TASKMASTER: AN INTERACTIVE, GRAPHICAL ENVIRONMENT FOR TASK SPECIFICATION, EXECUTION AND MONITORING

by

Kumbakonam S. Raghu

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

James D. Arthur, Chairman

Roger W. Ehrich

Deborah S. Hix

February 1988

Blacksburg, Virginia

.8

TASKMASTER: AN INTERACTIVE, GRAPHICAL ENVIRONMENT FOR TASK SPECIFICATION, EXECUTION AND MONITORING

by

Kumbakonam S. Raghu

James D. Arthur, Chairman

Computer Science and Applications

(ABSTRACT)

This thesis presents Taskmaster, an interactive, graphical environment for task specification, execution and monitoring. Taskmaster is an integrated user environment that employs a unique blend of the principles of Visual Programming, Tool Composition, Structured Analysis and Data Flow Computing to support user task specifications. Problem solving in the Taskmaster environment consists of decomposing the problem task into a partially ordered set of high-level subtasks. This decomposition is depicted graphically as a network in which the nodes correspond to the subtasks and the arcs represent the directed data paths between the nodes. The subtasks are successively decomposed into lower level subtasks until, at the lowest level, each network node is "bound" to a pre-existing tool from a tools database. Execution of the resulting network of software tools provides a task solution. Some of the novel features of the Taskmaster environment include 1) guidance to the user in the task specification process through menu-based interaction; 2) facility to interactively monitor the task network execution; 3) support for structured data flow among tools; and 4) enhanced support for reusability by providing operations to functionally abstract and reuse sub-networks of tools.

Key words and phrases: High-level Data Flow, Parallel Processing, Task Specification, Data Flow Graphs, Visual Programming, User Environment, Tool Composition, Tool Integration, Software Reusability, User Support Systems, Functional and Visual Abstraction, Structured Data Flow.

Acknowledgements

I would like to express my heartfelt gratitude to my advisor, Dr. James D. Arthur, for his inspirational guidance and enormous patience, and the generous help with his time, knowledge and experience. I would also like to thank Kelly C. Mulheren, currently at the Digital Equipment Corporation and formerly a graduate student at Virginia Tech, for the excellent design and the robust implementation of the GETS system. His help in familiarizing me with the GETS environment is also much appreciated. My thanks are also due to Dr. Roger Ehrich and Dr. Deborah Hix for their presence on my committee.

Partial funding for this research was provided by the Digital Equipment Corporation and the Virginia Center for Innovative Technology.

Table of Contents

List of Illustrations

Chapter 1

Introduction

1.1 New Challenges to Software Engineering

In the backdrop of incredible levels of sophistication achieved in both hardware and software over the past decade, a major challenge to Software Engineering has been to formulate techniques to keep the computer simple, intuitive and easy to use. This effort has not been easy because computers have come to permeate almost every walk of life and their audience spans the entire spectrum from novices to experts. Society's ever increasing dependence on computers and the continued dearth of qualified computer professionals has created an urgent need for methods and techniques to increase programmer productivity and decrease programming complexity. Programming methods need to be made more intuitive, closer to the mental process of problem solving, allowing the programmer to concentrate on the problem solution rather than on programming language syntax or semantics. Methods are needed to exploit the potential offered by hardware advancements, e.g. support for real-time interactive graphics, availability of inexpensive and abundant local computing

power and local area networks. These factors have contributed to a new emphasis on increasing the bandwidth of interaction between the computer and the user.

1.2 Evolutionary Trends in Software Engineering

In response to the aforementioned challenges to Software Engineering, several evolutionary trends have emerged. First and foremost is the evolution of the ***user environment***, a direct result of integrating independent software development tools into a single consistent interface. User environments are characterized by a user-friendly interface, cooperating internal components and a wide range of applicability [ARTH83]. A second important evolution is the discipline of ***visual programming*** which employs graphical representation of objects and allows for their "direct manipulation". The primary advantage of visual programming is the very high bandwidth of information transfer offered by techniques like iconic representation and use of logical windows. A third trend is the shift towards ***increased support and guidance for the user*** catalysed by the availability of abundant local computing power. This support not only includes the automation of routine aspects but also involves active support to the user in efficiently utilizing the system to perform the desired task. A fourth significant evolution is a programming methodology based on ***tool composition*** that advocates the development of large systems by the composition of pre-existing software modules. Tool composition facilitates *software reusability* which has been shown to be the most significant factor in improving software development productivity and quality [BOEH84]. All of these four trends are aimed primarily at increasing programmer productivity. A fifth trend which has evolved, aimed at better performance, is the concept of ***data flow*** computing as opposed to the conventional *control flow* computing. This thesis discusses a high-level task specification and execution environment, called ***Taskmaster***, which is a unique blend of all these five principles. It also addresses many of the important issues in the design, implementation and application of such an environment. The

rest of this chapter gives a brief overview of this environment and the motivations for this research followed by an outline of the later chapters.

1.3 Taskmaster Environment

Taskmaster is an interactive, graphical environment for task specification and execution. Task specification operations are supported through a collection of software tools present in a tools database. A *tool* as used here refers to a filter program (*a la* UNIX[1] *sort*) which performs a single high-level operation with minor variations. A more formal treatment of this topic can be found in Section 3.2.1 of this thesis and in other literature [ARTH83]. Each tool can have multiple input and output ports through which it communicates with other tools. In this environment, problem solving consists of decomposing the problem task into an ordered set of conceptually simple high-level subtasks. This decomposition is expressed graphically as a network in which the nodes correspond to the subtasks and the arcs represent the directed data paths between the nodes. The subtasks are successively decomposed to smaller subtasks until, at the lowest level, each network node is "bound" to one of the tools from a database containing all the available tools. Execution of the resulting network of software tools provides the problem solution.

The following example illustrates this programming paradigm. Suppose one desires to specify a task network to implement a matrix multiplication scheme with vector operations in order to exploit the parallelism offered by vectorization. Figure 1 illustrates the high-level task specification network for this scheme. The pre-multiplier matrix A is vectorized by row and the post-multiplier matrix B is vectorized by column. The product matrix elements are all individually computed in parallel by computing the scalar products of the corresponding row and column vectors. The

[1] UNIX is trademark of AT&T

product matrix elements are then composed to form the product matrix row vectors which are again composed to form the product matrix. Let the primitive tools vectorize_by_row, vectorize_by_column, combine_row_elements, combine_row_vectors and multiply_row_by_column be available in the tools database. Then for the case where both A and B are two-dimensional matrices, the task network can be refined to the topology shown in Figure 2. Though simplistic, this particular example does capture the flavor of a typical task specification in the Taskmaster environment.

The Taskmaster environment is novel because it

- provides a powerful set of operations to specify task networks either by stepwise refinement (top-down) or by successive abstraction from lower level functional units to higher level functional units (bottom-up) until the task network is fully specified,
- provides an interactive execution environment to monitor the task execution,
- supports structured data flow among tools, and
- allows inherent parallelism of algorithms to be conveniently expressed.

The Taskmaster environment is an integrated user environment composed of three cooperating components:

- the Network Editor,
- the Network Execution Monitor, and
- the Tools Database.

Physically, the environment is partitioned across two machines connected by a high speed communication link. The Network Editor resides on a VAXstation[2] I running MicroVMS 4.2 (local workstation). The Execution Monitor resides on a VAX 11/785 running Ultrix-32 (host computer). The Tools Database resides on the host machine but gets copied over to the local work-

[2] VAXstation, VAX, Ultrix and MicroVMS are all trademarks of the Digital Equipment Corporation.

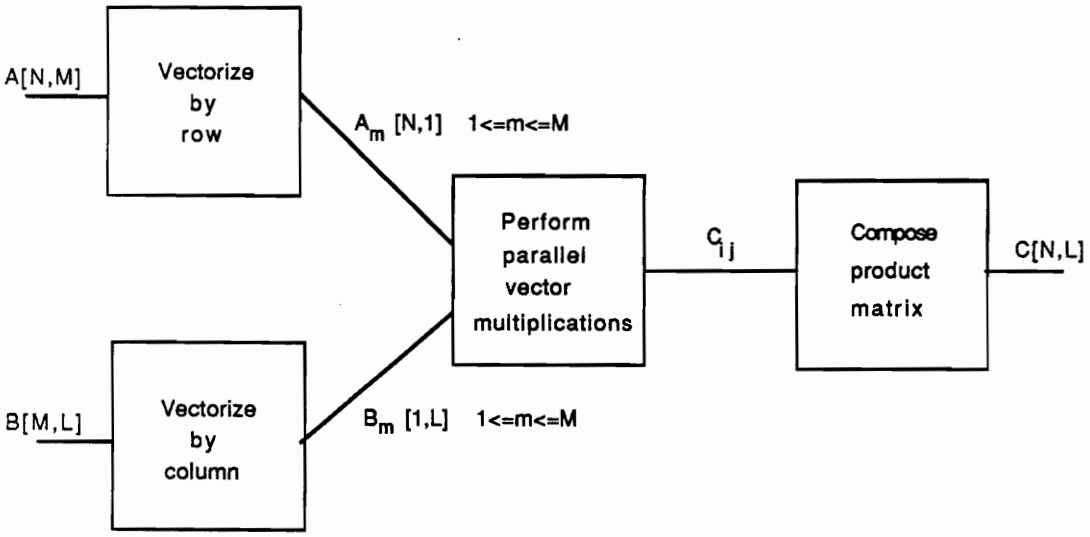


Figure 1. A high-level specification of vectorized matrix multiplication.

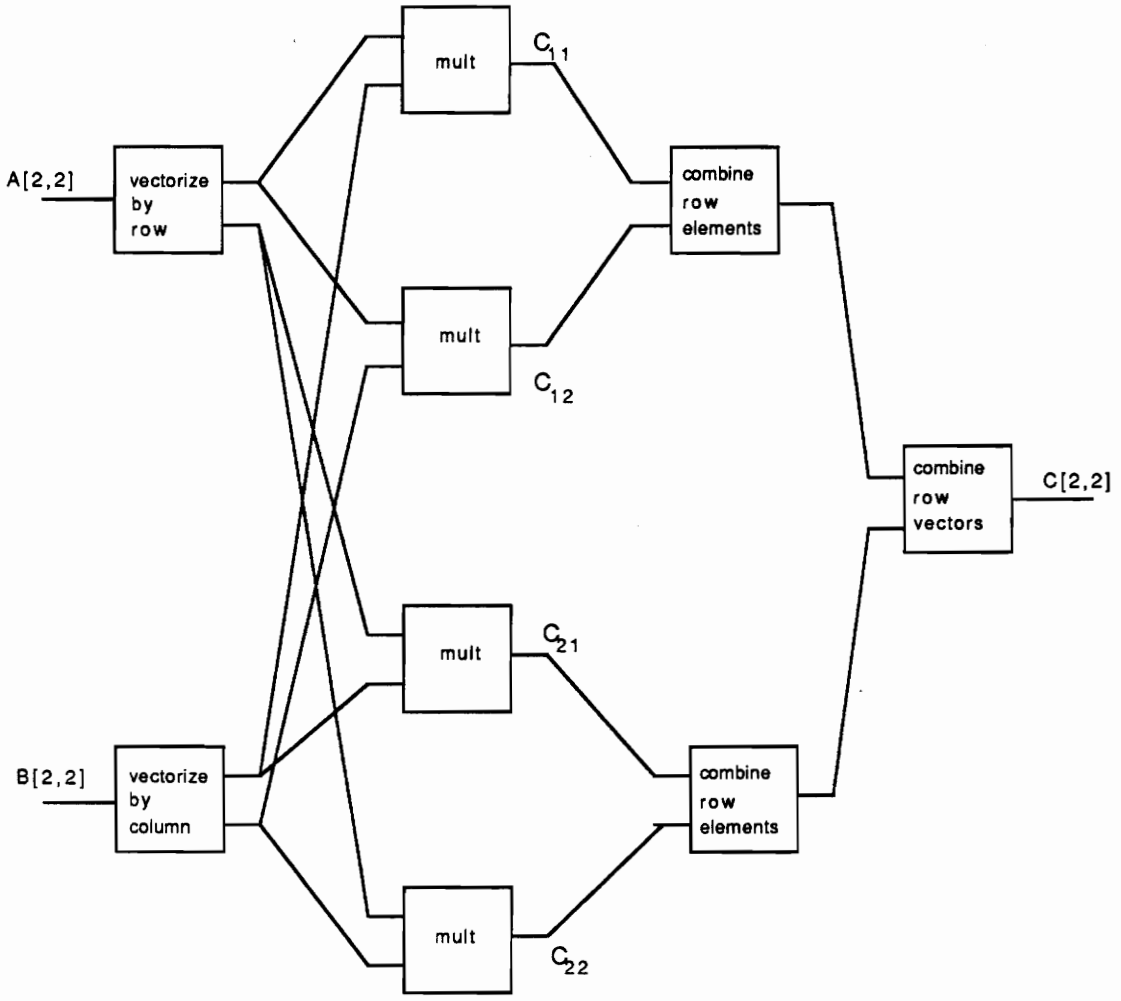


Figure 2. A fully refined task network for vectorized matrix multiplication.

station on every update. The current configuration has a single local workstation but we visualize a set of local workstations all connected to the host, in future. The overall system configuration is shown in Figure 3. The Network Editor provides a graphical interface for constructing task networks. It also guides the user through the task specification process by supporting stepwise task refinement through menu-based interaction. Once a task has been fully specified, the task network can then be executed, whereby its representation is forwarded to the Execution Monitor for network instantiation and monitoring. The Tools Database plays a supportive role in that it provides access to all the information pertaining to the basic tool set. This information includes a detailed description of each tool present in the database, its interface structure and the dialogue for refining its function.

1.4 Motivations for This Research

The Taskmaster environment is an enhanced version of GETS (Graphical Environment for Task Specification) [MULH86]. The motivations for this research are twofold. The first is the need for a high-level task specification environment which embraces the new trends discussed earlier in this chapter. The development of GETS and subsequently Taskmaster are aimed primarily at satisfying this need. The second motivation is to enhance the functionality of GETS to provide support for the following features:

- true top-down as well as bottom-up task specification,
- abstraction of a collection of tools into a composite tool,
- facility for structured data flow among the tools, and
- facility to interactively monitor the task execution.

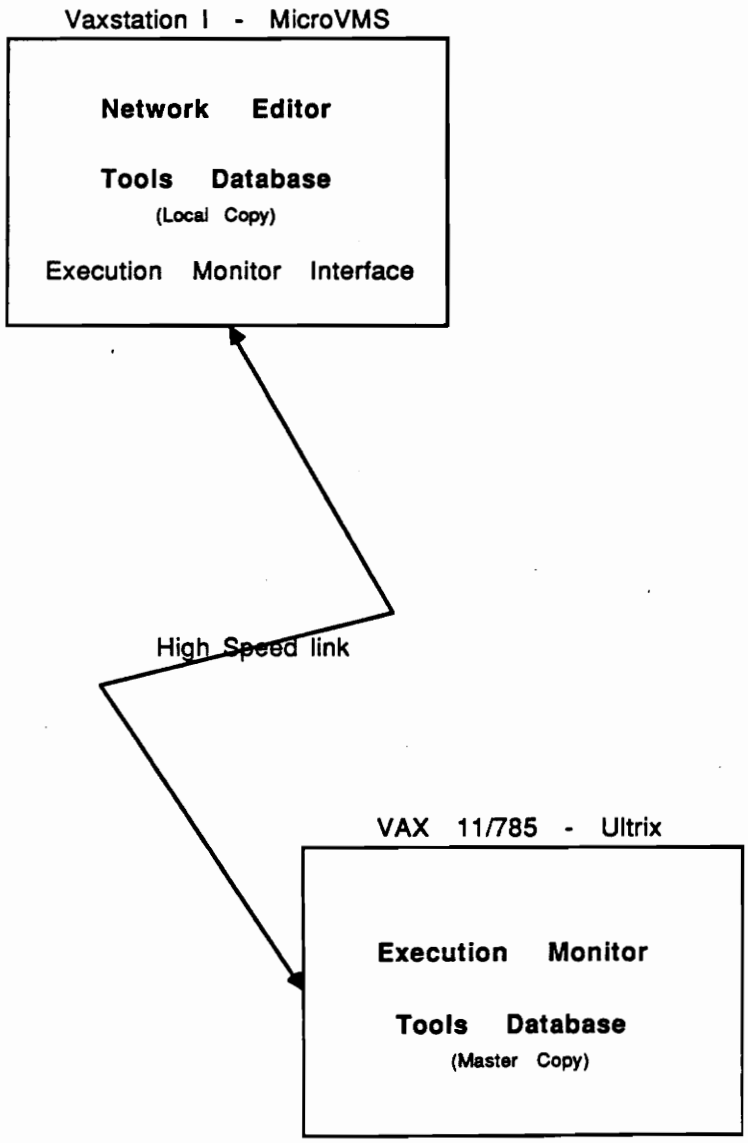


Figure 3. Overall system configuration

These additional features, as well as a more expansive application domain, distinguish the Taskmaster environment from the GETS environment and are the major contributions of the author. Decision paths, design rationale and implementation details follow in later chapters.

1.5 Plan of the Thesis

Chapter two presents a overview of user environments exemplifying the different concepts discussed in the beginning of this chapter. It also provides a more detailed description of GETS and the need for the enhancements made in the Taskmaster system. This chapter is intended to provide the background and motivation for the current research. Chapter three discusses the salient features of the Taskmaster user interface. The major topics of discussion are the support for abstraction and top-down specification in the Network Editor, interactive execution monitoring, execution time operations and usage statistics collection. Chapter four discusses the implementational and developmental aspects of the Taskmaster system and analyzes it from a system perspective. The important issue under consideration is the support for structured data flow among the tools. Also discussed are the execution status feedback mechanism, the internal representation of functionally abstracted sub-networks, and error handling and recovery. Chapter five profiles Taskmaster from an application perspective. It discusses potential application domains and their suitability to Taskmaster's programming philosophy. The later part of this chapter describes the two particular applications currently implemented and the lessons learned from this. Finally, Chapter six summarizes the contributions of this thesis, addresses the limitations of the current implementation and explores future research possibilities.

Chapter 2

Background

2.1 *Overview of User Environments*

A ***User Environment*** consists of a collection of software tools and an intelligent user-friendly interface which assists the user in utilizing the tools to accomplish a given task. User environments can be classified by several criteria such as degree of integration, domain of application, programming methodology and system architecture. These classification criteria are by no means mutually exclusive and there is considerable degree of overlap among the classes. This classification, however, provides a logical basis for comparison of the various environments. The importance of each of these criteria in relation to user environments is discussed below.

2.1.1 Degree of Integration

The term integration refers to the level of coupling and cooperation between the component elements of the environment. Based on this criterion, environments can be classified as tightly-coupled or loosely-coupled [HABE79]. Tightly coupled systems use the ***monolithic*** [SNOD86] approach to tool integration. The monolithic approach uses a centralized database which maintains descriptions of the current state of the system. Information sharing among tools is accomplished through this ***internal*** [ARTH83] channel which is the centralized database. The advantages of this approach are the ease of communication between the tools and the uniformity of the interface. In contrast to this, loosely-coupled systems consist of of a large number of small self-contained, separately-developed tools which communicate by means of channels ***external*** [ARTH83] to the system. The advantages of this so-called ***tool-kit*** [SNOD86] approach are the flexibility of combining forms and the ease of integration of new tools. UNIX is one example of a loosely-coupled system while Gandalf [HABE82,NOTK85], Smalltalk [TESL81], the Cornell Program Synthesizer [TEIT81], PECAN [REIS84] and Interlisp [TEIW81] are some of the well-known tightly-coupled environments. Lamb has suggested a model of tool integration intermediate to the above-mentioned approaches [LAMB83]. In this approach tools are developed independently as in the toolkit approach but are later combined in an organized manner based on a precise method of describing the tool interfaces like the Interface Description Language (IDL) [NEST82]. Taskmaster embraces this intermediate approach which is discussed in greater detail in the fourth chapter.

