Abstraction Mechanisms in Support of Top-Down and Bottom-Up Task Specification

K.S. Ragu and James D. Arthur

TR 88-15

Abstraction Mechanisms in Support of Top-Down and Bottom-Up Task Specification

K.S. Raghu and *James D. Arthur*

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

ABSTRACT

Abstraction is a powerful mechanism for describing objects and relationships from multiple, yet consistent, perspectives. When properly applied to interface design, abstraction mechanisms can provide the interaction *flexibility* and *simplicity* so desperately needed and demanded by today's diverse user community. Fundamental to achieving such goals has been the integration of visual programming techniques with a unique blend of abstraction mechanisms to support user interaction and task specification. The research presented in this paper describes crucial abstraction mechanisms employed within the Taskmaster environment to support top-down and bottom-up task specification. In particular, this paper (a) provides an overview of the Taskmaster environment, (b) describes top-down specification based on multi-level, menu-driven interaction and (c) describes bottom-up specification based on cutset identification and psuedo-tool concepts.

CR Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Interactive Programming Environments; H.1.2 [**User/Machine Interaction**]: Human information Processing

General Terms: Interface Abstractions, Top-Down and Bottom-up Task Specification, Partitioned Menu Networks, Tool Composites

Additional Keywords: Nodes, Arcs, Communication Paths, Cutsets

Abstraction Mechanisms in Support of Top-Down and Bottom-Up Task Specification

K.S. Raghu and *James D. Arthur*

Department of Computer Science

Virginia Tech

Blacksburg, VA 24061

1.0 Introduction

Abstraction is a powerful mechanism for describing objects and relationships from multiple, yet consistent, perspectives. When properly applied to interface design, abstraction mechanisms can provide the interaction *flexibility* and *simplicity* so desperately needed and demanded by today's diverse user community. In particular, user interaction and task specification need to resemble more closely the *mental* process of problem solving, allowing the user to concentrate on the problem solution rather than on programming language syntax or semantics. The research described in this paper reflects an effort to meet that challenge, not only in simplifying user/machine interaction but also in increasing the *bandwidth* of interaction between the computer and the user. Fundamental to achieving such goals has been the integration of visual programming techniques with a unique blend of abstraction mechanisms to support user interaction and task specification [REIS86].

The research vehicle for exploring specification abstractions has been the successive development of several prototype environments, culminating in the synthesis of *Taskmaster* - an interactive, graphical environment for task specification, execution, and monitoring. Although the

Taskmaster environment touts several novel features,e.g. visual programming, tool composition and structured inter-tool data flow computing, it derives its expressive power and interactive capabilities from the use of *complementary* abstraction mechanisms. In particular, concepts underlying functional abstractions, supported through visually-oriented icons and primitives, provide an integrated *top-down* and *bottom-up* task specification interface. That is, when specifying a task, a user has the option to:

- begin with a high-level task specification and, through a top-down process, successively refine that specification until lower level constituent tasks are bound to primitive operations supported by an underlying set of tools,
- initiate a bottom-up synthesis of a high-level task specification through successive abstractions of fully specified lower level subtasks into higher level subtasks until the high-level task is specified, or
- specify an initial task overview based on a sequence of partially ordered operations, use top-down specification to bind those operations to corresponding functional tools, and then use bottom-up specification to abstract (or "collapse") the overview into conceptually simpler forms representing higher level operations.

As outlined above, Taskmaster exploits several abstraction mechanisms in providing a flexible, user-directed approach to task specification. The focus of this paper is to present functional models, operational characteristics, and implementation considerations underlying the selection and integration of those mechanisms. Because the Taskmaster environment plays such a crucial role in our discussion, an overview of that system is presented in the next section. Included in the presentation is a description of the environment's major components that visually and textually support user task specification. Section 3 follows and presents a discussion of abstractions used in support of top-down task specification. The discussion includes a description of (a) *partitioned menu networks* in support of *multi-level, menu-based interaction,* and (b) the *expand node*

operation. Section 4 presents several models of *composite tool abstractions* and discusses their applicability to bottom-up user task specification. Included in this discussion is an overview of the *save tool-composite* and *attach tool-composite* operations. Finally, Section 5 provides a summary of the paper and briefly discusses the current status of the Taskmaster environment.

2.0 The Taskmaster Environment: An Overview

The Taskmaster environment has been a product of evolution. Its initial predecessor, OMNI [ARTJ87], was *textually* oriented and supported interactive user task specification based on a "loose" composition of program filters. Taskmaster's immediate predecessor, GETS [ARTJ88], exploited *graphics-based* task specification but, like OMNI, was still restricted to *rigid* specification constraints enforced by menu-based interaction. Learning from our experiences with OMNI and GETS, the Taskmaster environment has been purposely designed to support user task specification from a graphics-oriented perspective and to include abstraction mechanisms to overcome the interaction rigidity inherent to menu-based systems.

The Taskmaster environment 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 in this paper refers to a filter program (*a la* UNIX[1] *sort*) which performs a single operation with minor variations. Each tool can have multiple input and output ports through which it communicates with other tools. To "program" a given task within the Taskmaster environment, one decomposes the task into a partially ordered set of conceptually simple, high-level subtasks (or operations), and then composes a corresponding network of software tools that implement those subtasks. This decomposition/composition process is supported through and depicted as a graphical network in which nodes correspond to subtasks and arcs represent directed data paths between the nodes. The resulting network topology captures the

[1] Unix is a trademark of AT&T.

set and sequence of operations needed to compute a solution to the user task specification. Execution of that network of software tools, provides the problem solution.

2.1 A Task Specification Example

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 one task specification network for this scheme where:

(1) the pre-multiplier matrix A and the post-multiplier matrix B are vectorized by row and column, respectively. This activity is reflected in the overview by nodes bearing the operational names: *Vectorize by Row* and *Vectorize by Column.*

(2) the product matrix elements, C_{ij}, are computed in a *parallel fashion* based on the row and column vectors provided by the vectorize operations. This activity is portrayed in the overview by the node: *Multiply Vectors in Parallel.*

(3) the product matrix elements are composed into an NxL matrix C. This operation is implied by the node labelled: *Compose Product Matrix.*

We note that the network shown in Figure 1 is a task specification *overview* and is intended to exemplify operations at high level of granularity. These operations may or may not directly correspond to primitive functions supported by underlying environment tools.

