AN INTERACTIVE ENVIRONMENT FOR TOOL SELECTION, SPECIFICATION AND COMPOSITION

by

James D. Arthur

TR-86-2

January 1986

An Interactive Environment for Tool Selection, Specification And Composition*

James D. Arthur

Department of Computer Sciences
Virginia Polytechnic Institute
Blacksburg, VA 24061
(703) 961-7538

ABSTRACT

This paper describes a high-level, screen oriented programming environment that supports problem solving by tool selection and tool composition. Each tool is a powerful parameterized program that performs a single high-level operation (e.g., *sort* a file). To solve a given problem, the user first interacts with the system to compose a task *overview* consisting of a sequence of *generic* operations. Such sequences are called *compositions*. Once an overview is established, a second part of the environment interacts with the user to help expand the generic operations into a corresponding sequence of parameterized tool calls. When a composition is expanded to include details such as parameterization and punctuation it is called a *script*. This script, when executed by the underlying runtime system, computes a solution to the specified user task.

The current environment runs under the Unix operating system on a VAX 11/785, and uses a Bitgraph terminal with a 640×720 bitmap display and standard keyboard as the principal interface device.

* This work was supported in part by R.R. Donnelley & Sons, Chicago, IL

An Interactive Environment for Tool Selection, Specification, and Composition

---

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications - *methodologies, tools*; D.2.2 [**Software Engineering**]: Tools and Techniques - *user interfaces, modules and interfaces, software libraries, top-down programming*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.9 [**Software Engineering**]: Management - *productivity*

General Terms: Menu-Driven Systems, Man/Machine Interaction, User Interfaces, Tools-based Programming

Additional Keywords: Frames, Item selection, User response reversal, History, Tools

---

1. Introduction

Over the past decade, software tools have evolved from rudimentary, single function programs like loaders, assemblers, and device drivers, to more sophisticated collections of complementary tools that define self-contained, computational *environments*, e.g., Toolpack[1]. In tandem with this trend, the basic framework for specifying a task solution has changed from line-at-a-time coding to collecting and interfacing *existing* software modules. This approach is particularly attractive because it reduces coding errors and implementation time.

A powerful variation of the above approach is found in environments that support the ***runtime composition*** of independent software tools, e.g., Unix† with its composition mechanism – the "pipe". To "program" a given task in this environment, one decomposes the task into a sequence of conceptually simple, high-level operations, and then combines (composes) a corresponding sequence of software modules (tools) that implements these operations. Although the sequential nature of

† Unix is a trademark of AT&T Bell Laboratories

tool composition tends to restrict the applicable problem domain, with the proper set of software tools one can significantly minimize this restriction.

In general, this "toolkit" approach to task specification is particularly effective for an experienced user, but is often difficult for an intermediate user and usually more so for a novice user. As pointed out by Branstad[2], tools are usually designed for experts; many are poorly human engineered for the novice or even the journeyman programmer. The inherent truth of this observation, coupled with the author's opinion that programming via tools and tool composition can significantly increase productivity, provides the basic motivation for this research. The fundamental assumption is that to "program" effectively using tools and tool composition, the user must a) know what tools are available and how to use them, or b) have access to a software system that knows these details and can guide the user through an acceptable task specification. Because many such tools are needed to adequately support "tools based" programming, alternative (a) may be impractical for many users (e.g., the intermediate and novice users). Hence, a real need exists for an interactive environment that *knows* which tools are available, their capabilities, their invocation details, and how to compose them.

This paper describes Omni, an interactive environment for tool selection, specification, and composition. Omni embraces the "tools based" approach to problem solving and provides a user-friendly interface that guides the user through each task specification. Its novel features include the following.

A Two-Level User Interface. The primary interface is menu based and supports two levels of interaction. At the highest level, the system provides information about generic, high-level operations and guides the user through a task specification overview. Once a high-level overview has been established, the lower level interface assumes control of the dialogue and provides assistance with tool invocation and argument specification details. In effect, a composition is specified through the upper-level interface and transformed into a meaningful program through the lower-level interface. Advantages of the two-level approach include:

- Novice and Expert levels. A separate lower-level interface makes it possible for experts to access detailed information about specific tools directly, without going through the upper-levels of the menu system that are designed for a novice.

- Top-Down Design. The environment encourages top-down design by starting with the selection of abstract operations, moving toward composition, and ending with detailed parameter specification.

- Reduction of Menu Network Complexity. Because operation selection is distinct from tool invocation, we partition the menu network into disjoint sets, making its combined size smaller than that of a single monolithic network.

Ability to Edit Compositions. The system produces as output, a readable (textual) representation of a program that the user can edit, either with the built-in editor or one of his choosing.

Tool Database. Tool selection is an integral part of the composition process. As new tools are devised, the environment must know their capabilities. To make it easy to add new tools, Omni keeps all tool information in a database. This information includes a complete description of each tool's capability as well as the dialogue sequence necessary to refine its functional operation.

The Omni environment includes both *menu-driven* and *command-driven* interaction. This permits the user to follow menus until sufficient information has been found, and then to switch to text mode to fill in details. The system supports *incremental script generation* whereby the user receives immediate feedback from menu item selections, and an *in-line, screen-oriented editor* that enables the generated script to be modified at any time. The environment is also designed from and ergonomic standpoint: it provides error/crash recovery, a help subsystem, command aliasing, and function key rebinding.

2. Background

An *environment* can be described loosely as those parts of a system that the user perceives. This perception extends beyond the immediate user interface to facilities that provide ancillary user support. It includes the psychological "feel" of the system as well as the details of its functionality. In essence, the environment defines *how* one expresses computations as well as *what* one can express. A user environment consists of the methods, techniques, and tools that are used during the specification and completion of a given task.

