that the user enter a different paradigm.
2) Describing and accommodating data representation is difficult. Ideally, the composition system should know the structure of the data placed on a link by a part, and arriving at a part from a link. Such knowledge can be provided by the internal coding of the part itself or be associated independently of the part coding

with the library entry for the part. The former approach, though easier to implement, fails to satisfy the "immutable software" design goal and the latter approach increases the burden on the user who adds parts to the library.

3) DPCS does not suffice as an interactive command shell. Approaches to solving this limitation are discussed in the next section.

7.2. Future Work

There are many incompleted facets of our program composition system prototype, many of which are an entire research project in themselves. This section discusses some of them and proposes directions for future investigators to follow. PCS is an ongoing project; many of the items described here are work in progress or in the plans for the future. Current work is focussed on completing the extension of PCS into DPCS, the distributed programming version.

7.2.1. DPCS as an Interactive Shell

DPCS is a program development environment that a programmer uses to create applications. It's capabilities are limited, however, in that it cannot reasonably be used as an interactive command interpreter on a regular basis. One reason for this deficiency is that the programmer is required to create a complete program before any part of it can be invoked. All external connections must be explicitly tied to the boundary and decisions must be made explicitly about how to attach those boundary sockets to objects.

DPCS would be better suited to interactive use with the addition of more implicit actions and defaults. For example, if it would allow the execution of a partially constructed program graph, and attach all the external connections to windows, the programmer could quickly select a part or two and try them out. If links were automatically added, the process of constructing a program would be faster. In Figure 5.6, for example, it is clear from the parts placement how most of the sockets should be linked.

DPCS is oriented towards the construction of very large programs built up from parts. Interactive work is typically of a different nature: making queries on databases, manipulating file spaces, entering modal programs such as document preparation and spreadsheet programs, *etc.* DPCS could be augmented with a special library parts to support common interactive work, but its use would still be clumsy. DPCS uses modes extensively whereas most command interpreters are modeless (that is, they have only one mode). The sequence of events to build and run a single part program involve eight distinct steps, long process for a simple task, especially when compared to the simple "type command, hit RETURN" style of most command interpreters. The steps involve numerous spatial operations, moving the pointer from place to place, whereas typing requires very little spatial movement.

Questions concerning using DPCS as an interactive shell for regular use include the following:

1) Is a visual interface better than typing for a general-purpose command language interface?
2) What changes need to be made to DPCS to make it easier to use for smaller tasks?
3) Within DPCS, how does one maintain the parts databases in a way that allows fast access to commonly-used parts?

7.2.2. Debugging and Monitoring

Debugging support is an essential part of any programming environment. DPCS, by design, supports debugging through monitoring, that is, it allows the programmer to "watch" what is passing through the links. This feature is not implemented in the prototype. Ideally, one should be able to designate any link and ask the environment to show the data passing through it. There is no support for this capability in the links as defined by the virtual machine interface, though such capability, once designed, could be added. In lieu of built-in support, the programmer can insert taps manually into the program graph, such as shown in Figure 7.1. In this figure, the programmer has removed a link from the graph generated in Section 5 and inserted a module that replicates the data stream. Then, one replicated stream is spliced back into the main

