



University of Pennsylvania
ScholarlyCommons

Publicly Accessible Penn Dissertations

1986

A User Interface Management System Generator

Tamar Ezekiel Granor
University of Pennsylvania

Follow this and additional works at: <https://repository.upenn.edu/edissertations>

Recommended Citation

Granor, Tamar Ezekiel, "A User Interface Management System Generator" (1986). *Publicly Accessible Penn Dissertations*. 4282.
<https://repository.upenn.edu/edissertations/4282>

This paper is posted at ScholarlyCommons. <https://repository.upenn.edu/edissertations/4282>
For more information, please contact repository@pobox.upenn.edu.

A User Interface Management System Generator

Abstract

Much recent research has been focused on user interfaces. A major advance in interface design is the User Interface Management System (UIMS), which mediates between the application and the user.

Our research has resulted in a conceptual framework for interaction which permits the design and implementation of a UIMS generator system. This system, called Graphical User Interface Development Environment or GUIDE, allows an interface designer to specify interactively the user interface for an application.

The major issues addressed by this methodology are making interfaces implementable, modifiable and flexible, allowing for user variability, making interfaces consistent and allowing for application diversity within a user community.

The underlying goal of GUIDE is that interface designers should be able to specify interfaces as broadly as is possible with a manually-coded system. The specific goals of GUIDE are:

- The designer need not write any interface code.
- Action routines are provided by the designer or application implementator which implement the actions or operations of the application system. Action routines may have parameters.
- The designer is able to specify multiple control paths based on the state of the system and a profile of the user.
- Inclusion of help and prompt messages is as easy as possible.
- GUIDE's own interface may be generated with GUIDE.

GUIDE goes beyond previous efforts in UIMS design in the full parameter specification provided in the interface for application actions, in the ability to reference application global items in the interface, and in the pervasiveness of conditions throughout the system. A parser is built into GUIDE to parse conditions and provide type-checking.

The GUIDE framework describes interfaces in terms of three components:

- what the user sees of the application world (user-defined pictures and user-defined picture classes)
- what the user can do (tasks and tools)
- what happens when the user does something (actions and decisions)

These three are combined to form contexts which describe the state of the interface at any time.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Norman Badler

Comments

A User Interface Management System Generator

Tamar Ezekiel Granor

**University of Pennsylvania
1986**

A User Interface Management System Generator

TAMAR EZEKIEL GRANOR

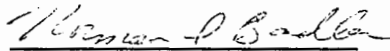
A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1986



Supervisor of Dissertation

Graduate Group Chairperson

COPYRIGHT ©
TAMAR EZEKIEL GRANOR
1986

This dissertation is dedicated to the memory of my grandfather, Walter Naphtali Ezekiel, who started a tradition, but didn't live to see it continued into the third generation.

ACKNOWLEDGEMENTS

A number of people have contributed to this work or have helped to make it a reality. I cannot express enough thanks to my advisor, Dr. Norman Badler, who suggested the idea in the first place, was available for endless consultation and understood the demands that marriage and motherhood placed on my time. In addition, the members of my dissertation committee, Dr. Jean Gallier, Dr. Tim Finin, Dr. Dale Miller, all of the University of Pennsylvania and Dr. James Foley of George Washington University all made valuable suggestions which improved the end result. Three students wrote parts of the system. Thanks to Jon Hanover for the window manager, Bruce Rabe for the toolkit code and Jill Smudski for the data structure drawing routines. The members of the Computer Graphics Research Group at Penn were all helpful; particular thanks to Jon Korein.

On the personal side, there are also a number of people to thank. My husband, Marshal, often seemed to have more faith in me than I did myself and worked hard to ensure that I had the time I needed to complete this work. My son, Solomon, gave up time with me for something he could not have understood. My parents, David and Ruth Ezekiel, let me know all my life that I could achieve anything academically that I aspired to. I also must thank my parents, my in-laws, Bernard and Marie Granor, my sister, Rachel Ezekiel and my sister-in-law, Alicia Granor, for lots of babysitting, without which this work might never have been completed.

ABSTRACT

A User Interface Management System Generator

Tamar Ezekiel Granor

Supervisor: Dr. Norman Badler

Much recent research has been focused on user interfaces. A major advance in interface design is the User Interface Management System (UIMS), which mediates between the application and the user.

Our research has resulted in a conceptual framework for interaction which permits the design and implementation of a UIMS generator system. This system, called Graphical User Interface Development Environment or GUIDE, allows an interface designer to specify interactively the user interface for an application.

The major issues addressed by this methodology are making interfaces implementable, modifiable and flexible, allowing for user variability, making interfaces consistent and allowing for application diversity within a user community.

The underlying goal of GUIDE is that interface designers should be able to specify interfaces as broadly as is possible with a manually-coded system. The specific goals of GUIDE are:

- The designer need not write any interface code.
- *Action routines* are provided by the designer or application implementator which implement the actions or operations of the application system. Action routines may have parameters.
- The designer is able to specify multiple control paths based on the state of the system and a profile of the user.
- Inclusion of help and prompt messages is as easy as possible.
- GUIDE's own interface may be generated with GUIDE.

GUIDE goes beyond previous efforts in UIMS design in the full parameter specification provided in the interface for application actions, in the ability to reference application global items in the interface, and in the pervasiveness of *conditions* throughout the system. A parser is built into GUIDE to parse conditions and provide type-checking.

The GUIDE framework describes interfaces in terms of three components:

- what the user sees of the application world (*user-defined pictures* and *user-defined picture classes*)
- what the user can do (*tasks* and *tools*)
- what happens when the user does something (*actions* and *decisions*)

These three are combined to form *contexts* which describe the state of the interface at any time.

Table of Contents

1. Introduction	1
1.1. What the user sees of the application world	3
1.2. What the user can do	4
1.3. What happens	5
1.4. Responding to Application States	8
1.5. Other Considerations	8
1.6. Output of a UIMS generator	9
1.7. GUIDE's interface	11
1.8. The generated interface	12
1.9. Organization	14
2. Survey of Other Research	15
2.1. Characteristics of an Ideal UIMS	16
2.1.1. Semantic Considerations	16
2.1.2. Interaction Considerations	17
2.1.3. Support Services	18
2.1.4. Specification Services	18
2.2. Interaction languages	20
2.3. Tool-oriented systems	23
2.4. Non-Graphics Systems	23
2.5. Other systems	24
2.6. Summary	26
2.7. Studies	26
2.7.1. Benesh Notation System	27
2.7.2. Grip-75	28
2.7.3. Spatial Data Management System	29
2.7.4. Lisa	30
2.7.5. The Lisp Machine	31
3. Tools and Tasks	33
3.1. The CORE Graphics System	33
3.2. Tools	34
3.3. Specific tools	35
3.3.1. Menus	35
3.3.2. Lists	37
3.3.3. Forms	39
3.3.4. Window Controller	40
3.3.5. Potentiometers	41
3.3.6. Picks	42
3.3.7. Keyboard	43
3.3.8. Locator with Button	43
3.3.9. Valuator with Button	44
3.3.10. Button	44

3.4. Tasks	45
3.5. Predefined Tasks	46
4. Contexts	47
4.1. User-defined pictures	47
4.2. The control path	48
4.3. Expressions and Conditions	48
5. Actions	52
5.1. Parameter gathering	52
5.1.1. Parameter contexts	52
5.1.2. Parameter expression lists	53
5.2. Fixed Actions	55
6. User Support	56
6.1. User Profiles	56
6.2. Helps	58
6.3. Prompts	58
6.4. Messages	58
7. The User Interface	60
7.1. Start-up	60
7.2. Appearance of the interface	61
8. Output	63
8.1. Data structures	63
8.2. Error file	63
8.3. Journal file	64
8.4. Interface	64
8.4.1. Context Code	64
8.4.2. Task routines	65
8.4.3. Tool Routines	65
8.4.4. Expression and Condition Code	68
8.4.5. Window controller extremes evaluation	72
8.4.6. Invocation Routines	73
8.4.7. Task Lookup Code	74
8.4.8. Environment	74
8.4.9. Data	75
9. Implementation	76
9.1. The organization of GUIDE	76
9.2. The structure of GUIDE	77
9.3. Choice of a graphics package	81
9.4. Representation of the Design	84
9.5. The Symbol Tables	84
9.6. Memory Management	86
9.7. Reading and Writing Designs	88
9.7.1. Overview of the solution	89
9.7.2. The Symbol Table	91
9.7.3. The Output Phase	91
9.7.4. The Input Phase	93

9.7.5. Problems	100
9.8. Copying Data Structures	100
9.9. Tool/Logical Device Mappings	101
9.10. The parser	102
10. Conclusions	103
10.1. Unavailable Features	103
10.1.1. Features not yet implemented	103
10.1.2. Inaccessible Features	104
10.2. Problems with defaults	105
10.3. Conversion Routines	105
10.4. Experience with GUIDE	106
10.5. Future Directions	106
10.5.1. Expression Functions	106
10.5.2. Tool Process Routines	106
10.5.3. Additional use of conditions	107
10.5.4. Changes to Default Task Values	107
10.5.5. Default Device Selection	107
10.5.6. Device Specification	108
10.5.7. Use of a database	108
10.5.8. Maintaining Session Status in the User Profile	108
APPENDIX A. Entities built into GUIDE	109
APPENDIX B. The Layout System - An Example	113
B.1. Environment File for the Layout System	113
B.2. Using GUIDE to create a design	116
APPENDIX C. Environment BNF	151

List of Tables

Table 2-1: Characteristics of UIMS Systems

19

List of Figures

Figure 1-1: Control Flow using GUIDE	12
Figure 1-2: Data Flow in a GUIDE-generated Interface	13
Figure 3-1: A menu	37
Figure 3-2: A list	38
Figure 3-3: A form	40
Figure 3-4: The window controller	41
Figure 3-5: A Horizontal Potentiometer	42
Figure 3-6: A keyboard tool	43
Figure 3-7: Buttons	44
Figure 4-1: GUIDE main context after selecting "create/edit context"	49
Figure 4-2: GUIDE context creation context	50
Figure 5-1: Layout system after selecting "Add symbol"	53
Figure 5-2: "Add symbol" parameter selection	54
Figure 7-1: Form for creating list tool instances	62
Figure 7-2: Picking a position within a list	62
Figure 8-1: Procedure generated for main layout context	66
Figure 8-2: Procedure to draw a task	68
Figure 8-3: Procedure to process a task	69
Figure 8-4: Procedure to set up and draw a tool	70
Figure 8-5: Procedure to echo and process input to a tool	71
Figure 8-6: Function to compute an expression value	71
Figure 8-7: Condition evaluation function generated by GUIDE	72
Figure 8-8: Procedure to compute window controller extremes	72
Figure 8-9: Context invocation procedure	73
Figure 8-10: Task look-up code generated for the layout system	74
Figure 9-1: Files input by GUIDE	78
Figure 9-2: Files output by GUIDE	79
Figure 9-3: Data flow in GUIDE	80
Figure 9-4: GUIDE runtime services	81
Figure 9-5: Processing after using GUIDE	82
Figure 9-6: Structure of a GUIDE-generated application	83
Figure 9-7: Records defining user-defined picture portion of contexts	85
Figure 9-8: Operations on context records	87
Figure 9-9: Some sample data structures	89
Figure 9-10: Sample data structures with additional fields	90
Figure 9-11: Clear routines for sample data structures	92
Figure 9-12: Output routines for sample data structures	93
Figure 9-13: Input routines for sample data structures	96
Figure B-1: Start context for GUIDE	120
Figure B-2: Main context for GUIDE	121
Figure B-3: Task creation context for GUIDE	122

Figure B-4: Tool creation context for GUIDE	123
Figure B-5: Tool creation context for GUIDE with instance command	124
Figure B-6: Menu creation context for GUIDE	125
Figure B-7: Menu item context for GUIDE	126
Figure B-8: Parameter context for "add item to list"	127
Figure B-9: Parameter context for "add list to item"	128
Figure B-10: Parameter context for "add tool"	130
Figure B-11: Main context for GUIDE with picture class commands	131
Figure B-12: User-defined picture creation context for GUIDE	132
Figure B-13: Parameter context for "add user picture"	133
Figure B-14: Context creation context for GUIDE	134
Figure B-15: Parameter context for "add predef td to context"	135
Figure B-16: Parameter context for "position tool"	136
Figure B-17: Parameter context for "make control path"	137
Figure B-18: Task creation context with action menu	138
Figure B-19: Parameter context for "Add context to list"	139
Figure B-20: Parameter context for "Add param context to action"	140
Figure B-21: Parameter context for "Enter actual param list"	141
Figure B-22: First parameter context for "Add action to task"	142
Figure B-23: Parameter context for "Generate code"	143
Figure B-24: Start context for layout system	144
Figure B-25: Main context for layout system	145
Figure B-26: Parameter context for "add symbol"	146
Figure B-27: Parameter context for "delete symbol"	147
Figure B-28: Parameter context for "change title"	148
Figure B-29: Parameter context for "change size"	149

CHAPTER I

Introduction

In recent years, much attention has been given to user interfaces. Researchers have studied what kinds of commands are most easily learned and remembered, (*e.g.*, [Barnard 82, Black 82]), what kinds of interfaces are easiest to use, (*e.g.*, [Card 82, Savage 82]) and how to make it easier to provide a good interface, (*e.g.*, [Buxton 83a, Kasik 82]). One approach to simplifying interface implementation is a user interface management system (UIMS), which mediates between the application and the user in much the same way that a data base management system mediates between the application and the data [Kasik 82, Thomas 83]. Several groups have implemented UIMS's [Bloom 83, Buxton 83a, Kamran 83, Green 85a, Kasik 82, Olsen 83a, Olsen 83b, Roach 82, Rubel 82, Wong 82].