Because people initially build tools to satisfy their own requirements, we have seen the growth of user environments that support software development. Among the earlier and more prominent environments are Unix, Interlisp[11], and Smalltalk[12]. More recently, environments like Gandalf[3] and The Cornell Program Synthesizer[4] have been introduced. In addition to basic software support such as compilers, loaders, and editors, these environments provide user facilities aimed at reducing task complexity and increasing productivity. Unix, for example, provides a user interface called a shell[5]. The shell is a command interpreter that accepts commands from the terminal and interprets them as requests to run a program.

Unlike Unix, which hosts several language compilers and interpreters, the Interlisp and Smalltalk environments focus on a single programming language. Interlisp is an environment that supports interactive Lisp programming, and provides an extensive set of user facilities. The more popular facilities include Masterscope and Dwim. Masterscope is an interactive facility for analyzing and cross-referencing user programs. It provides a global view of existing programs, calling sequences, and so forth. Dwim (Do What I Mean) is invoked when a system error is detected. It makes an intelligent guess as to what the user might have intended and then makes the correction. The Smalltalk environment is similar to Interlisp in that it provides many of the same basic programming functions, but is based on its own programming language bearing the same name. Smalltalk is largely written in its own language and supports what is called "modeless

operation". That is, Smalltalk allows the user to invoke many of its functions at any time. Except for Unix and Smalltalk, each of the above environments provides access to "smart" (or structured) editors, tailored for their respective language base. Each editor is aware of its associated language primitives and the rules that govern their usage. They provide a skeletal program, and the user provides the details. Additionally, Gandalf provides an extensive software (module) information facility. The user can determine the nature of a module or subsystem through an associated descriptor. These descriptors contain information such as the function provided by each module, its parameter requirements, and dependency modules.

The above mentioned environments all share a common purpose: to provide a user-friendly environment for specifying and computing solutions to user tasks. One disadvantage to working in these environments, however, is that they expect the user to either 1) *code* a task solution from language primitives, or 2) based on *a priori* knowledge, construct a task solution from existing software modules (or tools). Within the Omni environment, "tools based" programming is also fundamental to specifying task solutions. To minimize the need for *a priori* tool knowledge, however, Omni supports an interactive user interface that guides the user through tool selection, specification and composition.

To illustrate the full capabilities of the Omni environment, this paper presents the system from two perspectives. First, Omni is presented from a user's point of view. This includes a detail description of the primary user interface as well as ancillary facilities that promote a user-friendly environment. The second presentation describes why Omni is a *general-purpose*, problem-solving environment. It discusses the the basic elements that comprise Omni's application-independent design and illustrates how one defines an *application scope* for an Omni environment. Finally, a brief overview of an Omni-based file transformation environment is presented.

3. Omni: A Interface Design

From a user's perspective, the usefulness of a system is highly dependent on 1) the adequate use of visual aids in presenting and soliciting information, 2) the ease with which one can communicate

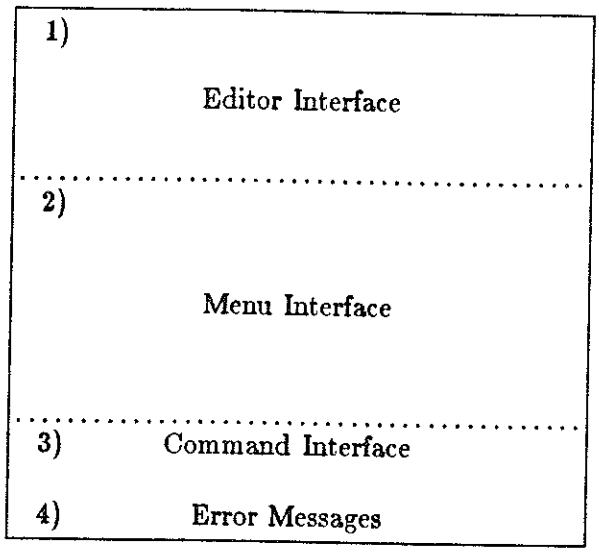

Figure 1
The Primary Screen Format

with the system, and 3) how forgiving the system is when a mistake is made. Because each of these issues individually presents a multi-facet design problem, they are separately addressed in the following section.

3.1 Screen Partitions

First impressions play an important role in the acceptance or rejection of a system[6]. A major portion of that impression is based on the user's visual perception of the system. In the Omni environment two distinct screen formats support visual prompts and user interaction.

The primary screen format, shown in Figure 1, is the first visual object presented to the user. It is partitioned into 4 regions called *windows*. Windows 1 through 3 are considered to be interactive windows, that is, they display information as well as accept responses. Only one of these windows, however, is active at any given time. Window 4 delineates the region where all error messages are displayed. When the user initiates an Omni session, the primary screen is displayed with window 2 having the active status. This window is initially active because it supports the principal communication vehicle, a menu-driven interface. Window 3 supports a command-driven interface whereby the user gains access to support functions such as *undo* and *history*. As discussed

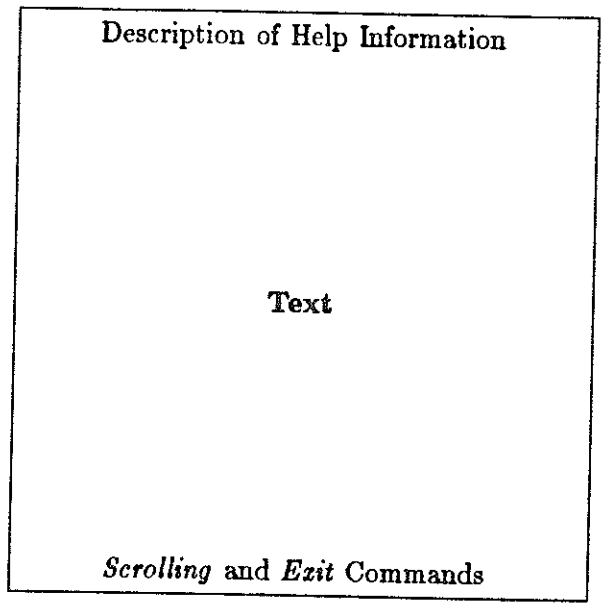

Figure 2
The Help Screen Format

later, windows 3 and 4 also define a conversational region. That is, they support a question and answer dialogue format.