2.1.2 Domain of Application

Environments have been developed to address specific application domains like Program Development, Computer Simulation, Project Management, Artificial Intelligence, Telecommunications, Computer Networking and so on. Environments that specifically aid the user in all phases of pro-

gram specification and development are known as ***programming environments***. UNIX is one example of a general purpose programming environment that supports several languages. PECAN [REIS84] and Gandalf [NOTK85] also support several languages but the former is aimed at programming-in-the-small while the latter is aimed at programming-in-the-large. On the other hand, GENESIS [RAMA85], Pictures [WASS87], Interlisp [TEIW81] and Smalltalk [TESL81] support a single target language. GENESIS and Pictures, for example, are mainly oriented towards the C programming language, Interlisp is primarily Lisp-oriented and the Smalltalk environment supports object-oriented programming in the Smalltalk language. Software development environments usually have a rich variety of software development tools for debugging, text editing, documentation, language translation and other program development phases. Many of these environments provide structured and intelligent editors, multiple views of the program and incremental compilation. Computer Simulation is another important domain of application of user environments. StarLite [COOK87] and AT & T's Performance Analysis Workstation [MELA85] are examples of simulation environments. StarLite is a visual simulation package for software prototyping. The Performance Analysis Workstation is also a visual-based environment for the simulation and modeling of complex systems. Toolpack [OSTE82] supports the development of mathematical software which constitutes one of the richest and most frequently used application domains. Taskmaster and GETS [MULH86] are designed to be generic high-level task specification systems suitable for orientation to a wide variety of application domains.

2.1.3 Programming Methodology

Environments have been developed to support specific programming methodologies like visual programming, functional programming, parallel programming, object-oriented programming and so forth. Visual programming environments are discussed in detail in Section 2.3 on Interactive, Graphical Programming Environments [IGPE]. VIM [DENN81] is an experimental multi-user computing environment based on the principles of functional programming and data flow computer

architecture. F-shell [SCHU83] for UNIX includes all the combining forms in Backus' FP language [BACK78]. GENESIS [RAMA86] provides support for some functional programming constructs. PISCES [PRAT85], Poker [SNYD84] and Prometheus [ARTH84b] are some environments for parallel programming. PISCES is an environment for parallel scientific computation which still retains sequential programming characteristics. Poker provides a program constructor for parallel programming with a graphical interface and uses a network-driven methodology very similar to Taskmaster. Programming by tool composition is becoming increasingly popular ever since the advent of UNIX which originally advocated this programming philosophy. Environments based on tool composition are addressed in a separate section (2.4) later in this chapter. Object-oriented programming is an evolutionary reusability-driven approach to programming that has emerged in recent years. Smalltalk is a well-known example of an object-oriented programming environment. While Taskmaster can be broadly classified as a visual programming environment based on tool composition, it is strongly influenced by the concepts of functional, parallel and object-oriented programming.

2.1.4 System Architecture

System architecture is another criterion for characterizing user environments. Open Systems Architecture (OSA) facilitating easy integration of tools is used in the Illinois Software Engineering Program (ISEP) [KAPL86]. OSA achieves easy integrability of tools into the environment by the use of independent objects (called *sorts*) to handle the inter-tool communication. Taskmaster and GETS use a similar approach where the tool integration is made easy by using a database of tools which encapsulates all the tool-relevant information. Environments based on data flow architecture, like Taskmaster and GETS, are becoming increasingly popular and are treated in more detail in the next section. To further elucidate the context for the current research, the next three sections discuss in detail environments based on data flow architecture, interactive graphical programming and tool composition respectively.

2.2 *Environments Based on Data Flow Architecture*

As mentioned in Chapter 1, the computing organization used by Taskmaster is based on the concept of high-level ***data flow***. The *data flow* model of computation is based on data-driven organization as opposed to the conventional control-driven organization. In the data flow model, modules communicate only by data flow and computations are "fired" only by the availability/arrival of data. There is no concept of the traditional shared data storage as intermediate or final results are passed as data tokens between consecutive stages of computation. The execution sequencing is constrained only by data dependencies among different computation stages. This concept of asynchrony is what gives the data flow architecture its high degree of concurrency [HWAN84]. The data flow organization has several attractive properties:

- program modularity,
- high degree of concurrency and implicit parallelism,
- locality and freedom from side-effects,
- tight control over program determinism [BART85], and
- increased programming productivity due to its inherent support for reusability and vectorization.

In the words of Wayne Stevens [STEV82,DEMA84]:

> The ability to reuse functions is critical to dramatic improvements in programming productivity... The ability to reuse functions is enhanced by making them more independent of one another. The data flow technique provides a higher level of independence between program modules than ways which pass data and control, such as the CALL statement.

This conclusion is supported by the high reusability of tools in UNIX which uses a data flow organization restricted to linear sequencing. Like UNIX, Taskmaster and GETS exploit the advantages offered by the data flow architecture but remove the restriction of linear sequencing. The remainder of this section will address some well-known environments based on data flow and how they compare with Taskmaster.

The Stream Machine [BART85] is a software architecture, based on data flow, to support the design, development and execution of real-time data acquisition and process control applications. A program for the Stream Machine consists of a set of concurrently executing modules communicating through streams of data. The basic data flow constructs have been augmented with some time-based operations for supporting process control software. The Stream Machine modules are coarse grained in that they are actually complete sequential programs. This organization is similar to FLUIDE [SKUB86] which uses multiple data and control flow. The FLUIDE organization differs from the conventional data flow model in that the modules are not defined in terms of the lowest level primitives. The modules in FLUIDE, represented as data flow graphs, may be refined into lower level graphs till sequential machine code is reached. This is very similar to the approach used by Taskmaster and GETS. Unlike Taskmaster, FLUIDE also provides for looping constructs. SYNCRO [DEMA84], another environment based on data flow architecture, implements a data flow command shell for the Lilith/Modula computer. SYNCRO is a two-dimensional command interpreter with a graphic command language and allows direct implementation of non-linear leveled data flow networks. It also uses the "named pipe" for inter-module communications. The data flow command shell of SYNCRO is very similar to the GETS application domain which implements a similar shell for UNIX. NIL [STRO86] is another environment based on data flow. NIL is a very high level programming language and environment, based on a network of processes, similar to Taskmaster. NIL, however, uses a completely object-oriented approach with all communication implemented via message passing. Another novel feature of NIL is that it treats both programs and data as objects and allows the communication of process objects among its modules (which are themselves processes).

2.3 *Interactive, Graphical Programming Environments*

It is widely accepted that the human mind is visually oriented and that people grasp information at faster rates when it is presented visually as opposed to being presented in a textual format [RAED85]. Visual programming capitalizes on this by using graphic representations of objects (icons) and allows for their "direct manipulation". IGPEs use pictures to depict various aspects of a program like control flow, data flow, data structure and topology [RAED85]. The flow chart is probably the best known example of graphical programming aids. Another well-known example of iconic representation is the use of "pronged" ends to distinguish different types of connectors. The user can directly perceive that a three-pronged plug will not fit a two-pronged connector [BROW86]. The BLOX methodology [GLIN86] is a novel mixed textual-graphical programming methodology which uses this concept. Visual programming also has the potential to enhance productivity by allowing the programmer to directly work with his conceptualizations in what is called ***conceptual programming*** [REIS86].

Taskmaster attempts to exploit the generic strength of visual programming, i.e. the high bandwidth of human-computer interaction, while using textual descriptions in a supplementary role. Environments like PECAN [REIS84], CEDAR [TEIW84] and Smalltalk [TESL81], though based on textual languages, still provide graphical capabilities in their user interfaces which make extensive use of logical windows. PECAN and GARDEN [REIS85] environments developed at Brown University provide multiple views of a program simultaneously and allow the user to choose among this plurality of representations. Similar to PECAN and GARDEN, the Omega [POWE83] system uses visual abstractions of textual based code and data but allows for several ways of displaying the objects. A similar approach of providing multiple levels of visual abstraction is employed by Taskmaster. In contrast to these mixed visual-textual environments, few purely visual-based environments exist. PICT [GLIN84] is a true visual programming environment which uses icons to represent everything and allows direct manipulation of these icons using a mouse or a joystick. The

user is not required to touch the keyboard at all once the system has been invoked. Show and Tell Language (STL) is another example of a completely icon-driven visual programming language aimed at "keyboardless" programming [GILL86]. The semantic model of STL is based on the concepts of data flow and completion. Programming in Pictures (Pip) [RAED84] is based on functional programming and permits great freedom to the user in designing the icons. PegaSys [MORI85] combines graphics with formal logic to provide support for program design and documentation. The Pegasys approach is unique in that pictures are mapped to form calculus formulae and the system provides automatic verification of the composition of the picture components for consistency. Tinkertoy [EDEL86] is a graphical programming environment oriented to Lisp and allows direct manipulation of icons representing Lisp expressions. The Dialog Management System (DMS) [EHRI86] and the state transition diagram systems of Jacob [JACO85] and Wasserman [WASS85] employ graphical techniques for building user interfaces. Visual programming is still very much in its infancy and much work needs to be done before its full potential is exploited.

2.4 *Environments Based on Tool Composition*

Tool composition is a programming paradigm where the specification of a problem solution is achieved by collecting and interfacing already existing software modules [ARTH84a]. This approach to software development using "software IC's" [COXB86], provides a direct parallel to the popular hardware design approach using integrated circuit (IC) chips. The advantages of this approach are the reduction of coding errors, reduction of implementation time and the inherent support for reusability. UNIX pipeline feature allows for linear composition of software tools. Even in this environment restricted to the composition of single-input, single-output filter tools, the tool composition paradigm has been found to be highly popular and useful. Tool composition goes extremely well with the UNIX philosophy [MCIL78] of

> Make each program do one thing well. To do a new job, build afresh rather than complicating old programs by adding new options. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter the output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

These simple tools are reusable components and their increasing popularity and use have been shown to have caused the largest long-term gains in software development productivity [BOEH84,MUNS85].

The OMNI environment [ARTH87] embraces the tool composition approach to problem solving. It provides an interactive environment for tool selection, specification and composition. To solve a given task, the user decomposes it into a sequence of simple, high-level operations which is converted by OMNI into a corresponding sequence of composed tools which implements the given task. OMNI also introduced the notion of using a tools database to guide the user in the task specification process. Whereas the OMNI environment is a textually oriented task specification system, GETS and consequently Taskmaster are visually-oriented systems. Section 2.5 discusses the GETS environment in more detail. SYNCRO environment mentioned earlier in this chapter allows tool composition but does not provide assistance to the user in the tool composition process. GENESIS supports functional composition of tools [RAMA86] which is more powerful than the linear composition offered by UNIX but is still limited by the single-output constraint on the tools. Unlike UNIX and GENESIS, Taskmaster and GETS allow multi-input, multi-output modules that can be used to model more sophisticated tools.

2.5 Graphical Environment for Task Specification

This section describes the Graphical Environment for Task Specification (GETS) [MULH86], the parent system from which the Taskmaster system has evolved. The GETS Environment is composed of three co-operating internal components: the Network Editor, the Network Execution Monitor and the Tools Database. The GETS system configuration is essentially the same as the

Taskmaster system configuration shown in Figure 3 except for the fact that the Tools Database resides on the local workstation rather than the remote host.

2.5.1 GETS User Interface - the Network Editor

The Network Editor provides an interactive, graphical interface for constructing and executing task specifications [MULH86]. Programming in the GETS environment, as in the Taskmaster environment, consists of transforming a conceptual task into a network whose nodes represent high-level operations (tools) and arcs represent the communication path between the nodes. The Network Editor supports this specification process by providing primitives to build generic networks and to specify the nodes and the arcs using a menu-driven interaction process defined within the Tools Database. The Network Editor also permits a user to send a fully specified task network to the Execution Monitor for instantiation.

The Network Editor is completety menu-driven and makes extensive use of logical windowing and mouse input. It also incorporates many of the human engineering principles related to graphical user interface design like direct manipulation, visual feedback, user error recovery, choice confirmation, default selection, operational and representational consistency, pop-up menus and so on. The Editor display consists of a large window displaying the topology of the network being edited and auxiliary pop-up windows for displaying

- menus,
- multiple views of nodes and arcs,
- user instructions and help messages,
- error messages,
- user confirmation requests, and
- textual information pertinent to tool and communication path specification.

Graphics support for windowing, mouse input and menu interaction are provided by the Graphical Kernel System (GKS 0/b) [DEC84]. The Network Editor supports the following set of editing primitives within the designated categories:

- category 1 - network topology construction and viewing

  create node, delete node, move node,

  create arc, delete arc,

  collapse, explode

  pan, zoom,
- category 2 - network specification and inquiry

  specify node, view node,

  specify communication path, view communication path,
- category 3 - network storage and retrieval

  save network, restore network,
- category 4 - user error recovery

  undo, and
- category 5 - network instantiation

  execute network.

The next paragraph discusses these editing primitives in more detail.

The network topology construction operations are used to operate on generic node and arc icons that serve as visual place-holders for the tools and their interface connections. Thus, an "unspecified" node created by the ***create node*** operation is just a place-holder and as yet has no semantic meaning with respect to the task being solved. The ***pan*** and ***zoom*** operations provide ***selective viewing*** in order to manage relative complexity of very large networks. The ***collapse*** operation allows one to abstract a sub-network performing some high-level operation into a single ***super-node***. The ***explode*** operation reverses the effect of the corresponding collapse operation. Note that the effect of the collapse and explode operations is only visual and no functional abstraction of the underlying

sub-network is attempted. That is, the collapsed "super-node" abstraction of GETS does not preserve any information about the collapsed sub-network in relation to the rest of the network.

The network specification operations are used to ***specify*** the unspecified node and arc icons. Node icons are specified by attaching to them a fully specified tool from the Tools Database. Arc icons are specified by making all the appropriate connections between the tools associated with the nodes connected by the arcs. Thus, an arc can be specified only after the nodes at both its ends are fully specified so that the interface is clearly defined. The network inquiry operation, ***view node***, provides a detailed view of the tool attached to a specified node. The ***view communication path*** operation provides a detailed view of the interface between the tools connected by an arc. GETS also provides for a textual view of each specified node containing the description of the associated tool, its attributes and its input and output ports. At anytime during a editing session, the current state of the network can be saved to disk or a previously saved network can be restored from disk using the ***save network*** and the ***restore network*** operations respectively. The ***undo*** operation supports error recovery by undoing the effect of the most recent topology modifying operation.

After a network is created and completely specified, the ***execute network*** operation can be used to send it to the Execution Monitor for instantiation. The Network Editor then sends an internal representation of the network to the Monitor after performing the following functions:

- makes automatic port connections for unspecified, yet unambiguous, arcs,
- attaches special invisible nodes to handle multi-input, multi-output "piping" operations that require merging of multiple input streams and duplication of multiple output streams, and
- performs consistency checking on the network flagging all unconnected ports in the network.

The Editor flags all the unconnected ports giving the user an opportunity to correct any oversights. Finally, the internal representation of the network is transferred over the high-speed link to the remote host where the Execution Monitor resides.

2.5.2 GETS Executive - the Execution Monitor

Problem solving in the GETS environment consists of (1) specifying the problem and (2) executing the solution [ARTH86]. The Network Execution Monitor supports the second stage of this process by performing the following functions:

1. reading the task network representation forwarded by the Network Editor,
2. validating the network,
3. spawning the network, and
4. monitoring the network execution.

The Execution Monitor runs each node as a separate process and the interprocess communication is achieved through the UNIX "pipe" facility. The Monitor performs a traversal of the network using the breadth-first search (BFS) algorithm, checking for network connectivity and data path consistency. After confirming network consistency, the Monitor instantiates the network by spawning a process for each node in the network, allocating a UNIX pipe for each arc and connecting pipes to the appropriate nodes [MULH86]. The node instantiation is also performed in the BFS traversal order in an attempt to satisfy certain interprocess communication constraints in the operating system. The network execution, however, is independent of the instantiation order because it is based on ***data flow***. The Execution Monitor also provides for simple execution monitoring using status messages to indicate the instantiation and the termination of each node-associated process.

2.5.3 GETS Knowledge Base - the Tools Database

The third component of the GETS environment is the Tools Database. This subsection discusses the significant role played by the Tools Database in the overall functionality of the GETS envi-

ronment. An essential characteristic of a ***user environment*** is its wide range of applicability [ARTH83]. To exhibit this characteristic, an environment needs to be as application-independent as possible. In the GETS environment, the Tools Database fulfills this need by encapsulating all application-specific information within itself and presenting it in a generic form to the other two components of the environment. Thus, to support a different application domain, only the Tools Database needs to be developed for the new application and can be integrated into the GETS environment transparent to the rest of the system.

The Tools Database contains information about all the tools available in the environment. This information includes tool communication requirements, tool arguments and complete textual descriptions of each tool and its input and output ports. The Tools Database also contains all the information supporting the multi-level, menu-based dialogue process for node specification. The Network Editor directly uses this information to drive the node specification process [ARTH85]. This notion of using a knowledge base to guide the user in the selection process is a novel feature first used in the OMNI environment mentioned earlier [ARTH83]. Intuitively, the node specification process can be viewed as a finite state machine driven by the Tools Database menu dialogue "table" [RABI59]. In the current implementation, the Tools Database resides on the local workstation and supports the file transformation environment. This application domain can be viewed as a ***data flow shell*** for UNIX and is similar to the SYNCRO environment which implements a similar shell for the Lilith/Modula [WIRT81] computer.

2.5.4 Limitations of GETS

The previous three subsections of this chapter provide a detailed description of the GETS environment. This subsection will address the limitations of the GETS environment mentioned in the first chapter:

To specify a task network in the GETS environment, the user needs to visualize a network of high-level operations which can, in turn, be directly transformed into executable form by the system. Within GETS, true top-down visualization of the problem solution is not supported. GETS provides limited support for top-down as well as bottom-up task specifications. The top-down specification is supported only in the node specification process - that too is limited by the depth of the menu tree for the node specification process. The bottom-up specification is limited by the lack of support for functional abstraction of a sub-network. Functional abstraction capability allows for an order of magnitude increase in productivity because one can define and reuse composite tools. Customization of the system also becomes possible. Taskmaster provides support for true top-down and bottom-up specifications by providing new primitives to operate on functionally abstracted sub-networks or ***pseudotools***. This facility for abstraction implies that the task networks of Taskmaster are multi-level as compared to the single-level networks of GETS.

Another important limitation of the GETS environment is that the data flow among tools is limited to a stream of characters. This limitation restricts the GETS application domain to text processing and related fields. To overcome this limitation, Taskmaster provides structured data flow among tools with provision for automatic type checking for structure compatibility of interfacing components.

A third significant limitation is that the GETS execution monitoring environment is not interactive and very simplistic. Taskmaster overcomes this limitation by providing an interactive execution monitoring environment and also supports several run-time operations.

A fourth limitation of GETS is the lack of support for looping constructs. This limitation still exists in the current implementation of Taskmaster but theoretical research addressing this aspect is near completion. To summarize, GETS supports the construction of task networks that are non-linear, acyclic, unidirectional and single-level.