This dissertation describes an interaction methodology that permits the design and implementation of a UIMS *generator* system. The system, called Graphical User Interface Development Environment (GUIDE), allows the system designer to specify the relationship between the interaction and the control path of his application. GUIDE then generates the code needed to form the UIMS, along with data describing specifics of the interface. The designer does not write the control code; GUIDE generates it.

The first question that must be addressed in creating such a system is what need is it attempting to fill; that is, what problem is it attempting to solve? In the area of interface design, a number of areas must be addressed:

- Implementability - There is the difficulty of implementing any interface. It tends to be a tedious and error-prone task. This problem is magnified when a system is to use graphics.
- Modifiability - Once an interface has been implemented, it is usually difficult to make substantial modifications without scrapping the entire interface and starting again. In fact, it is frequently difficult to make even small modifications. It is also desirable to be able to implement part of an interface and refine it before specifying the entire interface.
- Flexibility and User Variability - Even beyond the difficulty of implementing a single interface, many systems really need several alternative interfaces or alternative

paths within an interface depending on characteristics of the user and the state of the system.

- Intra-Application Consistency - It is desirable for an interface to have internal consistency; that is, the user should be able to do the same thing in the same way each time that it arises.
- Inter-Application Consistency - Within a user community, it is desirable to have consistency across the various interfaces used by the members of the community. This makes it easier for a user, knowledgeable in one system, to move to another.

Knowing what problems are to be addressed, we next must determine what is needed to solve them. In this case, we need a way for a designer to describe an interface. We want a method which is simple and will allow us to fulfill the considerations above. The interface which is described should be able to do anything a manually-coded interface can; using a generator should not limit the designer's expressiveness.

Based on these underlying goals, five specific goals have been formulated for GUIDE.

- The designer need not write any interface code.
- *Action routines* are provided by the designer or application implementor which implement the actions or operations of the application system. Action routines may have parameters.
- The designer is able to specify multiple control paths based on the state of the system and a profile of the user.
- Inclusion of help and prompt messages is as easy as possible.
- GUIDE's own interface may be generated with GUIDE.

To find a way to describe interfaces, we must look at what is contained in a user interface. There are three components:

- what the user sees of the application world;
- what the user can do;
- what happens when the user does something.

"What the user sees" refers to images derived from application data and data structures. "What the user can do" refers to the choices a user has at any time: what commands can be executed and what data can be modified. "What happens when the user does something"

refers to the control structure of the system. The choices the designer makes for each of these components at each point in the system determine the appearance and behavior of the interface. The next few sections discuss each of these components in detail.

1.1. What the user sees of the application world

A graphical user interface for any system must contain pictures which are the user's view of the application world. Generally, these pictures are graphical representations of application data structures. For example, in the room layout system described in Foley and van Dam [Foley 82], one such picture would show the current state of the room. In GUIDE, these pictures are called *user-defined pictures*. In a graphics editor, a picture of the object being edited would be a user-defined picture.

In Chapter 2, several existing systems are described in terms of GUIDE's structure. In doing these studies, it became apparent that there are frequently groups of objects of the same type, all of which should be displayed. In addition, these sets often can be modified dynamically (by the application or the user). For example, the Symbolics Lisp Machine has a set of "Lisp Listener" windows, any of which can be accessed by the user. The user may create and destroy "Lisp Listeners" as desired.

Individual items such as a single "Lisp Listener" are represented in GUIDE by user-defined pictures. To handle groups of pictures which are related, user-defined pictures are organized into *user-defined picture classes*. A user-defined picture class may contain several graphical representations of a single data structure, or several different objects of the same type. A class containing both a front and a top view of some object is an example of the former case; the class containing all of the Lisp Listeners is an example of the latter.

The interface performs no operations on the entire class. All operations on the class as a whole occur at the application level. The only operation which the interface itself performs on user-defined pictures is calling application routines to draw them. Any other operations which might be performed occur through the application system. The class organization is particularly useful in allowing the use of a single routine to draw any member of a class. This means that, for example, all Lisp Listeners could be drawn by a single routine, *i.e.* drawing is parameterized by class member.

A class may be either *static* or *dynamic*. The designer specifies the kind of class and the contents. For a static class, the designer specifies all of the members. For a dynamic class,

the designer provides information which enables the interface to access and draw the members of the class.

1.2. What the user can do

By definition, the user of an interactive system has one or more choices to make so that the application's action can be selectively controlled. In traditional non-graphical systems, the user is generally prompted for the next input and has no choice but to answer. It is possible that the prompt may be a menu asking what command to execute. Frequently, such a menu is *implicit* rather than *explicit*, that is, the command is entered and checked against a list of commands rather than chosen from a displayed list of commands. Most operating systems use implicit menus. However, the user must choose from that menu at that time and cannot do something entirely different. More recently, systems have attempted to be broader than this and allow the user to determine not only what command to execute next, but, in fact, from where the command is to be drawn. Frequently, the commands themselves are not explicit, as in a menu, but are implied by the action of the user. For example, completion of a form may imply execution of a routine processing that form. A UIMS system must be able to handle this latter case, with the user having multiple choices as to what input is to be entered next. Each of these choices is a *tool*. A tool is a technique for graphical interaction.

The set of tools available in GUIDE is pre-defined. Each tool has zero or more methods (routines) for initialization, drawing itself, echoing itself and processing input. A tool may also have a number of options controlling its appearance and performance. Some of the tools available in GUIDE are menus, lists, forms and potentiometers.

The designer may instantiate as many of each type of tool as desired. The designer chooses the characteristics for the instance which, in turn, determine which routines are used for drawing and echoing the tool and processing input to the tool. The designer may, if desired, provide the routines for drawing, echoing or processing a tool, although they must contain certain components, which will be discussed later.

It is not always desirable to enter the same kind of data using the same tool. Sometimes, one method may be preferable while, in another situation, another may be better. For example, sometimes a user wants to enter a number by typing it in and other times, a potentiometer may be preferred. The user's choice of tool may depend on such things as whether the user knows the desired value or is searching for it. It is also common for a user to need to enter the same value at different times. Having entered the value once, he should have the option of using the same value again without re-entering it.

Two of the ways in which systems can be consistent internally are by providing the same set of options for entering a value each time a value of that type is needed, and by using existing values either as defaults or as optional choices as much as possible. A UIMS should provide this capability of organizing tools for consistency.

GUIDE's *task* structure is designed to address exactly these issues. A task contains a number of tools, all of which can be used to enter the same value. For example, a "choose command" task might contain a menu tool and a keyboard tool. The ability to group tools into tasks simplifies the process of providing the same set of alternatives throughout a system. It also simplifies use of the value resulting from use of one of the tools. The task structure also allows for the possibility that entry of a value may be performed by choosing an existing value. A tool can be constructed which allows choosing from the set of existing values. A task may have a default value which is propagated to all of the tools within the task.

A small set of tasks is pre-defined, corresponding to frequently used real world tasks (e.g., `get_integer`). In addition, a designer may define as many tasks as desired. Each task contains one or more tools and, for each tool, must contain routines which convert between the output of the tool and the type desired by the task. Any instance of a predefined task may be edited to change the set of tools available either by adding or removing tools.

1.3. What happens

Within an application system, "what happens" refers to two things: what changes are made to the application data structures and "where do we go from here." A UIMS system must deal with both.

Since the designer must be able to specify alternatives for both cases, GUIDE provides for alternatives throughout by the use of *conditions*. Conditions are boolean expressions which may refer to the state of the system and the characteristics of the user. Conditions may refer to application variables. This provides a great deal of context sensitivity in the specification.

In order to modify application data structures, a UIMS must have a way of executing application-specific code. Several approaches have been used to do this. The most general approach is to allow the interface to invoke application routines with parameters based on user inputs and application variables. Since a program which does not use a UIMS can do this, a UIMS must at least do this if it is to be as useful. The application-specific procedures and functions which are invoked from the interface are known as *action routines*.

It is important that the UIMS not impose a particular style of interaction on the interface. One area in which this has been of paramount importance is in the specification of parameters to action routines. Existing UIMS's have not permitted full specification of parameters based on user inputs and application variables which are to be passed to application routines. This enforces an object-verb style of interaction, in which all of the "parameters" of an action must be made "current" by the user prior to invoking the action. (In English, this would be exemplified by "ball throw" instead of "throw ball.")

It is preferable to allow the designer to determine which items, if any, must be specified prior to selection of an action ("ball throw") and which can be specified after choosing the action ("throw ball"). This is done by permitting action routines to have parameters. The actual parameters for a routine can be specified as expressions and should permit reference to application variables and task values. In addition, the designer should be permitted to specify certain values which are to be obtained prior to computing the parameters. This allows the end-user to enter data which can be used in parameter computation after choosing the command. Prompting for values after the command has been specified corresponds to more traditional interactive systems. Of course, the designer may choose to use the object-verb form, or some combination of the two.

GUIDE provides this generality in specifying application actions. Each task may have one or more action routines associated with it. The action to be executed when a task is used depends on conditions specified by the designer. For example, a "choose command" task would likely have a separate action for each possible command.

Each routine may have parameters specified as expressions. Variables use in the expressions may refer both to task values and to application (*i.e.*, action routine) variables. The designer also may specify values to be collected prior to computing the parameters. More than one way of collecting inputs may be provided along with several ways of computing the parameters. In both cases, conditions are used to determine which option to select.

The second part of "what happens" is the specification of the control path of the system following execution of an application action. The ideal situation is to allow the designer to specify several possibilities depending on the state of the application and the characteristics of the user. That is, the control path may vary based on what has already happened and who is using the system.

The first thing which is needed in order to specify the control path is some kind of object

which indicates the state of the system at some point in time. That is, we need a structure which contains, for some state of the system, the three components under discussion (what the user sees, what the user can do, and what happens). The term used in GUIDE for such a structure is *context*.

Within each context, the designer can specify user-defined picture classes, tasks, actions and *control*. Control is specified by indicating a transfer to another (or the same) context. A stack of contexts is maintained. A transfer may indicate pushing or popping the stack, or may not change the stack. The stacking process allows for interruption of one command sequence by another.

Control is specified by inclusion of *decisions* which determine the new state of the system after an input from the end-user. A decision contains conditions plus instructions on the change in control if the conditions are met. The designer specifies the priority of the decision and the first decision to have its conditions fulfilled will be chosen. A null condition may be expressed, forcing the change in control to occur. This allows the designer to specify an unconditional change of context. The default, if all conditions fail, is to remain in the same context.

The mechanism for varying the control path and other features based on characteristics of the user is the *user profile*. The profile contains information about the user's personal characteristics and preferences. The values of these items for the specific user may be used in determining the control path. While it is easier to define the user profile as a fixed object, this reduces its usefulness. Instead, the designer should be permitted to specify the contents of the user profile, determining what factors differentiate the various users of the system. Similarly, allowing the designer to use any types for the fields provides additional generality. The designer may choose the representation which best distinguishes among users with different characteristics. Conditions used in the control path and at other locations may refer to fields of the user profile as well as task values and application variables.

The user profile in GUIDE may contain any information about the user which the designer chooses. Some items which might be included are the user's *skill level* (skill with the system) and *access level* (right to access information in the system). Other contents depend on the nature of the system being implemented, but will generally include the user's preferences in dealing with the application system.

The designer specifies the method whereby user profile fields receive initial values and are updated. Any user can access, at most, only his own profile. Access to each field in the

profile may be controlled by conditions based on the profile. In systems where security is an issue, access to some fields may be prohibited. For such systems, a separate system or sub-system must be provided to maintain the user profile data base.

1.4. Responding to Application States

The underlying goal for a UIMS generator is that the designer should be able to express as general an interface as by manual code generation. In particular, the interface should be able to respond to changes in the application world. It is possible to provide this kind of responsiveness by creating a large number of interface states corresponding to all of the possible application states. However, for any significant application, the number of states necessary to encode this information is prohibitive. A better approach to this problem is to allow the interface to refer to application variables in decision-making, so a single interface state can serve as front-end to a number of application states.

This latter approach is the one taken by GUIDE. The designer provides a file of application entity declarations to GUIDE. These are parsed to create a symbol table. Expressions and conditions in GUIDE are parsed using this symbol table and can therefore refer to application entities. In addition to the application objects, the symbol table contains a number of entities such as types and functions defined by GUIDE which are useful in most, if not all, applications. These entities can also be referenced in expressions and conditions.

1.5. Other Considerations

A good help system is an integral part of an interface. In order to have a good help system, the designer must provide *help messages* for a large class of objects. The design system must encourage the provision of these messages. In GUIDE, helps may be provided for tools, tasks and contexts. The help message associated with an object is created when the object itself is created. As with all other items, it may be edited freely. For each object, the designer may specify both a brief message and the name of a file containing a longer message. In addition, each object may have several help messages distinguished by conditions, allowing messages to be geared to individual users.

A *help task* is provided containing tools for triggering the help system. The designer may edit this task to eliminate any tools that he feels are inappropriate for the system.

The designer may also associate *prompt messages* with each context and tool. As with

helps, several prompts distinguished by conditions may be associated with each object. Prompts are displayed when the object with which they are associated is displayed.

The UIMS needs a way of handling semantic messages [Tanner 84]. If, for example, an input causes errors in the application routines, those routines must be able to inform the user of the error. Since those routines are not permitted to do any interactive output themselves, the UIMS needs a message system in which the application routines can signal an error and have the interface inform the user at the appropriate time. The designer also needs to be able to make control decisions based on whether or not an error has occurred.

In GUIDE, these messages could be passed through variables accessible to both GUIDE and the application. However, since this capability is likely to be necessary in almost every UIMS created, a message passing system is provided by GUIDE.