Omni provides a second screen format, illustrated in Figure 2, that incorporates the full screen length and displays *help* information to the user. Because the formatted information may exceed the length of the "help window", scrolling capabilities are provided. The user activates the help window by initiating a help command. A simple, one key response returns the user to the primary screen having the same configuration that preceded the user's request for information.

Although the display formats in Omni are based on *partitioned* screens (e.g., the primary screen format), "pop-up" windows, like those found on the Apollo and Sun workstations, are equally applicable. A primary objective of the Omni environment is to support tools-based programming – this can be achieved using either display format approach. We mention, however, that while "pop-up" windows can provide additional versatility, they also reduce software portability.

3.2 The User/Machine Interface

Proper screen formats are essential for a good user interface. Nevertheless, they are at most

aesthetically pleasing. The complexity of user problems and varied interface requirements place additional demands on support environments, and in particular on the dialogue facilities. As described below, the Omni system supports multiple communication formats.

The Two-Level, Menu-Based Interface. Omni provides an interactive problem solving environment based on tools and tool composition. Its design is tailored for the user who has a firm understanding of the problem to be solved, but needs assistance in selecting and composing the appropriate tools. Menu systems are ideal for such users, especially if the user is inexperienced or unfamiliar with the system[7].

The initial Omni prototype employed a conventional menu-driven interface. In this environment, each time a high-level operation is detected in the user task specification, a corresponding tool is selected and *immediately* expanded into a tool call via successive menu-based interaction. The primary disadvantage of such an approach is that the user must contend with tool particulars (arguments) while maintaining a global view of the intended sequence of high-level operations that effect a task solution.

The current prototype takes a two-level approach to task specification (see Figure 3). In effect the user specifies all generic task operations *before* addressing their individual details. Interaction at the upper level defines a sequence of high-level operations that outlines a task solution. In turn, interaction at the lower level provides the necessary details for constructing the appropriate sequence of tool calls. An additional advantage of the two-level approach is that it provides both novice and expert levels of interaction. The novice user can enter at the first level, select operations, and then explore the details through the second level. A more experienced user can enter directly at the lower level to determine details about a specific tool. This capability is particularly attractive because dependence on Omni decreases as the user becomes more proficient in the use of tools and tool composition.

As one might conclude from Figure 3, both interface levels share common software; in fact, they are the same mechanism. Their differing functions are precisely defined by the underlying