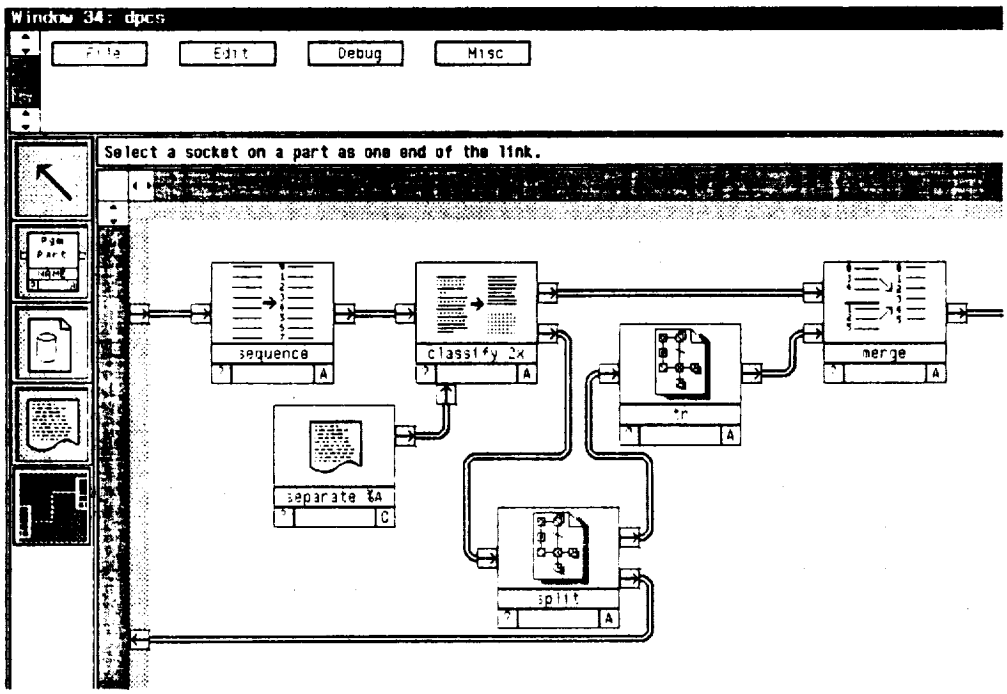


Figure 7.1: A Manually Inserted Tap

graph and the other is attached to the boundary where it can subsequently be associated with a window.

A problem that arises when considering how to tap into a communication link is buffering. When there is only one reader and one writer on a link, buffering is simplified because the communication reduces to the producer-consumer problem. However, with two readers (one added in for viewing the data), each consumer may read the data at different rates, and the slower one would govern the overall rate of data flowing through the pipe.

Another problem that arises is how to display the data. The example program in Section 5 avoided this problem by using only lines of text as the type of data passed between parts. If more complicated types are used, then they may not be as easily displayed. The extended types mechanism in the virtual machine (level 14) can provide help. This level provides a mechanism whereby type managers for extended types can register themselves with the system. Once that has been done, then any other program, such as DPCS, can use the facilities of that manager. Specifically, each type manager should include a routine for translating its own objects into a form that can be displayed, textually or graphically, on the workstation screen.

Unanswered questions about debugging and monitoring include the following:

1) What impact does monitoring the data in a link have on the global behavior of the execution of the graph?
2) Does restricting tap insertion to program load time, as opposed to run time, have a significant impact on the usefulness of this feature?
3) What operations must be added to the virtual machine to allow the programmer to insert taps on links after the program has started running?
4) What is the registration and naming mechanism for extended type managers in level 14 of the operating system?
5) What external representations are used for the data types and how are they best displayed? Can this be done using a universal mechanism?

7.2.3. Additional Topics

DPCS manages static data flow graphs. If dynamic graphs were supported, more powerful computations may be possible. The prototype establishes a clean break of control between graphical editing and program invocation; the editor does not impose its presence when the graph is executing. If dynamic node creation is allowed, and the dynamically-created nodes were to be shown on the sketch pad, the editor and invoker would necessarily be more tightly coupled.

DPCS contains no built-in control structures. We have found that conditionals and some loop forms can be simulated by classifiers (for conditionals) and stream-generating parts (for loops). Parts can be coded to compute on multiple input objects. It is trivial to create a part the generates a sequence of numbers, similar to the APL operator *iota*. Such a part exists in our prototype library. Other types of control

structures may exist, though. For example, a part duplication structure could exist. Part duplication would allow the programmer to specify, for any part on the sketch pad, that that part be replicated some number of times, and that each input object be directed to any one instance of the part. Replication in this way offers more opportunity for concurrency in the program, but adds the task of reassembling the outputs into a single stream.

7.3. Summary

In this report we have presented a model for parts-based programming, described the semantics and a visual syntax, and raised the issues surrounding making parts-based programming a reality. The following states the primary research contributions of this work.

The model of computing based on composing program parts into larger applications by joining them together with message-passing links has been presented. The model describes the semantics of pipe-based construction of representation-transformation program graphs that map into executable programs in a way that allows the transformation nodes to run on any available machine on the network.

A visual programming language described parts-based programs in a straightforward way that does not require programmers to manually convert a RT flow diagram into a textual representation. Instead, the RT diagram can be "drawn" on a sketch pad directly.

A working, usable prototype composition system that implements the visual language and maps programs onto a network of computers demonstrates the usefulness of our approach. This prototype allows programmers to enter their program diagrams, assign program parts to computer nodes, and invoke their programs.