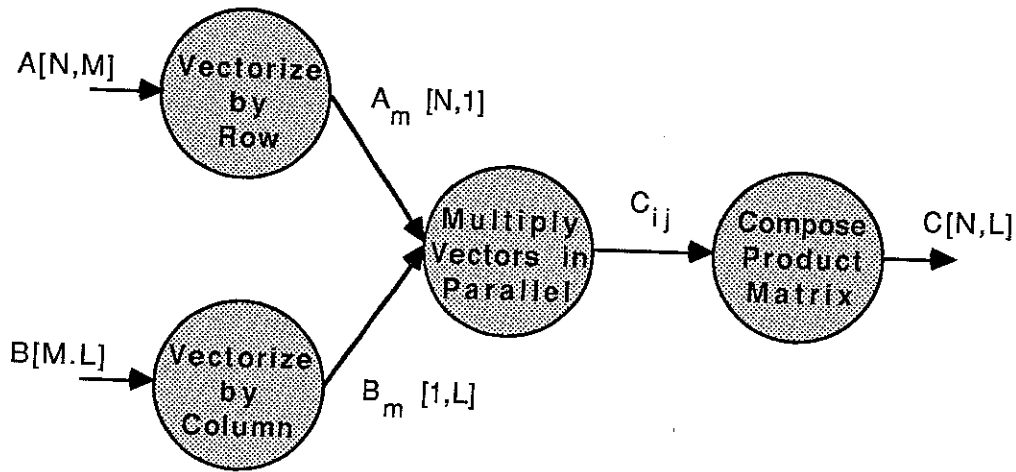

Figure 1

A High Level Specification of Vectorized Matrix Multiplication

Using the task specification overview as a basis, the user individually selects each node and successively refines that node until a primitive tool or predefined subtask is "bound" to it. The final network topology reflecting this refinement process and corresponding to the user's perception of a task "solution" is illustrated in Figure 2. Note, that in addition to refinement, node expansion is also performed on the *Multiply Vectors in Parallel* and the *Compose Product Matrix* nodes.

Though simplistic in nature, the above example does capture the flavor of a typical task specification within the Taskmaster environment. The conceptual simplicity, however, is attributable to the powerful network editing primitives (based on underlying abstraction mechanisms) provided to the user. For example, Taskmaster provides primitives to save *all* or *parts* of the task specified in Figure 2 as a functionally complete abstraction in the tools database. For all practical purposes, the above network can be saved (in its entirety) and later addressed as a *pseudotool* with 2 input ports and one output port. Hence, in a task specification where a 2x2 matrix multiplication operation is needed, the user simply creates a node in the network and

through the node specification process, binds that pseudotool to the node. Clearly, such a process could have been applied to the nodes, *Multiply Vectors in Parallel* and *Compose Product Matrix.*

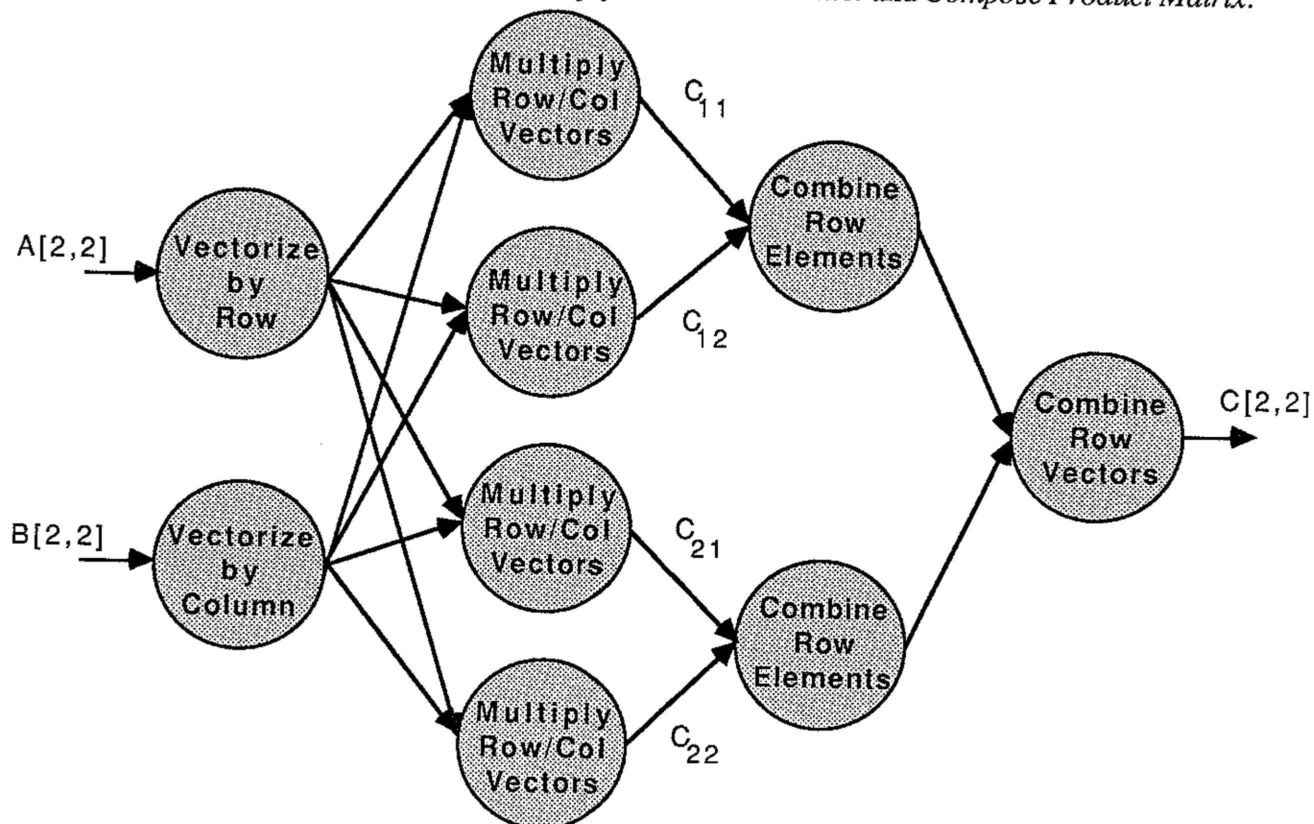

Figure 2

A Fully Refined Task Network for Vectorized Matrix Multiplication

2.2 The Major Components of the Taskmaster Environment

The Taskmaster environment is an integrated user support environment that exploits visual programming concepts, tool composition, and structured data flow. It is composed of three major cooperating components:

- the Network Editor,
- the Network Execution Monitor, and

- the Tools Database.

The Network Editor provides a graphical interface for constructing task networks. It guides the user through the task specification process by supporting top-down and bottom-up interaction formats. Once a task is fully specified, the corresponding network is ready for execution. A network 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.

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 workstation on every update. Although the current configuration has a single local workstation, in the future we visualize a set of local workstations all connected to the host. The overall system configuration is shown in Figure 3.