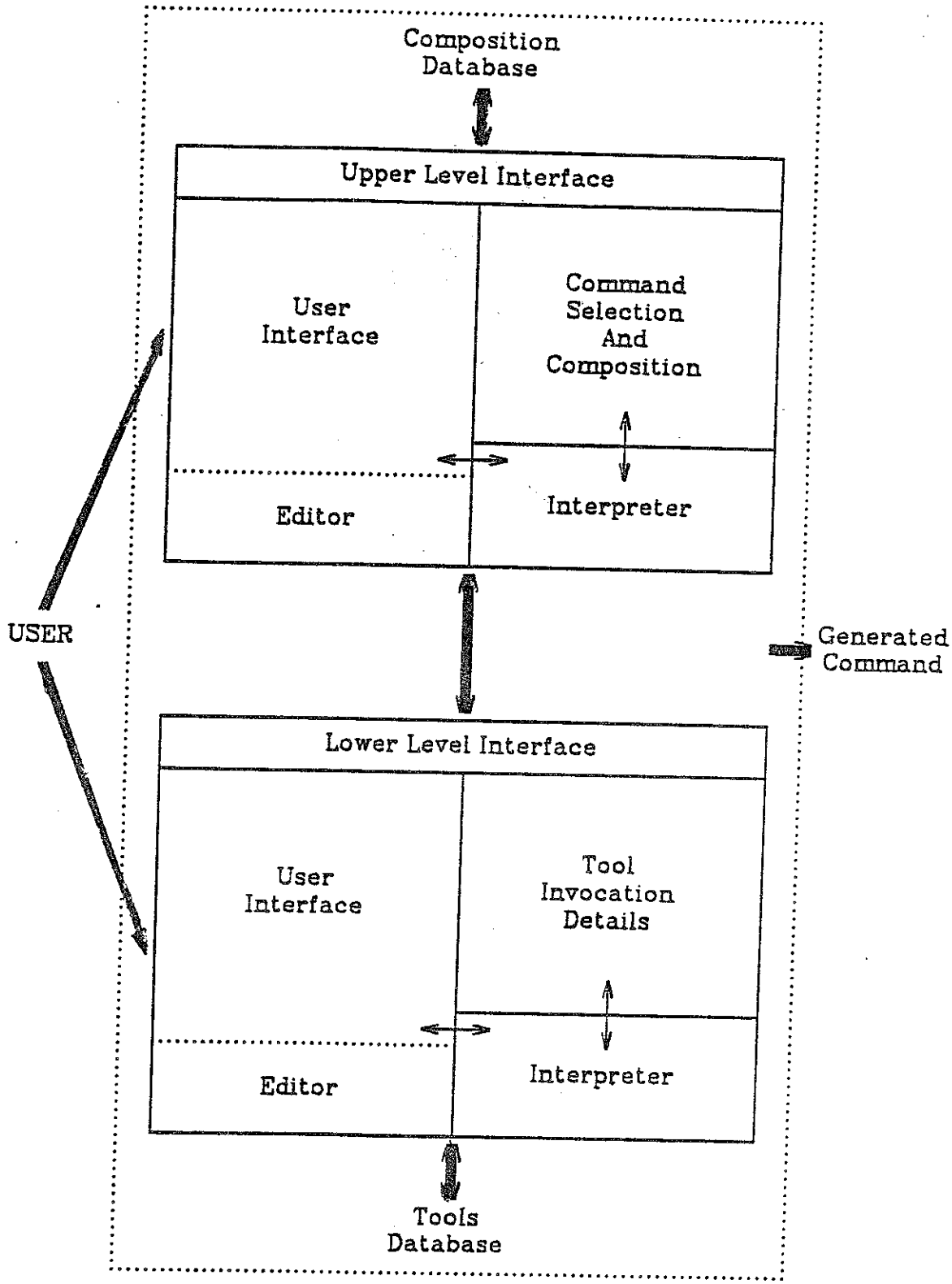

Figure 3
A Logical View of the Two-Level Interface

menu networks. When the user initiates an Omni session, an instance of the upper-level interface is created. A *composition database* supplies the menu network that defines the generic (high-level) operations supported by the Omni environment. Once a task overview is established by the upper-level interface, the lower-level interface assumes control, and sequentially processes the list of generic operations. For each generic operation, an instance of the lower-level interface is created and driven by the appropriate menu network obtained from a *tools database*. Figure 4 illustrates the logical relationship between the composition database and the tools database for a file transformation environment. Note that the composition database has only one menu network; it defines the generic operations. For each generic operation therein, a corresponding menu network is included in the tools database. These distinct networks define all available operations and their underlying tool invocation details.

From an implementation perspective, an added advantage of the two-level interface approach is that the "partitioned" menu networks are considerably smaller and more easily maintained than the monolithic counterpart. Moreover, the semantic actions that construct the tool calls can be logically partitioned; this promotes modular software construction.

Question and Answer Interface. Menu systems are very effective in guiding the user through decision processes. Effective interaction, however, often requires alternate forms of dialogue. Rather than base a user interface on one particular interaction format, Carlson[7] suggests a mixture. In addition to menu interaction, Omni supports a *conversation region* (windows 3 and 4) for system questions and user answers. In general, information that is not known *a priori* or cannot be deduced through menu interaction is requested through a question and answer dialogue[8]. Typical information solicited through the conversation region includes file names, record attributes, and numerical data.

The two interaction formats described above share a common trait: the *system* initiates the dialogue and the user responds. The third communication format, described next, allows the *user* to initiate the dialogue.