Finally, GETS does not allow for much experimentation with application domains due to the constraint of having to use the text stream paradigm of UNIX. Taskmaster, by providing for inter-tool communication using structured streams, has opened up a whole new domain of potential applications.

Chapter 3

Salient Features of the Taskmaster User Interface

3.1 Introduction

The purpose of this chapter is to analyze the Taskmaster user interface. The main focus of the discussion will be on the features of the user interface that are novel to the Taskmaster environment and those features of the GETS environment which have been significantly enhanced or modified. The Taskmaster user interface incorporates all the generic network editing primitives available in the GETS environment discussed in Section 2.5.1. In addition to these, the Taskmaster interface has powerful primitives to support true top-down and bottom-up task specifications. Top-down task specification involves the successive decomposition of a task into lower level subtasks until the lowest level subtasks are directly identifiable with available tools in the Tools Database. Bottom-up task specification involves successive abstractions of fully specified lower level subtasks into higher level subtasks till the task is fully specified. Figure 4 shows the starting configuration of the Taskmaster user interface with a menu showing all the available editing primitives.

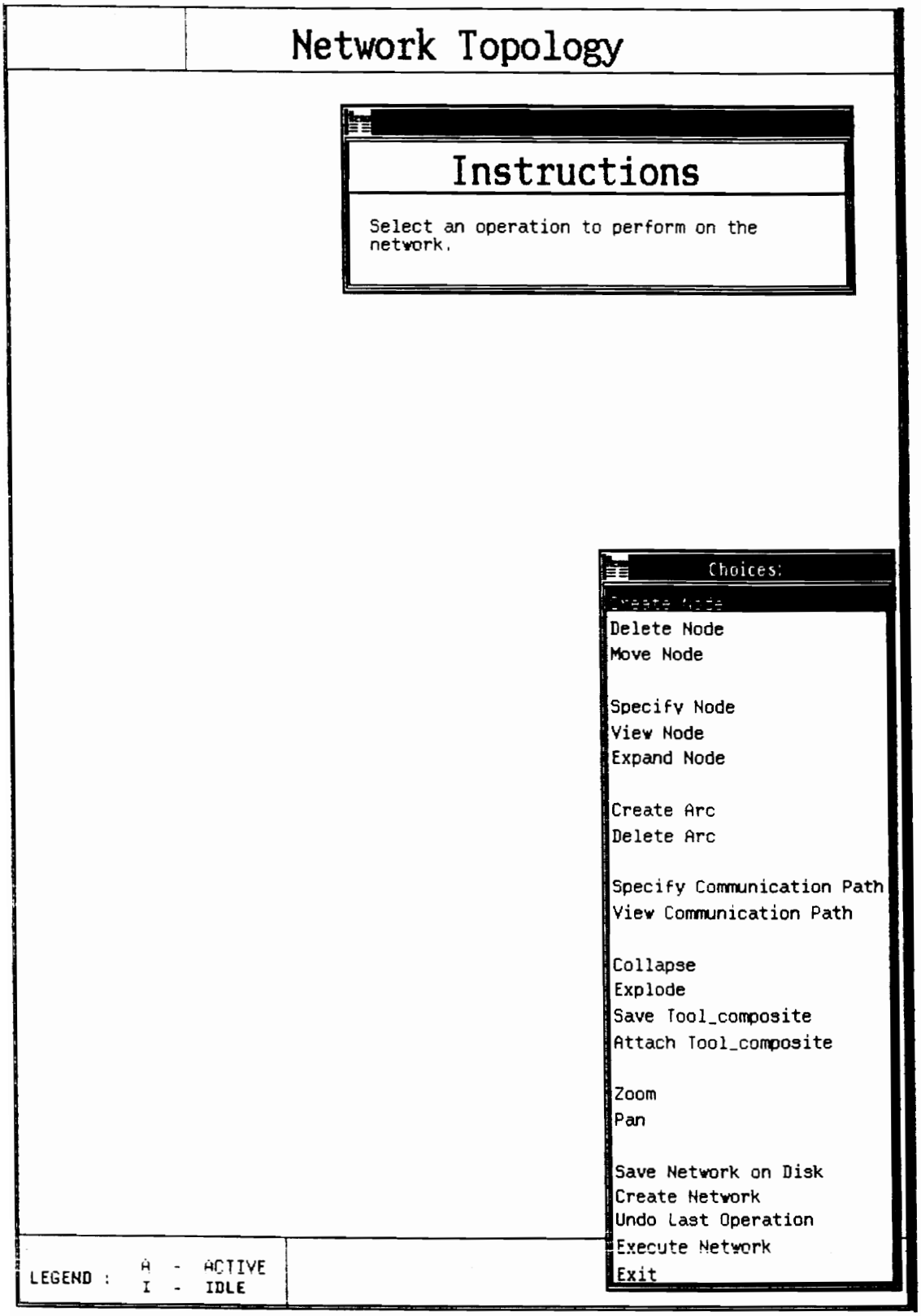


Figure 4. Starting configuration of the Taskmaster user interface.

In support of levels of abstraction, the Network Editor provides three new primitives: ***save tool-composite***, ***attach tool-composite*** and ***expand node***. The ***save tool_composite*** operation allows the user to delineate a portion of the network that performs some high-level operation and store it for later reuse. Using the ***attach tool-composite*** operation the user can reuse previously defined composite tools and integrate them into the task network currently being edited. These primitives are discussed in detail in Section 3.2 which addresses the support for abstraction in the Network Editor. The ***expand node*** operation supports top-down specification by allowing the user to edit a subtask network separately and integrate it into the task network. This operation is discussed in detail in Section 3.3 which is on the support for top-down specification. Another salient feature of the Taskmaster environment is the facility to visually monitor the task network execution. This interactive execution monitoring facility provides the user (located at the local workstation) with real-time, visual feedback of the execution status of the task network (being executed on the remote host). This facility is discussed in detail in Section 3.4. In addition to this, Taskmaster provides the user with *execution-time* operations which allow the user to change the current view of the task network while it is being executed. Section 3.5 discusses the execution-time operations which form a subset of the normal editing operations. Finally, Section 3.6 discusses the usage statistics collection facility provided by the Network Editor.

3.2 Support for Abstraction in the Network Editor

This section explains the functionality and the underlying concepts of the powerful abstraction capabilities provided by the Network Editor. The first subsection defines the terminology and the concepts needed to provide a framework for discussing the abstraction mechanisms. The next subsection presents the different models used for abstraction and the last subsection describes the editing primitives supporting abstraction from a user's perspective.

3.2.1 Terminology and Concepts

Abstraction is itself an abstract term which has manifold meanings. Abstraction as used here means the hiding of unnecessary detail or equivalently showing only the aspects essential to solving a given problem. It is important to note that the criteria used in abstraction are dependent on the projected use of the abstracted object or the target environment. Abstraction is the best way to deal with complexity since it reduces the apparent complexity by the elimination of irrelevant detail. Of particular interest here is the abstraction of a collection of tools forming a sub-network into a composite tool. The ***tool*** is the basic entity in the tool composition programming paradigm and performs a simple high-level operation. Each tool has one or more ***ports*** with which it communicates with other tools via ***links***. A composite tool or a ***tool-composite*** is a collection of tools grouped together forming a new tool. With particular reference to the Taskmaster environment, a tool has a direct representation in the Tools Database. A tool-composite, on the other hand, has no direct representation in the Tools Database and exists only by virtue of its creation by the user. Hence a tool-composite is in effect a ***pseudotool***; the two terms will be used synonymously hereafter. A ***cutset*** of tools is a sub-network of tools delineated from the whole by a closed polygon. An ***internal link*** of a cutset is a connection with both its ends within the cutset. An ***intersecting link*** of a cutset is a connection with only one end inside of the cutset. An ***internal port*** of a cutset is a port within the cutset which has only internal links. An ***external port*** of a cutset is a port within the cutset with at least one non-internal link. Figure 5 shows a network where a closed polygon forms a cutset comprising the tools labelled C, D, E and F and links labelled e, f, g, h and i. Links e, f, g, h and i are internal links while the rest are intersecting links. Ports labelled 4, 5, 6, 13 and 14 are external ports and ports labelled 7, 8, 9, 10, 11 and 12 are internal ports.

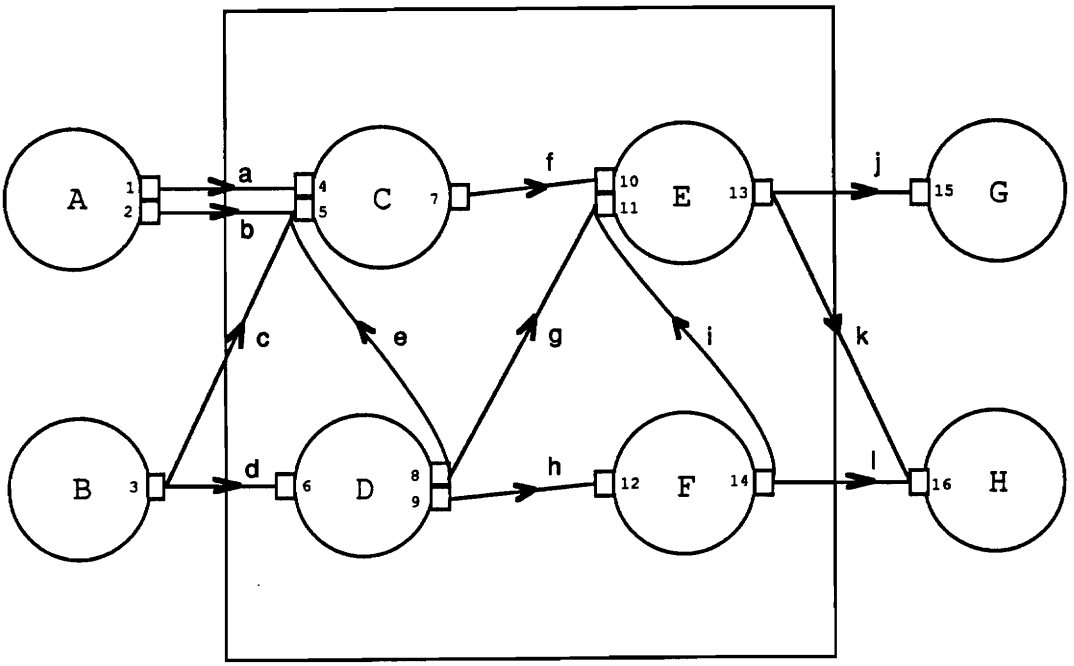


Figure 5. Example network to illustrate cutset and other definitions.

3.2.2 Models for the Abstraction of a Cutset

Any cutset can be considered to have two distinct contexts. The first is its internal context which includes only the internal links and all the tools comprising the cutset. The second is the external context of a cutset which is information about the cutset vis-a-vis the rest of the network. The very term abstraction of a cutset automatically implies the preservation of its internal context. In terms of the external context one can identify three different abstraction models for a cutset:

- no external context preservation model (M1),
- external functional context preservation model (M2), and
- external relational context preservation model (M3).

Referring to Figure 6, the M1 abstraction can be constructed by removing all intersecting links from the cutset and then using all the unconnected ports to form the interface. The M2 abstraction can be constructed by using all unique external ports of the cutset to form the interface. The M3 abstraction can be constructed by using a port for every intersecting link of the cutset. Figure 6 shows the three abstraction models for the cutset of Figure 5. As can be seen from the figure, the M1 abstraction has 2 input ports and 1 output port, the M2 abstraction has 3 input ports and 2 output ports while the M3 abstraction has 4 input ports and 3 output ports.

Recall from Section 3.2.1 that the abstraction model is dependent on the target environment. Model M1, which preserves no external context information, can be used in an environment where grouping is the only essential information that needs to be preserved. A good example of such an abstraction is provided by the "cut" and "paste" operations of MacDraw[3]. The collapse operation in GETS also uses the M1 model of abstraction. Model M2 preserves some external context information *viz.* the functional context of the cutset. An attractive feature of M2 abstraction is that it supports reusability at the functional level. Model M3 preserves all the external context infor-

[3] MacDraw is a trademark of Apple Computers, Inc.

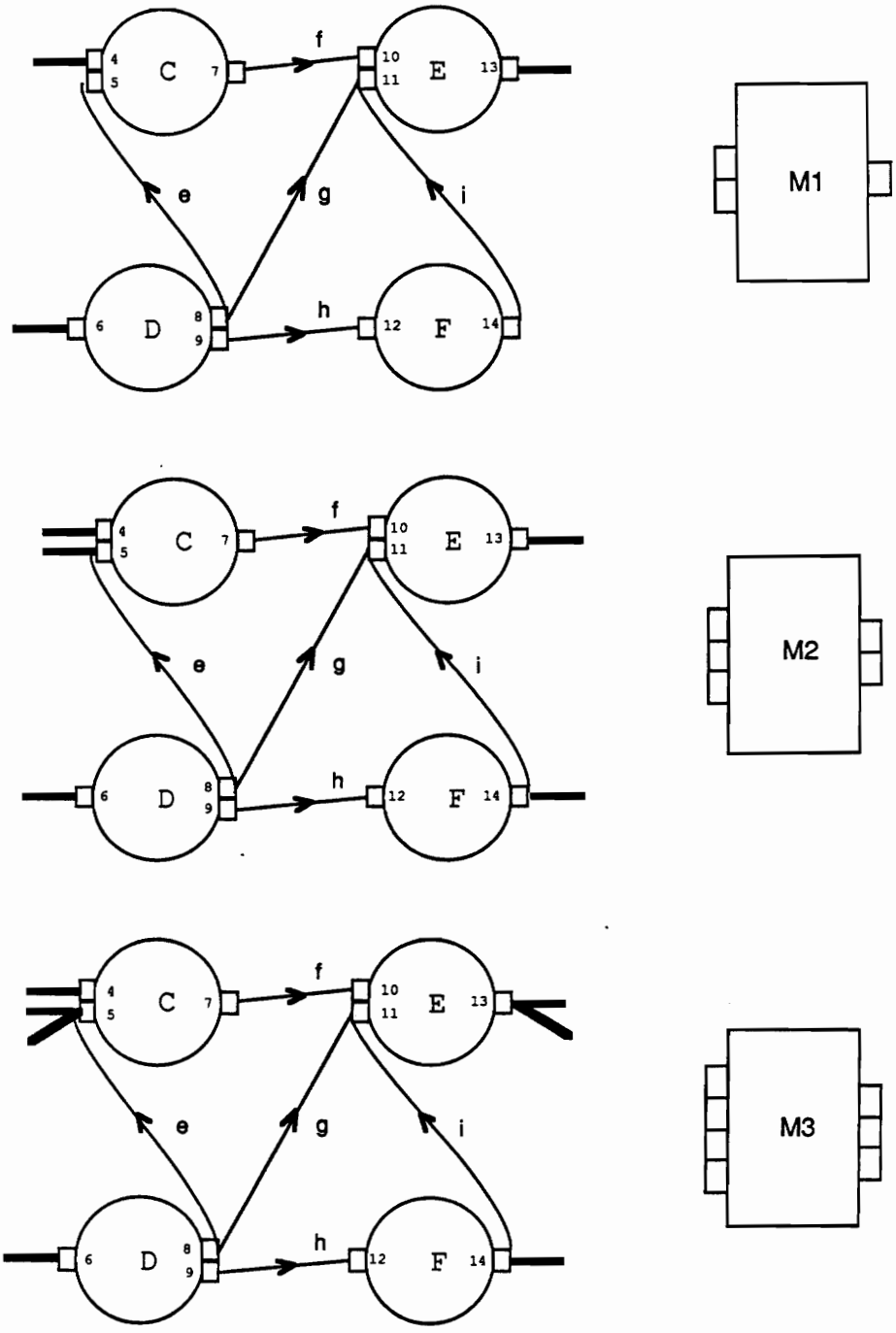


Figure 6. The three abstraction models of the cutset of Figure 5.

mation and as such is extremely context-specific. It would be ideal for applications requiring storage and reuse in the same or similar external context. In the Taskmaster environment, the abstraction facility is intended to functionally and visually abstract a tool-composite performing an identifiable high-level subtask. Hence model M2, which preserves the external functional context, seems to be best-suited for this purpose. Another important point of interest is the fact that the Taskmaster environment allows user-defined descriptions of pseudotools and hence all the external context information can be preserved textually if required. The next subsection discusses the actual implementation of the M2 model of abstraction in Taskmaster.

3.2.3 Editor Operations Supporting Abstraction

As mentioned in Section 3.2.1, the Network Editor has two new primitives supporting abstraction of tool-composites: ***save tool-composite*** and ***attach tool-composite***. In addition to these two operations, the ***collapse*** and ***explode*** operations have also been enhanced from their GETS incarnation to support the pseudotool abstraction. Recall that in the GETS environment, the ***collapse*** operation groups a set of nodes into a single "super-node" in the network topology display. This grouping corresponds to the M1 model of abstraction defined in the previous paragraph. But though the internal context is preserved, the ***view node*** operation does not work with "super-nodes" and hence the effect of the GETS ***collapse*** operation is only visual and abstracts no other information. In Taskmaster, there are three different ways to bring a pseudotool into a network:

- using the ***collapse*** operation to define a new pseudotool *in-place*,
- using the ***attach tool-composite*** operation to *import* a pre-defined pseudotool (defined with the ***save tool-composite*** operation), and
- using the ***expand node*** operation to construct a separate sub-network and abstract it into a new pseudotool.

The ***explode*** operation not only reverses the effect of the corresponding ***collapse*** done previously but also "explodes" pseudotools brought in either by the ***attach tool-composite*** or the ***expand node*** operation. The remainder of this section illustrates the working of each of these primitives, except ***expand node*** which is discussed in Section 3.3.

3.2.3.1 Collapse Operation

The collapse operation performs an in-place M2 abstraction of a collection of tools identified by the user. The resulting "super-node" implements some high-level operation. The collapse operation can be used recursively in the sense that it may be applied to a cutset which already contains collapsed pseudotools. The effect of the collapse operation is to redraw the network with the collapsed tool-composite being represented by a special "super-node" icon containing the user-supplied label. Figure 7 shows a cutset in the process of being collapsed. The *rubber band* polyline drawn with the mouse to delineate the cutset can also be observed in the figure. Figure 8 provides the detailed view of the two tools included in the cutset of Figure 7. Figure 9 shows the network after the collapse operation is completed. The "super-node" icon with a brick-pattern ring represents the newly defined pseudotool named ***solve***. The inset in Figure 8 shows the detailed view of the abstracted pseudotool. The Network Editor automatically pulls in the port descriptions for the pseudotool from the corresponding nested tools. The pseudotool name and description are solicited from the user before performing the collapse. Other than the ***specify node*** operation, all the other generic node operations can be performed on the "super-node". Now, if the user chooses to explode the ***solve*** pseudotool, the network will be redrawn to show its pre-collapse state of Figure 7.