GUIDE maintains a list of messages. Each message consists of a number, a message text and a location. The designer may decide whether the number is to be used for an message code, a severity code or something else entirely. The message text is intended to describe the message and the location is intended to tell where the message occurred. GUIDE's runtime support package contains routines to clear the message list, to add a message to the list, to check for messages with various characteristics (first, last, highest, lowest) and to provide message reports.

The application routines may use these routines to signal errors and behave appropriately when an error occurs. The designer may check for the occurrence of messages in conditions. In the interface generated by GUIDE, the message list is cleared at the start of each context and messages are reported and cleared following each command sequence.

1.6. Output of a UIMS generator

Once a designer has specified what the interface to his system is to look like, the UIMS generator must do something with the specification to create the interface. In general, a generator can take either of two approaches: it may generate code in some programming language or it may generate tables to be used by a driver. It is also possible to use some combination of the two techniques: generate some code and some data. This is the approach taken by GUIDE. This hybrid technique allows the use of stored data where it is efficient, but generates code for most of the system. The amount of code needed for the driver is minimized.

The choice of generating code to provide a compiled interface rather than generating data to be used by an interface interpreter was based on several considerations. Most of these had to do with allowing as much generality as possible at a low cost to the designer.

To allow access to application routines in an interpreted system, the designer would have to provide a routine which invokes each accessible routine with appropriate parameters. Passing parameters would be far more complicated and it is likely that each routine would be restricted to a single set of actual parameters. Variable parameters would be able to be specified only by the designer's putting them into calls in the invocation routine. The code generation approach, on the other hand, allows multiple calls to any routine with as many different parameter sets as desired. It is no problem to specify both value and variable parameters. The only action required of the designer is to provide the headers of all routines used, so type checking can be performed on the actual parameters. This requirement is likely to be necessary in an interpreted system as well.

The second area in which the compiled approach is superior to the interpreted is in allowing variables to be accessible both to the application and the interface. In an interpreted system, interface access to application variables would require maintenance of a symbol table at run-time. This symbol table would need knowledge of the underlying programming language of the application routines in order to access variables when referenced. In the compiled approach, references to variables in the interface are resolved by the compiler and linker, system-supplied facilities. No symbol table is necessary at run-time.

The compiled approach also greatly simplifies the evaluation of expressions and conditions used in the interface. The generated code is simply invoked at the appropriate time, again taking advantage of system services. In an interpreted system, the driver would need to include code to evaluate expressions, which would be represented in some intermediate form. In addition, as above, a symbol table would be needed to provide access to variables.

The last advantage of generating code is that the interface can be modified directly as well as through GUIDE. Although caution is needed in making such changes, the designer could, for example, change the name of an application routine to be invoked or change an actual parameter. This is useful if very small changes are quickly needed.

In addition to generating the interface code, a UIMS generator must be able to store a design so that it may be developed over several sessions or modified at a later date.

The output from GUIDE consists of a number of files. The designer may request that his

design be stored in a file for later examination or modification. A journal file containing a complete record of each session using GUIDE is also created. Various errors may occur in the interface design; error messages will be stored in a third file. Lastly, the major output of GUIDE consists of Pascal code and data which can be linked with the designer's code and the code implementing the tools to form the complete application system.

1.7. GUIDE's interface

Another question that arises in defining a UIMS generator system is how the designer is to specify the design from which the interface will be generated. The most attractive solution is to have an interactive, graphical interface to the generator. The ability to use GUIDE to provide such an interface is one of the goals of GUIDE. This has been achieved by using the action routines for GUIDE as a subroutine package to build a design and generate the appropriate interface. In fact, this is how the prototype interface to GUIDE was created. The graphical and interactive components of GUIDE have been separated from the application code which operates on the data structures. This is the structure which a UIMS is intended to promote.

A prototype user interface for GUIDE has been designed using GUIDE itself. The prototype system is primarily menu-driven, with extensive use of forms, especially for instantiation of tools, tasks and other objects. Some information may be entered by picking items from the display.

The methods for creating objects (*e.g.*, tools, tasks) and editing them are substantially identical. For most objects in GUIDE, the user is presented with a form containing the current values for that object. Any desired changes can then be made. When the object is initially created, many of the fields will be empty and require input. With this technique, no commands specific to editing need to be learned.

It is hoped that a better interface for GUIDE, which takes more advantage of GUIDE's flexibility, will be designed in the near future. The designer of the new interface will use the prototype version, both as a starting point for the next version and to access GUIDE.

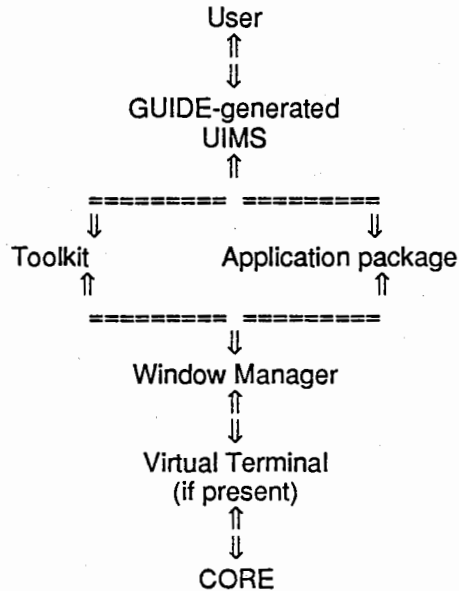


Figure 1-1: Control Flow using GUIDE

1.8. The generated interface

GUIDE generates code and data which, when combined with the application code and some runtime support, forms the application system. In doing so, it is analogous to a compiler-compiler (*e.g.*, [Johnson 75]). The user of a compiler-compiler provides a grammar which is processed, and semantic routines. The compiler-compiler takes the grammar and produces as output the tables needed to compile programs. The output is used with the semantic routines by an end-user (to compile his program). With GUIDE, the user (designer) provides a description of the desired interface which is processed to produce the interface itself. This interface is then used, together with other code provided by the designer (the semantic routines), by the end-user. However, the description of the interface is developed interactively, rather than specified as a grammar (as in [Olsen 83b]).

The Graphical Input Interaction Techniques Workshop [Thomas 83] characterized UIMS's along three scales: internal vs. external control, single- vs. multi-thread control and simple vs. hierarchical dialogues.

GUIDE generates external control UIMS's, meaning that the UIMS is in control and invokes the application as a slave. It permits multiple threads of control, allowing the user to

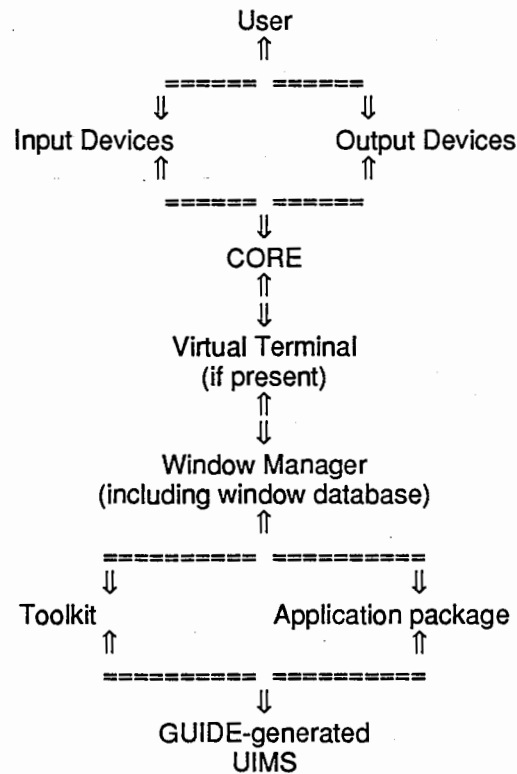


Figure 1-2: Data Flow in a GUIDE-generated Interface

have several commands under construction at the same time. GUIDE provides hierarchical tools, thereby allowing hierarchical dialogues.

The control hierarchy for a system with a GUIDE-generated interface is shown in Figure 1-1. The presence of a window manager is assumed; it is expected to handle the details of placing the viewports (*windows*) on the screen and identifying viewports when a *pick* is made.

The user is in control of the system. When he does something (*i.e.*, causes an event), the GUIDE-generated interface is activated. It invokes the toolkit to determine what the user did. The toolkit, in turn, invokes the window manager and so forth until the appropriate hardware driver is queried. The interface can also invoke the application package, which has access to the window manager. GUIDE uses the CORE graphics system [GSPC 79] to perform its own drawing and assumes that the application uses CORE for its drawing. The GUIDE methodology does not depend on CORE, but the low-level graphics routines use the CORE system to handle input.

The flow of data in a system using GUIDE is shown in Figure 1-2. Data goes from the user to an input device, triggering a CORE event. CORE, in turn, informs the virtual terminal, which passes the information to the window manager and so forth until the GUIDE-generated interface is reached. For output, the interface invokes either the toolkit or the application package, which pass the data down the hierarchy until it is displayed.

1.9. Organization

Throughout this dissertation, examples will be drawn from the furniture layout system described by Foley and van Dam [Foley 82], which consists of manipulating a pre-defined set of symbols representing furniture (desk, chair, *etc.*) until a satisfactory layout is achieved. The user may add symbols to or delete symbols from the layout, may title or re-draw the current layout and may change the window into the layout world. Other examples will be cited as needed.

The remainder of this document is organized as follows. Chapter 2 discusses other research in the area of user interface specifications and looks at some existing systems in GUIDE's terms. Chapter 3 discusses tools and tasks. Contexts, decisions, expressions and conditions are covered in Chapter 4. Actions and parameter specification are discussed in Chapter 5. Chapter 6 discusses user profiles, helps, prompts and message handling. Chapter 7 covers the GUIDE user interface. Chapter 8 discusses the output produced by GUIDE and Chapter 9 covers implementation issues. Conclusions, contributions and future directions for this research are discussed in Chapter 10.

CHAPTER II

Survey of Other Research

A number of researchers have studied the area of interface specification. The approach taken by most [Hanau 80, Hayes 85, Kamran 83, Kasik 82, Olsen 83a, Olsen 83b, Schuler 85] has been to design languages which can be used to specify either the interface of a system or the entire system. A few have approached interface design from the tool level [Anson 82, Green 82, Van den Bos 78]. Several researchers [Bass 85, Heffler 82] have designed interactive systems for specifying interfaces, but these systems cannot handle applications using graphical input. A few systems have been implemented which allow interactive design of graphical user interfaces [Buxton 83a, Green 85a, Roach 82, Rubel 82, Wong 82]. None of these systems, however, allows as much freedom to the designer as GUIDE.

Since the introduction of user interface management systems, attempts have been made to classify them [Kasik 84] and to provide models [Seeheim 84]. In particular, the Seeheim Workshop on User Interface Management provided a comprehensive model of UIMS's. This model divides a UIMS into three components: presentation, dialogue control and application interface model. The presentation component controls the external appearance of the interface. The dialogue control component interprets input and invokes application routines. It also converts data from the application to a form understandable by the user. Control logic is contained in this component. One system [Green 85b, Green 85a] has been implemented which conforms to this model.

The next section discusses a collection of characteristics either necessary or useful to an ideal UIMS system. The characteristics are divided into four categories: items relating to specification of semantics, that is, what happens when the user does something, items relating to interaction tools, items relating to user support services and items relating to specification of the interface, that is, support for the designer. Those items relating to semantics are considered necessary for the ideal UIMS generator. Items in the other groups are useful. Table 2-1 examines existing UIMS systems and shows which of these characteristics each possesses. Items marked with an 'L' in this table are limited in the particular system and are discussed in the text. The tool-based systems are omitted from the table as none of them actually constitutes a UIMS system. The sections which follow summarize some of the existing systems

and evaluate them in terms of the amount of generality which can be expressed in the interface. The last section in this chapter examines some existing interactive systems and attempts to describe them in GUIDE's terms.

2.1. Characteristics of an Ideal UIMS

2.1.1. Semantic Considerations

1. Allows any semantic specification - A UIMS system must be able to specify the actions and control structure of the application. The remaining items in this group are specific parts of semantic specification.
2. Provides access to application variables - As discussed in chapter 1, the ability to refer to application variables in decision-making is crucial to specifying an interface in a reasonable number of states.
3. Allows choice among multiple control paths following a user action and execution of application routine - If a system is to respond to user actions and their results, the designer must be able to specify selection among several states to follow any given state. Such factors as the user's choice of command, the result of the application action and the state of the application all may be used to determine the new state.
4. Provides access to application routines - The UIMS must allow execution of routines which perform the work of the application.
5. Allows choice among several application routines following a user action - The application action which follows a user action may depend on such factors as the state of the application, the characteristics of the user and the results of previous user actions. The designer should be able to specify such distinctions without factoring his command set to require distinct user actions for each case.
6. Allows parameters to be passed to application routines - The application routines must have access to user inputs. In an external control UIMS, this access must be provided without explicit requests from the application.

7. Allows choice among parameter sets to be passed to application routines - Again, the set of parameters to be passed may depend on a number of factors like those mentioned above. The UIMS must allow the designer to distinguish among possible parameter lists.
8. Allows collection of values for parameter computation - The UIMS should not constrain the designer to any one interaction style, in particular, the parameter-verb style. The designer must be able to indicate, for each application action, what user inputs are necessary in order to perform the action.
9. Allows choice among methods of collecting values for parameter computation - As with the other semantic items, the list of user inputs needed for parameter computation may vary. The designer must be able to specify the various cases and the conditions controlling each of them.

2.1.2. Interaction Considerations

10. Allows at least menu, form filling, direct manipulation and command language interaction techniques - These tools provide a minimal set of operations for interaction. The absence of any one of these makes some simple interactions difficult to perform.
11. Allows a single interaction task to be implemented by multiple techniques (tools) - The ability to represent a single task by several tools makes it easier for the designer to provide alternatives to the user and to provide consistency throughout an interface.
12. Allows non-hierarchical menus - In many applications, some commands should be available in more than one set of commands. A hierarchical menu system constrains each item to appear in a single set.
13. Allows optional items in menus - Option items allow a menu to be sensitive to the kinds of factors discussed under semantic considerations.
14. Allows optional fields in forms - Optional fields in forms also allow sensitivity to these factors. In addition, they allow a form to develop as it is filled in, that is, the response to an item can determine whether or not some other items appear.