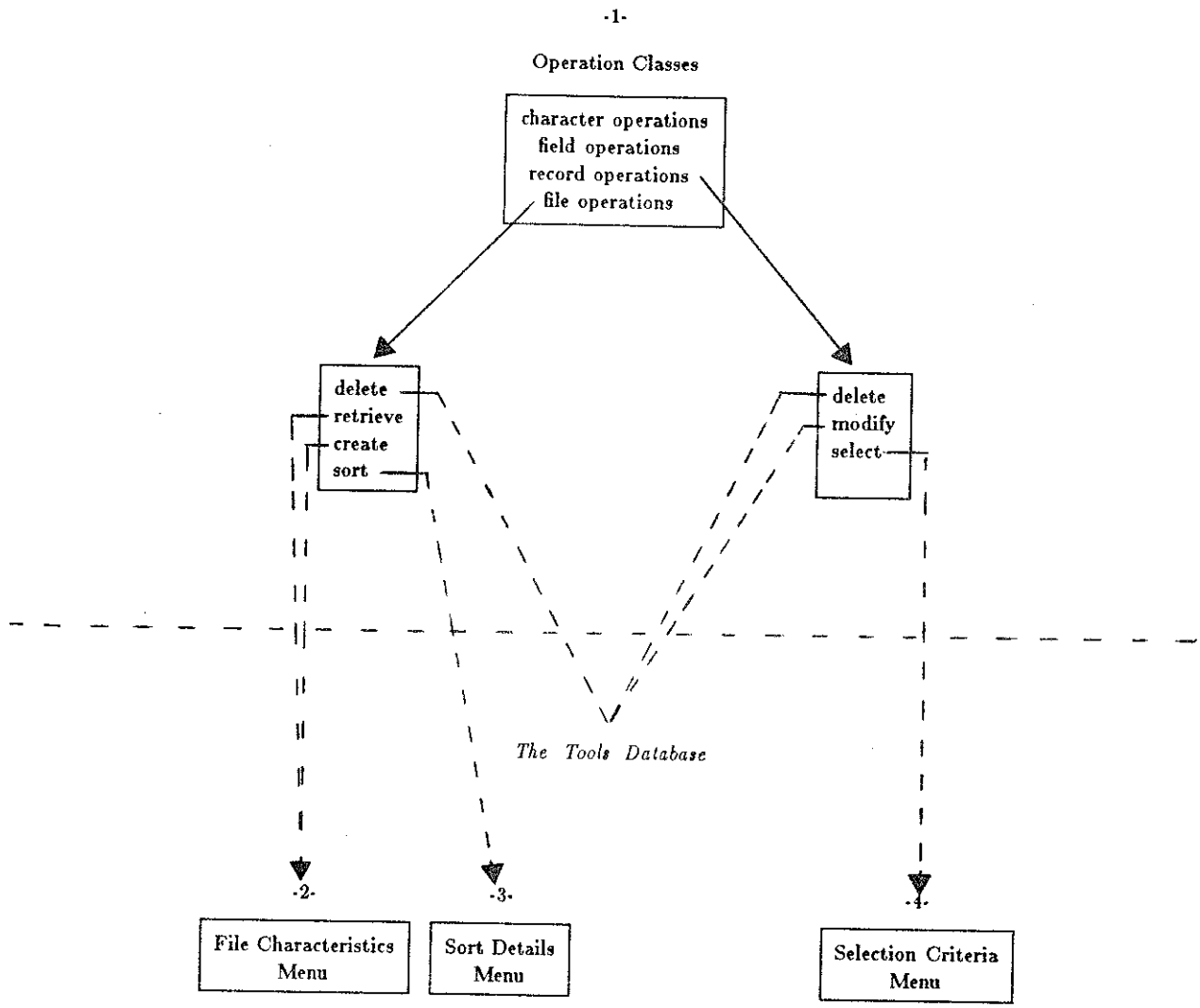

Figure 4
Logical Relationship Between the Composition and Tools Databases

Command-Driven Interface. In general, Omni guides the user through a task specification. Situations arise, however, where the user needs to control the dialogue process. For example, the user may require information about a specific frame item. Omni provides a command-driven interface, supported through window 3, that allows the user to gain control of and initiate an interactive dialogue. Instead of selecting a menu item or responding to a question, the user can request *command mode*, at which point the system relinquishes control of the dialogue process. The command facility provides access to support functions such as *history*, *undo*, and *help*, as well as function key

rebinding and aliasing capabilities.

Editor Interaction. As the user specifies a task, a corresponding script is incrementally constructed in window 1. This script, when executed by the underlying runtime system, computes a solution to the specified task. Although Omni supports user response reversal, script editing may still be necessary. For example the user may want to insert an operation not yet known to Omni but supported by the runtime system. Editing facilities, supported through window 1, permit modification of this script.

The Omni editor is screen-oriented and provides a wide variety of editing operations. They include character and line operations as well as of those operations that permit script viewing and user movement, e.g., scrolling, tabbing, and 4-way cursor movement. Additionally, in an effort to minimize user errors, the Omni editor requires the user to *select* line mode before initiating any line operation. This approach is recommended by both Engel[8] and Good[9] . Because users are fallible, however, the Omni editor also provides a *cancel* operation that restores the data to the pre-edit configuration, and then terminates the edit session.

3.3 Help Facilities

By matching each type of dialogue requirement with the proper interface format, Omni achieves a highly flexible, user-oriented interface. To further promote a "user-friendly" environment, Omni also supports a *help* facility that provides information about the user support functions as well as the current menu and menu items. In Omni, the *help* command is initiated through the command window and provides access to the following information.

Help for User Support Functions. For simplicity, the unadorned *help* command (or "?") displays a brief description of each user support function. If a more detailed explanation is required, the user invokes *help* with a function name as an argument.

Help for Menus. Because a menu heading alone may not be sufficient in describing its purpose, Omni provides access to detail information for each menu and each item within a menu. Both

types of information are accessed through the *help menu* command; the current position of the cursor determines which type of information is displayed.

3.4 Error Prevention and Recovery

The Omni environment is designed to minimize user errors as well as assist in correcting them when they occur. Considerations for error prevention include keystroke confirmation and immediate user feedback, while error recovery relies heavily on the *undo* and the editor *cancel* operations.

Error Prevention. To aid in reducing user errors, Omni adopts the "Simplicity and Consistency" approach[10]. The simplicity of user interaction refers to how many concepts the user must learn to function in a given environment. The menu-driven interface requires only item selections. Moreover, Omni allows the user to rebind function keys and define aliases during an interactive session or at start-up time via a reconfiguration file. In recognizing the fact that each user has individual preferences, Omni provides a set of support functions to encourage the user to redefine a more "natural" set of function keys. Aliasing commands provide additional flexibility by allowing the user to define synonyms for any given key sequence.

Consistency refers to similarity among command formats. In Omni, any user command can be invoked by simply typing its name. In general, if a command requires additional information, the user is prompted through the standard conversation region.

Error Recovery. It is generally accepted that even with good error prevention facilities, users still make mistakes. In Omni, such situations primarily occur during an item selection process or during an edit session. The *undo* and *cancel* operations, respectively, address these types of errors.

The principal function of the *undo* operation is to provide a method for reversing the effects of an item selection. The existence of an *undo* operation also encourages learning and experimentation. The user knows that if an item is selected out of curiosity the selection can be undone. In Omni,

the user is permitted to *undo* an arbitrary sequence of immediately preceding item selections that can be reviewed via the *history* command. The user executes the *history* command, determines the number of item selections to negate, and then issues the corresponding number of *undos*.

The *cancel* operation is a second error recovery mechanism and is accessible from the editor. Most errors introduced during an editing session are minor and relatively easy to correct; nevertheless, major errors can occur. For example, the user may inadvertently delete some necessary text or lines of data. During any given edit session, if a mistake occurs and simple recovery is not possible, the user can *cancel* the edit session. The *cancel* operation terminates the edit session and restores the script to its pre-edit configuration. The user can then reinvoke the editor and supply the correct sequence of modifications.

3.5 Immediate Visual Feedback

The features described above can be found in various systems that support a "user-friendly" interface. Because Omni is intended to be highly interactive, several design decisions have favored immediate visual feedback to user responses. For example, during a user task specification Omni detects each subtask as early as possible and *immediately* displays the corresponding generic function name. After all subtasks are detected and displayed, additional user interaction results in similar visual feedback during the expansion phase for each generic function. This *incremental* approach to script generation has two major advantages. First, it expands the feedback mechanism provided during normal interface activities. Second, incremental script generation serves as a learning device. As tool calls are developed, the user acquires familiarity with the available tools and their invocation criteria. This reduces later dependence on the Omni environment.

4. Omni: A General Purpose Problem Solving Environment

In the previous sections we have discussed several aspects of Omni that present it as a user-friendly problem solving environment. We have concentrated on characteristics that are common to all Omni environments. This section discusses those elements that make each environment

unique, that is, the elements that define the *application scope* of an Omni environment. We begin by discussing Omni from a general, application-independent perspective, and then enumerate those elements that define an application scope. The last section presents an overview of one particular problem solving environment supported by the Omni system.