Finally, we have identified issues that have not yet been resolved concerning parts based programming on a distributed system. One of the most difficult is the accommodation of data representation heterogeneity present on a network offering a diversity of computing resources. The next stage in the development of the prototype is to investigate practical solutions to the representation heterogeneity problem that are coherent with the other components of the prototype.

LIST OF REFERENCES

[Acce86] Accetta, Mike, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," *Usenix Technical Conference*, Atlanta, Georgia, pp. 93-113 (1986).

[Acke79] Ackerman, W. B. and J. B. Dennis, "VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual," MIT/LCS/TR-218, Massachusetts Institute for Technology, Cambridge, MA (June, 1979).

[Acke82] Ackerman, W. B., "Data Flow Languages," *Computer*, 15(2), pp. 15-25 (February, 1982).

[Andr83] Andrews, Gregory R. and Fred B. Schneider, "Concepts and Notation for Concurrent Programming," *Computing Surveys*, 15(1), pp. 3-43 (March, 1983).

[Andr88] Andrews, Gregory R., Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kevin Nilsen, Titus Purdin, and Gregg Townsend, "An Overview of the SR Language and Implementation," *ACM Transactions on Programming Languages and Systems*, 10(1), pp. 51-86 (January, 1988).

[Appl85] Apple Computer, Inc., *Inside Macintosh, Volume I*, Addison-Wesley, Menlo Park, CA (1985).

[Babb85] Babb, R. G. II, "Programming the HEP with Large-Grain Data Flow Techniques," pp. 203-227 in *Parallel MIMD Computation: HEP Supercomputer and its Applications*, ed. Janusz S. Kowalik, The MIT Press (1985).

[Baec75] Baecker, R. M., "Two Systems Which Produce Animated Representations of the Execution of Computer Programs," *ACM SIGCSE Bulletin*, 7(1), pp. 158-167 (February, 1975).

[Baec81] Baecker, R., *Sorting Out Sorting*, Dynamic Graphics Project, Computer Systems Research Group, University of Toronto, Toronto, Canada (1981).

[Banc88] Bancroft, Gordon and Fergus Merrit, "3-D Graphics Applications in Fluid Flow Simulations," *Proceedings of the 2nd IEEE Conference on Computer Workstations*, Santa Clara, CA, pp. 142-147 (March 7-10, 1988).

[Baro86] Barok, Amnon and On G. Paradise, "MOS - Scaling Up UNIX," *Usenix Technical Conference*, Atlanta, Georgia, pp. 93-113 (1986).

[Batc85] Batcher, Kenneth E., "The Massively Parallel Processor System Overview," pp. 142-149 in *The Massively Parallel Processor*, ed. J. L. Potter, The MIT Press (1985).

[Bela84] Belady, L. A. and K. Hosokawa, "Visualization of Independence and Dependence for Program Concurrency," *Proceedings of the 1984 IEEE Workshop on Visual Languages*, Hiroshima, Japan, pp. 59-63 (1984).

[Bere86] Beretta, M., P. Mussio, and M. Protti, "Icons: Interpretation and Use," *1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, pp. 149-158 (June, 1986).

[Bers87] Bershad, Brian N., Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Transactions on Software Engineering*, SE-13(8), pp. 880-894 (August, 1987).

[Bour78] Bourne, S. R., "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.*, 57(6), pp. 1971-1990 (1978).

[Brow84] Brown, Robert L., Peter J. Denning, and Walter F. Tichy, "Advanced Operating Systems," *IEEE Computer*, 17(10), pp. 173-190 (October, 1984).

[Brow85a] Brown, G. P., R. T. Carling, C. F. Herot, D. A. Kramlich, and P. Souza, "Program Visualization: Graphical Support for Program Development," *IEEE Computer*, 18(8), pp. 27-35 (August, 1985).

[Brow85b] Brown, Marc H. and Robert Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, 2(1), pp. 29-39 (January, 1985).

[Brow88a] Brown, Robert L., Joe Barerra, and William Lynch, *Implementation of a Concurrent C and Debugger at RIACS*, RIACS (1988). (In preparation).