3.2.3.2 Save Tool-Composite and Attach Tool-Composite Operations

The save tool-composite operation performs an M2 model abstraction of the cutset similar to the collapse. While the collapse operation does an in-place replacement of the cutset with the

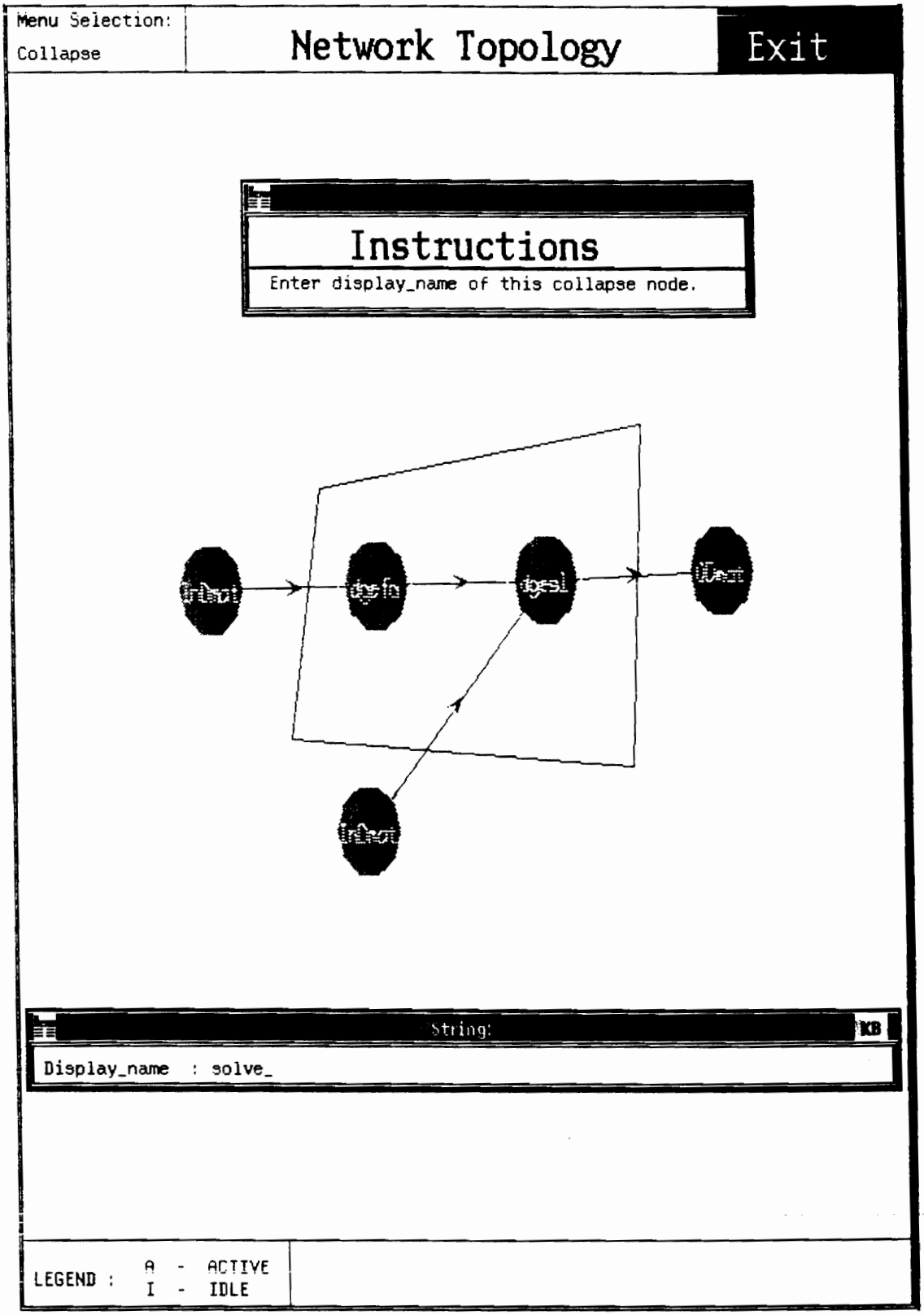
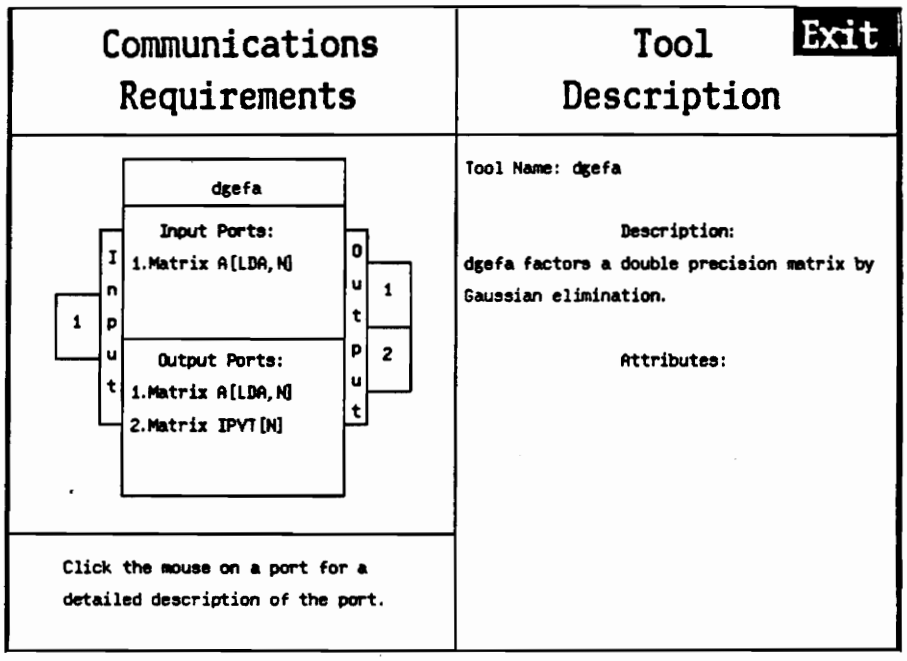


Figure 7. Network showing a cutset before the collapse operation.



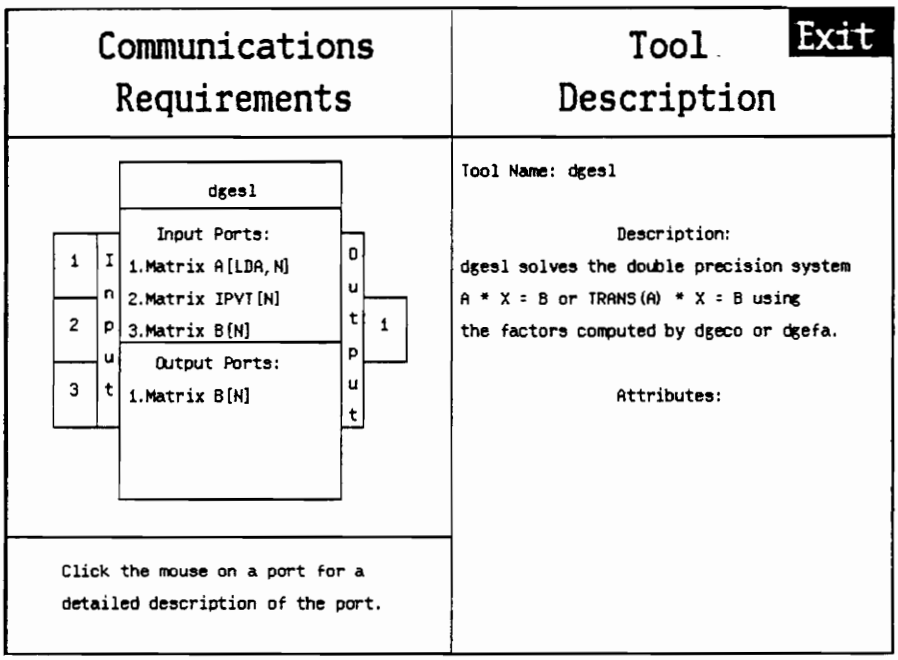


Figure 8. Detailed view of the two tools in the cutset of Figure 7.

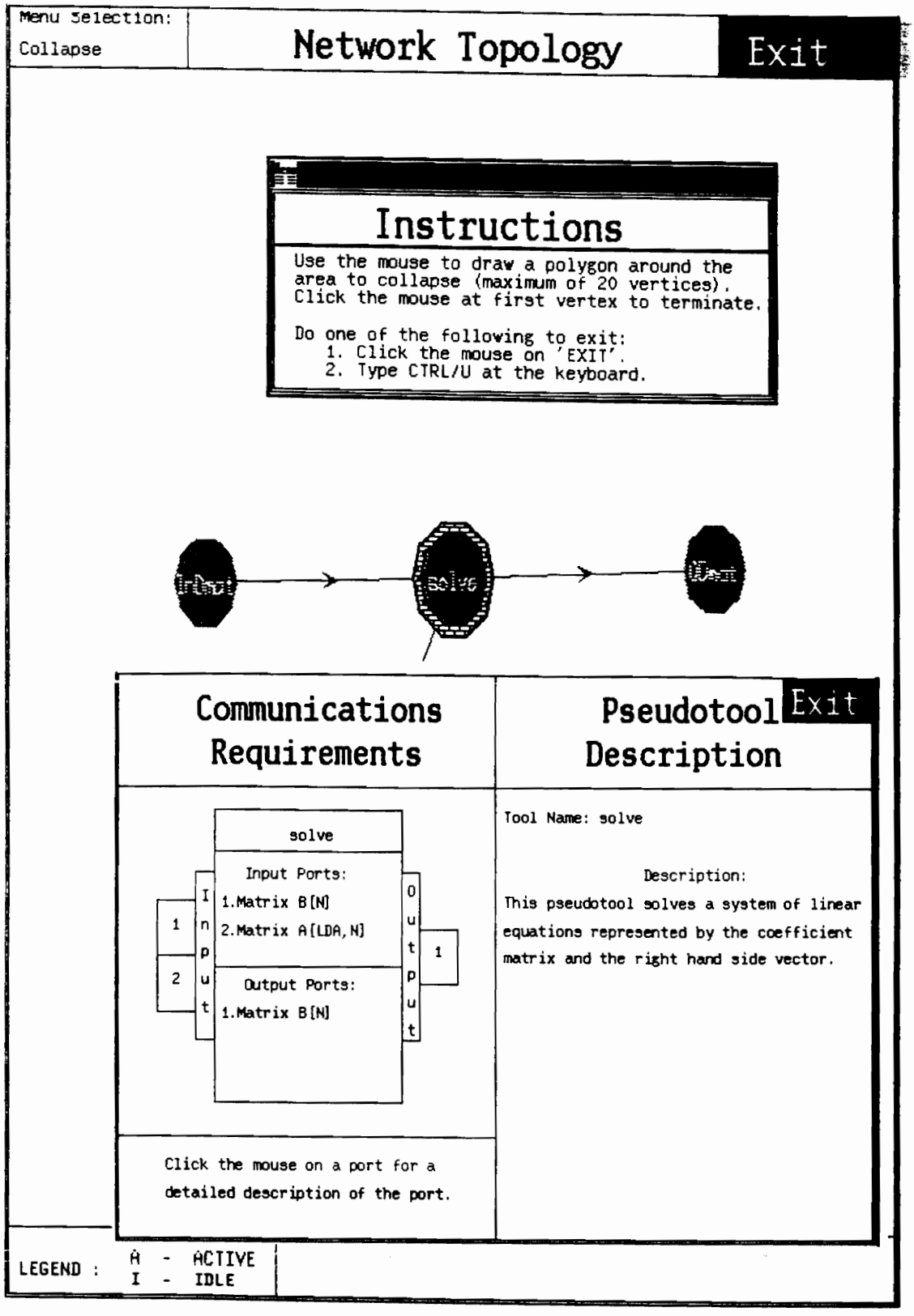


Figure 9. Network topology display after the collapse operation.

pseudotool, the save tool-composite builds the pseudotool based on the cutset and stores it for later reuse. The network topology display is not altered by a save tool-composite operation. The attach tool-composite operation is used to attach an already saved pseudotool to an unspecified node, and hence, is one way of specifying an unspecified node. Figure 10 shows a network with the unspecified node (highlighted) to which the pseudotool is to be attached. Figure 11 shows the same network after the pseudotool is incorporated into the network using the attach tool-composite operation.

3.2.3.3 Explode Operation

The explode operation reverses the effect of a previous collapse operation. It also expands an M2 abstraction of pseudotools brought in either by the attach tool-composite or the expand node operation (these two operations bring in sub-networks abstracted outside of the current context). This raises the problem of overlap when the explode operation is done in-place. Overlap is a problem caused by the possible physical overlaying of the node icons corresponding to the constituent nested tools of the pseudotool being exploded, over one or more of the node icons already being displayed. The Taskmaster environment strives to give the user complete freedom in the location and placement of the node and arc icons. Hence it does not normally perform any "pretty printing" operations on the user-defined networks. In the particular case of performing an explode operation on a pseudotool abstracted outside the current context, however, a radius transformation was initially employed to migrate all the nodes outwards in order to avoid possible confusion due to overlap. This radius transformation, however, tends to disorient the users of their intuitive positional context. So, in the interest of consistency, the current implementation does not attempt to alter the user-defined positional context of the network. Figure 12 shows the network topology display after the explode operation on the ***solve*** pseudotool of Figure 11.

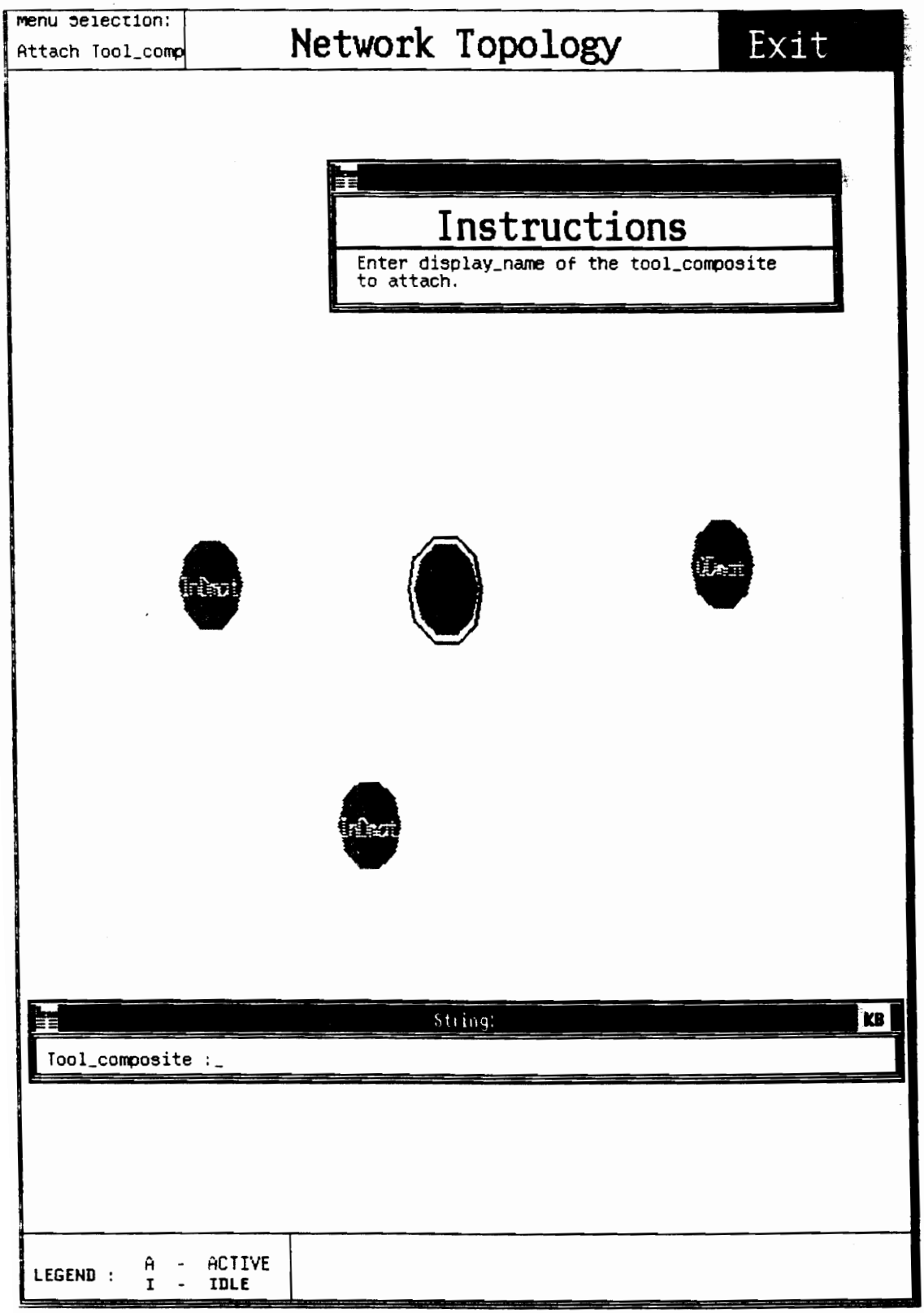


Figure 10. Network before the attach tool-composite operation.

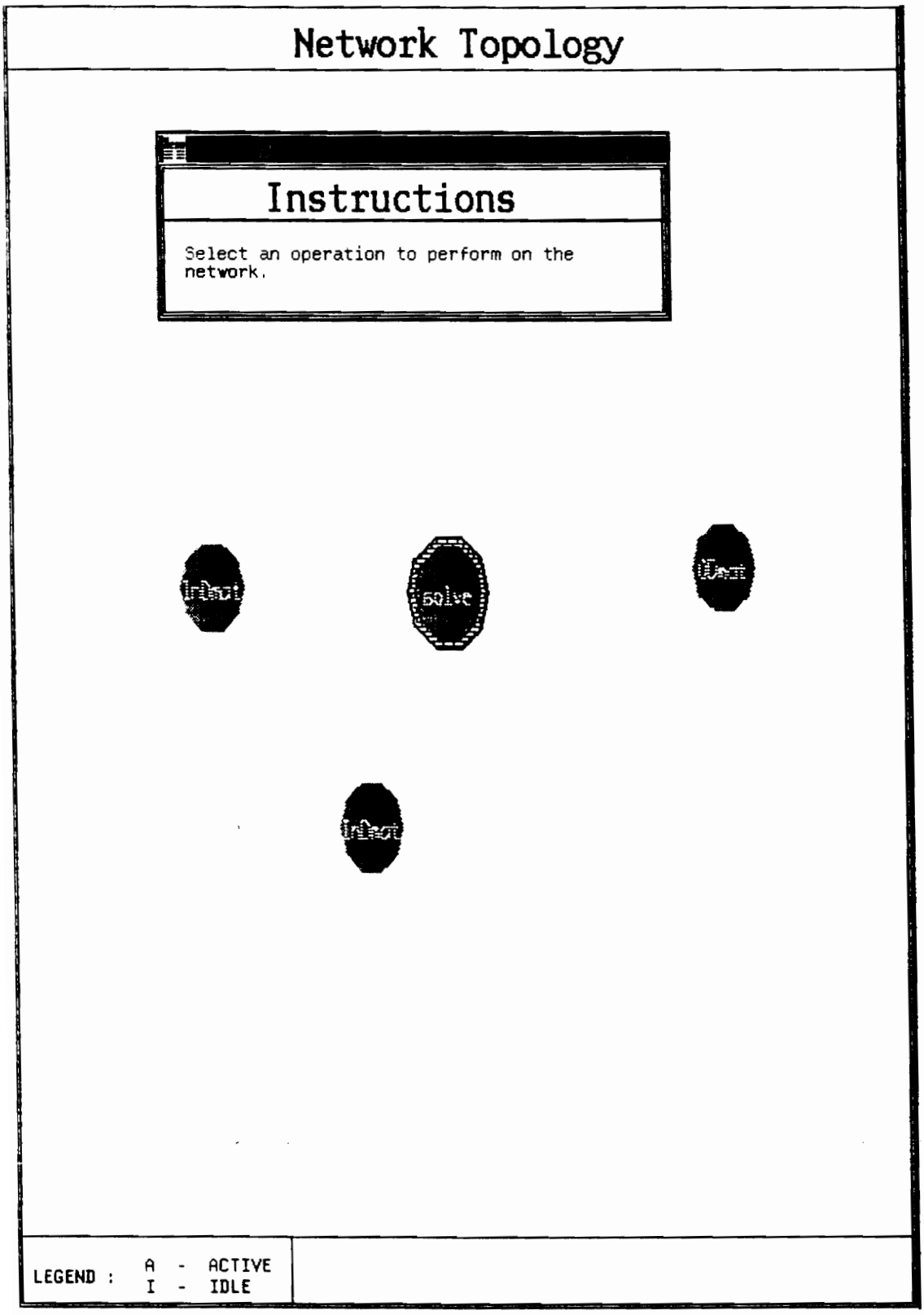


Figure 11. Network after the attach tool-composite operation.