4.1 Defining an Application for Omni

Each Omni environment is constructed from a basic, application-independent framework. This framework is an integrated set of software modules that supports the fundamental operations of tool selection, tool specification, and tool composition. Three elements are required to transform this general purpose framework into a specific problem solving environment:

- a set of frames (menus),
- a set of operations (actions) associated with each frame item, and
- a set of software tools.

In the primary user interface, the user specifies a given task by selecting items from successively displayed frames. In essence, frames describe a network that defines a *class* of problems solvable in an Omni environment, i.e., the application scope of the environment. The *composition database* contains the set of frames that outlines all generic (high-level) operations supported by underlying tools. The *tools database* contains an additional set of frames that defines logical processes for expanding each high-level operation into a tool call.

Associated with each frame item is an operation (action) that assists in the synthesis of a corresponding tool call. These operations can be grouped according to the functions they perform. First, they can *deduce* information and store it for later reference. Second, item operations can *prompt* the user for details that cannot be inferred from an item selection, e.g., a file name. The third group of operations addresses the incremental construction of the tool script. For example, if

an item selection completes a subtask specification, the corresponding operation modifies the tool script accordingly.

Frames, items, and item operations define a procedure for task specification and script generation. In essence, they provide a complete description of the problem set addressed by an Omni environment. In addition to providing a method for *specifying* a task and *implementing* a solution, the Omni environment contains the necessary tools for *computing* the solution. That is, it contains the set of tools that perform the selected operations.

4.2 A File Transformation Environment

The current prototype application environment addresses file transformation tasks. The frame networks, item operations, and software tools define and support character, string, record, and file oriented operations. The following paragraphs provide a brief overview of this environment by first describing the "tools approach" to file transformation and then providing a simple example.

4.2.1 The Tools Approach to File Transformations.

Suppose we are given a file F and want to construct file F' by applying transformations t_1, t_2 , and t_3. One approach is to successively apply all three transformations to *each* record $r \in F$, $1 \leq i \leq |F|$. Such an approach requires one pass through the file F, but many invocations of the transformation routines. An alternate approach is to apply transformation t_1 to *all* records in F, apply t_2 to the results of the first transformation, and then apply t_3 to the results of the t_2 transformation. This approach requires three passes through the records but only one invocation of each transformation routine.

The second approach is consistent with the underlying principles of tools and tool composition. If tools T_1, T_2, and T_3 implement the transformations t_1, t_2, and t_3 respectively, and "$\odot$" denotes composition, then the script "T_1 *filename* $\odot$ T_2 $\odot$ T_3" describes the complete transformation process. From an Omni perspective, the transformations are high-level operations supported by the tools T_j. The script is a sequence of tool calls that specifies a solution to the user's task. The following paragraphs illustrate this approach using the Omni prototype environment.

4.2.2 A File Transformation Example

The prototype file transformation environment provides tool-based, generic operations that address data sets from three basic perspectives. Data can be viewed and manipulated as (1) a single entity, or *file*, (2) a collection of character sequences terminated by a special delimiter, i.e. *records*, or (3) simply as a stream of characters without any inherent structure. Transformation operations included in category (1) are file oriented, that is, the smallest manipulative element is the file itself. Elements of this category include *retrieving* a file for subsequent operations, *sorting* a file, and *displaying* a file on some output device. Transformation operations found in category (2) view files as a sequence of records, i.e., records are considered as the atomic unit. Such operations include *selecting* specified records for further processing, *deleting* specified records, and record *modification*. Although these operations may seem rather simple, they possess extensive capabilities. They must understand the distinction between variable and fixed length record formats, and be able to choose records based on contents therein. This implies an additional understanding of *fields*, field *delimiters*, and comparison operators. Finally, the third category of transformation operations addresses character and string manipulation. These operations consider each file as simple text without any perceived structure. They include, *deleting* and *replacing* specified strings as well as *inserting* text according to given specifications. By considering a character as a string of length one, these operations generalize to simple character manipulation. Although there exist several other refinement levels, the ones mentioned above provide a basic set of operations sufficient for our experimental purposes.

Figures 5-12 illustrate a selected subset of screen displays for a simple task specification. The task is to retrieve a file of student records, select all records that have a graduation date of "1983" (it is assumed that each record contains only one date), and sort the records by the student's last name (starting in position 10).

Because the prototype environment supports a two-level interface, the user first specifies a task overview in terms of generic operations. Conceptually, one appropriate sequence is: (1) *retrieve* the

student file, (2) *select* the appropriate records, and (3) *sort* the resulting file. Figure 5 illustrates the category of file system operations available to the user. Because the first operation is to retrieve a file, the user selects frame item 1, "File Operations." Figure 6 illustrates the resulting frame which details all available file operations. By selecting the *Retrieve* item, the user specifies the first generic operation of the task overview. This selection is immediately placed in window 1 (as shown in Figure 7), and the user returns to the basic "File System Operation" frame. By selecting "Record Operations" next (and traversing the ensuing frame subnetwork), following with a selection of "File Operations", the user completes the task *overview* that is shown in window 1 of Figure 8. It is noted that the constructed sequence corresponds precisely to the conceptual sequence stated above.

The user signifies that the task overview is complete by providing a *null* response to the "File System Operations" frame, that is, the user presses the Newline (or Return) key. Subsequently, the second-level interface assumes control and initiates an expansion of the first generic operation (*Retrieve_file*) into its effective tool counterpart (shown in Figure 9). Note that *Retrieve_file*, the generic operation originally displayed in window 1, has been replaced by "cat", the corresponding Unix tool name. Additionally, the screen heading now displays the generic operation being expanded. The user selects frame item 1, "Retrieve 1 (One) File", and is immediately prompted for the file name (Figure 10). After entering the appropriate file name, window 1 is updated and the next generic operation is selected for expansion (illustrated in Figure 11). Note that the entries in window 1 are assuming the form of tools composed by the "pipe" operation. Continued interaction at the second level results in the final script:

cat student_file | *grep* "1983" | *sort* +0.9

shown in Figure 12. This script, when executed by the Unix shell, computes a solution to the specified task.