[Brow88b] Brown, Robert L., "DPCS User's Manual," RIACS TR 88-XX, Research Institute for Advanced Computer Science (1988). (In preparation)

[Catt86] Cattaneo, G., A. Guerico, S. Levialdi, and G. Tortora, "IconLisp: An Example of a Visual Programming Language," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 22-25 IEEE Computer Society, (June, 1986).

[Chan86] Chang, S. K., Q. Y. Shi, and C. W. Yan, "Iconic Indexing by 2D Strings," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 12-21 IEEE Computer Society, (June, 1986).

[Chan87] Chang, Shi-Kuo, "Visual Languages: A Tutorial and Survey," *IEEE Software*, 4(1), pp. 29-39 (January, 1987).

[Chen86] Cheng, K. Y., C. C. Hsu, I. P. Lin, M. C. Lu, and M. S. Hwu, "VIPS: A Visual Programmer Synthesizer," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 92-98 (June, 1986).

[Cher78] Cheriton, David C., "Multi-process Structuring and the Toth Operating System," Ph.D. Thesis, University of Waterloo (1978).

[Cher84] Cheriton, David R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, 1(2), pp. 19-42 (April, 1984).

[Chio87] Chiola, Giovanni, "GreatSPN User's Manual," Draft 30/9/87, Dipartimento do Informatica, Universita degli Studi di Torino, Torino, Italy (September, 1987).

[Come84] Comer, Douglas, *Operating System Design, the XINU Approach*, Prentice-Hall, Englewood Cliffs, NJ (1984).

[Conw63] Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler," *Communication of the ACM*, 6(7), pp. 396-408 (July, 1963).

[Davi82] Davis, A. L. and R. M. Keller, "Data Flow Program Graphs," *Computer*, 15(2), pp. 26-41 (February, 1982).

[Denn66] Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, 9(3), pp. 143-155 (March, 1966).

[Denn80] Dennis, T. Don, "A Capability Architecture," Ph.D. Thesis, Purdue University, West Lafayette, IN (1980).

[Denn83] Denning, Peter J. and Robert L. Brown, "Should Distributed Systems be Hidden?," CSD-TR-426, Purdue University, West Lafayette, IN (1983).

[Denn84] Denning, Peter J. and Robert L. Brown, "Operating Systems," *Scientific American*, 251(3), pp. 94-106 (September, 1984).

[Denn86] Denning, Peter J., "Parallel Computing and its Evolution," *editorial in Communications of the ACM*, 29(12), pp. 1163-1167 (December, 1986).

[Diaz80] Diaz-Herrera, J. L. and R. C. Flude., "PASCAL/HSD: A Graphical Programming System," *Proceedings of COMSAC 80*, Los Alamitos, California, pp. 723-728 (1980).

[Dijk68] Dijkstra, Edsger W., "The Structure of the THE-Multiprogramming System," *Communications of the ACM*, 11(5), pp. 341-346 (May, 1968).

[Dong86] Dongarra, J. and D. Sorensen, "A Portable Environment for Developing Parallel Fortran Programs," Technical Memorandumn No. 79, Argonne National Laboratory, Argonne, Illinois (July, 1986).

[Engl68] Englebart, Doug C. and W. K. English, "A Research Center for Augmenting Human Intellect," *Proceedings of the 1968 Fall Joint Computer Conference*, San Francisco, pp. 395-410 AFIPS, (1968).

[Finz84] Finzer, W. and L. Gould, "Programming by Rehearsal," *Byte*, 9(6), pp. 187-210 (1984).

[Form86] Forman, Ira R., "Raddle: An Informal Introduction," TR No. STP-182-85, MCC, Austin, TX (February, 1986).

[Glin84] Glinert, Ephraim P. and Steven L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *IEEE Computer*, 17(11), pp. 7-25 (November, 1984).

[Glin86] Glinert, E. P., "Towards Second Generation Interactive, Graphical Programming Environments," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 61-70 (June, 1986).

[Gold84] Goldberg, Adele, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Menlo Park (1984).

[Gome85] Gomez, Julian E., "Twixt: A 3D Animation System," *Computers and Graphics*, 9(3), pp. 291-298 (1985).

[Graf87a] Graf, Mike, "A Visual Environment for the Design of Distributed Systems," *1987 Visual Language Workshop*, Linkoping, Sweden, (August, 1987).

[Graf87b] Graf, Mike, "The Design of a Distributed System Using a Visual Language," TR No. STP-319-87, MCC, Austin, TX (October, 1987).