3.3 *Editor Support for Top-down Specification*

The previous section discusses the support for abstraction in the Network Editor from a bottom-up perspective. The Network Editor also provides support for top-down specification involving functional decomposition of the problem task. The ***expand node*** operation allows the user to edit a separate network to implement a high-level operation and integrate it into the current network as a pseudotool. The expand node operation can be used recursively up to a depth of five levels. The recursion depth has been limited to control the complexity of the display. The unspecified node of Figure 11 can be specified using the expand node operation to build the ***solve*** pseudotool. Figure 13 shows the display configuration after the user has chosen to "expand" the unspecified node of Figure 11. As in the case of the collapse and save tool-composite operations, the system solicits the pseudotool name and description from the user. The user is supplied with a separate "node expansion" window to edit the sub-network to be integrated. All the normal editing primitives, including the undo and the tool-composite operations, are available to the user in the node expansion window. The node expansion window is logically the same as the network topology window and is represented physically by a window two-thirds the size of the network topology window. If a node in a node expansion window is to be expanded, a new node expansion window appears overlaying the previous one. Only one of these windows is active at any particular instant and the underlying inactive windows are clearly identified with an "inactive" message near the top right hand corner.

3.4 *Interactive Execution Monitoring*

As mentioned in Section 2.5.2, the GETS environment uses a very simplistic scheme to monitor the task network execution. The user at the local workstation has no direct means of knowing the status of the execution without actually logging on to the remote host where the Execution Monitor

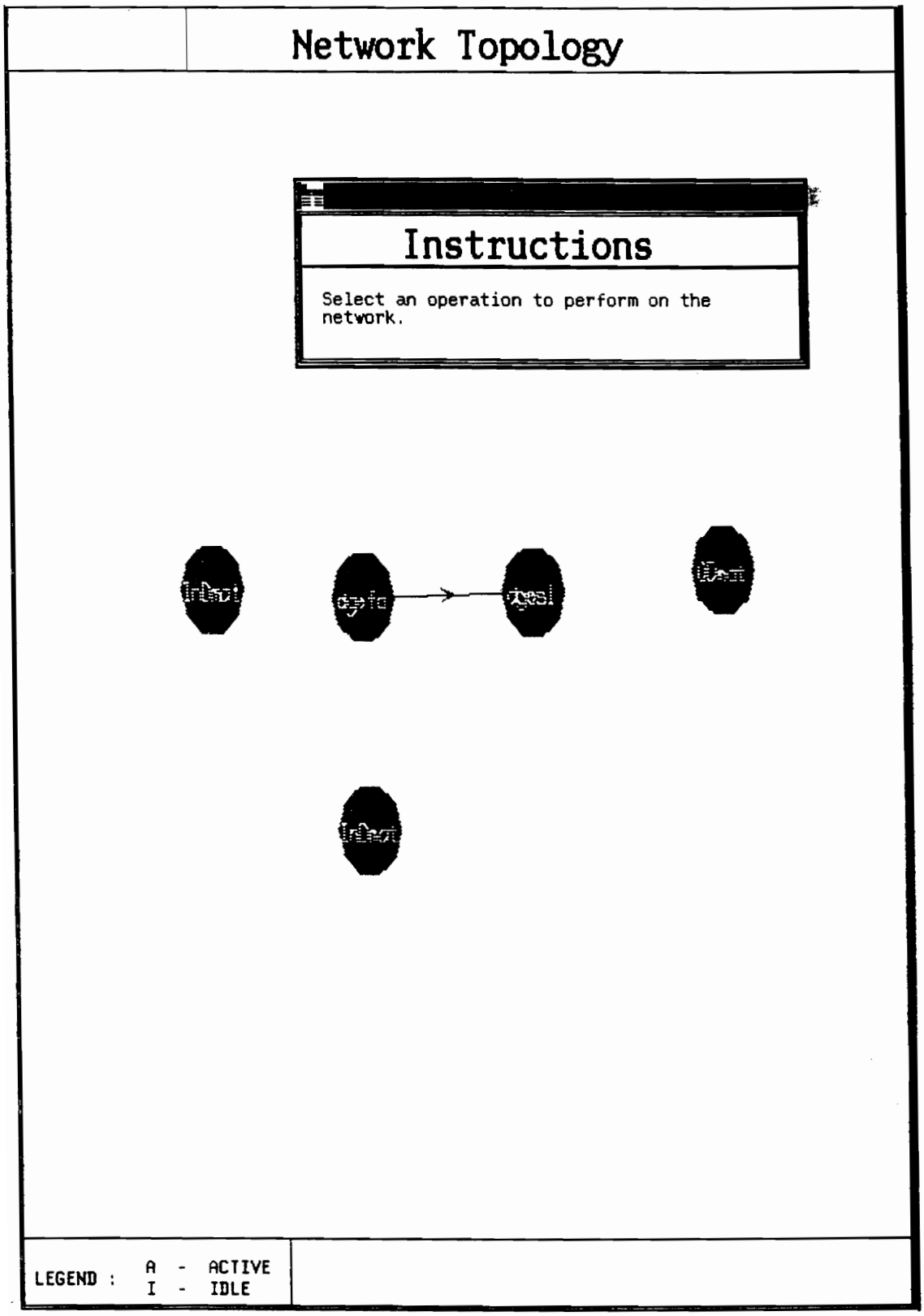


Figure 12. Network after the explode operation on the pseudotool of Figure 11.

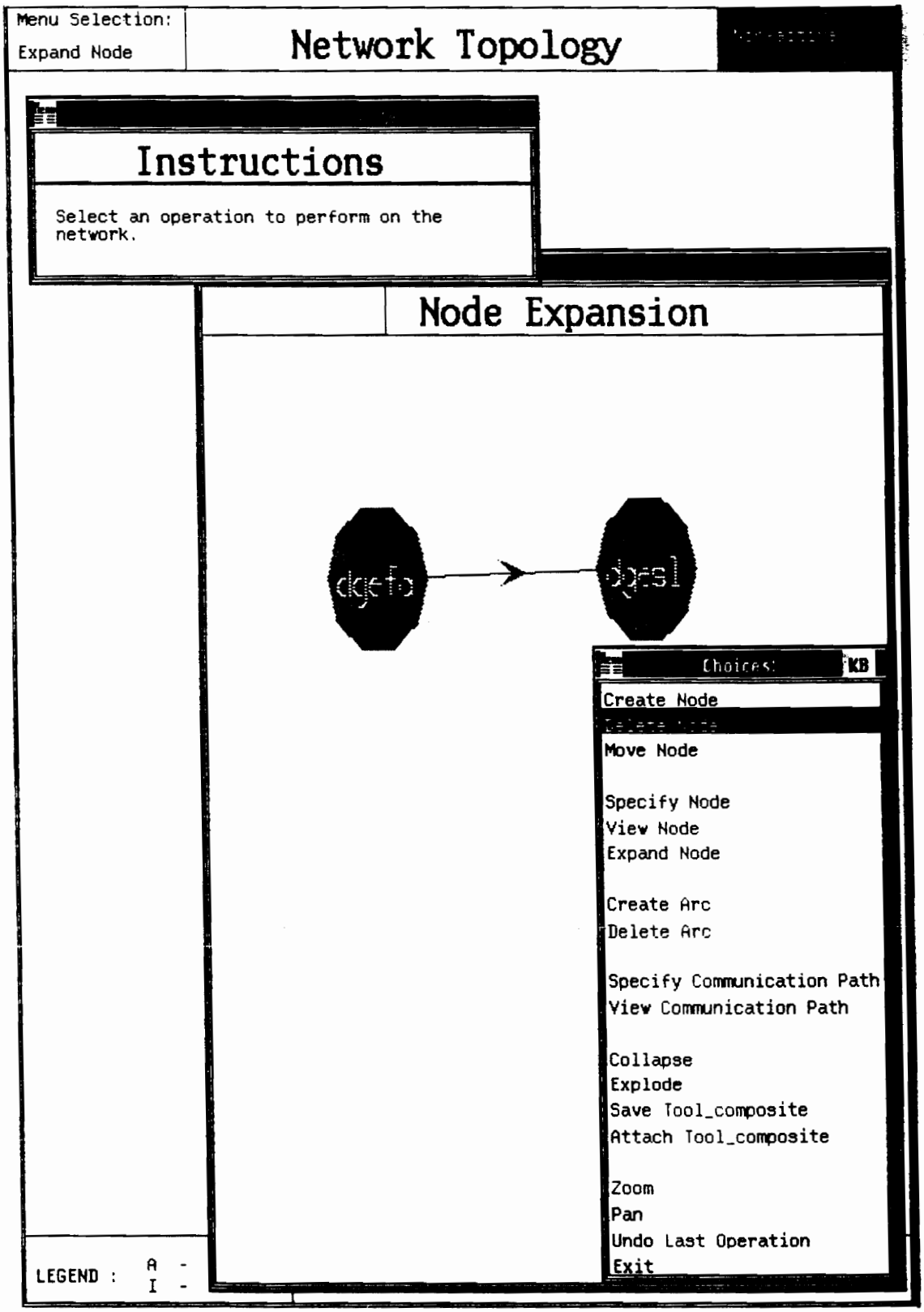


Figure 13. Network display during the expand node operation.

is resident. Taskmaster attempts to remedy this situation by providing an interactive facility to monitor the execution. In keeping with its *what-you-see-is-what-you-get* design philosophy, the feedback to the user is visual and intuitive. The Taskmaster environment provides the user with feedback on the instantiation of the network by signalling the initiation and termination of each node-associated process. In addition to this, Taskmaster system keeps the user continually posted with the actual execution status of every node-associated process. The remainder of this section discusses the network instantiation feedback and the network execution feedback from the user's viewpoint.

The purpose of the network instantiation feedback is to signal the user at the initiation and the termination of each node-associated process executing on the remote host. After a task network has been forwarded to the Execution Monitor for instantiation, every node in the network is in one of the following three states:

- *pre-instantiation* state where the node-associated process is yet to be spawned,
- *execution* state where the node-associated process has been spawned and is executing, and
- *post-execution* state where the node-associated process has completed its execution.

In prototyping the Taskmaster environment, we have explored two different visual approaches to presenting this information to the user. In the first approach, no visual distinction is made between the pre-instantiation and the post-execution states. In the execution state however, a node is highlighted by a surrounding pulsating ring. The underlying graphics support (GKS 0/b) does not provide for a batch screen update facility and hence it is not possible to synchronize the pulsation of the highlighting rings. This asynchronous flashing tends to be too distracting to the user and hence an alternative approach, which would be more aesthetically pleasing and less intrusive to the user, has been devised. In this approach, each of the three states is distinguished by filling the node icon with a different shade of gray. The pre-instantiation state is denoted by mild 75 percent white-on-black shade, the execution state by a 100 percent white filling and the post-execution state by a dark 25 percent white-on-black shade. The effect of this shading is that the user sees the node

becoming bright when its associated process gets spawned and retain the brightness till the process terminates, when it becomes darker and merges into the background. This approach achieves the goal of clearly differentiating the three node states and depicting the instantiation status of the network in a non-intrusive manner. Figure 14 shows a fully specified task network before execution and Figure 15 shows the same network in execution.

In addition to the feedback on the network instantiation status, Taskmaster also provides the user with incremental feedback of the execution status of each node-associated process. Thus every node in the execution state is marked to be ***active*** or ***idle*** and the display is continually updated to reflect any status changes. This can be seen in the network shown in Figure 15 where the active nodes are marked with the letter 'A' and the idle nodes nodes with the letter 'I'. This is in contrast to an earlier implementation where there was no incremental monitoring of the execution status and a process was assumed to be active from the time it is spawned till the time it terminates.

3.5 Execution-Time Operations

After specifying a task network and choosing to execute it, the user may wish to change the current view of the network. Taskmaster allows the user to perform many of the editing operations while the network is being executed. A similar feature of editing during execution is available in the PECAN [REIS84] system. The editing operations that modify actual network topology (like the specification operations) are not allowed during the monitoring phase. But the user's visual representation of the network topology can still be modified. Figure 16 illustrates the menu of the available subset of editor operations in the monitoring phase. The menu of editing operations is invoked by typing the user interrupt control sequence from the keyboard. The operations not available to the user during the execution are:

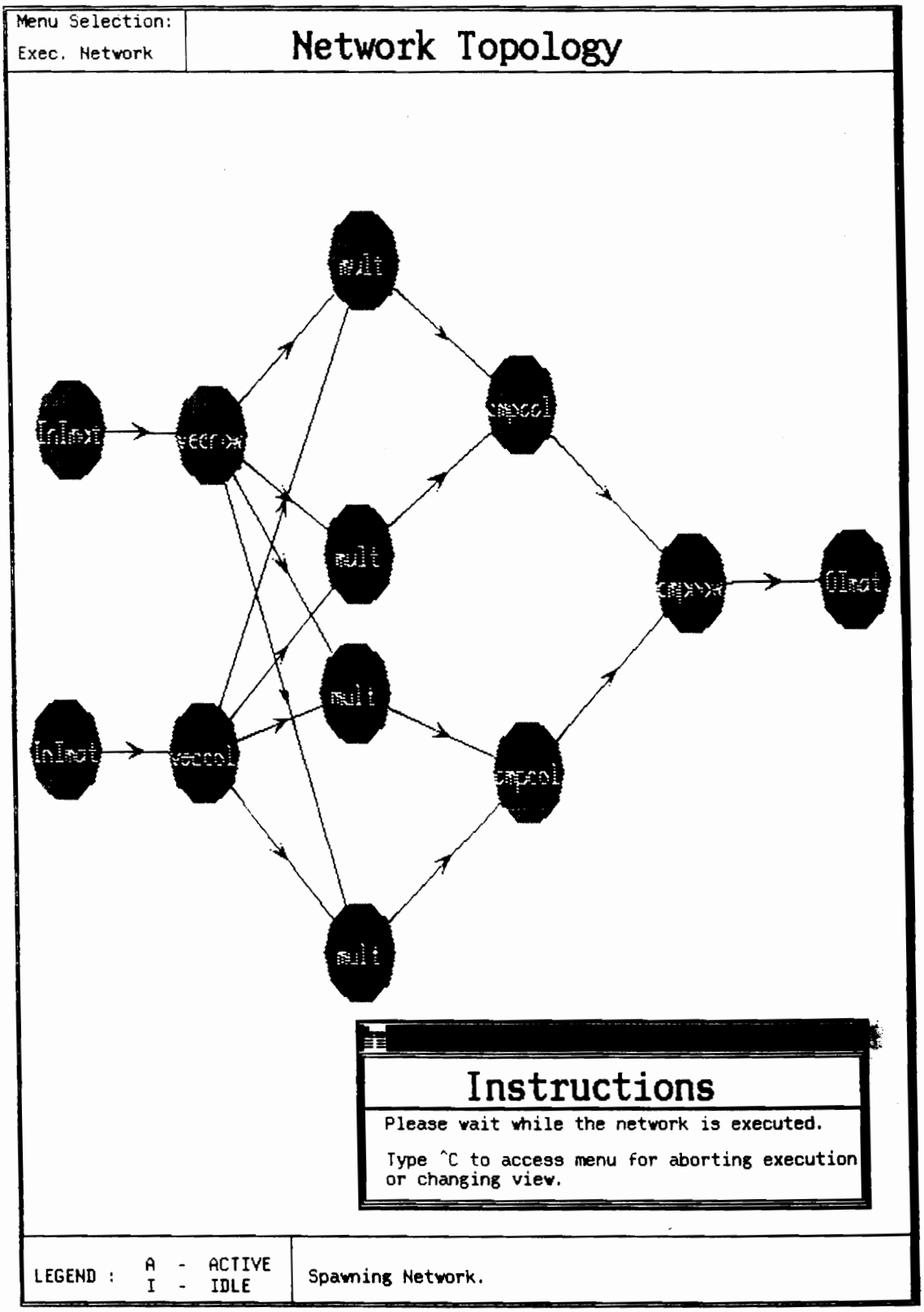


Figure 14. Example task network before execution.

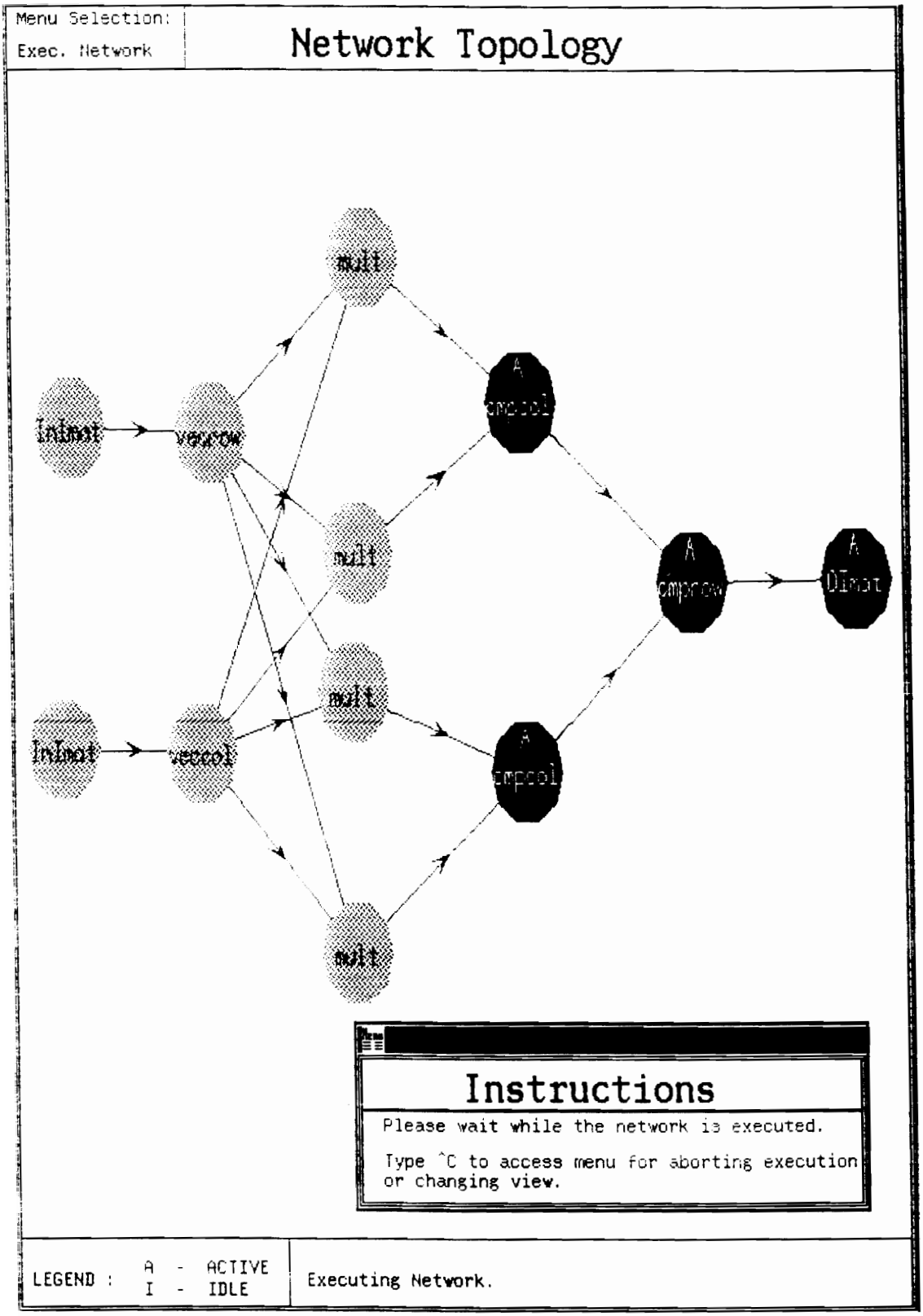


Figure 15. Execution status feedback of the task network of Figure 14.