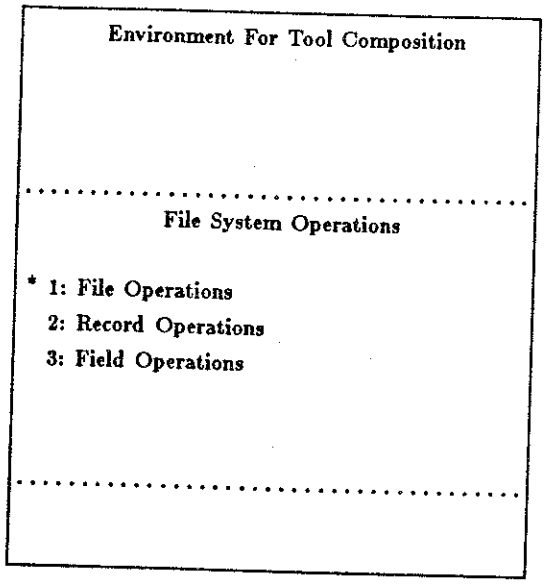

Figure 5
The Set of File System Operations

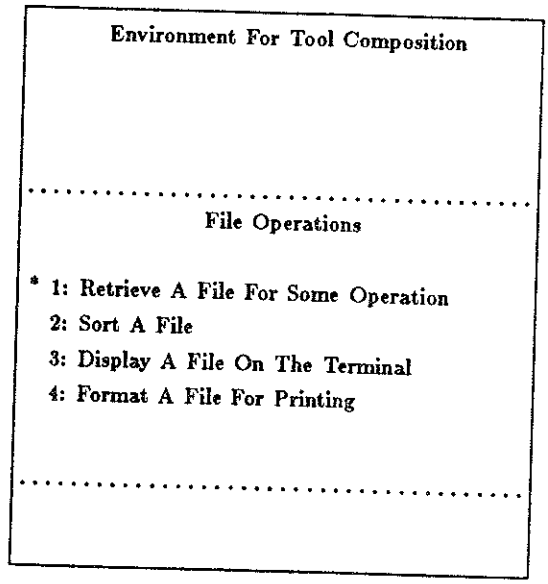

Figure 6
The Set of File Operations

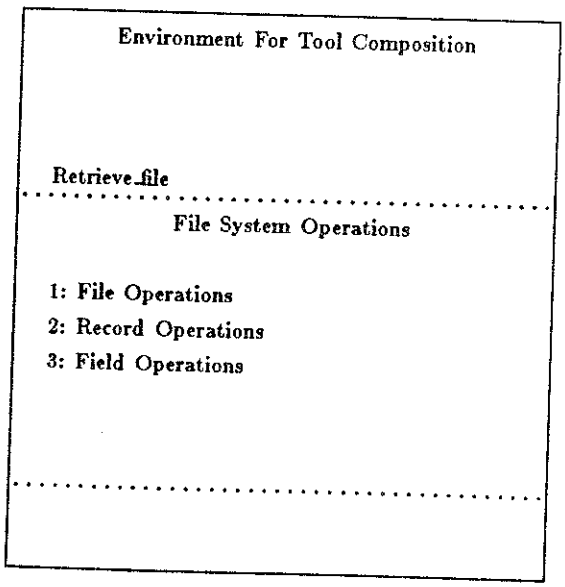

Figure 7
The Screen Display After Selecting Retrieve_file

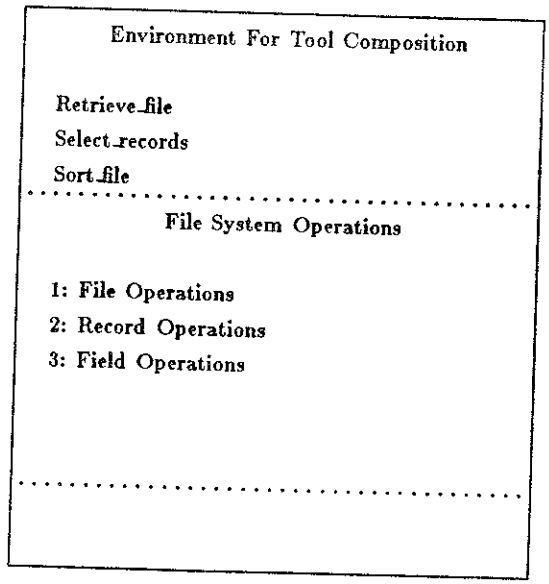

Figure 8
A Completed Task Overview

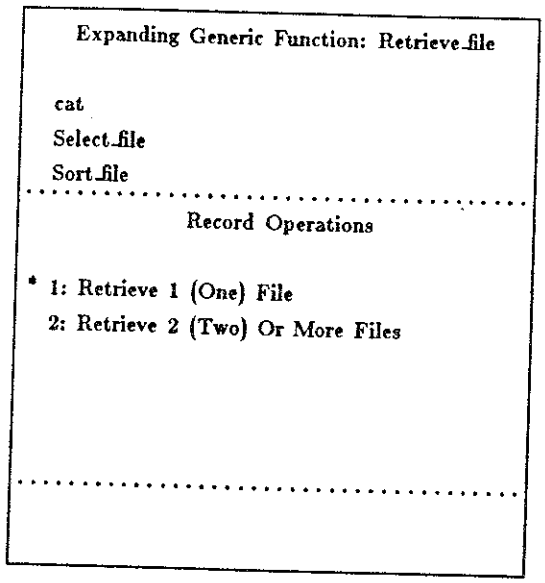

Figure 9
Expanding Generic Function: Retrieve_file

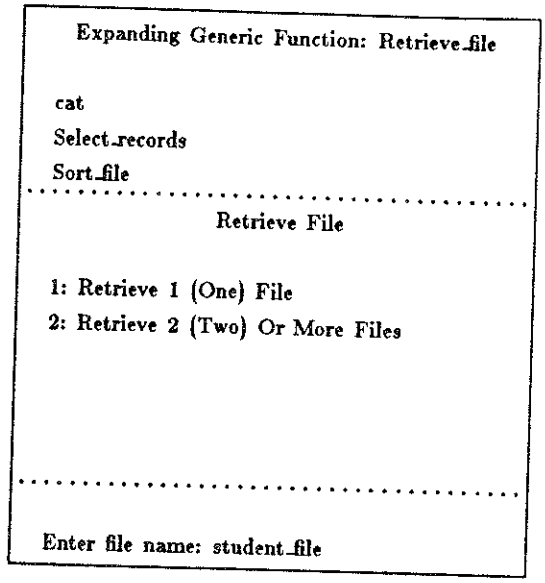

Figure 10
Soliciting a File Name

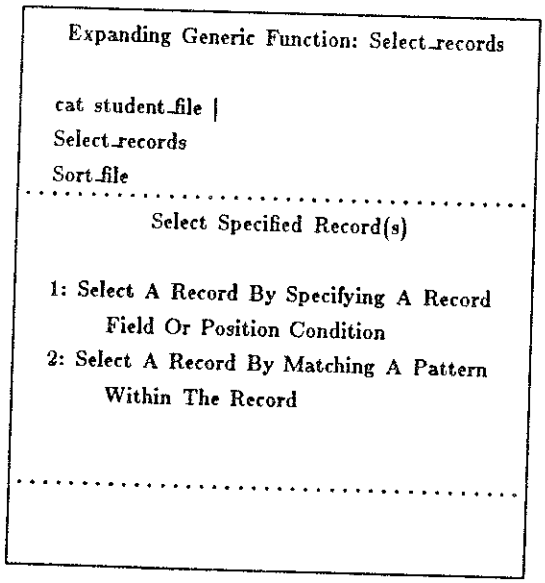

Figure 11
Expanding Generic Function: Select_records

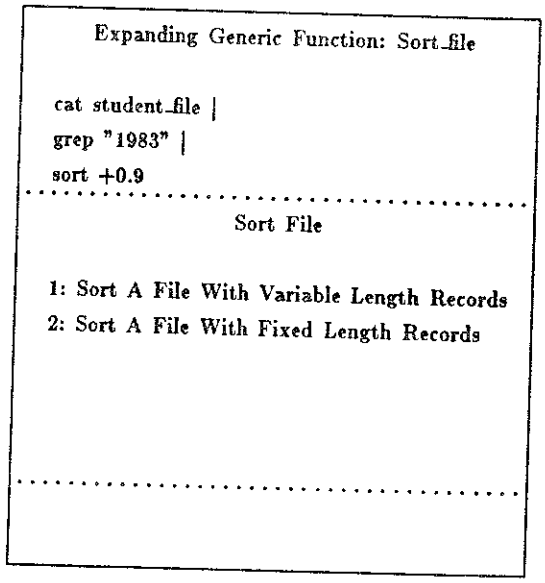

Figure 13
A Completed Task Specification

5. Concluding Remarks

This paper characterizes Omni as a user-friendly environment that supports task specification through a host of interactive formats. A primary motivation for developing Omni is to provide a highly flexible, user support environment that reduces the time and effort required to solve computational problems. In general the user response to using Omni has been highly favorable. Suggestions do indicate, however, that at times the repetitiveness of *textually* oriented menu interaction can be boring. To address this problem, our current research assumes the the use iconic structures and graphical networks for specifying a task. In developing Omni the approach has been practical: 1) the two-level interface encourages a truly top-down approach to task specification and, 2) the underlying concepts of tools-based programming encourages task specification through the use of more natural, high-level operations. From an implementation perspective, Omni consists of several software modules that perform specific functions. The interface between modules is designed to operate efficiently and succinctly. The result is a synergistic *interactive environment for tool selection, specification and composition.*