[Habe76] Habermann, A. Nico, Lawrence Flon, and Lee W. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communications of the ACM*, 19(5), pp. 266-272 (May, 1976).

[Haeb86] Haeberli, Paul E., "A Data-Flow Manager for an Interactive Programming Environment," *USENIX 1986 Summer Technical Conference*, Atlanta, Georgia, pp. 419-418 (1986).

[Hagi84] Hagiwara, Nor ko and Kanji Iwamoto, "A Graphical Tool for Hierarchical Software Design," *1984 Workshop on Visual Languages*, Hiroshima, Japan, pp. 42-46 (September, 1984).

[Hail86] Hailpern, Brent, "Multiparadigm Languages and Environments," *IEEE Software*, 3(1), pp. 6-9 (January, 1986).

[Hill85] Hillis, W. D., *The Connection Machine*, The MIT Press, Cambridge, Mass. (1985).

[Hira86] Hirakawa, Mashhito, Noriaki Monden, Iwao Yoshimoto, Minoru Tanaka, and Tadao Ichikawa, "HI-VISUAL: A Language Supporting Visual Interaction in Programming," pp. 233-259 in *Visual Languages*, ed. Shi-Kuo Chang, Tadao Ichikawa, Panos A. Ligomenides, Plenum Press, New York (1986).

[Hoar78] Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, 21(8), pp. 666-677 (August, 1978).

[IBM67] IBM, "IBM Operating System/360 Concepts and Facilities (Excerpts)," pp. 598-646 in *Programming Systems and Languages*, ed. Saul Rosen, McGraw-Hill, New York (1967).

[Jaco85] Jacob, R. J. K., "A State Transition Language for Visual Programming," *IEEE Computer*, 18(8), pp. 51-59 (August 1985).

[Jord85] Jordan, Harry F., "Parallel Computation with the Force," ICASE Report No. 85-45, Institute for Computer Applications in Science and Engineering, Hampton, Virginia (October, 1985).

[Kahn77] Kahn, Gilles and David B. MacQueen, "Coroutines and Networks of Parallel Processes," *Proceedings of the IFIP Congress*, pp. 993-998 North Holland, (1977).

[Kahn81] Kahn, Kevin C., William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, Fred J. Pollack, and Michael R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, CA, pp. 127-136 (December, 1981).

[Kern78] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

[Korf86] Korfhage, Robert R. and Margaret A. Korfhage, "Criteria for Iconic Languages," pp. 207-231 in *Visual Languages*, ed. Shi-Kuo Chang, Tadao Ichikawa, Panos A. Ligomenides, Plenum Press, New York (1986).

[Lant80] Lantz, Keith A., "Uniform Interfaces for Distributed Systems," TR63, University of Rochester, Rochester, NY (May, 1980).

[Lars84] Larson, John A. and J. B. Wallick, "An Interface for Novice and Infrequent Database Management Systems Users," *Proceedings National Computer Conference*, pp. 523-529 (1984).

[Lars86] Larson, James A., "Visual Languages for Database Users," pp. 127-147 in *Visual Languages*, ed. Shi-Kuo Chang, Tadao Ichikawa, Panos A. Ligomenides, Plenum Press, New York (1986).

[Lesk78] Lesk, M. E., "Some Applications of Inverted Indexes on the UNIX System," in *UNIX Programmer's Manual*, ed. K. Thompson and D. M. Ritchie, Bell Laboratories (1978). Seventh Edition.

[Lisk72] Liskov, Barbara H., "The Design of the Venus Operating System," *Communications of the ACM*, 15(3), pp. 144-149 (March, 1972).

[Loza79] Lozano-Perez, Tomas and Michael A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Communications for the ACM*, 22(10), pp. 560-570 (October, 1979).

[Luck80] Luckman, David C. and Wolfgang Polak, "A Practical Method of Documenting and Verifying Ada Programs with Packages," *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, Boston, pp. 113-122 (December, 1980).

[Main86] Mainstay, *V.I.P.: Visual Interactive Programming*, Mainstay, Agoura Hills, CA (1986).

[McCa63] McCarthy, J., F.T. Corbato, and M. M. Daggett, "The Linking Segment Subprogram and Linking Loader," *Communications of the ACM*, 6(7), pp. 391-395 (July, 1963).