2.2.1 Taskmaster User Interface: The Network Editor

The Network Editor provides an interactive, graphical interface for constructing task specifications. Programming in the Taskmaster environment consists of transforming a conceptual task into a network whose nodes represent operations (tools) and arcs represent the communication path between the nodes. The Network Editor supports this specification process by providing editor primitives for building *generic* networks and specifying the nodes and the arcs through a menu-based interaction process defined *within* the Tools Database. For clarity, a generic network

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

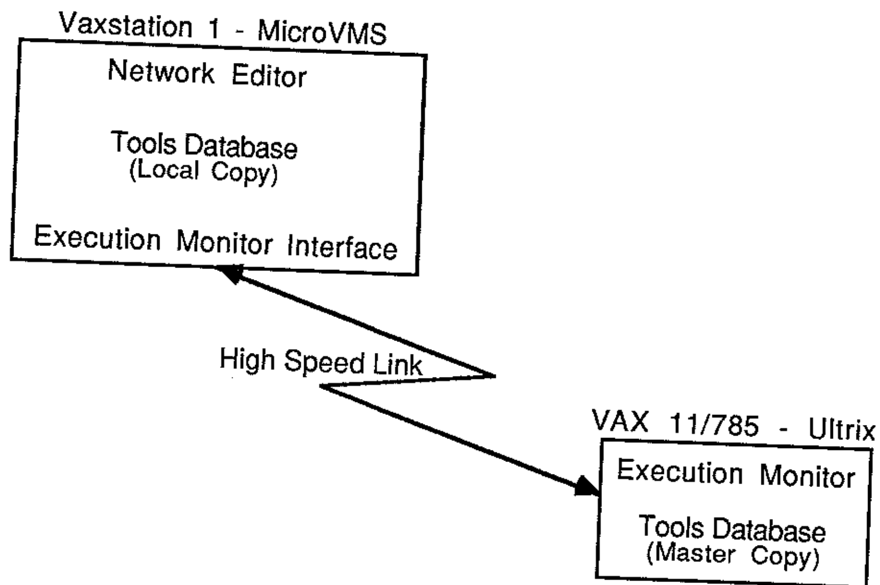

Figure 3

Taskmaster System Configuration

is viewed as a directed graph with *unspecified* nodes and arcs. The specification process associates with each node and arc a corresponding operation and tool port assignment, respectively. Figure 4 illustrates the generic network from which the fully *specified* network in Figure 2 is derived. The Network Editor also permits a user to send a fully specified task network to the Execution Monitor for instantiation.

The Network Editor is primarily menu-driven and makes extensive use of logical windowing and mouse input. Similar to the Dialogue Management System [EHRR86] and PICT [GLIE84], the Network Editor incorporates many of the human engineering principles related to graphical user interface design. For example, ergonomic features include direct manipulation, visual feedback, user error recovery, choice confirmation, default selection, operational and representational consistency, pop-up menus and so forth. The Editor display consists of a large window detailing the topology of the network being edited and auxiliary pop-up windows for displaying

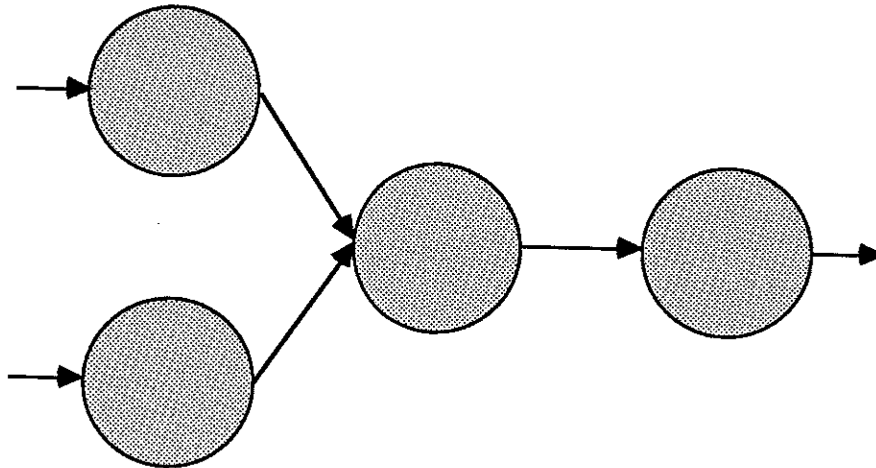

Figure 4

Generic Network for Vectorized Matrix Multiplication

- 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.

The Network Editor supports many editing primitives, the majority of which support either network *construction*, network *specification*, or network *inquiry* operations. The network topology construction operations are used to create and operate on node and arc icons. The unspecified icons serve as visual place-holders for the tools and their interface connections. Thus, an unspecified node created by the *create node* operation has no initial semantic meaning with respect to the task being solved. *Pan* and *zoom* operations provide selective viewing for managing relative complexity of very large networks. The *collapse* operation allows one to abstract a subnetwork performing some high-level operation into a single "super-node". The *explode*

operation reverses the effect of the corresponding collapse operation. The *expand* operation provides a new network topology window in which to define a subnetwork associated with the node being expanded. Sections 3 and 4 presents a more detailed discussion of the expand, collapse and explode operations relative to specifying functional abstractions.

The network specification operations are used to *specify* the node and arc icons. Node icons are specified by attaching to them a fully specified tool or pseudotool 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 both incident nodes are fully specified so that the corresponding tool interfaces are clearly defined.

The network inquiry operations provide characteristic information based on the current specification status of nodes and arcs. The *view node* operation provides a detailed view of the tool attached to a specified node. Taskmaster also provides a textual view of each specified node containing the description of the associated tool, its attributes and its input and output ports. The *view communication path* operation provides a detailed view of the interface between the tools connected by an arc.

Additional operations not discussed above provide for various "backup" and "restore" capabilities. At anytime during an 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. Moreover, the *save tool-composite* and *attach tool-composite* allow the user to identify, save, and retrieve (sub)networks as *functional abstractions.* These two operations, fundamental in defining and reusing pseudotools, are discussed more fully in Section 4. The *undo* operation supports error recovery by undoing the effect of the most recent topology modifying operation.

Finally, after a network is created and completely specified, the *execute network* operation can be used to send it to the Execution Monitor for instantiation. Upon selection of this operation, the Network Editor performs consistency checks and network validation, and then sends an internal representation of the network to the Monitor for instantiation. The internal representation of the network is transferred over the high-speed link to the remote host where the Execution Monitor resides.

2.2.2 Taskmaster Executive - the Execution Monitor

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

- reading the task network representation forwarded by the Network Editor,
- validating the network,
- spawning computational processes based on the network topology, and
- monitoring the network execution.

Before initiating execution of the network, the Monitor first performs a modified breadth-first traversal (BFS) of the network 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 those pipes to the appropriate nodes. The node instantiation is also performed in the BFS traversal order in an attempt to satisfy certain interprocess communication constraints imposed by the operating system. The network execution, however, is independent of the instantiation order because it is based solely on *data flow*. The Execution Monitor also provides for execution monitoring using status messages to indicate the instantiation, execution and termination of each node-associated process.

2.2.3 Taskmaster Knowledge Base - the Tools Database

The third component of the Taskmaster environment is the Tools Database. In the Taskmaster environment, the Tools Database plays a major role in isolating and encapsulating all application-specific information, and presenting it in a generic form to the other two components of the environment. This approach has significant advantages in that:

- defining a new application domain requires only that the Tools Database be redefined accordingly, and
- integrating the new Database into the Taskmaster environment is an operation that is transparent to the rest of the system.

More specifically, 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. 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. Effectively, the specification process can be viewed as a finite state machine driven by the Tools Database menu dialogue "table" [RABI59].