- specify node,
- specify communication path,
- expand node,
- attach tool-composite, and
- restore network.

The user also has the choice to abort the execution of the network anytime during the execution. The Execution Monitor immediately aborts the execution of the network regardless of the current stage of execution.

3.6 Usage Statistics Collection

The effectiveness of a user interface can be assessed only after thorough testing over an extended period of time among a good sample of the intended audience. Hence, a facility to log the usage data automatically can provide input data for later statistical analysis of operation usage, based on various criteria. Some of the interesting factors for analysis are:

- the frequency of use of the undo operation,
- the frequency of a particular operation being undone,
- the extent to which the reusability and customizability features are used, and
- the type of specification used (top-down, bottom-up or hybrid).

The Taskmaster user interface incorporates automatic logging of the usage parameters. The parameters logged are the date and time of each editor session and all the editor operations used in that session, in the order of their use. This usage data logging facility has been provided to support future work on evaluating the effectiveness of Taskmaster's user interface.

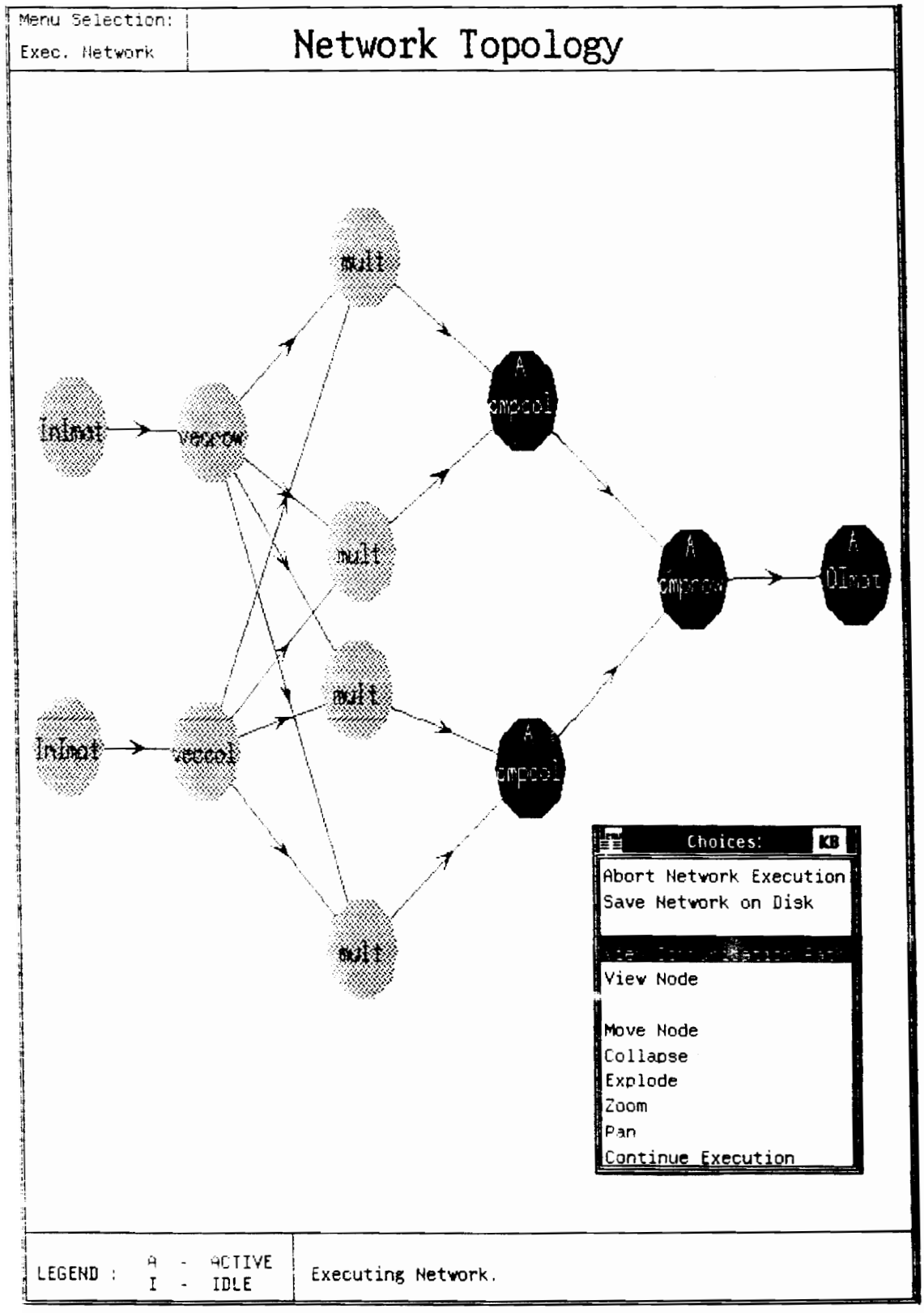


Figure 16. Menu of editor operations available during the monitoring phase.

Chapter 4

Implementational and Developmental Aspects of the Taskmaster System

4.1 Introduction

Chapter 3 discussed the Taskmaster system from a user's perspective. This chapter addresses the Taskmaster environment from a system perspective. It discusses the major internal aspects critical to the design as well as the implementation of the Taskmaster system. The main issues under consideration are the support for handling structured data, the execution status feedback mechanism and the internal representation of pseudotools. Section 4.2 which is on structured data flow, develops the background and motivation for supporting inter-tool communication using streams of structured data elements. It also discusses different approaches to providing this support. Next, Section 4.3 describes the approach adopted by the current implementation and the role of the Tools Database in supporting this approach. This section also includes a comparison of Taskmaster's approach with related work in structured data flow. Section 4.4 analyzes the implementational as-

pects of the execution status feedback mechanism described in Section 3.4. Section 4.5 discusses exception handling facility provided by Taskmaster. Finally, Section 4.6 discusses the internal representation used for a pseudotool in order to achieve operational consistency.

4.2 Structured Data Flow

This section provides the background, the motivation and the mechanisms for supporting structured data flow. The first subsection briefly describes the context and then brings out the need for supporting structured data flow. The next subsection discusses various design approaches to support structuring and the implementation considerations involved in each of the approaches.

4.2.1 Terminology, Background and Motivation

Recall from Chapter 2 that the Taskmaster environment is based on the ***data flow*** model of computation and employs the tool composition programming paradigm. As explained in Chapter 2, an intrinsic characteristic of such an environment is the use of ***external*** channels to achieve inter-tool communication because the communicating tools have no *a priori* knowledge of one another. An ***external*** channel allows for a *dynamic* binding between the communicating tools unlike an ***internal*** channel which imposes a *static* binding [ARTH83]. These external channels can be considered as active entities that receive data from the sender and forward it to the receiver. External channels usually provide data buffering by acting as intermediate repositories for communication data passed between modules [ARTH83]. By the term ***structured data flow***, we refer to the ability of the communication channel to recognize and process streams of *structured data* elements. A ***structured data element*** can be defined to be a grouping of one or more related data items into a single logical structure. For example, a ***record*** in the file transformation environment is a structured data element

consisting of a collection of ***fields***. By analogy with the file transformation domain, the term *record* will henceforth be used to denote a *structured data element*. Note that records can be *embedded* in the sense that they can contain sub-groupings of data elements.

The previous paragraph elucidates the terminology and the concepts related to structured data flow. This paragraph explains the need for structuring. Recall from Section 2.5.3 that Taskmaster and GETS employ the UNIX pipe facility to perform inter-process communication between tool-processes. The UNIX pipe facility imposes no structure on the data being communicated through it and handles the data as a simple character stream. In applications where the tools operate on "records" of related data, the above approach is not suitable. The domain of linear algebra is an intuitive example of an application domain that employs structured data elements, i.e. matrices. The application domain supported by GETS consists of file tranformation operations on simple character streams [MULH86]. Hence the need for structuring is not apparent. Taskmaster, however, is targeted towards a wider range of applicability and hence needs to provide support for structured data flow. Also, the granularity of the tools in the tool composition approach is "coarse" in the sense that a tool usually performs a high-level operation. This coarse-grained nature of the tools usually implies that the data objects operated on by the tools are also coarse-grained, i.e. they are composed of groupings of data elements relevant to the finer (lower level) aspects of the high-level operation. The next subsection discusses some of the approaches to structuring the data communication channel.

4.2.2 Approaches to Structuring the Data Communication Channel

Two distinct approaches can be considered to address the problem of structuring the communication channel:

- the ***object-oriented approach***, and

- the ***embedded structure descriptor*** approach.

Both of these approaches require the use of a (pre-processor, post-processor) pair to implement the communication protocol between the communicating tool-ports and also require access to port-related information from the Tools Database. Note that this discussion on structuring the communication channel is primarily concerned with *external* channels. Structuring is not a major concern for *internal* channels because the communicating tools possess *a priori* knowledge of each other's interfaces. The remainder of this subsection discusses each of the two approaches individually and in comparison.

4.2.2.1 Object-oriented Approach

In the object-oriented approach, a post-processor is attached to the output port of the producer node and a pre-processor is attached to the input port of the consumer node. On instantiation of the producer node, the post-processor associated with each of its output ports queries the Tools Database and obtains the definition of the output data object type. Using this information it creates an instance of that type corresponding to the particular data element being output. The post-processor then sends the object handle (pointer) down the communication channel to the pre-processor. The object instances could be created in either locally-shared or globally-shared address spaces. Note that an important requirement of this approach is that the tools share the same address space (locally or globally). The advantage of this approach is the high degree of reduction in transmitted data because only pointers need to be passed between the communicating ports. This protocol requires both the post-processor and the pre-processor to access the Tools Database.

4.2.2.2 *Embedded Structure Descriptor Approach*

The embedded structure descriptor approach also requires a post-processor and a pre-processor to be provided at the opposite ends of a communication link. As in the object-oriented approach, the post-processor obtains the data structure description by querying the Tools Database. In this approach, the post-processor interleaves/embeds structure descriptors within the data stream to enable the pre-processor to decode the structures. All the information required by the pre-processor to decode the structures is available in the data stream itself and hence the pre-processor need not access the Tools Database. The disadvantage of this approach is the increase in the amount of transmitted data due to the addition of structure descriptors.

4.2.3 Issues Impacting Structured Data Flow

The previous subsection of this chapter presents two approaches to structuring the communication channel after providing the requisite background and motivation. This subsection will address many of the important issues related to the choice of the structuring approach. The issues under consideration are

- binding time and coupling strength,
- time and space efficiency,
- openness or ease of integration,
- granularity of the tools, and
- degree of homogeneity of the data stream.

The remainder of this section discusses each of these issues and how it relates to the choice of the structuring approach.

4.2.3.1 *Binding Time and Coupling Strength*

Static (or early) binding implies that all the semantic information related to the tool interfaces is precisely known beforehand. Static binding enhances reliability because thorough verification of the compatibility of interfacing components becomes possible. ***Dynamic*** (or late) binding, on the other hand, assumes no *a priori* knowledge of the tool interfaces. The advantages of dynamic binding are reusability and wide range of applicability due its flexibility in changing the representation. The concept of coupling is closely related to the concept of binding time. Binding time criterion is based on the idea of tools being the primary interaction entities while coupling shifts the focus to the communication channel as the primary interaction entity. Intuitively, the strength of coupling is inversely related to the lateness of binding. In other words, the earlier the binding the stronger the coupling. Concerning the two approaches to structuring discussed earlier in this chapter, both approaches use dynamic binding. The strength of coupling, however, is moderate in the sense that the coupling is indirect via the Tools Database.

4.2.3.2 *Time and Space Efficiency*

Another factor affecting the choice of the structuring approach is the efficiency of implementation with respect to both time and space. The object-oriented approach is much more time-efficient than the embedded structure descriptor approach because of the reduction in transmitted data. It may or may not be more space-efficient depending on the overhead incurred in the implementation of the object definition facility.

4.2.3.3 *Openness or Ease of Integration*

Ease of integration or "openness" of the architecture refers to the level of effort involved in adding (or deleting) tools to (from) the environment at arbitrary points in time [KAPL86]. The use of in-

dependent objects (pre-processor and post-processor) to implement the structuring, contributes to a very open architecture. Any new tool can be easily integrated into the environment, at any time, by completely and precisely defining its communication requirements. Both the structuring approaches mentioned earlier support easy integrability, by transferring the burden of interfacing away from the tools and on to independent communicating objects. Also, if data structure conversion is required to link two tools with different interfaces, a *format translator* tool can be interposed between the tools to act as an "interpreter".

4.2.3.4 Granularity of the Tools

As discussed in Section 4.2, granularity of the tools is an important issue affecting the structuring approach. In general, the more sophisticated the tool, the more complex is its interface. In the two approaches to structuring discussed earlier, the pre-processor and the post-processor, in conjunction with the Tools Database, implement the structuring. In the object-oriented approach, the granularity of the tool determines the complexity of the object definition capability. This effect is not very significant because the object definition capability can be easily augmented to handle complex objects. In the embedded structure descriptor approach, the granularity plays a more significant role because the complexity of the data structure directly affects the complexity of the encoding scheme.

4.2.3.5 Degree of Homogeneity of the Data Stream

By *homogeneity* of a data stream we refer to the similarity among various structured data elements forming a data stream. A fully homogeneous stream consists of identical structured data elements. A heterogeneous stream consists of structured data elements that are physically different from each other in one or more characteristics (e.g., length, type, precision, etc.). The very term object-oriented implies that this approach can handle heterogeneous streams because encapsulation is one

of the inherent features of the object-oriented approach. Embedded structure descriptor approach can also handle heterogeneity to a limited extent but with increased encoding complexity.

4.3 *Taskmaster's Approach to Structured Data Flow*

The previous section discusses the fundamental concepts related to structured data flow. Using this framework, this section will address the approach adopted by Taskmaster to support structured inter-tool data flow. Taskmaster employs the embedded structure descriptor approach to structuring. The first subsection deals with the design and implementation aspects in adopting this approach. Next, Subsection 4.3.2 discusses the role of the Tools Database in supporting this approach. Finally, Subsection 4.3.3 makes a comparative analysis of Taskmaster's approach with related work in structured data flow.

4.3.1 Design and Implementation Aspects

As mentioned above, Taskmaster employs the embedded structure description method to structuring the communication channel. The object-oriented approach is automatically eliminated from consideration because of its requirement of shared memory between processes. UNIX, the operating system on which the Taskmaster execution environment is built, does not provide support for inter-process memory sharing. Taskmaster provides the following standard set of stream primitives which includes the simple character stream format of UNIX:

- integer,
- character (binary),
- floating point number,

- double precision floating point number,
- multiple integer,
- multiple character (text),
- multiple floating point number, and
- multiple double precision floating point number.

An ***integer stream*** is a stream of integer elements. A ***multiple integer stream*** is a stream of structured data elements containing one or more integers delimited by a special delimiter integer. The definitions of the other stream primitives follow by analogy with the above definitions. A ***character stream*** can also be termed as a ***binary stream*** because it consists of a stream of bytes or binary data. Similarly, a ***multiple character stream*** is analagous to a ***text stream*** in that text records are usually defined as one or more characters delimited by the newline character. The multiple streams supported are examples of limited heterogenous streams (mentioned in Section 4.2.3.5) in that each structured data element can be of variable length. The encoding information consists of a header prefixed to every variable length structure. The header contains information about the actual length of the variable length structure. This information is used not only by the pre-processor to reconstruct the structures, but also by the "merge" tool. The "merge" tool is a special tool that merges multiple inputs to the same port making sure that the stream interleaving preserves structure integrity. The "dup" tool is another special tool that produces multiple copies of the output for output ports with multiple connections. The post-processor, the pre-processor, the "merge" tool and the "dup" tool are special in the sense that they are system tools which are invisible to the user. The Network Editor automatically integrates them into a user-built task network at the appropriate locations. Note that the "merge" tool has an important role in the structuring process because of its critical function of preserving the record integrity.

4.3.2 Tools Database Support

This section discusses the support provided by the Tools Database in the structuring process. The Tools Database acts as a system dictionary and holds all the information pertaining to the communication requirements of the input and the output ports of each tool. Referring back to the "software IC" analogy of Section 2.4, the Tools Database plays the role of an IC "cook book". When a new tool is to be added, its communication requirements need to be precisely defined to the Tools Database in terms of one of the available standard stream formats. This information is used later for a variety of purposes:

- automatic type checking of structure compatibility of the input and output ports before making a connection requested by the user,
- encoding of the embedded structure information into the outgoing data stream by the post-processor, and
- preservation of the structure integrity in the merging of multiple input streams by the "merge" tool.

From the viewpoint of structuring, the primary role of the Tools Database is its object-oriented nature. By encapsulating all the tool communication requirements within it, the Tools Database allows dynamic binding without sacrificing verification. The responsibility of type checking is shifted to the system (the supplier) rather than the user (the consumer) in accordance with the object-oriented programming paradigm [COXB86]. The logical culmination of this approach would be the use of an object-oriented tool description language (functionally similar to the Interface Description Language (IDL) [NEST82]) to describe the structures.

4.3.3 Comparison with Related Work

The previous sections describe Taskmaster's solution to the problem of structuring the data communication channel. This subsection compares this solution approach in relation with other work in structuring.

One area of related research is on module interconnection in conventional programming languages focusing on specifying interconnections between separately or jointly compiled modules sharing the same address space [SNOD86]. Special languages (e.g. MIL75 [DERE76], INTERCOL [TICH79], C/Mesa [MITC79]) have been proposed to address this problem. Extensions to existing languages (e.g. Anna [LUCK84], PIC/Ada [WOLF84]) have also been attempted. Another important approach to inter-module communication is offered by DIANA [GOOS83], an interface formalism for passing Ada programs between modules, using attributed abstract syntax trees. This research, however, addresses interconnections of tools where the granularity is larger and the modules more disjoint. The conventional efforts at specifying the interactions in terms of shared variables, types and procedures [SNOD86] are not sufficient in this realm of programming based on tool composition and data flow.

The UNC IDL Toolkit [SNOD86] proposes extensions to the Interface Description Language (IDL) [NEST82] in order to connect modules together. This addresses interconnections between tools with the same granularity as Taskmaster but involves compilation and linking of the tools whose interfaces are described in IDL. Taskmaster employs a different approach where the binding is dynamic but still allows for interface type checking. This is achieved by the use of a Tools Database to encapsulate the tool interface specifications.