15. Creates a graphical interface - The availability of graphical tools expands the range of applications for which a UIMS is useful.

2.1.3. Support Services

16. Allows choice among several helps and prompts for an object depending on user and/or system state - Helps and prompts are another area in which the system's behavior may depend on factors like the user's characteristics and the state of the application.
17. Allows specification of a user profile to distinguish among users - An interface is far more useful if it can respond to each user as an individual instead of treating all users the same way. A user profile described by the designer allows the system to make distinctions among users based on factors appropriate to the application.

2.1.4. Specification Services

18. Interface may be designed interactively - Interactive specification of the interface avoids the necessity of learning a special-purpose language.
19. Specification package contains an expression parser - The inclusion of a parser to check expressions for syntactic correctness allows errors to be corrected when they occur and reduces the number of errors in the generated interface.
20. Interface may be edited after testing - First attempts at interfaces generally do not result in ideal interfaces. The ability to try an interface and then modify it allows the designer to incorporate feedback from users as well as correct any errors with a minimal effort. It is interesting to note that this is the only characteristic possessed by all of the UIMS systems.

	GUIDE	Olsen	AIH	COUSIN	ADM	TIGER	IDS
1. Semantics	X	X	X		X	X	X
2. Variables	X						
3. Multiple paths	X			N/A			
4. Appl. Routines	X	X	X	N/A	X	X	
5. Choice of Rtns	X		X	N/A			
6. Pass Params	X	X	X	N/A	?	X	
7. Choose Params	X			N/A	?	?	
8. Collect Params	X		X	N/A	?	X	
9. Mult. Collect	X			N/A	?		
10. Minimal tools	X			X	X		
11. 2-level input	X		X		X		
12. Menus random	X						
13. Opt. menus	X				X	X	
14. Opt. fields	X						
15. Graphical Int	X	X	X		X	X	X
16. Helps/Prompts	X		L				
17. User profile	X		L				
18. Interactive	X		X				
19. Parser	X						
20. Editing	X	X	X	X	X	X	X

	Bass	Heffler	DMS	Alberta	MENULAY	BLOX	FLAIR
1. Semantics		X	X	X	X	X	L
2. Variables							
3. Multiple paths			X				
4. Appl. Routines		X	X	X	X	X	
5. Choice of Rtns			?	X			
6. Pass Params		X	?	X	L	X	
7. Choose Params			?				
8. Collect Params		X					
9. Mult. Collect							
10. Minimal tools			?	?			
11. 2-level input							
12. Menus random		L			X		
13. Opt. menus			?				X
14. Opt. fields	X		?				
15. Graphical Int			X	X	X	X	X
16. Helps/Prompts							
17. User profile					L		
18. Interactive	X	X	X	L	X	X	X
19. Parser							L
20. Editing	X	X	X	X	X	X	X

Key:

X - system has this feature

? - unknown whether system has this feature

L - system has this feature in a limited form

Table 2-1: Characteristics of UIMS Systems

2.2. Interaction languages

Olsen [Olsen 83a, Olsen 83b] describes a system called SYNGRAPH which generates graphical user interfaces from BNF-like descriptions of the input language's grammar. The interface grammar is automatically analyzed to provide "menu management, simulated device management, prompting, echoing, error handling, backing up over erroneous inputs and canceling a sequence of interactive inputs to return to a known home state." Semantic specification is extremely limited. This system does allow use of Pascal statements by including these statements in the grammatical specification. Parameters may be passed. However, no provision is made for collecting the values needed for parameters at that time. The set of tools is defined only at the device level and does not include form filling, which is a higher-level tool. User services do not include alternate helps and prompts. The system also does not allow specification of distinct paths based on user characteristics. The technique described for graphical rubout is very similar to that proposed for the abort command in GUIDE. Specification of the interface is not performed interactively, although the grammar may be edited after testing the interface. SYNGRAPH's applicability is limited as the inability to specify collection of inputs for parameters restricts designers to the parameter-verb approach. For some applications, this would make an interface extremely difficult to use.

A group at George Washington University [Kamran 83, Feldman 82] has implemented a UIMS called the Abstract Interaction Handler (AIH). This system includes an interaction language along with a program generator for this language. This language is based on transition networks and is used for describing the syntactic rules of the dialogue. The system also includes an interpreter for the interaction language, which activates application modules and passes inputs from the user to the application modules, a *Screen Handler* which imposes additional structure on CORE segments and *Style Modules* which are used to enforce uniformity on the front end of an application. A great deal of semantic specification is available; however, no access to application variables is provided. In addition, AIH associates a single control path with each command and apparently allows a single specification of parameters for the action. The basic unit of interaction in AIH is an *interaction task* which is defined as "a way of using a physical input device to input a certain type of word coupled with the simplest forms of feedback." This corresponds fairly closely to GUIDE's notion of a *tool*. Menus provided by AIH are hierarchical only. There is no form-filling. The style modules are similar to GUIDE's user profiles, but are less powerful in that they influence only the appearance of the interaction and not the control flow. They also affect the helps and prompts provided by the interface. Like GUIDE, AIH allows a choice of several interaction techniques ("tools") to be used to carry out a task. The interface may be specified interactively and may be edited after testing. The lack of

access to application variables combined with a single control path for each command means that the number of states needed to specify a system of any size would be unreasonably high.

The COUSIN system at Carnegie-Mellon [Hayes 85] is a UIMS for personal workstations. The designer of an interface describes it using an attribute/value notation. The description contains slots, which correspond roughly to GUIDE's tasks. Each slot contains a single interaction mode. The interaction modes correspond roughly to GUIDE's tools, although the set is quite different since COUSIN is not a graphics-based system. The interfaces created using COUSIN are not of the external control variety created by most of the UIMS systems, including GUIDE. COUSIN uses a mixed strategy, which requires the application to contain the control code and to explicitly request values from the UIMS. This means that the interface specification contains no semantics. Application actions are invoked by the application and parameters are requested from the interface by procedure calls. Therefore, the application code requires a great deal more information about the interface in COUSIN than in GUIDE and other external control UIMS's. Although the interfaces created with COUSIN are not graphical, the set of tools is quite broad. Tools are available which can be made to perform as each of the basic tools. Neither of the support services is available. The interface is not specified interactively, but can be edited. COUSIN is limited in applicability by its lack of graphical capabilities and by the requirement that the application make action calls and retrieve parameters. The latter requirement means that application programmers must be aware of interface terminology rather than writing code which is independent of the interface.

ADM [Schulert 85] authored by Schulert, Rogers and Hamilton has many similarities to GUIDE. Interaction is specified in terms of tasks and techniques, with ADM's tasks similar to GUIDE's tasks and ADM's techniques analogous to GUIDE's tools. Tasks are organized into states, which are similar to GUIDE's contexts. Semantic specification is limited. A task may call an application routine when it is used. No provision is made for conditional choice of application routines and it is unclear whether parameters may be collected and passed. No conditions are permitted in specifying transitions between states. The methods provided for interacting in ADM are quite general. All of the basic tool types appear to be available along with several others. Menus, however, have only a single level. ADM provides graphical interfaces through access to a graphics package. The ability to get help is provided, but with no alternate selections. A single task may contain several techniques as in GUIDE. ADM has no notion of a user profile; in order to provide distinctions between users, multiple distinct interfaces must be created. To specify an interface with ADM, the designer writes a description in a special language provided for this purpose. This description may be edited. As with AIH, the designer using ADM is likely to need an extremely large number of states to represent a system of any size, since there is no conditional transition to states.

Kasik's UIMS [Kasik 82] called TIGER provides a dialogue specification language (TICCL) and an interpreter which takes the compiled output from TICCL and displays all command sequences to the user. TICCL is used to "define and organize interactive dialogue sequences." The runtime interpreter handles all input and output for the interactive dialogue. It accepts the compiled TICCL as input. All issues of form, position and style are handled by the interpreter. Processing is interrupt-driven. An interrupt may invoke an application routine and parameters may be passed. There is no indication that multiple parameter sets may be specified or that multiple methods of collecting parameters may be provided. Control provided by TICCL is hierarchical, although the user may choose a command at a higher level, execute it and then return to the original position. Control differentiated by conditions is not provided. The interface has no access to application variables. TIGER provides graphical interfaces; however, the set of tools available is limited. Support services such as alternate helps and prompts and a user profile are not present. The interface is specified by writing a TICCL program which may be edited. Like AIH and ADM, TIGER's usefulness is limited by the inability of the designer to specify multiple control paths and the lack of access to application variables. In any significant application, states will proliferate unreasonably.

One of the earliest efforts in interactive system generators was a system called IDS (Interactive Dialogue Synthesizer) by Hanau and Lenorovitz [Hanau 80]. This system takes a BNF-like description of the interface and parses it to produce the interface. Semantic actions are specified as literal strings which can be either printed out or executed in a reverse Polish notation. Calls to application routines are not possible. No method is provided for specifying control paths based on conditions, and no access to application variables is provided. A large collection of tools is provided, although forms cannot be specified. Alternate helps and prompts cannot be provided. No distinctions are drawn among users. The interface description is not processed interactively, but can be edited. IDS contains many interesting features, but reflects its early entry into the field with its extremely limited semantics. The inability to invoke application routines means that only simple semantics (those which can be expressed in the RPN provided) can be specified.

In addition to the limitations described for each system, the necessity of writing a program, grammar or declaration for the interface, implicit in the language-based systems, would seem to limit their applicability to smaller systems. It seems likely that writing the appropriate description for a large system will turn out to be almost as difficult and error-prone as writing traditional interface code for the same size system, using a tool library. For smaller systems, the language-based approach provides an improvement over traditional methods.

2.3. Tool-oriented systems

Several researchers have attempted to define user interfaces in terms of the tools being used. None of these systems can be used to completely specify a user interface, as none of them include any semantic specification.

Green [Green 83, Green 82] has a system for prototyping user interfaces by putting together *building blocks* from a catalogue of interaction techniques. Each building block can represent either input, output, interaction or control. Each building block can have input and output connections that allow data to flow between blocks. This system provides no connections for the application and is intended only as a prototyping tool.

Anson [Anson 82] provides a language to describe interaction in terms of devices. Devices can be hierarchically defined and, therefore, entire systems can be described using this language. As of 1982, no compiler for this language had been implemented. This language describes only the interactions between the user and the devices. It does not allow specification of application pictures or control. It corresponds to the task portion of GUIDE only.

Van den Bos [Van den Bos 78] devised a notation for defining higher-level tools in terms of more primitive tools. As in Anson's system, this notation handles only the task side of the interaction.

2.4. Non-Graphics Systems

Bass [Bass 85] describes a system for defining and displaying forms on a screen. This system allows the user to divide the display into components and provides a full-screen editor for laying out the components and specifying relationships among components. The system, like GUIDE, allows the use of conditions to determine the contents of a form. This system deals only with the syntactic and lexical levels of the interface and leaves all semantic issues to the application. Keyboard-based forms are the only tool provided. Like the tool-based systems, this system cannot be used to specify a complete interface, due to its lack of semantic specification.

A system for interactively creating menu systems has been implemented by Heffler [Heffler 82]. This system is designed for use with non-graphical terminals and does not incorporate any devices other than a keyboard. The system creates a set of tables used by a run-time driver. Some semantic specification is provided. When executing the generated

menus, the system can prompt for a list of parameters for an action to be invoked. However, the parameters are passed through pointers to character strings, which limits both the kind of parameters and the types of parameters which can be passed. Control flow is fixed and the designer is not allowed to specify the control path based on the result of an action. Interaction occurs only through menus and traditional prompting. Any menu in this system may be followed by any other; however, the items in a menu belong only to that menu. Therefore, if the same item is desired in more than one menu, it must be defined each time. Help is integrated into this system. No distinctions are drawn among users. The menu specifications are entered interactively and may be edited. This system is limited in applicability by its limited semantics. The fixed control flow makes many interaction sequences difficult to specify, while the limitation on parameters constrains the actions which can be specified.

2.5. Other systems

The University of Alberta UIMS [Green 85b] is implemented in conformance with the model described at the Seeheim Workshop on User Interface Management [Seeheim 84]. In the Alberta system, the dialogue can be defined using an event language, a recursive transition network or a context-free grammar. The internal representation of dialogues is event-based. The application interface model determines how the application is viewed by the interface and controls the interface's access to the application structures. The Alberta system allows a moderate amount of semantic specification. Although application routines can be invoked, there is no indication that parameters can be passed other than the token or tokens which caused the call. There is no access to application variables and only a single control path. The Alberta UIMS creates graphical interfaces; it is unclear what tools are provided other than menus. No user profile is included and alternate helps and prompts are not available. Only the presentation component can be defined interactively, but all three components may be edited. The Alberta UIMS represents a first attempt to fulfill the Seeheim model. As such, it is a major step. However, to make an ideal UIMS, the application interface component needs expansion to allow access to application variables and the dialogue component should allow more control specification. The absence of these features is likely to cause a proliferation of states in the control structure.