3.0 Abstractions in Support of Top-Down Task Specifications

In specifying a task, the user first defines a generic network reflecting a conceptual ordering of one or more high-level operations. The initial network can be a single node representing the entire

task or a network of nodes representing a task specification overview. From this initial configuration, top-down specification can be employed to refine the network. Top-down task specification is the successive decomposition of a task into lower level subtasks until the lowest level subtasks are directly identifiable with available tools or pseudotools defined in the tools database. In the Taskmaster environment, top-down task specification is supported through two distinct interaction formats, each embracing different decomposition philosophies and employing distinct abstraction mechanisms. The first approach exploits *partitioned* menu networks through a *multi-level,* menu-based interface [ARTJ85]. As with any menu-based system, the specification/decomposition paths are *predefined.* The second approach employs node *expansion* activities and supports *user-directed* specification/decomposition. As discussed below, both approaches embrace top-down decomposition. Multi-level, menu-based interaction, however, assumes that each node being specified represents one operation and will be attached to a single tool. Node expansion, on the other hand, assumes just the opposite.

3.1 Top-Down Specification through Multi-Level, Menu-Based Interaction

Multi-level, menu-based interaction assumes that the node being specified is to be directly bound to a primitive tool defined in the tools database. Furthermore, the interaction process is restricted to predefined sets of refinement paths that correspond to the the underlying menu network hierarchy. The novelty of this approach is not the menu-based interaction *per se,* but the specification sequence induced by a *partitioning* of the underlying menu network. That is, the predefined menu network is partitioned into multiple levels, where each level (or layer) represents a refinement abstraction across the *entire* menu network. As illustrated below, the partitioning induces *interface layers* that permit the user to

- specify a task overview based on predefined high-level operations, and then

- successively refines that overview in a "lock step" fashion through subsequent menu-based interaction.

Although we choose to restrict the discussion of partitioned networks and the multi-level, menu-based interaction to systems that support user task specification, the concepts presented in this section are applicable to most general menu-driven systems and their corresponding application domains.

In specifying a task, the user first constructs a generic network that provides a framework for sequencing and specifying a conceptual set of operations. Through conventional menu-based interaction, for each node in the generic network the user selects the appropriate sequence of menu frame items that *identifies* the high-level operation, *binds* that node to a tool which implements the operation, and then *refines* the execution behavior of that tool. This scenario implies that a selected node is fully specified before another node is considered. For example, suppose that a user has access to a menu-driven, *file transformation* system and wants to retrieve a file, select certain records from a specified file, sort them, and then save them for later processing. First, the user selects the sequence of frame items whose corresponding actions solicits the name of the file to be *retrieved* and infers all associated physical attributes. Next the user chooses a sequence of frame items that indicates the *select-record* operation as well as the criteria for selecting the appropriate records. The user then chooses a sequence of menu items that leads to a description of the *sort* operation and all refinements that specify the desired sort sequence. Finally, frame items are selected that denote the *file-save* operation and that define all characteristics relating to the destination file. Figure 5 illustrates one possible network to accomplish the above specified task. In this fully specified network *cat* is the file retrieval tool, *grep* is the select tool, *sort* is the sort tool and *fileit* is file creation tool.

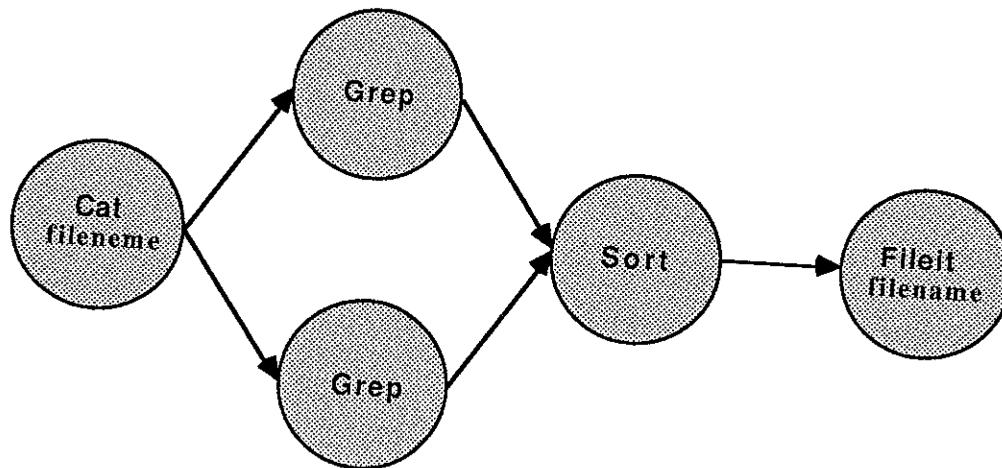

Figure 5

Fully Specified Network to Sort and Save Selected Records

The problem with the specification approach described above is that the user is *forced* to select one node at a time and fully specify its operational details before moving to another node in the network. Such rigidity, enforced by conventional menu networks, tends to obscure the user's *overall* perception of the task solution. For complex network topologies, forcing the user to contend with details before firmly establishing a task overview can have adverse, if not devastating repercussions.

In the Taskmaster environment, however, menu interaction is based on partitioned menu networks that support and encourage partial node specification through defined interface layers. Intuitively, an interface layer can be viewed as a "slice" through the menu network that delimits menu frames possessing a common level of specification. For the above file transformation example, a (simplified) conventional menu network might look similar to the one illustrated in Figure 6. In the Taskmaster file transformation environment, however, Figure 7 shows the same menu network after partitioning. Note that the network defined at Hierarchical Level 1 "terminates" with the selection of a high-level operation. Hence, the user can traverse the menu network in a conventional manner, specify a task overview (without being encumbered by