The Open Systems Architecture (OSA) proposed in the Illinois Software Engineering Program (ISEP) [KAPL86] is similar to Taskmaster's as well as that of the UNC IDL toolkit. Specifically, the "sort" concept of ISEP is analogous to the "structure" concept of UNC IDL and the "record"

concept of Taskmaster. The "communication objects" of ISEP are similar to the post-processor and pre-processor used by Taskmaster. Similar to ISEP's proposal for use of standard "sort morphisms" (data conversion functions), Taskmaster employs standard stream formats. ISEP proposes three solutions to the problem of tool recognition [KAPL86]. Taskmaster's approach to structuring, using the Tools Database to support tool recognition, is essentially the same as the third solution proposed by ISEP.

Polylith [PURT85,PURT85a] is a *software bus* that supports communication between modules written in diverse languages. Polylith addresses modules whose granularity is somewhat smaller than the tools in the Taskmaster environment. Polylith has an internal language for describing structures and also provides a graphical editor (similar to Taskmaster) for interconnection of modules.

Toolpack [OSTE83], an environment for the development of mathematical software, uses the Integrated System of Tools (IST) which is based on passing structures between tools. IST uses a centralized database similar to Taskmaster. But unlike Taskmaster, Toolpack handles structuring by providing more management support in the form of a file system and a command language interface [OSTE83,SNOD86] and is specific to a particular language.

4.4 Implementation of the Execution Status Feedback Mechanism

The previous sections discuss the support provided by Taskmaster for structured data flow. This section addresses another important structural improvement in the Taskmaster system, i.e. the execution status feedback mechanism. The user's view of the execution status feedback has already

been described in Section 3.4, which discusses interactive execution monitoring. This section will discuss the execution feedback mechanism from an implementation perspective.

Recall from Figure 1 that the Taskmaster environment is physically partitioned across two machines with the Network Editor located on the local workstation and the Execution Monitor and the Tools Database residing on the remote host. Networking support for communication between the machines is provided by the DECnet Network [VNET86,UNET85]. The Execution Monitor is defined as a DECnet object on the remote host and this object definition facility is used by the Network Editor to directly communicate with the Execution Monitor. DECnet objects provide known general-purpose network services. This mechanism is used to open a two-way, buffered communication channel between the Network Editor and the Execution Monitor. The Network Editor and the Execution Monitor communicate with each other using the following message types:

Network Editor messages

- OK,
- ABORT_EXECUTION,
- START_OF_DATA, and
- END_OF_DATA.

Execution Monitor messages

- OK,
- PROCESS_SPAWN_DATA,
- EXECUTION_ABORTED,
- PROCESS_STATUS_DATA,
- END_OF_EXECUTION, and
- ERROR.

The Execution Monitor maintains a table of the current state of each process. The *process status* command of UNIX is used to periodically obtain the execution state of each tool-associated process. The current state is compared with the previous state and if there is a change, the Execution

Monitor reports it to the Network Editor. This enables the Network Editor to continually monitor the network execution. Note that this model is very similar to the distributed client-server model used by the X Window system [SCHE86].

4.5 Exception Handling

As mentioned in the previous section, the Network Editor and the Execution Monitor communicate with each other using a set of pre-defined messages. The Network Editor controls the entire execution process using the Execution Monitor. At any time during the execution, if an exception has occurred, the Execution Monitor sends the ERROR message to the Editor indicating the nature of the exception. Taskmaster also provides a framework where the tools can directly signal exceptions to the Network Editor. This option however, needs to be specified at compile-time. The Network Editor can abort the Network Execution at any point in time by sending the ABORT_EXECUTION message to the Monitor. The Monitor then kills all the tool processes before exiting. Everytime the user executes a network, a new instance of the Execution Monitor monitors the execution.

4.6 Internal Representation of Tool-Composites

Finally, to conclude this chapter, this section discusses the internal representation used by the Network Editor to characterize tool-composites. When a new pseudotool is to be defined, the Network Editor saves the current network representation in a network stack. The Editor then marks the nodes and links outside the cutset for deletion. After this, the deleted node and arc labels are

reused and a "cleaned-up" version of the network structures describing the cutset is constructed. The "collapse" algorithm is used to mark all the nodes inside the cutset, as collapse members. The Editor builds a list of "pseudo-input" and "pseudo-output" ports analagous to the structure used for a basic tool. If the pseudotool abstraction is *in-place*, the "super-node" location is computed to be the centroid of the sub-network delineated by the cutset. If the abstraction of the sub-network is to be saved for later *import* outside of the current context, the "super-node" location is translated to the origin. The user-supplied pseudotool label and description are used to fill the corresponding fields for the basic tool which are usually extracted from the Tools Database.

Chapter 5

Taskmaster - An Application Perspective

The earlier chapters of this thesis have discussed the Taskmaster Environment from the user and system perspectives. To complete the picture, this chapter will address Taskmaster from an application perspective. Effective development of applications is the ultimate goal of any programming system, especially so in the case of user environments. An environment, whatever its capabilities, is only as good as the applications supported by it. Thus, it is the purpose of this chapter to profile Taskmaster from an application developer's perspective. The first section of this chapter discusses the factors to be considered in determining the suitability of an application domain to Taskmaster's problem solving philosophy. The next section presents an in-depth discussion of the two application domains currently supported by the Taskmaster environment. To conclude the chapter, the last section explores many potential application domains suitable for implementation in the Taskmaster environment.

5.1 Criteria Determining Suitability of Application Domains

A number of factors need to be considered in order to determine whether a particular application domain is suitable for development within the Taskmaster environment. This section analyzes some of the important criteria used to determine an application's suitability to Taskmaster's programming paradigm. The criteria under consideration are:

- reusability,
- composability,
- structuring, and
- time dependency.

The remainder of this section discusses each of these criteria individually.

5.1.1 Reusability

Software reusability is the central concept underlying the tool composition programming paradigm employed by Taskmaster. Recall from Section 1.2 that software reusability has been shown to be the most significant factor in improving software development productivity and quality [BOEH84]. Reusability is being used as a catch-all term to address a spectrum of approaches from reusable pieces of program code, to reusable program design, to reusable program specifications [WIRR84]. With respect to Taskmaster, reusability refers to the first approach of employing reusable building blocks. The current implementation of Taskmaster assumes that the tools available in the Tools Database are generic and reusable. It is left to the application-developer to ensure that the tools are

developed with reusability in mind. Developing reusable software involves the application of the *black box* approach where modules (tools) can be used based solely on their functional and interface descriptions and without requiring understanding of any internal aspects. Particularly addressing the problem of "converting" an existing application for use in the Taskmaster environment, care must be taken to ensure that the functionality of the application is exported while removing/hiding any context-specific information.

5.1.2 Composability

Composability refers to methods of combining existing reusable components to construct a bigger, more specialized component [RAMA86]. With reference to Taskmaster, composability requires the availability of *format translators* (referred to in 4.2.3.3) to convert the data from one format to another. These format translators are needed to interface ports with incompatible input/output data stream formats.

5.1.3 Structuring

Another factor that needs to be considered before an application is targeted for Taskmaster is the feasibility of expressing the input and the output interface structures in terms of the available standard streams. In general, this implies that applications that do not easily fit within the scope of the standard streams provided by Taskmaster can still be used in the Taskmaster environment but will not utilize the advantages provided by the type checking facility.

5.1.4 Time Dependency

Recall from Section 2.2 that Taskmaster uses a high-level data flow organization in which the execution sequencing is constrained only by the data dependencies between the different computation stages. This concept of asynchrony, while contributing to its high degree of concurrency, also limits Taskmaster to applications that are purely deterministic i.e. do not contain timing constraints. Non-deterministic extensions to the data flow architecture (similar to the extensions in Stream Machine [BART85]) can be incorporated into Taskmaster but will result in losing the benefits of purely deterministic computations.

5.2 Two Prototype Implementations

The previous section discusses the criteria to be considered for suitability of an application to Taskmaster's programming philosophy. This section presents the prototype implementations of two different application domains for the Taskmaster environment:

- a data flow command shell for UNIX, and
- programming with the LINPACK [DONG79] software tools.

The remainder of this section discusses each of the above applications individually.

5.2.1 A Data Flow Command Shell for UNIX

Recall from Section 2.3 that the pipe mechanism provided by the conventional command shells of UNIX allows for the linear composition of tools. While this linear composition mechanism is adequate for single-input, single-output tools, it is not general enough for the composition of many other tools available in the UNIX environment that do not satisfy this constraint. Some common examples of tools that do not satisfy the single-input, single-output constraint are *comp, diff, grep, sort*, etc.. Tools like *comp* and *diff* necessarily require more than one input while tools like *grep* and *sort* can handle multiple inputs (and outputs) optionally. A purely *linguistic* command language (like the conventional command shells for UNIX) cannot exploit the two-dimensional capabilities available in these tools. Hence, one feels the need for a two-dimensional, i.e. visual-based, command language in order to take full advantage of the capabilities of the system. This need is the basic motivation behind the development of GETS and subsequently Taskmaster. Thus, a data flow shell for file transformation operations in UNIX is an automatic choice for application in the Taskmaster and GETS environments. The remainder of this section discusses the nature and scope of this application with respect to the capabilities and limitations of Taskmaster.

UNIX provides a highly composable and reusable set of tools because reusability and composability were part of the original design considerations underlying its development. UNIX also assumes that tools communicate among themselves using streams of text. This is easily mapped to the multiple character (text) standard stream provided by Taskmaster by defining the record delimiter to be the newline character. Thus, a visual data flow shell is easily implemented that not only exploits the two-dimensional capabilities of the available tools but also provides an environment conducive to the development of new and more sophisticated tools. All the capabilities provided by the conventional command shells are still available with the only requirement being that they be explicitly installed in the Tools Database. File transformation operations can be grouped under the following three categories, based on the granularity of their operands:

- file operations,
- record operations, and
- character operations.

In order to be consistent, all file transformation operations that perform filtering are assumed to be pure filters. That is, they explicitly read the input data via their input ports and write the output data to their output ports. This assumption is needed to avoid multiple instances of the same tool being installed in order to simulate the tool incarnations that use non-standard input (or output). Special tools that implicitly act as sources (or sinks) are used to indirectly perform disk input (or output). The operating system commands like *ls* and *ps* that act as natural sources can also be used to feed data into a network. Experience has shown that this visual approach to specifying the data flow is very intuitive and provides enhanced support for reusability compared to the conventional command language interfaces that are restricted to linear composition.

5.2.2 Programming with the LINPACK Software Tools

The previous subsection discusses the prototype implementation of a data flow command shell for UNIX. This subsection discusses the prototype implementation of another application domain i.e. mathematical software - in particular, the LINPACK software package for linear algebra. LINPACK [DONG79] is a collection of routines, implemented in FORTRAN, that solve various systems of linear algebraic equations. LINPACK has been chosen as an application for Taskmaster because it

- is a popular and frequently used library of mathematical software,
- provides an opportunity to illustrate Taskmaster's structuring facility, and
- provides an opportunity to analyze the effort involved in "converting" a pre-existing application for use in the Taskmaster environment.

The remainder of this subsection discusses this application from an implementation viewpoint.

The choice of LINPACK as the target application poses many challenges. Firstly, LINPACK is implemented in the FORTRAN language for use as subroutines callable from other applications. Hence, these routines do not directly fit into the model of a *tool* used in the tool composition programming paradigm. Secondly, unlike the UNIX tools, the LINPACK routines are not originally designed to be used as filters (particularly in the form of independently compiled modules). Instead, the emphasis in their design is on machine independence and optimum efficiency under different operating environments. In particular, there is no notion of standard input and output stream descriptors. Thirdly, LINPACK uses matrices of various data types (e.g. real, double precision, complex, etc.) and storage forms (e.g. General, Triangular, Tridiagonal, etc.). In addition, the dimensions of the matrices are also variable. A consistent methodology is needed to represent the above-mentioned types and forms within the framework of standard streams provided by Taskmaster.

To address the problems mentioned in the previous paragraph, a novel approach involving the encapsulation of each FORTRAN subroutine within an interface shell, has been devised. This interface shell (implemented in the C programming language) takes care of the communication requirements of the tool in a flexible, yet consistent manner. Note that the interface shell is not an independent communication object (like the pre-processor and post-processor of Section 4.2.2) but an integral part of the tool. An additional advantage of this approach is that this interface shell can be used to hide the idiosyncrasies resulting from the language-specific implementation. For example, FORTRAN's column major array storage can be made transparent to the user.

Addressing the problem of variable dimension matrices, two possible solutions can be formulated:

1. the dimensions are treated as separate data distinct from the matrix and separate ports (input and output) are assigned exclusively for the dimensions, or

2. the dimensions and the matrix are treated as a single logical structure and are conveyed through the same port.

Although the first approach appears to be acceptable at first sight, it requires an additional port (input or output) for every matrix data stream. Considering the fact that the dimension data usually consists of only two elements (for every two-dimensional matrix), the communication and specification overhead incurred by using an additional port is prohibitive. Hence, the second approach of treating the dimension information as part of the matrix data, is preferable from the tool design perspective. In order to fit into one of the standard stream paradigms of Taskmaster, the dimensions which are integer values, are forcibly type cast to conform to the data type of the matrix elements.

To conclude, this section has analyzed the specific issues involved in the prototype implementation of two different applications. The data flow command shell for UNIX is, as expected, perfectly suited to Taskmaster's problem solving philosophy. The LINPACK application domain, on the other hand, provides highly reusable tools but with limited composability. This seems to be more a result of the functionality of the tools rather than any of the factors involved in the "conversion". The domain of mathematical software does seem to be amenable to easy incorporation into the Taskmaster environment. In light of the experience gained from these prototype implementations, the next section will discuss other potential application domains suitable for incorporation into the Taskmaster environment.

5.3 Potential Application Domains

Based on the experience gained from the two prototype implementations discussed in the previous section, this section will assess the suitability of several other potential application domains. The

application domains under consideration for incorporation into the Taskmaster environment are individually assessed in the following subsections.

5.3.1 Mathematical Software

Mathematical software appears to a highly suitable domain for "conversion" to Taskmaster if the LINPACK routines can be considered to be typifying this domain. Mathematical software is probably one of the few domains where reusability has been enormously successful. This success is primarily because of the generality and wide applicability of the operations involved. The data types involved are easily mapped into one of the standard stream paradigms of Taskmaster except for one notable exception, i.e. complex numbers. Augmenting the standard streams of Taskmaster to support complex numbers should be a straightforward procedure. Specific instances of mathematical software under consideration as applications for Taskmaster are two locally developed and frequently used systems: the Linear Algebra and Systems (LAS) [BING87] and the General Image Processing System (GIPSY) [KRUS81,GARL86].

5.3.2 Algorithm Animation

Algorithm animation is another promising application domain for Taskmaster, given the visual and intuitive nature of its execution monitoring facility. Algorithms for vectorization, multi-processor data assignment and partitioning, pipelining, routing in interconnection networks, etc. can be visually represented and examined. The vectorized matrix multiplication scheme of Figure 1 is an example of this application.

5.3.3 Hardware Description Languages

Hardware description languages like ISP [BELL79] and CASL [MAXE79] are used to model hardware components to support Computer Aided Design (CAD) of highly sophisticated hardware. ISP is a language for the specification, evaluation and verification of computer instruction sets. An instruction is described by a condition and an action sequence. CASL is a Computer Architecture Specification Language designed for hardware description at the register-transfer level. Languages like ISP and CASL are textual-based languages used to describe objects that can be more intuitively represented in a visual form. IDES (Integrated DEsign System) [FOUL85] is a comprehensive environment for the computer-aided design of concurrent hardware systems. IDES employs an ALGOL-like high-level language called 'G' to provide algorithmic descriptions that are translated to code in a hardware description language called HDL. Taskmaster, with a little augmentation, can support this domain of design and modeling of complex hardware. Visual specification of these descriptions can provide a more elegant and intuitive design environment. It is interesting to note that IDES uses a hardware component database functionally very similar to Taskmaster's Tools Database.

5.3.4 Computer Simulation

The Taskmaster environment, given its inherent support for functional and visual abstraction, is well-suited for the simulation and prototyping of embedded systems software. Software prototyping involves the investigation of the properties of a software system in an environment different from that of the target hardware [COOK87]. Although the Taskmaster system does not provide an in-built clocking mechanism, it does allow application developers to implement their own synchronization schemes. The actual details of how this synchronization can be accomplished is treated in the next chapter.

5.3.5 Process Control and Other Real-Time Applications

Process control is an example of a real-time application. Real-time applications involve severe timing constraints. As distinct from simulation, actual control of embedded system processes involves direct interaction with hardware signals. Thus, one can install hardware device drivers as "tools" in the Taskmaster environment in order to directly interact with hardware signals. Again, the purely deterministic data-driven model will need to be augmented to include time-based primitives in order to meet real-time requirements. The Stream Machine [BART85] paradigm discussed in Section 2.2 can be used as a starting point in this regard.

Chapter 6

Conclusion

This thesis presents Taskmaster, an interactive, graphical environment for high-level task specification, execution and monitoring. The primary goal of this research is to develop a high-level problem solving environment that

- capitalizes on the generic strength of Visual Programming, i.e. its high bandwidth of human-computer interaction,
- embraces the Tool Composition programming paradigm in order to increase programming productivity,
- exploits the advantages offered by the data flow model of computation,
- supports the top-down and the bottom-up approaches to program design, and
- provides guidance to the user in constructing the program.

The development of Taskmaster is the culmination of the research towards achieving this goal. The remainder of this chapter reflects on the results of this research. In particular, this chapter

- highlights the major research contributions of this work,

- discusses the limitations of the current implementation and suggests possible enhancements to overcome these limitations, and
- explores future research possibilities.

6.1 Contributions of This Thesis

6.1.1 Functional and Visual Abstraction of Tool-Composites

Functional and visual abstraction of a sub-network of tools into a composite *pseudotool* is a novel concept introduced by this thesis. The importance of this concept stems from its

- enhanced support for *software reusability* by providing the necessary abstraction mechanisms to store and reuse sub-networks of tools,
- support for *layered design* of embedded systems either by the top-down or the bottom-up approach, and
- support for *customization* of the environment with user-defined building blocks.

In order to support this *pseudotool* abstraction, a significant amount of research was devoted to characterizing the abstraction of a sub-network of tools. A primary contribution of this research is the development of the three abstraction models for cutsets, discussed in Chapter 3. These models provide a framework for characterizing the abstraction of cutsets in terms of the amount of external context information that is preserved.

6.1.2 Structured Data Flow and Tool Integration

The second major contribution of this thesis is the identification of the issues involved in supporting inter-tool communication of structured data. Two different approaches to structuring have been identified and the text stream paradigm of UNIX has been generalized to encompass the notion of *structured streams*. Easy integrability of tools is achieved by the use of independent communicating objects (post-processor and pre-processor) to implement the inter-tool communication thereby decoupling the tools from their communication interfaces.

6.1.3 Application Development and Tool Design

The third major contribution of this thesis is in the area of application development. Several criteria determining the suitability of an application domain to Taskmaster's problem solving philosophy have been identified, including reusability, composability, time dependency and the structure of the data. This thesis also provides some intuitive insights on tool design based on the experience gained from the prototype implementation of two example applications.

6.2 Limitations of Taskmaster and Suggested Enhancements

Although the current implementation of Taskmaster fulfills the original objectives of this research, it is not without its limitations. This section examines each of these limitations and, in most cases, suggests possible enhancements to overcome the limitation:

1. The Network Editor lacks support for making direct arc connections with *pseudotool* ports. This requires the user to explode a *pseudotool* until all its external ports are available for connection. A possible enhancement to address this problem is the addition of a *pseudo-arc* concept into the current model and augmenting the editor operations to support this abstraction.