LIST OF REFERENCES

1. L. Osterweil, "Toolpack – An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering*, Vol. 9, No. 6, November, 1983, pp. 673-685.

2. M. Branstad and R. Adrion, "NBS Programming Environment Workshop Report," *A.C.M. SIGSOFT, Software Engineering Notes*, Vol. 6, No. 4, August, 1981, pp. 3-15.

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

4. T. Teitelbaum and T. Reps, "CPS – The Cornell Program Synthesizer," *Communications of the A.C.M.*, Vol. 24, No. 9, September, 1981, pp. 563-573.

5. S. Bourne, "The UNIX Shell," *Bell System Technical Journal*, No. 6 (Part 2), July-August, 1978, pp. 1971-1990.

6. T. Carey, "User Differences in Interface Designs," *IEEE Computer*, Vol. 15, No. 11, November, 1982, pp. 14-20.

7. E. Carlson, "Developing The User Interface For Decision Support Systems," IBM Research Report RJ3112, IBM Research Laboratory, San Jose, CA, April, 1981.

8. S. Engel, R. Granda, "Guidelines For Man/Display Interfaces," IBM Technical Report TR-00.2720, Poughkeepsie Laboratory, Poughkeepsie, NY, December, 1975.

9. M. Good, "Etude and the Folklore of User Interface Design," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, Vol. 16, No. 6, June, 1981, Portland, OR, pp. 34-43.

10. G. Perlman, "The Design Of An Interface To A Programming System," University Of California, San Diego Technical Report 8105, November, 1981.

11. W. Teitelman *et al*, *The Interlisp Reference Manual*, XEROX Palo Alto Research Center, Palo Alto, CA, October, 1978.

12. A. Goldberg, *Smalltalk-80 - The Interactive Programming Environment*, Addison Wesley, 1984.