[Merr81] Merriam-Webster, *Webster's Third New International Dictionary of the English Language Unabridged*, G & C Merriam Company, Springfield, MA (1981).

[Mill84] Mills, Charles C. and Anthony I. Wasserman, "A Transition Diagram Editor," *Proceedings of the 1984 USENIX Summer Conference*, Salt Lake City, Utah, pp. 287-296 (1984).

[Mitc85] Mitchell, Chad, Robert Gardner, Chet Wood, and Boyd Edwards, *Concertware+*, Great Wave Software, Menlo Park, CA (September, 1985).

[Mond84] Monden, Noriaki, Iwao Yoshimoto, Mashhito Hirakawa, Minoru Tanaka, and Tadao Ichikawa, "HI-VISUAL: A Language Supporting Visual Interaction in Programming," *1984 Workshop on Visual Languages*, Hiroshima, Japan, pp. 199-205 (September, 1984).

[Mont86] Montalvo, F. S., "Diagram Understanding: Associating Symbolic Descriptions with Images," *1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, pp. 4-11 (June, 1986).

[Mori85] Moriconi, Mark and Dwight F. Hare, "Visualizing Program Designs Through PegaSys," *IEEE Computer*, 18(8), pp. 72-85 (August, 1985).

[Muus87] Muuss, Michael, Phillip Dykstra, Keith Applin, Gary Moss, Edwin Davisson, Paul Stay, and Charles Kennedy, "Ballistic Research Laboratory CAD Package: A Solid Modelling System and Ray-Tracing Benchmark Distribution Package," Release 1.12, Ballistic Research Laboratory, Aberdeen Proving Ground, Maryland (January, 1987).

[Myer83] Myers, B. A., "Incense: A System for Displaying Data Structures," *Computer Graphics*, 17(3), pp. 115-126 (July, 1983).

[Nass73] Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming," *ACM Sigplan Notices*, 8(8), pp. 12-26 (August, 1973).

[Nels81] Nelson, Bruce Jay, "Remote Procedure Call," CSL-81-9, Xerox PARC, Palo Alto (May, 1981).

[Neum80] Neumann, Peter G., Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System: its Applications, and Proofs," Computer Science Laboratory Report CSL-116, SRI International, Menlo Park, CA (May, 1980).

[Parn69] Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proceedings of the 24th ACM Conference*, pp. 379-385 (1969).

[Parn72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, 15(12), pp. 1053-1058 (December, 1972).

[Parn74] Parnas, D. L., "On a 'Buzzword': Hierarchical Structure," *1974 Proceedings of the IFIP Congress*, Amsterdam, The Netherlands, pp. 336-339 (1974).

[Parn79] Parnas, David L., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, SE-5(2), pp. 129-137 (March, 1979).

[Pete86] Peterson, John W., Rod G. Bogart, and Spencer W. Thomas, "The Utah Raster Toolkit," *USENIX Workshop on Graphics*, Monterey, CA, (November, 1986).

[Petr87] Petrebic, Peter and Phil Goldman, "A Debugger-based System for Graphical Display and Editing of Data Structures," *Proceedings of the 1987 USENIX Summer Conference*, Phoenix, Arizona, pp. 147-158 (June, 1987).

[Pong83] Pong, M. C. and N. Ng, "PIGS - A System for Programming with Interactive Graphical Support," *Software Practice and Experience*, 13(9), pp. 847-855 (1983).

[Pong86] Pong, M. C., "A Graphical Language for Concurrent Programming," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 26-33 (June, 1986).

[Pope81] Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Pacific Grove, CA, pp. 169-177 (December, 1981).

[Prat85] Pratt, Terrence W., "Pisces: An Environment for Parallel Scientific Computation," ICASE Report No. 85-12, Institute for Computer Applications in Science and Engineering, Hampton, Virginia (February, 1985).

[Purv83] Purvy, R., J. Farrell, and P. Klose, "The Design of Star's Records Processing: Data Processing for the Noncomputer Professional," *ACM Transactions on Office Automation Systems*, 1, pp. 3-24 (1983).

[Raed85] Raeder, G., "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, 18(8), pp. 11-25 (1985).

[Reis85] Reiss, S. P., "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, 11(3), pp. 276-285 (1985).

[Rich87] Richmond, Barry, Peter Vescuso, and Steven Peterson, *Stella for Business*, High Performance Systems, Inc., Lyme, New Hampshire (1987).