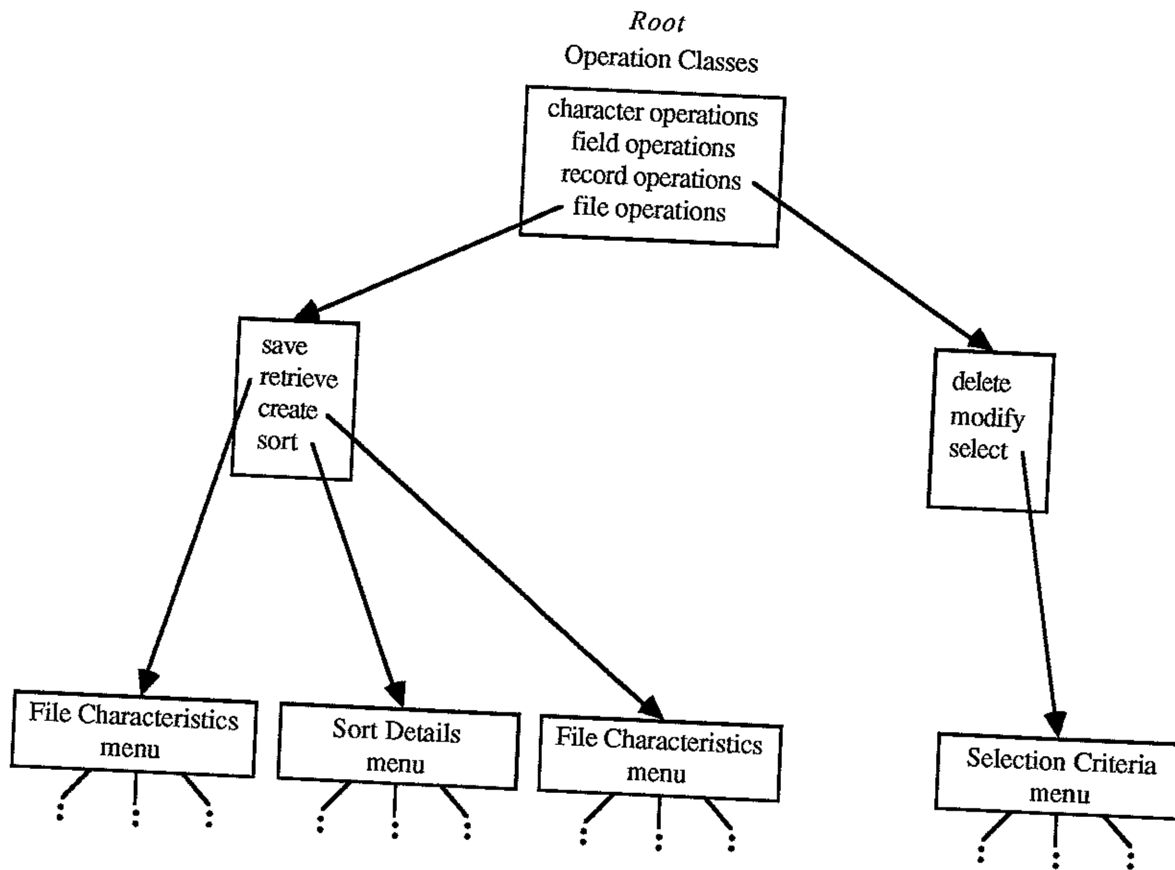

Figure 6

Conventional Menu Network for File Transformation

refinement details), and then *continue* with individual node refinement through interaction guided by the second-level menu networks. That is, the user first constructs a generic network (Figure 8a), specifies an overview by associating high-level operations with each node (Figure 8b), and then refines each high-level operation through continued menu interaction on the second hierarchical level. The final result is a fully specified network identical to the one shown in Figure 5. We emphasize that partitioned networks provide the *capability* for the user to specify an overview through the Taskmaster interface. The final choice remains with the *user* as to whether the specification sequence follows "breadth-first" overview or the conventional "depth-first" orientation.

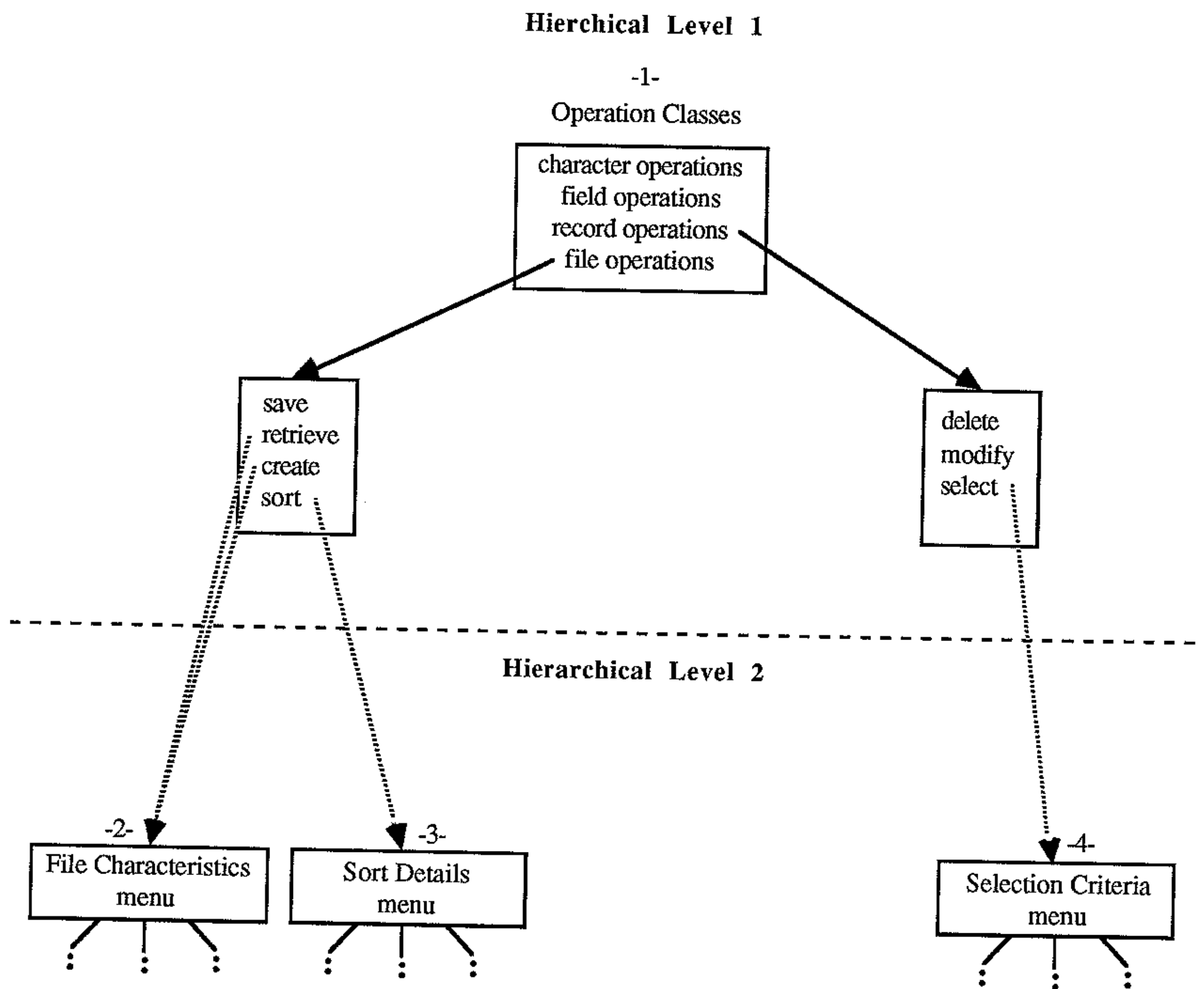

Figure 7

Partitioned Menu Network Supporting File Transformation

Although partitioned menu networks is a powerful mechanism for supporting multi-level, menu-based interaction, the user is still forced to follow a set of paths defined by the menu (sub)networks. The next section describes an alternate top-down specification scenario that allows the *user* to control the specification process.