DMS, designed at Virginia Polytechnic Institute [Roach 82, Hartson 84], divides an application into three components: control, dialogue and computation. The dialogue component handles input and output. The computational component contains the application code. The control component supervises the order in which dialogue and computation occur. DMS

provides a Graphical Programming Language (GPL) which is used to describe the control structure of the system. A graphical editor is provided for GPL. While application routines can be invoked and parameters passed, the authors do not state whether any options are permitted in choosing the action routines and parameters. They also do not state whether collection of values after action selection is permitted. A package called AIDE (Author's Interactive Dialogue Environment) allows the designer to specify the dialogue component using a small set of tools. No user profile is provided and no alternatives may be provided for helps and prompts. The interface is specified interactively. DMS provides a Behavioral Demonstrator which allows the execution of the control structure even prior to its completion. The application code need not be present. Following such a test, the interface may be edited. Although DMS does allow several control paths following an action, the inability to access application variables will cause an unreasonable number of states to be needed for many applications,

Buxton and his colleagues at the University of Toronto [Buxton 83a, Buxton 83b] have implemented a UIMS, called MENULAY, which allows interactive layout of menus. MENULAY generates a C program which can be combined with the designer's routines to form the complete system. Some semantic specification is provided. Menu items may be associated with application functions, as in GUIDE. Each menu item may be associated with a single action. Parameters may be passed to action routines, but only a single set may be specified. There is no provision for collecting values to compute parameters. MENULAY provides only menu-based interactions and a small set of interaction tools. Menus may be arbitrarily structured. MENULAY distinguishes among its users based on skill, but there is no indication that the systems so created can use skill considerations. MENULAY interfaces are created interactively and may be edited. MENULAY is similar to many of the other systems in that its limited semantic specification is likely to make it difficult for designers to specify larger systems.

Rubel [Rubel 82] has a commercially-available system called BLOX which can be used to interactively specify an interface based on finite state machines. The system then builds the tables needed for the interaction. The designer may use pre-programmed functions or provide his own application routines. Although application routines may be invoked, only a single set of parameters may be specified. Control flow is not based on the result of actions and there is no access to application variables. BLOX provides graphics with a two-dimensional CORE system. The set of tools does not include forms. Help is provided, but no alternative messages may be specified. The interface is specified interactively and may be edited. BLOX has limited applicability due primarily to its weak semantic specification. As with other systems, specification of large systems is likely to be hampered by the limited control flow and lack of access to application variables.

FLAIR [Wong 82] provides a *Dialog Design Language* which is a "menu-driven system that aids and directs the designer through a coherent and orderly translation of his scenario into a form that is executable." FLAIR provides access to a large set of devices, but has no interaction with any other programming language. Designers must use pre-programmed actions. FLAIR, intended primarily as a prototyping system, would seem to be somewhere in between a UIMS and a user interface. It allows the user to define scenarios dynamically and to create menus, but these can apparently be used only within the entire FLAIR system. The additions are entered interactively and may be edited. A built-in calculator utility allows the designer/user to perform any necessary computations while specifying his scenario. FLAIR's usefulness is limited by the fact that it cannot create new interfaces, only make additions to an existing package.

2.6. Summary

A number of the systems described have proven to be useful to their implementors. However, none of these contains all of the features considered necessary for the "ideal" UIMS generator. Only GUIDE has all of these features and it is the only one to provide access to application variables in the interface.

It also should be noted that, while it was not designed with the Seeheim model [Seeheim 84] in mind, GUIDE does contain all of the components in the model. The boundaries between the components are not as clean as in systems defined to conform to the model. GUIDE's tools and user-defined pictures, together with the run-time tool drawing code, constitute the presentation component. The dialogue control component is composed of the task, context, decision and action structures plus the run-time tool processing code. The symbol table constructed from the application environment provides the application interface model.

2.7. Studies

In order to determine what features to incorporate into GUIDE, several existing interactive systems were chosen for their diversity and studied. The goal was to determine what kinds of features are common to interaction and to see whether the initial model for GUIDE could handle a wide range of interactive systems. Several modifications to GUIDE's design were direct results of these studies.

At present, as discussed in section 9.3, GUIDE cannot track locators and pick tools with anything other than a tracking cross. This limitation is a function of the underlying graphics package and will not be considered in the following discussion.

The systems examined were the Benesh notation editor of Singh [Singh 84], GRIP-75 [University of North Carolina 81], the Spatial Data Management System [Herot 84], Apple's LISA computer and the Symbolics Lisp Machine. The first three are application systems while the last two are operating systems. The following sections describe each of the systems and discuss how each could be implemented in GUIDE. It should be noted that a system, implemented using GUIDE, may not be identical to the existing version. This section explores whether or not GUIDE can perform the kinds of interactions necessary to define these systems.

2.7.1. Benesh Notation System

The Benesh Notation Editor [Singh 84] is an interactive system for creating and modifying dance scores in Benesh Movement Notation. The user interactively modifies a current frame which describes a single set of motions. The user may also edit the entire score of frames to insert, delete or re-organize frames. The system is primarily menu-driven with several special purpose graphically-based tools.

The Benesh system contains a menu of commands which can be selected by the user. It also contains a menu which is composed of the human body divided into various regions. The current body part being specified is selected from this menu. Choice of an item from this menu implicitly executes the command "position body part" for the chosen body part. Both of these types of menus (*i.e.*, command list vs. pick part) can be represented in GUIDE. Echoing for both of these menus is performed by shading the chosen item.

Various symbols can also be positioned in the current frame by selecting appropriate commands. The choices of symbols for each command can be represented in GUIDE by lists.

Positioning is done with a tablet puck. As the user moves the puck on the tablet, a symbol representing the body part being positioned is moved in the current frame. The symbol displayed always shows what the frame would look like if the position of the puck were selected. The designer could provide a routine to choose the appropriate symbol and draw it. Then, GUIDE could use such a routine to echo the movement of the puck.

The specification of head and torso positions is somewhat more complicated than that for other body parts. The user must select the head or the torso from the body menu, and then manipulate an icon with the puck until the desired position of the head or torso is found. Then, the appropriate symbol is generated (by the action routine) and the user places it in the right location in the frame, as with any other body part. In GUIDE, the selection of the head or torso

would place the user in a context for orienting that body part. When that is done, an action routine to create the appropriate object would be invoked. Control would then flow to a context for positioning the head or torso in the frame. Following this context, control would return to the main context.

Modification of the score can be handled very simply by GUIDE. A window controller (see section 3.3.4) should be provided to allow the user to see all or part of the score. The user may select one or more frames to make them current and then select the desired command from the menu. To make more than one frame current at the same time requires the use of one task to initiate the operation and a second to terminate it.

2.7.2. Grip-75

The Grip-75 system [University of North Carolina 81] is an interactive system for building molecular models. It is designed for use by crystallographers who are working from a graphical representation of the electron density of the molecule. The molecule under construction is displayed and the user may manipulate it with a collection of tools until a satisfactory placement is achieved. A command menu is provided to initiate actions, but within an action, menus are used as little as possible. Commands are orthogonal so that the user may construct more than one command sequence at a time. A light pen is used for picking sub-sections of the molecule and choosing commands from the menu. Some commands are implied by the use of particular tools, *e.g.*, special devices are provided for translation and rotation of molecular sub-sections.

When the user selects a molecular sub-section, its "twistable bonds" are automatically linked to a set of dials. Then rotation of these dials implies rotation of the corresponding bond in the molecule. The user may change viewpoint by dials or joysticks.

Grip-75 can be modelled with GUIDE. Selection of a command from the main menu changes the context so that subsequent actions can be interpreted with respect to the selected command.

The most interesting problem in modelling Grip-75 is the linkage of twistable bonds to dials. We must deal with the strong possibility that the number of bonds does not equal the number of dials. If there are fewer bonds, this is not a serious problem. We only need to decide whether some bonds should be linked to more than one dial. If there are more bonds than dials, however, we must decide which bonds get linked and how to access those which are not linked. The action routine for the "select sub-image" command is responsible for returning a

list of the bonds. One way to handle the problem is to require that routine to return the bonds in a priority order. Then, we can also provide a command to switch between the main set of linkages and a subset. An alternative method to handle this problem is by having the user pick the bond to be twisted before using the dial. Then, any dial can be used for any bond. The physical dials can be associated with valuator with button tools.

Grip-75 also permits the user to indicate his preferred method for stereo viewing. This choice can be maintained in the user profile.

2.7.3. Spatial Data Management System

The Spatial Data Management System (SDMS) [Herot 84] is an interactive interface to a database management system. It uses three screens to allow a user to examine a collection of databases. One screen provides a *world view map*, showing what items are contained in the world. The second screen provides a close-up of the current position in the world. The third screen is used for an interactive graphical editor. A joystick allows the user to move around within the world. The middle screen always contains a close-up of the *current* part of the world. The user can zoom in and out on both the first and second screens by twisting a dial on the joystick. The user may also touch a point on a screen to make the object displayed there current.

The keyboard is used for typing in queries to the DBMS. The commands which can be typed allow visual response from a command, for example, "blink."

The user can choose among several methods of displaying the contents of a database. A set of templates is used to generate icons for entries in the database. The user may create new templates with the interactive graphical editor. The graphical editor can also be used to make notations on the display itself, for example, to mark an item to be examined later.

SDMS can be represented in GUIDE by extending GUIDE's viewports to indicate on what surface they are to be displayed. (Multiple view surfaces are provided in the CORE specification, but are not currently implemented in our version.) Almost everything drawn in SDMS is a user-defined picture. The world map tool, while not included in the initial implementation of GUIDE, is certainly a possibility for a later version. Alternatively, it can be replaced by a window controller. (The window controller is fully defined in chapter 3 and an example is shown in appendix B. Use of the world map tool, that is, moving within the world view, should change values which are parameters to the current view drawing tool. The keyboard is associated with a command parser tool.

Zooming in and out can be handled by treating the dial on the joystick as a valuator and associating with it an action routine which modifies the view of the world.

A current *view type* must be maintained to indicate the method of displaying the database contents. This value can be placed in the user profile and passed as a parameter to the routines which draw the views of the database.

2.7.4. Lisa

Apple's Lisa [Apple 83] running the Desktop Manager is a personal computer designed for office applications. The Desktop Manager contains a collection of packages useful to business managers. Included are a spreadsheet program (LisaCalc), a graphing package (LisaGraph), a graphics package (LisaDraw), a text processor (LisaWrite) and several others. All interaction is performed through a one-button mouse and the keyboard.

A menu of command classes is always displayed at the top of the screen, regardless of the currently active package. The list of command classes depends on which package is active. The user may see any of the commands in a class by selecting the class from the menu. The sub-menu remains visible until the user releases the mouse button. If it is released when the mouse is over any of the commands, that command is executed.

Current objects are widely used in Lisa. For example, the current type style for text is maintained. The user may change the currents affecting the active package by selecting an item (the appropriate command class) from the main menu and then selecting the desired value for that object.

Lisa distinguishes between *single* and *double* clicks of the mouse. When the mouse button is depressed and released twice within a certain period of time (the length is user-determined), a *double click* is signalled. At some points, a single click selects a item while a double click executes it.

Lisa has a user profile called *preferences* including items such as the length of time between clicks for *double clicking*, the period of time with no activity until the screen dims and other similar things. The user may change any of these items at any time.

To represent the Lisa system via GUIDE, one or more contexts are used for each package including the operating system itself. Selection of the icon for a package from a menu of packages changes to the (main) context for that package. The set of *windows* for each of the

various packages constitutes a user-defined picture class. Since the user may create an arbitrary number of windows for each package, these classes are of the dynamic type. Each *window* which has been created appears on an iconic menu. (Icons are drawn by designer-supplied routines.) Selection of the icon for a window implies a change of context to the context for the package with which the window was created. The selected window becomes the current member of that class. Since Lisa uses the Smalltalk [Tesler 81] model of overlapping windows, selection of a window which is partially obscured also implies a change of context to the underlying package. Selection is performed with a pick tool.

Some of the command classes in the menu at the top of the screen are actually commands and the sets of items displayed when these commands are selected can be represented by lists in GUIDE.

The single click/double click can be modelled in GUIDE by attaching a condition to each menu item for which the double click has meaning. It is also necessary to distinguish between pressing the button and releasing it, since these represent different tools in LISA. This must be handled in the mapping from physical to logical devices.

The slightly different meanings of commands depending on the application and the different sets of commands within categories can be handled by GUIDE quite simply by using appropriate conditions. Menu items which are "personalized" based on current applications values (e.g., "Open bob" instead of "Open file", where "bob" is the current file) can be specified in GUIDE by using generated menu items.

Lisa uses a number of iconic menus. GUIDE provides the capability to specify icons for menu items. Lisa also displays unavailable menu items in half-tones. The designer in GUIDE may specify the omission type for menu items.

2.7.5. The Lisp Machine

The Symbolics Lisp machine [Symbolics 85] is a single-user system designed for efficient processing of Lisp. The operating system is based on the Smalltalk model [Tesler 81] of overlapping windows on a screen, representing a desktop. There are several types (classes) of windows available. In most of them, everything typed in at the keyboard is sent to a Lisp interpreter. A three-button mouse is provided for moving around and picking points from the screen. A number of pop-up menus are used for control of the display. When a menu is displayed, a one-line prompt message is shown for the item currently under the mouse. The

same prompt line is used to indicate what a push on each button would do at any time. Multiple button pushes have special meaning in some situations (generally, a return to a higher level). When this is so, the prompt line lists these as well. An abort key is provided to abort the current process. The user may scroll up and down within a window by placing the mouse over a scrolling tool provided at the edges.

As with other operating systems, the Lisp machine is not an ideal candidate for modelling with GUIDE; however, it is possible. The various types of windows can be implemented as user-defined picture classes. The context would depend upon the class of the window in which the user is working. In most contexts, the keyboard tools would invoke a Lisp interpreter as their action routine.

One menu hierarchy, that used for invoking processes and modifying screen contents, is available in all contexts, although the method of accessing it varies in the number of button pushes required.

In order to display the one-line prompts based on the mouse position, movement of the mouse more than a specified threshold must be considered to be an event. Prompting can then be specified with conditions based on the mouse position.