[Ritc74] Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, 17(7), pp. 365-375 (July, 1974).

[Rock83] Rockart, John F. and Lauren S. Flannery, "The Management of End User Computing," *Communications of the ACM*, 26(10), pp. 776-784 (October, 1983).

[Rohr84] Rohr, Gabriele, "Understanding Visual Symbols," *1984 IEEE Computer Society Workshop on Visual Languages*, Hiroshima, Japan, pp. 184-191 (December, 1984).

[Schw86] Schwetman, Herb, "PPL: A Parallel Programming Language Based on C," MCC Technical Report No. PP-096-86, Microelectronics and Computer Technology Corporation, Austin, Texas (March, 1986).

[Seit85] Seitz, Charles L., "The Cosmic Cube," *Communications of the ACM*, 28(1), pp. 22-33 (January, 1985).

[Shne83] Shneiderman, Ben, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*, 15(8), pp. 57-69 (August, 1983).

[Shu84] Shu, Nan C., "A Forms-Oriented and Visual-Directed Application Development System for Non-Programmers," *1984 Workshop on Visual Languages*, Hiroshima, Japan, pp. 162-170 (September, 1984).

[Shu85] Shu, N. C., "FORMAL: A Forms-Oriented and Visual-Directed Application System," *IEEE Computer*, 18(8), pp. 38-49 (August, 1985).

[Sobe88] Sobek, S., M. Azam, and J. C. Browne, *Architectural and Language Independent Parallel Programming: A Feasibility Demonstration*, Department of Computer Sciences, University of Texas, Austin, Texas (January 15, 1988).

[Spit78] Spitzen, Jay M., Karl N. Levitt, and Lawrence Robinson, "An Example of Hierarchical Design and Proof," *Communications of the ACM*, 21(12), pp. 1064-1075 (December, 1978).

[Sun86] Sun Microsystems, Inc., "SunView Programmer's Guide," Part No. 800-1345-10, Sun Microsystems (September 1986).

[Suth63] Sutherland, I. B., "Sketchpad, a Man-Machine Graphical Interface," *Proceedings of the AFIPS Conference, SJCC*, 23, Reston, Virginia, pp. 329-346 AFIPS Press, (1963).

[Tane81] Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ (1981).

[Tani82] Tanimoto, S. L. and E. P. Glinert, "Programs Made of Pictures: Interactive Graphics Makes Programming Easy," Technical Report 82-03-03, Department of Computer Science, FR-35, University of Washington, Seattle, Washington (March, 1982).

[Tani86] Tanimoto, Steven L. and Marcia S. Runyan, "PLAY: An Iconic Programming System for Children," pp. 191-205 in *Visual Languages*, ed. Shi-Kuo Chang, Tadao Ichikawa, Panos A. Ligomenides, Plenum Press, New York (1986).

[Walk83] Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, Bretton Woods, NH, pp. 49-70 (October, 1983).

[Will88] Wills, Craig E., "Service Execution in a Distributed Environment," Ph.D. Dissertation, Department of Computer Sciences, Purdue University, West Lafayette, IN (1988).

[Wulf81] Wulf, William A., Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill (1981).

[Yao84] Yao, S. B., A. R. Hevner, Z. Shi, and D. Luo, "FORMMANAGER: An Office Forms Management System," *ACM Transactions on Office Automation Systems*, 2(3), pp. 235-262 (1984).

[Yosh86] Yoshimoto, I., N. Monden, M. Hirakawa, M. Tanaka, and T. Ichikawa, "Interactive Iconic Programming Facility in HI-VISUAL," *1986 Workshop on Visual Languages*, Dallas, Texas, pp. 34-41 (June, 1986).

[Your75] Yourdon, Edward, *Techniques of Program Structure and Design*, Prentice-Hall, Englewood Cliffs (1975).

[Zloo81] Zloof, M. M., "QBE/OBE: A Language for Office and Business Automation," *IEEE Computer*, 14(4), pp. 13-22 (May, 1981).

RIACS

Mail Stop 230-5
NASA Ames Research Center
Moffett Field, CA 94035
(415) 694-6363

The Research Institute for Advanced Computer Science
is operated by
Universities Space Research Association
The American City Building
Suite 311
Columbia, MD 21044
(301) 730-2656