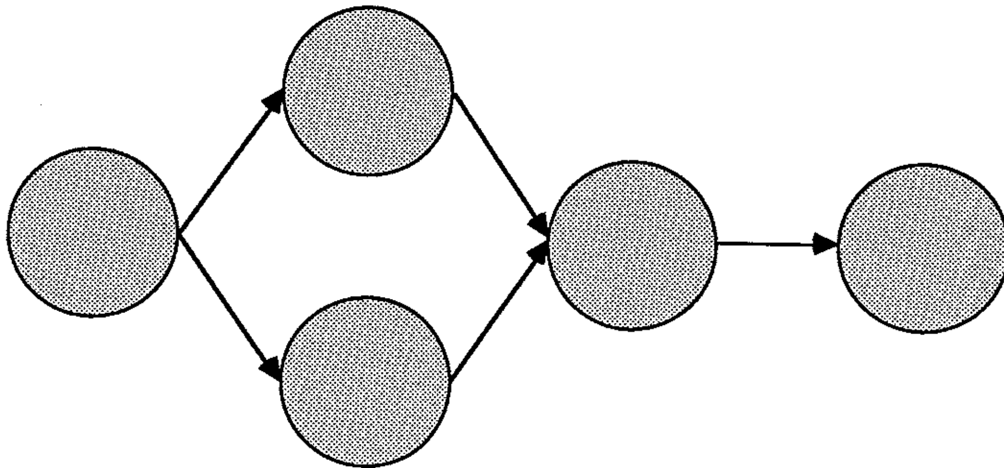

Figure 8a

Generic Network to Sort and Save Selected Records

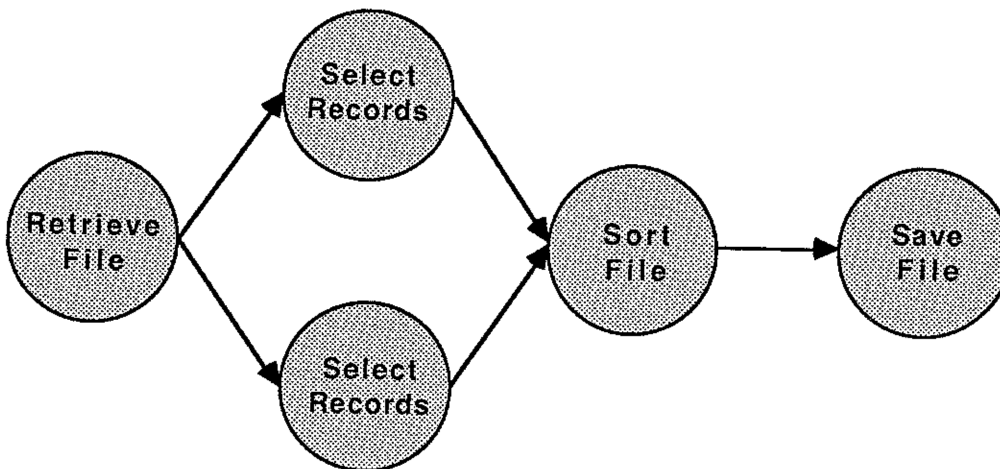

Figure 8b

Overview Network to Sort and Save Selected Records

3.2 Top-Down Specification Through Node Expansion

Given a generic network topology, top-down task specification entails

- the selection of an unspecified node, and
- the binding of that node to a tool or pseudotool through an iterative refinement process.

As described in the previous section, one method for associating a tool with a selected network node is through menu-based interaction. With this approach, however, the user is forced to follow a predefined set of paths leading to the selection of a tool in the database. To introduce more flexibility in the top-down specification process and to encourage user creativity, the Taskmaster environment provides a node *expansion* primitive that allows the *user* to direct the specification process. Intuitively, *expand node* "opens up" a single, unspecified network node and permits the user to fully specify a subnetwork within that node.

The node expansion operation provides the user with a separate "node expansion" window to edit the subnetwork to be integrated. This window is two-thirds the size of the network topology window but logically the same. The expand node operation is recursive to five levels and only limited there for controlling the complexity of the display. Thus, if a node in an expansion window is also selected for expansion, a new node expansion window appears, overlaying the previous one. Only one expansion window is active at any particular instant and the underlying inactive windows are clearly identified as such.

In effect, the user can specify multiple levels of abstraction reflecting his own perception of an operation or task, and have all levels appear as one node at the outermost level. Moreover, because all normal editing operations are supported by the expansion window, the user can choose the method by which each individual subnetwork node is subsequently specified.

4.0 Abstractions in Support of Bottom-Up Task Specification

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. In many instances top-down specification is most natural, e.g. when concentrating on the specification of a single network node. Within the framework of task specification, however, it is often convenient for the user to consider groups of nodes as a single abstraction supporting one high-level operation. Although not specifically stated, the expand node operation provides such a view but from a top-down perspective.

Bottom-up task specification involves successive abstractions of fully specified lower level subtasks into higher level subtasks. At the lowest level of abstraction a network is specified where each node is directly bound to a tool or pseudotool in the database. The specified network usually defines some low-level, yet not quite primitive, function. Continuing in a bottom-up fashion, this network is collapsed into a "super-node" and becomes a single node in a higher level network. This successive abstraction toward higher level functionalities culminates with a fully specified subnetwork that performs a specific a *user defined* function.

The remainder of this section describes how successive abstraction is integrated into the Taskmaster environment. In particular, Section 4.1 provides a discussion of the three models considered in defining the semantic framework associated with successive abstraction. Section 4.2 describes the editing primitives supporting abstraction from a user's perspective.

4.1 Models of Abstraction based on Cutsets

Abstraction is itself an abstract term which has manifold meanings. Abstraction as used here means the hiding of unnecessary detail, or equivalently, showing only those 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 following paragraph defines terms that will be helpful in describing our models of abstraction.

As described earlier, the *tool* is the basic entity in the tool composition paradigm and performs a single 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, or *pseudotool*. 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 9 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.

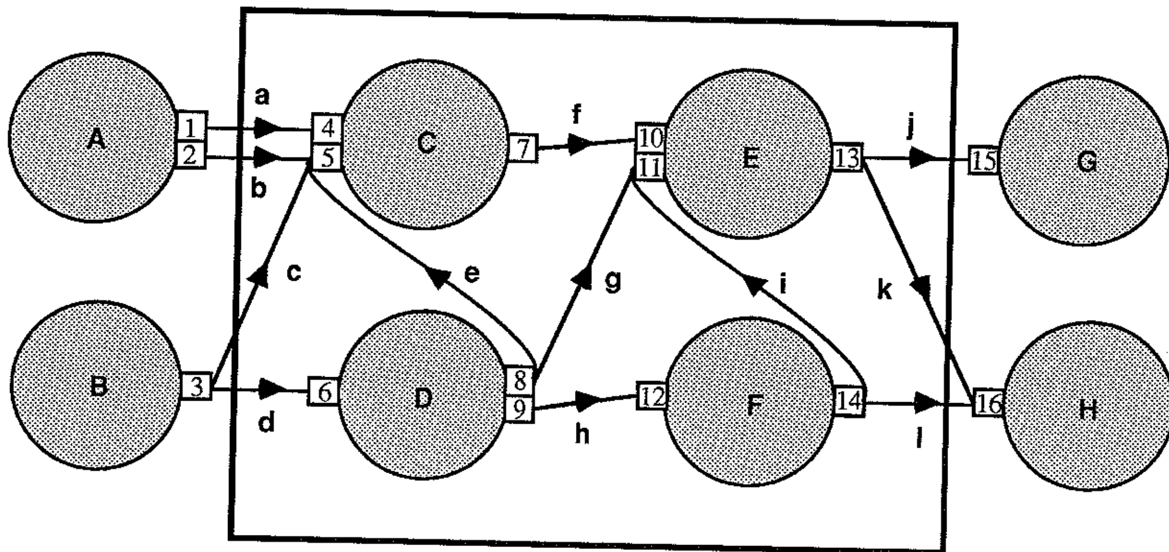

Figure 9

Example Network to Illustrate Cutset and Other Definitions

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 entails 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:

- model $\mathbf{M_1}$ which preserves no external context,
- model $\mathbf{M_2}$ which preserves external, functional context, and
- model $\mathbf{M_3}$ which preserves external, relational context.

Referring to Figure 10, the M_1 abstraction is constructed by removing all intersecting links from the cutset and then using *all* the unconnected ports to form the interface. The M_2 abstraction is constructed by using all unique external ports of the cutset to form the interface. The M_3

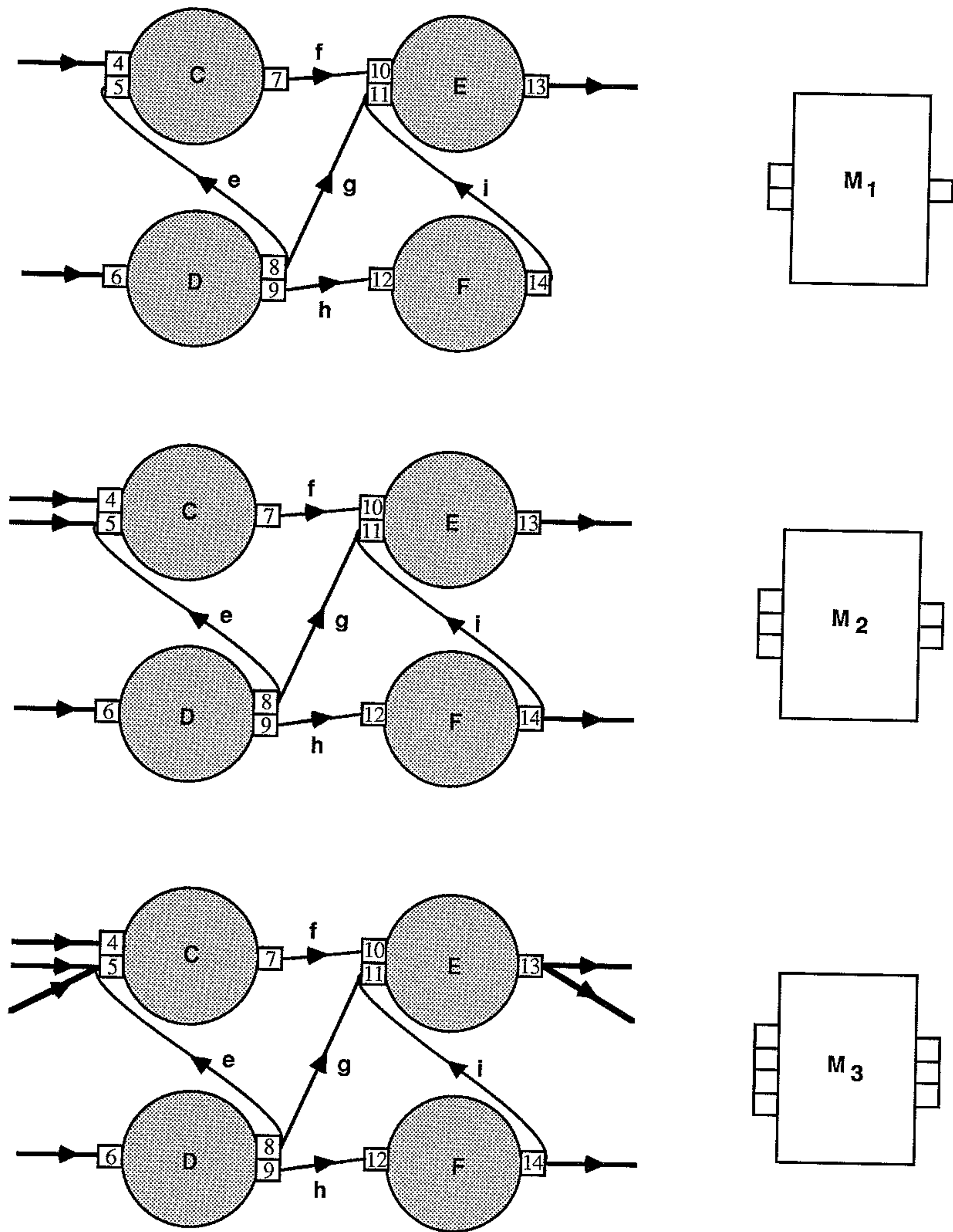

Figure 10

The Three Abstraction Models of the Custet of Figure 9

abstraction is constructed by using a port for every intersecting link of the cutset. As can be seen from the figure, the M_1 abstraction has 2 input ports and 1 output port, the M_2 abstraction has 3 input ports and 2 output ports while the M_3 abstraction has 4 input ports and 3 output ports.

Models of abstraction are used to capture the important aspects of their target environment. Model M_1, 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]. Model M_2 preserves some external context information, i.e. the functional context of the cutset. An attractive feature of M_2 abstraction is that it supports reusability at the functional level. Model M_3 preserves all the external context information and as such is extremely context-specific. It is 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 M_2, 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. Hence all the external context information can be preserved textually if required. The next subsection discusses the actual implementation of the M_2 model of abstraction in Taskmaster.

4.2 Editor Operations Supporting the M_2 Abstraction

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 also support the pseudotool abstraction. In Taskmaster, there are three different ways to integrate a pseudotool into a network:

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

- 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 how of each of these primitives operate, except for expand node which is discussed in Section 3.2.