CHAPTER III

Tools and Tasks

Tools are techniques for graphical interaction. The set of tools available in GUIDE is pre-defined. Each tool has one or more methods for each of initializing, drawing, echoing and processing it. Tools may have other characteristics controlling their appearance and performance. In GUIDE, the designer instantiates tools as needed. Every tool instance has associated with it particular methods for initialization, drawing, echoing and processing, and values for its particular characteristics which are selected by the designer.

A task is a set of tools, any one of which can be used to produce a desired result. Each task has associated with it a set of actions, one of which is to be executed when the task is used based on the conditions specified by the designer. A small set of tasks, including "exit" and "help", is pre-defined and the designer can define others as needed.

3.1. The CORE Graphics System

GUIDE uses a "standard" graphics system, CORE [GSPC 79], to handle graphical input and output. The choice of CORE was made primarily because of its local availability and device independence.

Objects to be drawn using CORE are organized into named *segments*. Within a segment, an integer *pickid* may be used to distinguish among components of the object. The segment and pickid allow one to determine exactly what part of a drawing is located at some point.

CORE uses two coordinate systems. *Normalized device coordinates* (NDC) refer to positions on the screen with the origin at the lower left corner. *World coordinates* (WCS) are determined by the user's drawing routines and are arbitrary. The user establishes a viewport on the screen by specifying its boundaries in NDC space. A window indicating what portion of the application world is visible in the viewport is specified in world coordinates. Given these specifications, CORE determines a mapping between NDC and WCS, allowing the drawing routines to work in world coordinates.

CORE provides six logical devices for input: *pick*, *keyboard*, *button*, *stroke*, *valuator* and *locator*. A pick device returns the segment and pickid of the object located at a picked point. A keyboard returns alphanumeric input. A button allows choice among alternatives. A stroke device provides a series of positions in NDC space. A valuator returns a scalar value. A locator returns the coordinates of a picked point in NDC space. Each logical device may be mapped to a physical device through an appropriate software driver.

Pick, keyboard, button and stroke devices are *event* devices, meaning that use of the associated physical device signals an event to the application. Locators and valuator are *sampled* devices, meaning that their values are read on request.

3.2. Tools

The set of tools contains the five of the six CORE [GSPC 79] logical input devices plus higher-level devices built using the CORE concepts. Sampled devices are modified to perform as event devices by the addition of a button. Some of the higher-level tools available are menus, lists, potentiometers, window controllers and forms. For each tool, GUIDE contains a routine which initializes the tool, a routine which draws the tool (if it appears on the screen), a routine which echoes the tool and a routine which processes the input received by the tool. The designer may specify alternate routines to perform these functions, provided they contain certain things necessary to the GUIDE runtime system. All routines are parameterized to provide for multiple instances of the tool type. In addition, each tool has a data structure which contains the names of these routines, plus any characteristics which affect the appearance or operation of the tool. For example, a menu may be displayed vertically, horizontally or in some configuration specified by the designer.

An initialization routine for each tool type performs operations which must be completed before drawing the tool. For example, lists may be generated by executing a routine which constructs a list. The set-up for a list constructed in this way would invoke the appropriate routine and then convert the result to the appropriate form for drawing.

The drawing routine for each tool displays that tool according to its characteristics, such as orientation and color. Some tools, such as a pick, have no display.

An echo is a perceptible (visible or audible) reaction to the use of a tool. Every tool has one or more possible echoes. For example, a commonly used echo for a menu is to highlight the chosen item. In this system, each tool type has one or more echo types available. Upon

instantiating a tool, the designer may specify which echo type to use, if there is more than one. Normally, there will be a default echo for a tool, which is the echo commonly associated with that tool type.

Some echoes will occur in the viewport containing the tool (*e.g.*, highlighting of a menu item), some will occur in other viewports (*e.g.*, a window controller changes the viewport with which it is associated) and some will be off the screen entirely (*e.g.*, lighting a button). In the last two cases, it is the designer's responsibility to specify what viewport or object is affected when the tool is instantiated.

The processing routine for a tool converts the raw data received by the CORE system into the type that the tool expects and performs any necessary updating of the tool data structure. For example, the processing routine for a list uses the segment and pickid of the picked point to determine which item was chosen. Then, the list instance is updated to have that item as its current value.

3.3. Specific tools

This section describes each of the tools available in the prototype implementation of GUIDE. For each tool, the characteristics which may be specified by the designer are described. In addition, the designer may specify colors for background, text and line drawing for each tool. These default to acceptable values if no others are specified. Many characteristics other than those listed below could be added to the appropriate tools. A few features are described which are supported by GUIDE, but have not yet been implemented. These features are listed in section 10.1.

3.3.1. Menus

Menus are generally used to specify the control path of a system. The user chooses options from a menu of those available at some point. The system acts on the option chosen and then presents a menu of the options subsequently available. In general, it is expected that the options in a menu will be application commands.

While many systems have strictly hierarchical menu structures, there are some systems in which an option may appear at many levels. In order to deal with this case, menus in GUIDE are not restricted to a tree structure. Each item in a menu structure contains a list of those items which belong to its *follow list*. The follow list for an item contains those items which

should be displayed if the item is chosen. (The term *sub-menu* does not apply since one item may appear both in the same menu as another item and in the second item's follow list.) This method allows total generality of menu structures. For example, in GUIDE, the menu item for creating or editing a tool appears in both the top-level menu and in the menu for creating or editing a task.

A stack is maintained for each menu at run-time, containing each item which causes a change in the list of items presented. When the user is not at the starting list of a menu, an item is provided to go back to the previous list. Choice of this item pops an items off of the stack for that menu.

A condition may be associated with each item in a follow list which determines whether it is actually available for selection. An item may have a different condition for each list in which it appears. The condition may be based on application variables, task values (see section 3.4) and user profile values (see chapter 6.1).

The designer determines, for each menu, what is done when items are to be omitted. An item may be omitted from the display, its space may be left blank or the item may appear in half-tone (as in LISA [Apple 83]). Additional possibilities may be added in later versions of GUIDE.

The designer also has control over details regarding the appearance of the menu. The menu may be displayed horizontally, vertically or in some other arrangement. The designer may also specify whether a menu display is to be paged, and, if so, how many items constitute a page and what message should inform the user that more items are available. If the menu is to be displayed in an arrangement other than horizontal or vertical, the designer specifies the location of each item. The designer also specifies the background color for the menu and the color of each item. Each item may even be a different color. The text for a menu item may be generated by calling a routine to return it. For such items, the designer specifies the routine and parameters, as well as the text color. Items may also be drawn as icons. The designer must specify a routine to draw the icon. A single menu may contain predefined textual items, generated textual items and iconic items. Figure 3-1 shows the menu for the layout system. It is unpagged and vertical.

An alternate form of menu which has not been included in the initial version of GUIDE is the implicit menu such as a command parser. With such a menu, no items are displayed and the user selects a command by typing it in. These menus could be incorporated into GUIDE by

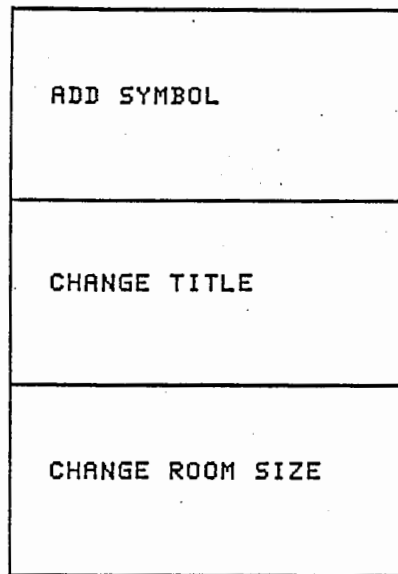


Figure 3-1: A menu

modification of the draw, echo and process routines for menus. The non-hierarchical structure of menus would still apply.

3.3.2. Lists

Lists are used for selection of an object from a set. Lists may be either static or generated. A static list is totally specified by the designer. Alternatively, a designer may provide the name of a routine which constructs the list and returns it to the interface. This is useful for presenting lists of application or operating system structures, such as files. In general, it is expected that lists will contain objects rather than commands.

Every list instance has a value which is the item most recently picked from the list. The initial value of the list may be set by specifying a default value at the task level (see section 3.4).

As with menus, the designer may specify a number of items regarding the display of each list. He may indicate the orientation of the list (horizontal, vertical or other) and whether or not the list is paged. He also indicates whether the individual items should be numbered or labelled or both. For numbered lists, the designer specifies a starting number. In both cases, the designer specifies a terminator character, such as a colon, to follow the number or label. The designer may specify a name for the list and the background color of the list. For generated lists, the designer specifies a single color for the items.

For static lists, the designer may specify for each item, a label, a value, a color and a position in the list. Individual items may be represented by icons, in which case the designer specifies a routine to draw the icon. As with menus, a single list can contain both textual and iconic items.

In addition, a static list may be structured hierarchically, such that each item may have a list of sub-items. When an item is chosen, its sub-list is presented for selection. This continues until the bottom of the hierarchy is reached. The task containing the list is not considered to have been invoked until the bottom of the hierarchy is reached. This allows the designer to organize a list into categories each of which contains a list of items. This structure can be used as an alternative to or in conjunction with paging. For hierarchical lists, a stack similar to that used for menus is necessary. A "previous item" selection then allows the user to move up in the hierarchy. Figure 3-2 shows the list of tools available in GUIDE. This list is vertical and paged.

TOOL TYPES
FORM
LIST
MENU
-- MORE --

Figure 3-2: A list

The major difference between menus and lists is that list structures are strictly hierarchical while menus may be non-hierarchical. In addition, the designer may specify conditions for inclusion of individual menu items in a follow list, while the contents of a list level are fixed.

While it is expected that menus will be used for commands and lists will contain objects, there is nothing in GUIDE which forces the designer to make this distinction. It is possible to specify a menu of objects or a list of commands and to specify appropriate actions as the result of their use.

3.3.3. Forms

Forms are an extremely general and useful tool. The designer may construct a form consisting of tasks to be filled in by the user. The user fills in the form by choosing its various fields and using the appropriate tools to provide values for them. There are two ways in which completion of a form may be indicated. The designer indicates which method is to be used for each form. The first method (*implicit termination*) simply monitors the fields, and when all fields have been given values, signals completion. With the second method (*explicit termination*), when every field has been given a value, a specified task (the *termination task*) appears in the form's window (in addition to the other fields). The user indicates that he is finished filling in the form by using the *termination task*. This permits modifications to the form prior to explicit acceptance of all form values.

This tool provides a method whereby a designer can ensure that a number of tasks receive values without specifying the order in which this occurs. It also gives the user a chance to enter and modify values until they are correct. The user may use the tasks in the form in any order; tasks other than the form task may even be used between form field tasks. Forms are extremely useful for entering parameters.

The fields in a form need not all be textual; any other kinds of tasks may be included. A form may even contain other form tasks. The designer may specify initial and default values for any or all of the fields by specifying these values for the task associated with the field. If all fields have initial values, explicit termination must be used and the "termination task" will appear initially. If the initial values are all acceptable to the user, he may choose the "termination task" and make no changes.

Each field in a form may have a condition associated with it. The condition determines whether or not that field is displayed at any time. The conditions are re-evaluated each time a field is modified. These conditions allow the designer more control over the form filling and make forms useful for filling in variant data structures.

The task containing the form is notified that the form has been used only when termination is signalled. At that time, the action associated with the task is executed.

As with menus and lists, the designer specifies the orientation of the form. It may be horizontal, vertical or positioned by the designer. The designer also provides a name for each field. Since the tools within the field tasks will have their own characteristics, no further specification is needed at the form level. Figure 3-3 shows the form used to initially define the characteristics of the room in the layout system.

ROOM TITLE ENTER TITLE	
ROOM WIDTH ENTER WIDTH	20.000
ROOM LENGTH ENTER LENGTH	20.000
	DONE

Figure 3-3: A form

3.3.4. Window Controller

There are several techniques which can be used to provide a full or partial view of a user-defined world. These include provision of a global view as in video games and SDMS [Herot 84], entry of explicit limits for display, *contextual includes*, such as "make the window large enough to include items A, B, C and D" and the window controller. The window controller is a particularly powerful example of this class.

The window controller is used as an auxiliary to a user-defined picture (see section 4.1). It permits the user to control what part of the world is displayed in its assigned viewport.

The window controller is divided into five regions (figure 3-4). Each region corresponds to a different command. The commands are:

- A. shrink window (*i.e.*, zoom in);
- B. expand window (*i.e.*, zoom out);
- C. translate window;
- D. move window to world extreme;
- E. extend extremes and move to new extreme.

The first two commands use a scale factor specified by the designer. The last three commands use the position of the chosen point within the region to determine what part of the world is to be displayed. The controller is considered to be a map of the world. Choice of a point in the controller indicates that the corresponding region of the world is to be displayed. Translation is always proportional to the amount of the world currently displayed. A complete description of the operation of the window controller appears in [Badler 84].

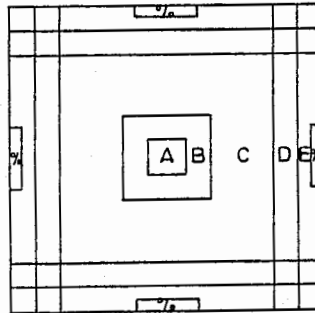


Figure 3-4: The window controller

The designer of a system must indicate the user-defined picture associated with the controller. Ordinarily, use of the window controller will not affect the control path. The modification of the window is the echo of the window controller.

The designer may specify the relative sizes of the regions, as well as the scale factor for zooming and the percentage by which to extend extremes. He also must specify an expression which represents the world extreme in each direction.

3.3.5. Potentiometers

Potentiometers are tools for entering real number values. They are software implementations of dials and slides such as those found on ovens and radios. Potentiometers are based on the CORE valuator device.