2. In the current implementation, the Network Editor does not provide the capability to modify existing *pseudotools* even to incorporate minor changes. An *edit tool-composite* facility will provide support for "partial" reusability analogous to the inheritance mechanism of object-oriented programming.

3. Currently, the Network Editor flags all nodes with unconnected ports in a task network chosen for execution. This verification is intended to provide the user with an opportunity to correct any oversights. In this verification process, however, the user is forced to re-examine all the ports that were *intentionally* left unconnected ("don't care" outputs). This deficiency can be easily remedied by providing a pre-defined "don't care" output sink (null device) to which the user can explicitly connect any such inconsequential outputs.

4. Taskmaster, in its current implementation, attempts no optimization of user-defined task networks. Data flow dependency analysis techniques can be used to optimize the network for maximal parallelism.

5. Although the Taskmaster environment uses a computing organization based on data flow, it does not provide any of the control primitives used in data flow graphs (e.g. decider, T-gate, F-gate, etc. [HWAN84]). These primitives can be easily incorporated into the Taskmaster environment by defining simple tools that emulate their function.

6. The structured data flow approach used by Taskmaster currently does not provide support for embedded (user-defined or otherwise) structures. Taskmaster, however, provides the requisite framework and only the encoding scheme needs to be enhanced to handle embedded struc-

tures. Another alternative is to implement the object-oriented approach of Section 4.2.2.1 using files to simulate inter-process memory sharing.

7. At the present time, Taskmaster does not provide any guidance to the user in the retrieval of metafiles (pseudotools and complete networks saved to disk). Addition of an indexing scheme for better storage and retrieval of metafiles and a metafile browser (similar to the Smalltalk system browser) will provide more support for reuse.

8. In the current implementation of the Taskmaster environment, the user work area on the remote host cannot be accessed from within the environment. To remedy this situation, a system window which provides access to the operating system facilities on the remote host can be incorporated using a "pseudo-login" mechanism.

6.3 Future Research Possibilities

The previous section addresses the limitations of Taskmaster and suggests enhancements to overcome these limitations. This section will address the broader issue of future research. The remainder of this section focuses on some of the major topics for future research:

1. Extending the deterministic data flow model to allow time-based non-deterministic operations is an area which needs to be addressed in depth. Time-based operations will necessitate a clocking mechanism for synchronization. Distributed synchronization using individual local clocks or centralized synchronization with a single global clock are both viable options [LAMP78]. As mentioned in the previous chapter, a synchronization mechanism is a prerequisite for the application of Taskmaster to inherently time-based areas like process control, network monitoring and computer simulation.

2. The support for iterative processing is another topic which deserves scrutiny. Using the data flow control primitives mentioned in the previous section offers one method for controlling iterative processing. These control primitives, however, involve low-level operations. A high-level framework to support iterative processing in a data-driven environment [SKUB86] needs to be provided.

3. Dynamic reconfiguration of task networks during the execution phase is another area for future research. A dynamic reconfiguration facility will be a powerful addition to the environment in supporting reliability and fault-tolerance.

4. An object-oriented tool description language offers another promising area for further research. The domains of visual programming, object-oriented programming, tool composition and data flow have much to offer in supplementing as well as complementing each other and constitute a rich and almost unexplored area for collaborative research.

5. Evaluating the effectiveness of Taskmaster's user interface is another area for future research. The usage statistics collection facility provided by Taskmaster can be used to analyze the usage based on various criteria.

List of References

[ARTH87] J. D. Arthur and D. E. Comer, "An Interactive Environment for Tool Selection, Specification, and Composition," *Interactive Journal of Man-Machine Studies,* Vol. 26, No. 5, May 1987, pp 581-596.

[ARTH86] J. D. Arthur, R. W. Ehrich, and K. C. Mulheren, "A Network Specification and Execution Environment," *Software and Hardware Applications of Microcomputers,* The International Society for Mini and Microcomputers, January 1985, pp. 132-136.

[ARTH85] J. D. Arthur, "Partitioned Frame Networks for Multi-Level, Menu-Based Interaction," *Proc. of the Fourth Annual International Conference on Computers and Communications,* IEEE, Phoenix, Az., March 1985, pp. 34-39.

[ARTH84a] J. D. Arthur and D. Comer, "An Interactive Environment Based on Tool Composition," *The IEEE Computer Society's Eighth International Computer Software and Applications Conference,* Chicago, IL, November 1984, pp. 28-36.

[ARTH84b] J. D. Arthur and D. A. Reed, "Prometheus: An Interactive Environment for the Development and Execution of Functional Programs," *The IEEE COMPSAC 84,* Chicago, IL, November 1984, pp. 44-56.

[ARTH83] J. D. Arthur, "An Interactive Environment for Tool Selection, Specification, and Composition," Ph.D. Dissertation, Purdue University, Indiana, 1983.

[BACK78] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM,* Vol. 21, No. 8, August 1978, pp. 613-641.

[BART85] P. Barth et al., "The Stream Machine: A Data Flow Architecture for Real-Time Applications," *The IEEE Computer Society's Eighth International Conference on Software Engineering,* London, UK, August 1985, pp. 103-111.

[BELL79] Bell and Newell, "The ISPS Computer Description Language," *Technical Report CMU-CS-79-137,* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1979.

[BING87] S. Bingulac, "Linear ALgebra and Systems: Interactive Coversational Language Used for Computer-Aided Design of Control Systems," LAS User Documentation, Virginia Tech, Blacksburg, Virginia, 1987.

[BOEH84] B. W. Boehm et al., "A Software Development Environment for Improving Productivity," *IEEE Computer,* Vol. 17, No. 6, June 1984, pp. 30-42.

[BROW86] G. P. Brown, "Visual Programming-in-the-large: A Practical Concept?," *IEEE COMPSAC 86,* October 1986, pp. 406.

[COOK87] R. P. Cook and R. J. Auletta, "StarLite, A Visual Simulation Package for Software Prototyping," *Proc. ACM Sigsoft/Sigplan Software Engg. Symposium on Practical Software Development Environments,* December 1986, printed as *Sigplan Notices,* Vol. 22, No. 1, January 1987, pp. 102-110.

[COXB86] B. J. Cox, *Object Oriented Programming: An Evolutionary Approach,* Addison-Wesley, 1986.

[DELI84] N. M. Delisle, D. E. Menicosy and M. D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proc. ACM Sigsoft/Sigplan Software Engg. Symp. on Practical Software Development Environments,* April 1984, printed as *Sigplan Notices,* Vol. 19, No. 5, May 1984, pp. 49-56.

[DEMA84] T. DeMarco and A. Soceneantu, "SYNCRO: A Dataflow Command Shell for the Lilith/Modula Computer," *IEEE Transactions on Software Engineering,* 1984, pp. 207-213.

[DENN81] J. B. Dennis et al., "VIM: An Experimental Multi-user System Supporting Functional Programming," *Proceedings of 1981 Workshop on High-Level Computer Architecture,* 1981.

[DERE76] F. DeRemer and H. H. Kron, "Programming-in-the-Large vs. Programming-in-the-Small," *IEEE Transactions on Software Engineering,* Vol. 2, No. 2, June 1976, pp.80-86.

[DOCK86] T. W. G. Docker and G. Tate, "Executable Data Flow Diagrams," *Software Engineering 86,* (Ed. D. Barnes and P. Brown), Peter Peregrinus Ltd., 1986, pp. 352-370.

[DONG79] J. J. Dongarra et al., *Linpack User's Guide,* SIAM, Philadelphia, Pennsylvania, 1979.

[EDEL86] M. Edel, "The Tinkertoy Graphical Programming Environment," *The IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference,* October 1986, pp. 466-472.

[EHRI86] R. Ehrich and R. Williges, *Human-Computer Dialogue Design,* Vol. 2, Elsevier Amsterdam, 1986.

[FOUL85] P. W. Foulk, *CAD of Concurrent Computers,* Research Studies Press Ltd., John Wiley & Sons Inc., 1985.

[GARL86] E. Garland and R. W. Ehrich, "A GIPSY Primer," Spatial Data Analysis Laboratory, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, October 1986.

[GILL86] W. D. Gill and T. D. Kimura, "Parsing Two-Dimensional Languages," *IEEE COMPSAC,* October 1986, pp. 472-477.

[GLIN86] E. P. Glinert, "An Interactive Graphical Programming Environments: Six Open Problems and a Posiible Partial Solution," *IEEE COMPSAC 86,* October 1986, pp. 408-410.

[GLIN84] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment," *IEEE Computer,* Vol. 17 No. 11, November 1984, pp. 7-25.

[GML85] "Introduction to GML," Document No. SC01, User Services Department, Virginia Tech Computing Center, Blcaksburg, Virginia, July 1985.

[GOOD81] J. W. Goodwin, "Why Programming Environments Need Dynamic Data Types," *IEEE Transactions on Software Engineering,* Vol. 7, No. 5, September 1981, pp. 451-457.

[GOOS83] G. G. Goos et al., "DIANA: An Intermediate Language for Ada," *Vol. 161 of Lecture Notes in Computer Science,* Springer-Verlag, 1983.

[GRAF85] R. B. Grafton and T. Ichigawa, "Visual Programming," *IEEE Computer,* August 1985, pp. 6-9.

[HABE82] A. N. Habermann and D. Notkin, "The Gandalf Software Development Environment," Carnegie-Mellon University Technical Report, Computer Science Department, January 1982.

[HABE79] A. N. Habermann, "Tools for Software Systems Construction," *Software Development Tools,,* Springer-Verlag, 1979, pp. 10-21.

[HALL86] P. A. V. Hall, "Reusable and Reconfigurable Software Using C," *Software Engineering 86,* (Ed. D. Barnes and P. Brown), Peter Peregrinus Ltd., 1986, pp. 352-370.

[HORO78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,,* Computer Science Press, 1978.

[HWAN84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill 1984.

[JACO85] R. J. K. Jacob, "A State Transition Diagram Language for Visual Programming," *IEEE Computer,* Vol. 18, No. 8, August 1985, pp. 51-59.

[KAPL86] S. M. Kaplan et al., "An Architecture for Tool Integration," *Advanced Programming Environments,* Proceedings of an International Workshop, Trondheim, Norway, June 1986, pp. 290-314.

[KERN81] B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," *IEEE Computer,* Vol. 14, No. 4, April 1981, pp. 12-24.

[KRIE86] M. Krieger and R. Joannis, "Process Activity Diagrams: A Tool for the Specification and Design of Multiple Microprocessor Systems," *Software and Hardware Applications of Microcomputers,* The International Society for Mini and Microcomputers, January 1985, pp. 157-161.

[KRUS81] S. Krusemark, "GIPSY Applications Programmer Guide," SDA81-7, Spatial Data Analysis Laboratory, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, December 1981 (Rev. February 1983).

[LAMB83] D. A. Lamb, "Sharing Intermediate Representations: The Interface Description Language," Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, May 1983.

[LAMP78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM,* Vol. 21, No. 7, July 1978, pp.558-565.

[LOND85] R. L. London and R. A. Duisberg, "Animating Programs Using Smalltalk," *IEEE Computer,* Vol. 18, No. 8, August 1985, pp. 61-71.

[LUCK84] D. C. Luckham and F. W. Von Henke, "An Overview of Anna, a Specification language for Ada," *Proceedings of the IEEE Computer Society's 1984 Conference on Ada Applications and Environments,* St.Paul, Minnesota, October 1984, pp. 116-127.

[MAXE79] G. F. Maxey and E. I. Organick, "CASL - A Language for Automating the Implementation of Computer Architecture," *Proceedings of the Fourth International Symposium on Hardware Description Languages,* Palo Alto, California, 1979, pp. 102-108.

[MCIL78] M. D. McIlroy et al., *Bell System technical Journal,* Vol. 6, 1978, pp. 1902-1904.

[MELA85] B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation," *IEEE Computer,* Vol. 18, No. 8, August 1985, pp. 87-94.

[MITC79] J. G. Mitchell et al., "Mesa Language Manual, Version 5.0," *Technical Report CSL-79-3,* Xerox Palo Alto Research Center, April 1979.

[MONK84] A. Monk, *Fundamentals of Human-Computer Interaction,* Academic Press, Inc., 1984.

[MORI85] M. Moriconi and D. F. Hare, "Visualizing Program Design Through PegaSys," *IEEE Computer,* Vol. 18, No. 8, August 1985, pp. 72-85.

[MULH86] K. C. Mulheren, "An Interactive, Visual-based Environment for Task Specification," M.S. Project Report, August 1986.

[MUNS85] W. Munsil, "The Language Translation Task: Toward Reusable Components," *Proc. of the Fourth Annual International Conference on Computers and Communications,* IEEE, Phoenix, Az., March 1985, pp. 46-52.

[NEST82] J. R. Nestor, W. A. Wulf and D. A. Lamb, "IDL Formal Description, Draft Version 2.0," *Technical Report,* Computer Science Dept., Carnegie-Mellon University, June 1982.

[NEWM79] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill Book Company, 1979.

[NOTK85] D. Notkin, "The GANDALF Project," *Journal of Systems and Software,* Vol. 5, No. 2, May 1985, pp. 91-106.

[OSTE83] L. J. Osterweil and W. R. Cowell, "The Toolpack/IST Programming Environment," *IEEE SOFTFAIR,* July 1983, pp. 326-333.

[OSTE82] L. J. Osterweil, "Toolpack - An Experimental Software Development Research Project," *Proceedings of the IEEE Computer Society's 6th International Conference on Software Engineering,* September 1982, pp. 166-175.

[POWE83] M. L. Powell and M. A. Linton, "Visual Abstraction in an Interactive Programming Environment," *Proceedings of SIGPLAN Symposium on Programming Language Issues in Software Systems 83,* printed as *SIGPLAN Notices,* Vol. 18, No. 6, June 1983, pp. 14-21.

[PRAT85] T. W. Pratt, "PISCES: An Environment for Parallel Scientific Computation," *IEEE Software,* July 1985, pp. 7-20.

[PURT85a] J. M. Purtilo, "On Specifying an Environment," *IEEE COMPSAC 85,* October 1985, pp. 456-463.

[PURT85] J. M. Purtilo, "Polylith: An Environment to Support Management of Tool Interfaces," *Proceedings of the ACM SIGPLAN symposium on language issues in Programming Environments,* ACM SIGPLAN Notices, Vol. 20, No. 7, July 1985.

[RABI59] M. O. Rabin and D. Scott, "Finite Automata and Their Decision Problems," *IBM Journal of Research and Development,* Vol. 3, 1959, pp. 114-125.

[RAED85] G. Raeder, "A Survey of Current Graphical Programming Techniques," *IEEE Computer,* Vol. 18, No. 8, August 1985, pp. 11-25.

[RAED84] G. Raeder, "Programming in Pictures," Ph.D. Dissertation, University of Southern California, Los Angeles, California, November 1984.

[RAMA86] C. V. Ramamurthy et al., "Support for Reusability in GENESIS," *IEEE COMPSAC 86,* October 1986, pp. 299-305.

[RAMA85] C. V. Ramamurthy et al., "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software," *IEEE COMPSAC 85,* October 1985, pp. 472-479.

[REIS86] S. P. Reiss, "GARDEN Tools: Support for Graphical Programming," *Advanced Programming Environments,* Proceedings of an International Workshop, Trondheim, Norway, June 1986, pp. 290-314.

[REIS84] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proc. ACM Sigsoft/Sigplan Software Engg. Symp. on Practical Software Development Environments,* April 1984, printed as *Sigplan Notices,* Vol. 19, No. 5., May 1984, pp. 30-41.

[SCHE86] R. W. Scheifler and J. Gettys, "The X Windows System," *ACM Transactions on Graphics #63, Special Issue on User Interface Software,* October 1986.

[SCHU83] J. Schultis, "A Functional Shell," *Proceedings of the SIGPLAN 1983 Symposium on Programming Language Issues in Software Systems,* SIGPLAN Notices, Vol. 18, No. 6, June 1983, pp. 202-211.

[SHAW86] M. Shaw, "Beyond Programming-in-the-Large: The Next Challenges for Software Engineering," *Advanced Programming Environments,* Proceedings of an International Workshop, Trondheim, Norway, June 1986, pp. 519-536.

[SKUB86] J. J. Skubich, "FLUIDE: A Multiple Data and Control Flow Computing Organization," *IEEE COMPSAC 86,* October 1986, pp. 221-227.

[SNOD86] R. Snodgrass and K. Shannon, "Supporting Flexible and Efficient Tool Integration," *Advanced Programming Environments,* Proceedings of an International Workshop, Trondheim, Norway, June 1986, pp. 290-314.

[SNYD84] L. Snyder, "Parallel Programming and the Poker Programming Environment," *IEEE Computer,* July 1984, pp. 27-36.

[STEV82] W. Stevens, "How data flow can improve application development productivity," *IBM Systems Journal,* Vol. 21, No. 2, 1982.

[STRO86] R. Strom and S. Yemini, "NIL: A Very High Level Programming Language and Environment," *Application Development Systems,* (Ed. T. L. Kunii), Springer-Verlag, 1986, pp. 40-52.

[TEIT81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer," *Communications of the ACM,* Vol. 24, No. 9, September 1981, pp. 563-573.

[TEIW84] W. Teitelman, "A Tour Through Cedar," *IEEE Software,* Vol. 1, No. 2, April 1984, pp. 44-73.

[TEIW81] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *IEEE Computer,* Vol. 14, No. 4, April 1981, pp. 25-33.

[TESL81] L. Tesler, "The Smalltalk Environment," *Byte,* Vol. 6, No. 8, August 1981, pp. 90-146.

[TICH79] W. F. Tichy, "Software Development Control Based on Module Interconnection," *Proceedings of the 4th International Conference on Software Engineering,* Munich, West Germany, September 1979, pp. 29-41.

[UNET85] *The DECnet-Ultrix User's and Programmer's Guide,* Digital Equipment Corporation, July 1985.

[VNET86] *The VAX/VMS Networking Manual,* Digital Equipment Corporation, April 1986.

[WASS87] A. I. Wasserman, "A Graphical, Extensible Integrated Environment for Software Development," *Proc. ACM Sigsoft/Sigplan Software Engg. Symposium on Practical Software Development Environments,* December 1986, printed as *Sigplan Notices,* Vol. 22, No. 1, January 1987, pp. 131-142.

[WASS85] A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Transactions on Software Engineering,* Vol. 11, No. 8, August 1985, pp. 699-713.

[WIRR84] R. Wirth, "Software Reusability," *IEEE COMPSAC 84,* October 1984, pp. 470.

[WIRT81] N. Wirth, "Lilith: A Personal Computer for the Software Engineer," *Proceedings of the Fifth International Conference on Software Engineering,* Long Beach, California, March 1981.

[WOLF84] A. L. Wolf et al., "An Ada Experiment for Programming-in-the-Large," *Proceedings of the IEEE Computer Society's 1984 Conference on Ada Applications and Environments,* St.Paul, Minnesota, October 1984.

Vita

Kumbakonam S. Raghu was born in Madras, India on April the Nineteenth, Nineteen Hundred and Sixty Two. He received the degree of Bachelor of Technology in Electrical Engineering from the Indian Institute of Technology, Madras, in July 1984, and joined the Department of Computer Science, Virginia Polytechnic Institute and State University as a graduate student in January 1985. Since March 1987, Mr. Raghu has been working as a Software Engineer with Online Computer Systems, Inc., Germantown, Maryland.