4.2.1 The Collapse and Explode Operations

The collapse operation performs an in-place M_2 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 11a 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 11b shows the network after the collapse operation is completed. The "super-node" icon with a brick-pattern ring represents the newly defined pseudotool named *Process File.*

During the collapse operation 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

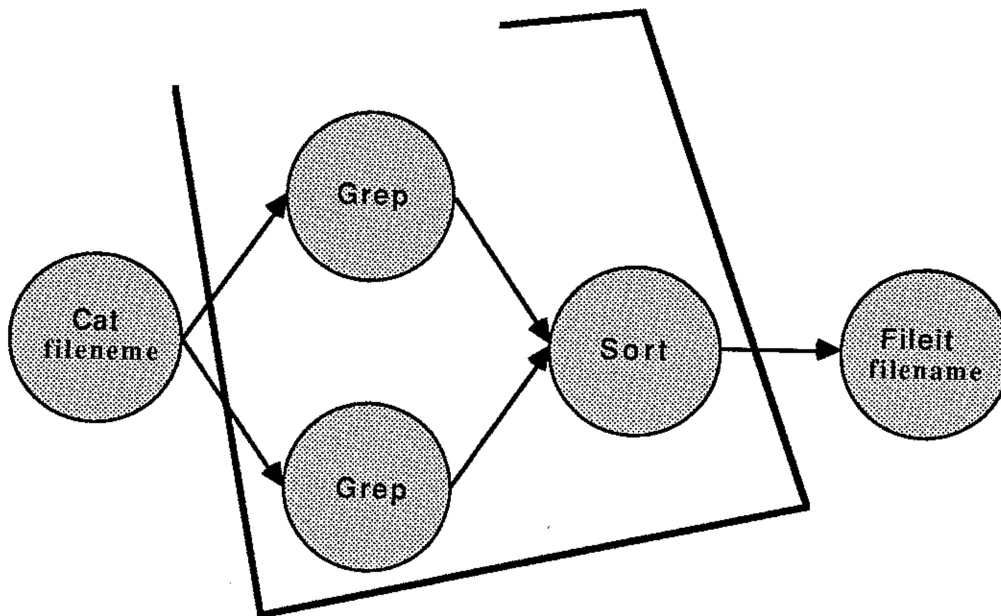

Figure 11a

Network Showing a Cutset Before the Collapse Operation

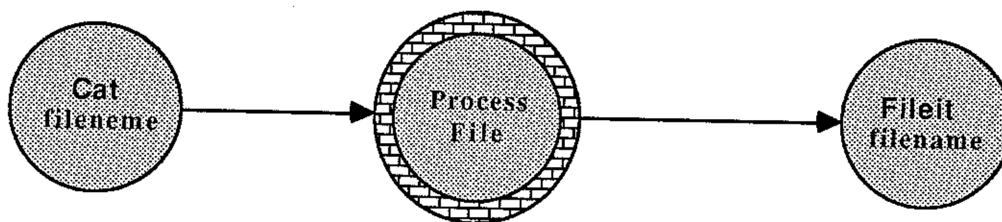

Figure 11b

Network Topology Display After the Collapse Operation

other generic node operations can be performed on the "super-node". If the user chooses to *explode* the *Process File* pseudotool, the network will be redrawn to show its pre-collapse state of Figure 11a.

4.2.2 The Save Tool-Composite and Attach Tool-Composite Operations

The *save tool-composite* operation performs an M_2 model abstraction of the cutset similar to the collapse. While the collapse operation does an in-place replacement of the cutset with the 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.

5.0 Summary and Conclusions

Abstraction is a powerful tool for succinctly describing, modelling and implementing complex operations. In particular, abstraction is extremely useful in characterizing the complexities of user/machine interaction as related to tools-based, task specification. Currently two prototype Taskmaster applications exploit abstraction in support of user task specification:

- a Unix-based, dataflow command shell supporting file transformation task specifications, and
- a matrix manipulation environment supported through selected LINPACK [DONJ79] routines.

Knowledge gained from the synthesis of these two application environments, and their current use as experimental test-beds, has contributed significantly toward the development of *complementary* interface abstractions. A realization of those abstractions within the Taskmaster environment embody a unique blend of top-down and bottom-up task specification capabilities, all oriented around visual programming concepts.

Top-down task specification is achieved through the successive refinement of a graphical network where nodes represent operations and arcs correspond to communication paths between those operations. In support of the top-down specification process, Taskmaster effectively exploits the inherent powers of *multi-level*, menu-based interaction and the *node expansion* operation. The multi-level, menu-based interface employs *partitioned* networks to minimize the adverse impact of rigid network traversal paths so prevalent in conventional menu-based interaction. In particular, the *interface layers* induced by partitioned networks define logical network boundaries that enable the user to choose a depth-first or breadth-first task specification approach. Even with such flexibility, however, menu-based interaction requires the binding of one node to one tool. To relax this restriction, Taskmaster utilizes a second top-down specification mechanism, *node expansion*. Node expansion allows the user to associate fully specified subnetworks (or subtasks) with individual nodes.

Bottom-up task specification is achieved through the *successive abstraction* of fully specified lower level networks into higher level operations. The significance of this bottom-up abstraction process is that the user can define powerful pseudotools, fundamental to the application environment, and combine them to form a higher level operations. Additional applications of successive abstraction lead to the desired task specification and, as a by-product, a powerful set of *reusable* pseudotools. As touted by Boehm [BOEB84] and Munsil [MUNW85], the provision for reusable components can have a significantly beneficial impact on productivity.

In summary, our work in user task specification has provided significant insights into the complexities of user/machine interaction. Although the research described in this paper presents only one aspect of a multi-faceted problem, it is a *crucial* aspect. Research addressing interface abstractions as well as the many other issues underlying "user-friendly" systems must continue if the user community is to enjoy simplicity and power in interactive, man/machine dialogue.

References

[ARTJ88] Arthur, J., "GETS: A Graphical Environment for Task Specification," *Proceedings of The Seventh Annual Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1988, To Appear.

[ARTJ87] Arthur, J. and Comer, D., "An Interactive Environment for Tool Selection, Specification, and Composition," *International Journal of Man-Machine Studies*, Vol. 26., No. 5, May 1987, pp. 581-596.

[ARTJ85] Arthur, J., "Partitioned Frame Networks for Multi-Level, Menu-Based Interaction", *Proceedings of the Fourth Annual Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1985, pp. 34-39.

[BOEB84] Boehm, B., et al., "A Software Development Environment for Improving Productivity," *IEEE Computer*, Vol. 17 No. 6, June 1984, pp. 30-42.

[DORJ79] Dongarra, J., et al., *Linpack User's Guide*, SIAM, Philadelphia, Pennsylvania, 1979.

[EHRR86] Ehrich, R. and Williges, R., *Human Computer Dialogue Design*, Vol. 2, Elsevier Amsterdam, 1986.

[GLIE84] Glinert, E. and Tanimoto, S., "Pict: An Interactive Graphical Programming Environment," *IEEE Computer*, Vol. 11, No. 11, November 1984, pp.7-25.

[MUNW85] Munsil, W., "The Language Translation Task: Toward Reusable Components," *Proceedings of the Fourth Annual Phoenix Conference on Computers and Communications*, Phoenix, Arizona, March 1985, pp. 46-52.

[RABM59] Rabin, M. and Scott, D., "Finite Automata and Their Decision Problems," *IBM Journal of Research and Development*, Vol. 3, 1959, pp. 114-125.

[REIS86] Reiss, S., "GARDEN Tools: Support for Graphical Programming," *Proceedings of the Workshop on Advanced Programming Environments*, Trondheim, Norway, June 1986, pp. 519-536.