A potentiometer has low and high boundaries between which the selected value is scaled. It also may have labels for the low and high ends as well as for the entire potentiometer. The designer specifies all of these values. The designer also specifies the color for drawing the potentiometer. Figure 3-5 shows a horizontal potentiometer.

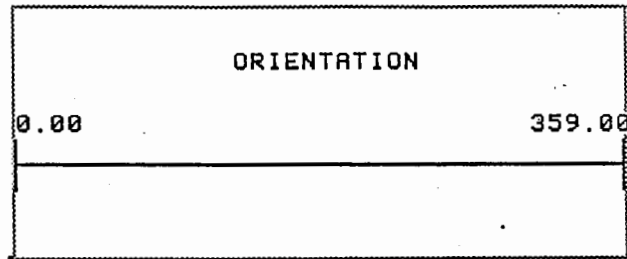


Figure 3-5: A Horizontal Potentiometer

A user positions the cursor in the desired position on the potentiometer, then signals an event. The event is echoed with the value which this selection gives the potentiometer. Processing for a potentiometer assigns the new value to the instance.

3.3.6. Picks

GUIDE provides two kinds of pick tools: a pick and a pick with locator. The pick returns only the segment and pickid of the selected point. The pick with locator returns the segment and pickid plus the coordinates of the point in world coordinates. In all other respects, the two tool types are identical.

The designer may select a cursor type for each pick instance. In addition, a user-defined picture class is specified over which the pick operates. Only when the selected point is within a member of the picture class for some pick tool is an event assumed to be a use of that tool. This method of handling picks is similar to the filter concept used in PHIGS [ANSI 85].

In future versions of GUIDE, the designer may have control over the pick aperture of each pick, allowing specification of how close a match must be made.

The echo of a pick shows the cursor at the picked point. If a pick hits neither a tool nor a user-defined picture class specified for a pick tool, the user is notified and no action occurs.

3.3.7. Keyboard

The keyboard tool in GUIDE corresponds to the CORE keyboard device. The designer may specify a label or message to be displayed in the tool's viewport position on the screen. This message generally serves as the prompt for this tool, although an ordinary prompt may also be specified (see section 6.3). The user enters a string of characters, which becomes the value of the instance. The string is echoed on the screen along with the original message.

Figure 3-6 shows a keyboard tool before and after it is used.

CONTEXT NAME

a) A keyboard before use

CONTEXT NAME

MAIN_CONTEXT

b) The same keyboard after use

Figure 3-6: A keyboard tool

3.3.8. Locator with Button

A locator with button is based on the CORE locator device. The addition of a button converts the device from sampled to event.

A locator with button returns a point when the appropriate button is pushed. As with picks, each locator with button instance is associated with a user-defined picture class within which the point is to be selected. The echo displays the cursor at the chosen point. The tool value is the chosen point in world coordinates. The designer may specify the type of cursor to draw.

3.3.9. Valuator with Button

A valuator with button, based on the CORE valuator device, is another tool which is converted from sampled to event by the addition of a button. This tool is generally mapped to a hardware device such as a dial which actually produces numeric values. It returns a real value scaled according to high and low values specified by the designer.

The echo of a valuator with button is display of the entered value in the viewport specified for the tool. The exact performance of a valuator with button depends upon the hardware device to which it is mapped.

3.3.10. Button

A button tool in GUIDE corresponds to a CORE button device. When used, the button returns a designer-specified value. A button tool may be mapped to either a real button or to a position on the screen.

The echo of a software button is generally a highlighting of the button. The echo of a hardware button depends on the button. Figure 3-7 shows some software buttons.

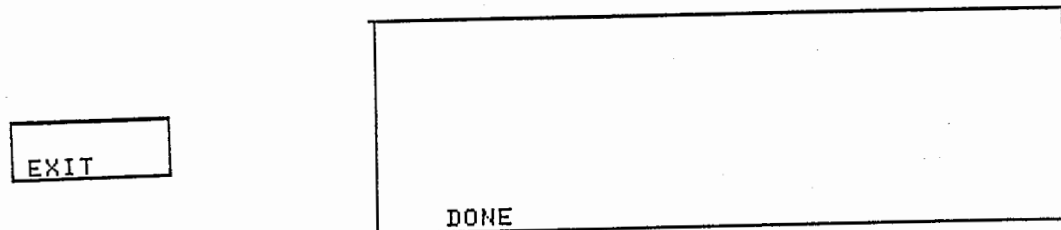


Figure 3-7: Buttons

3.4. Tasks

A *task* is a set of tools, any one of which can be used to achieve the same end. Tasks provide the user with options regarding the way in which an item is entered. Every tool instance may be contained in at most one task.

Each task in GUIDE contains a list of one or more tools which implement it, optional initial and default values and a value type. A task also contains a list of action routines, one of which is invoked when the task is used, depending on conditions specified by the designer.

The value of a task may be of any type, built-in or defined in CORE or the application environment. Structured types may be used. For example, a task to input a color value might have as its type an array representing the color value.

The designer must specify, for each tool in the task, a routine which converts from the tool value type to the task value type. If default or initial values for the task are specified, conversions from the task value type to the tool value types must also be provided. Some type conversion routines are provided. In particular, routines which convert between and simply pass along types provided by GUIDE, are included.

The initial and default values for a task are specified as expressions (see section 4.3). The initial value must be computable at the start of the application session. The default value may depend on input values. The task is given the initial value at the beginning of the session. The default value is assigned at the beginning of any context containing the task. In both cases, the tools within the task are assigned appropriate values, computed by applying the appropriate conversion routine to the task value. For example, the task for specifying the title of a room defaults to the current title of the room. This value is passed to the keyboard tool used to implement the task.

The designer specifies an action routine to be invoked when the task is used. Conditions (see section 4.3) may be used to determine which of several routines to invoke. The designer also may specify parameters for the action routine (see chapter 5).

3.5. Predefined Tasks

A few commonly used tasks are provided in the tool kit. The designer may instantiate and edit these as well as create his own tasks. There are also frequently tasks in an application system which need to be present all or most of the time. In order to simplify the process of installing such tasks in the contexts of the application, the designer may create and edit a list of *permanent tasks*. The list is automatically copied into each context as it is created. The designer may edit the list in any context but in any case must add the decision logic accompanying these tasks in each context.

In addition, there are some tasks which are common to almost every system, such as "exit" and "help." These tasks also have the characteristic that their actions come not from the application, but from the interface control. To make inclusion of these tasks simple, they are built into GUIDE along with appropriate decision logic and may be installed by the designer into any context or the permanent task list. Because these tasks carry decision logic with them, they are known as *task-consequences*. The "exit" task-consequence has the effect of popping a context off of the context stack and transferring control to it. If the context stack is empty, "exit" is interpreted as "end the application session." The "help" task-consequence determines for what object help is desired and displays the appropriate help message. After the display, the control path is not affected.

CHAPTER IV

Contexts

Contexts describe the state of the application system at any time. A context consists of user-defined pictures and tasks and decisions: user-defined pictures indicate what can be seen of the application on the screen; tasks are what the user can do; decisions indicate the control path by creating connections between contexts. Tasks were discussed in section 3.4. User-defined pictures and decisions are discussed below.

4.1. User-defined pictures

User-defined pictures are those items which the designer wants to display which are not tools. Generally, these are graphical representations of application data structures. For each user-defined picture, the designer must provide a name and a drawing routine, as well as a viewport in each context in which the picture appears. The GUIDE-generated interface will display each user-defined picture at its specified location.

User-defined pictures are organized into classes. In general, the members of a user-defined picture class represent different graphical instantiations of the same data type. The contents of a class may be static or dynamic. A static class contains a group of one or more pictures specified by the designer. For example, the class containing the room in the layout system is static. The contents of a dynamic class may be modified when the application system is running. The class containing all of the Lisp Listener windows on the Lisp Machine (see section 2.7.5) is dynamic.

For a dynamic class, the designer provides the name of a variable which points to the beginning of the class; the name of a function which, when given a pointer to a member of the class, returns the next member of the class; the name of a routine which draws members of the class and the name of the pointer type for the class. This structure allows the application system to be in control of the list of items in a class. The interface accesses this list only when necessary. Also, the overhead of maintaining two copies of the list is avoided, since the interface directly accesses the application copy of the list.

4.2. The control path

The control structure of a system indicates what context is to be active at any given time. The control structure is formed by creating connections between contexts. A connection between two contexts indicates that in the first context, if a certain task is used, and then if certain conditions are met, we follow the first context with the second. There are three ways in which a new context may gain control. First, the designer may choose to push the first context onto a context stack, allowing an action to interrupt another action. Second, the connection between two contexts may indicate that if all conditions are met, the second context is to be popped from the context stack and re-activated. In this case, the first context is terminated without stacking. Third, we may leave the first context neither popping nor pushing.

Any number of contexts may be stacked. The designer must indicate which variables, if any, should have their values restored when the context is popped. The values are saved when the first context is pushed onto the stack and restored when the context is popped from the stack. For example, in GUIDE, when the "create/edit context" command is selected, control transfers to a context for editing contexts. The previous context is stacked and the value of the "choose_command" task is saved. Figure 4-1 shows the main context of GUIDE after "create/edit context" has been selected. Figure 4-2 shows the context creation context which follows it.

The ability to either stack contexts or visit them in sequence and to mix these two methods freely gives the designer a great deal of control over the control structure. The designer determines what actions may interrupt others and which can be performed only in sequence. In this way, the designer may prevent some problems from arising. The only kind of path which cannot be specified is returning to a stacked context without modifying the stack.

Whenever we enter a context, the screen is cleared and all objects in that context are drawn. The interface then awaits input for processing.

4.3. Expressions and Conditions

Expressions in GUIDE correspond to the usual programming notion of expressions: evaluable sequences of operators and operands. Every expression has a type, such as boolean. Expressions are used to specify conditions, parameters, initial and default values for tasks and values to be restored when a context is popped.

EXIT

COMMANDS	DESIGN NAME	ENTER IDENTIFIER	LAYOUT
CREATE/EDIT TOOL			
CREATE/EDIT TASK	START ROUTINE ENTER NAME WITH PARAMS		
CREATE/EDIT CONTEXT			
MAKE CONTROL PATH			
GENERATE CODE	END ROUTINE	ENTER NAME WITH PARAMS	
STORE DESIGN			
READ DESIGN	START CONTEXT CONTEXT NAME		
CREATE/EDIT USER			
CREATE/EDIT USER			
CREATE/EDIT USER		DONE	

Figure 4-1: GUIDE main context after selecting "create/edit context"

Logical, relational and arithmetic operators may be used in an expression. The syntax of GUIDE's expressions follows that of Pascal. Pascal's precedence rules are observed. In addition, an exponentiation operator ("**") has been provided with precedence between NOT and multiplying operators. Constants may be used, and functions may be invoked with appropriate parameters. The terms of an expression may refer to any global variable in the application system, any field of the user profile, the value of any task and to previously defined expressions.

- Application variables are referenced exactly as they appear in the application system.
- User profile fields are referred to as PROFILE.<field specification>. This corresponds to the notation used in the interface generated by GUIDE.
- Task values are referred to as <task-name>.<any field specification>. Again, this corresponds to the notation used in the interface. It is the designer's responsibility to ensure that a referenced task has a value.
- Expressions may be named and then referred to by name. The name is an

EXIT

COMMANDS	
PREVIOUS MENU	
CREATE/EDIT TOOL	
CREATE EDIT TASK	
CREATE/EDIT USER PICTURE	
CREATE/EDIT USER PICTURE CLASS	
ADD TASK	
REMOVE TASK	
ADD USER PICTURE CLASS	
REMOVE USER PICTURE CLASS	
ADD PREDEF TO CONTEXT	
NAME OF CONTEXT: MAIN_CONTEXT	

Figure 4-2: GUIDE context creation context

arbitrary length identifier. For example, in the layout system, the expression 'cur_room.room_size.x' might be named 'room_width'. Use of the name of an expression in another expression creates a reference to, rather than a copy of, the original expression. This means that later editing of the expression affects all uses. If a copy is desired, a command may be used to copy an expression.

All expressions are checked for syntactic correctness and type matching when they are entered. A parser is integrated with GUIDE for this purpose. The parser ensures that all expressions obey the type rules of Pascal and returns the type of the expression. In order for expressions to use application variables and constants and to have values of application types, the designer must provide GUIDE with a file containing those declarations. This file is discussed in detail in Chapter 7.

This capability of referring to application entities allows parameters, task default and initial values to be drawn or computed from application variables, greatly increasing the functionality of the interface. The ability to specify application defined types as the types for tasks reduces the amount of work application routines must do, although the designer must provide routines to

convert between built-in types and the application types. The effect of access to application definitions on conditions is discussed below.

A number of types and functions are built into GUIDE and may be referenced in the interface. These are shown in appendix A.

Conditions are used in a number of places in GUIDE. They can determine the control path, the set of items to be presented in a menu, list or form, which action routine to execute at some time, the manner in which the parameters for an action routine are collected or computed, and which of a set of helps or prompts to display. Conditions in the interface system are boolean expressions. The designer may assign a name to any condition, thereafter allowing reference to it by name. Again, names may be arbitrary-length identifiers. This simplifies specification of frequently-used conditions. For example, if the user profile contained a field called 'skill' of type real, the designer might create a condition called 'NAIVE' defined by the expression 'profile.skill<1.5'. Conditions may also be copied.

The ability to refer to application variables in conditions is one of the strongest features of GUIDE. It allows a great deal of context-sensitivity in the interface specification. Although the designer could, in theory, get the same sensitivity by using a large number of GUIDE's contexts to encode the state information, it is impractical to do so. Access to the application variables allows a single context to represent a large number of states.

CHAPTER V

Actions

Most of the useful processing in any system will occur in the designer-supplied action routines. The interface simply provides a way to determine what action routine to execute at any time. GUIDE, therefore, provides a method for indicating what action routine should be executed upon a given user action.

Most tasks allow the designer to specify actions that should occur when the task is used. For example, a "choose command" task will usually contain distinct actions to be executed for each possible choice. The designer specifies the action to be taken by supplying the name of an action routine (which has been or will be written) and methods for collecting and computing the necessary parameters.

5.1. Parameter gathering

Specification of parameters consists of two parts. First, the designer may provide a series of specialized contexts which allow the user to enter the necessary input values. Then, the designer may provide a list of expressions for composing the input values and any other relevant values into the actual parameters. The designer may provide several methods each for collecting and computing parameters and indicate which method is to be used by specifying conditions for each.

5.1.1. Parameter contexts

A specialized form of context is used to specify the screen organization during parameter gathering. These parameter contexts contain a special list of parameter tasks as well as any other tasks and user pictures desired. The list of parameter tasks contains those tasks which provide values for parameter computation. Parameter tasks are not permitted to have actions of their own. The designer may specify that several such contexts are to be presented in sequence, with each one collecting some of the values needed to compute the parameter values. The designer may also specify several lists of parameter contexts with conditions to determine which list is to be used. Figure 5-1 shows the main context of the layout system after

the "add symbol" command has been chosen. Figure 5-2 shows the parameter context which follows it.

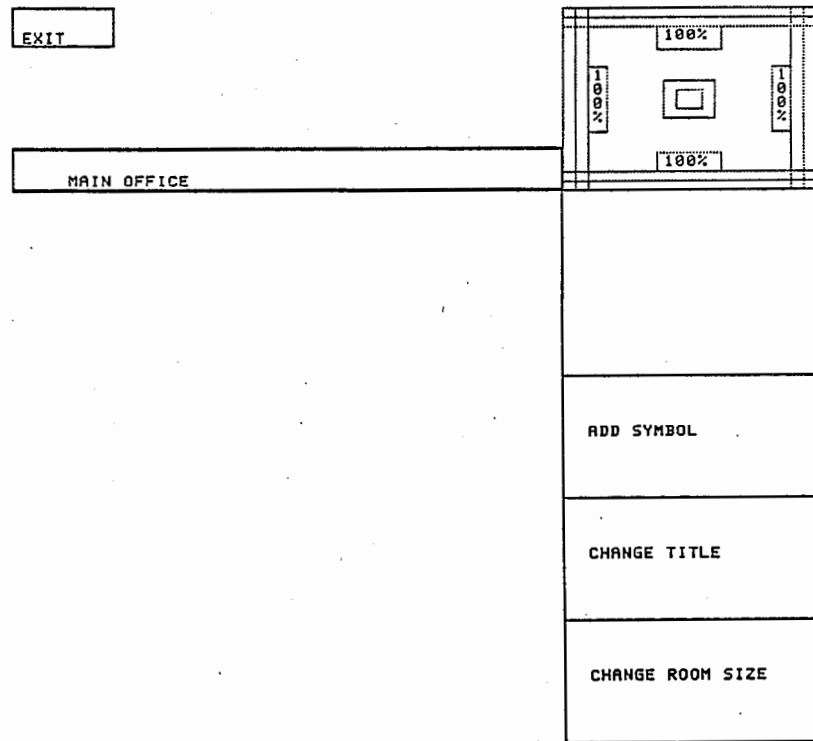


Figure 5-1: Layout system after selecting "Add symbol"

Transfer of control to a parameter context does an implicit push on the context stack.

5.1.2. Parameter expression lists

The actual parameter list for an action contains an expression (see section 4.3) for each formal parameter to the routine. This list of expressions is called a *parameter expression list*. The designer may specify several alternative parameter expression lists for each action, distinguishing them by conditions.

In the file containing declarations from the application system (see section 7.1), the designer also specifies the headers for any application routines which will be invoked by the interface. When a parameter expression list is entered, it is checked against the formal parameter list for the specified action routine to determine whether the types match. If they do not match, the designer is notified and given the opportunity to modify the parameter expression

EXIT		ORIENTATION		<div>100%</div> <div>100%</div>	
0.00		359.00			
MAIN OFFICE			PICK A POINT IN VIEWPORT ROOM_PIC		
			SYMBOLS		
			WINDOW		
			DOOR		
			DESK		
			CHAIR		
			PARTITION		
			DIVIDER		
			BOOKCASE		
			FLOORPLANT		

Figure 5-2: "Add-symbol" parameter selection

list. If the specified action routine is not found in the header file, the designer is notified. At any time, the designer may enter the editor as a sub-process, suspending GUIDE and returning to the current state when he is through editing. Thus, a designer may edit or create the appropriate action routines when an error occurs and may request that GUIDE re-check a particular parameter expression list.

At the time that the designer exits GUIDE, prototype headers are created for any routines for which the formal parameter list and any parameter expression lists do not match. Prototype headers are also created for routines which were not found in the header file. These headers are put into a file set aside for this purpose (see section 8.2), so the designer can see what the header should look like when code is written for the routine.

5.2. Fixed Actions

In a number of places, the designer needs to specify a routine for which the user cannot provide input values. Such routines include those to draw, echo and process tools, the routines to execute when starting and finishing an application session, and the routines to convert between tool and task values. These routines may take parameters, but the values of the actual parameters are not under user control. To handle these cases, *fixed action routines* are used. For each fixed action routine, the designer may specify a list of parameter expressions. As with regular actions, upon exit from GUIDE, prototype headers are created for any fixed action routines which are undefined, and for those with parameter mismatches.

The routines which perform conversions between tasks and tools must be able to access tool values. There is no notation for the designer to do this both because it is unnecessary and because such a notation would, of necessity, be cumbersome. Since each type of tool has different kinds of values, a designer would have to know how to refer to values of each kind. However, it is a simple matter for GUIDE to generate the appropriate references. When specifying a conversion routine, the designer should not specify parameters. Instead, GUIDE generates appropriate actual parameters for these routines in the interface. For tool to task conversion, GUIDE generates a list of tool values depending on the tool type, and a reference to the task. It is assumed that the formal parameter corresponding to the task value is variable. For task to tool conversion, GUIDE generates a reference to the task value, followed by references to the tool values appropriate to the tool type. In this case, the formal parameters corresponding to the tool values are assumed to be variable.

CHAPTER VI

User Support

Several facilities are provided in GUIDE to simplify the designer's job. The user profile allows the designer to customize the interface for each user. In addition, the designer may specify helps and prompts and may pass textual messages between the application and the interface.

6.1. User Profiles

One of the most difficult problems facing the interface designer is providing an interface that is appropriate for every user of a system. The approach GUIDE takes to this problem is the provision of a *user profile*. The user profile contains information about the user's preferences in dealing with the application. It also may contain data regarding the user's skill with the system and right to access information in the system. The user profile affects the interface when its fields are referenced in conditions. Thus, the designer can personalize the system for all of the users: multiple control paths may be specified; some menu items may be restricted to certain classes of users; and even different application actions or parameters may be specified for different users.

The designer specifies the fields of the user profile. He must also specify the type of each field and whether or not the user may update it. For modifiable fields, a condition based on the current values of the fields may be provided, so that only users with certain characteristics may change some fields of their own profiles. Except as discussed below, a user may update only his own profile.

The designer must specify the name of a file to contain the user profile records and should then ensure that the file is maintained as securely as the operating system allows. The designer indicates whether a user's profile must be in the user profile file before he can use the application system. If not, the user may simply begin using the system by providing values for his own user profile. In that case, the designer specifies default values for all fields and indicates by conditions which fields may be initialized by the new user. These may be the same as the conditions for updating given above.

If the user is to have access to his own profile, the designer must provide the opportunity to perform updating. When the designer specifies the profile, GUIDE automatically creates a form for updating the profile and includes it in the interface. The designer must then specify inclusion of this form in some context to allow updating of the profile. The designer is also responsible for ensuring that updating does not lead to a user's getting stuck at some point in the interface. If updating is permitted at any time, and the profile is used to determine the control path, it is possible for a user to enter a state, change his profile and then be unable to leave that state because his updated profile does not meet any of the conditions for exiting. For some systems, therefore, it may be desirable to permit updating of the profile only at the beginning and end of a session.

If access to some fields is restricted, the designer will need to provide a separate system (or sub-system) for maintaining the user profile file. This (sub-)system can also be implemented using GUIDE, and can use a condition based on the user profile to determine who may use it. In this way, the security of the user profiles depends on the operating system rather than on GUIDE.

Any field of the user profile may be referred to in conditions. If the designer, in the course of editing, attempts to remove a field of the profile which is referred to by any conditions, a conflict will be signalled.

In the layout system, the user profile might distinguish between architects and designers with architects able to modify the room size and designers able to work on rooms only as defined. The profile would contain a field

```
user_type:user_types;
```

where

```
user_types = (architect,designer);
```

Then, the "change room size" command would be included in the menu list only if the condition

```
profile.user_type=architect
```

were true.

6.2. Helps

One of the goals of this system is to make inclusion of helps in the designer's system as easy as possible. This is done by integrating creation of helps into the creation of the interface.

Helps may be specified for each tool, task, and context in the design. Each help may contain both a short message and the name of a file containing a longer message. If only one is specified, that one will constitute the entire help. If both are given, then the short message will be presented first, with the option of requesting the additional information from the file. The designer may provide several help messages, distinguished by conditions, for any item. For example, if the user profile contained a language field, the system could provide helps in English for one user, while providing them in Spanish for another.

6.3. Prompts

GUIDE permits the designer to specify prompt messages for each tool and context distinguished by conditions. Since help is thoroughly integrated into the system and conditions can be used to selectively present or withhold prompts, it is felt that extensive prompting is not necessary.

For tools, the designer indicates the placement of the prompt within the tool viewport. For contexts, both a viewport and a position within the viewport are specified.

6.4. Messages

When GUIDE invokes an application routine, it is possible that errors may occur in the action routine. That is, the application routine may find that it is unable to complete the desired action or that some condition occurs of which the user should be made aware. GUIDE provides a message system to allow the application to inform the user of such situations.

GUIDE maintains a list of messages which have occurred. Each message has three components: a number, a textual message and a location. The designer may choose to use the numbers either for individual message codes or for severity codes. The textual message is appropriate text regarding the situation and possible suggestions for alternatives and the location is the name (or some other identification) of the routine where the situation occurred.

Any routine can add a message to the list by calling the message posting routine with the

three components. In addition, the designer may call routines to clear the message list, to check for messages and to report the messages. Both checking and reporting can look for messages according to position or code. For example, the designer may check for the lowest numbered message or the last message to have been posted.

Application routines may check if a message has been posted by calling one of the check routines. In addition, since the check routines are functions, they may be used by the designer in conditions in the interface. The GUIDE-generated interface clears messages at the beginning of each context and reports and clears all messages after each input is processed.

CHAPTER VII

The User Interface

One of the goals of this system is that its user interface should be designed using this system. The initial version of the user interface has been designed with this goal in mind. In fact, some of the capabilities desired for this interface have affected the design of this system, most notably in the structure of menus.

7.1. Start-up

At the beginning of any session with GUIDE, the designer must provide the names of two files. These files contain the environment for the application system, one showing the declarations and the other containing the compiled environment which is to be inherited by the interface. An environment is an extension provided in VAX Pascal [VAX-11 Pascal 82] to allow for separate compilation of modules containing Pascal routines. Inheritance of an environment by a module directs the compiler to behave as though the contents of the environment were declared in that module. The environment provided by the designer must contain the declarations for all constants, types and variables which will be referred to in the interface, as well as the headers for all those routines which will be accessed by the interface. This environment will be used to check parameter list specifications against the actual routines for correct number and types of parameters, and to check expressions for type matching and result type. Appendix B.1 shows the environment provided for the layout system.

The environment file provided to GUIDE should follow the syntax for a module in VAX Pascal. At present, however, the parser cannot handle included files or any attributes for variables other than "global" and "external". Procedure and function headers should specify "extern;" instead of the body. The module may specify inheritance of other modules; however, the contents of those modules will not be included in the symbol table. The BNF for the environment file is shown in appendix C.

In later versions of GUIDE, the user will also be able to provide names for two other files. (Default files will be used if the user does not provide names). The first will be used for error messages. In particular, prototype headers for action routines with parameter mismatches (see

section 8.2) will go into this file. The second file will be used for a journal of the user's session (see section 8.3). For now, a default file is used for the prototype headers and no journal file is created.

7.2. Appearance of the interface

The nature of GUIDE is such that there is no one correct interface. Since GUIDE can be used to generate its own interface, a large set of interfaces is possible. For the prototype implementation, one interface has been chosen, primarily for definitional simplicity. This interface is described in this section.

The interface to the design system is menu-driven. At any time, the designer is presented with a selection of available options. Some of the options cause the stacking of the current process in order to do something else. For example, when creating a context, the user may opt to create a task. This will stack the context creation until the task has been created. After returning to the creation of the context which was stacked, the user may include the newly-created task in the context. Similarly, creation of a control path item can be interrupted by creation of a context which can then be used in that control path item.

Forms are very widely used in this system. Creation and editing of tools, task, control path entries and many other objects use forms. Figure 7-1 shows the form used to create a list tool instance. This allows the user to edit an item in the same manner as it was originally created. When an item is presented for editing, its current values are displayed and the user may change as many (or as few) as desired. When an item is initially presented for creation, any default values are displayed. The user can then enter and modify the values until the desired set is achieved. For example, help information can be specified for those objects for which help is available. The user's ability to create the help text at the same time as the structures is expected to encourage inclusion of helps.

In some cases, the designer specifies objects by picking them from the display. For example, when adding a menu item to a list of items, the position within the list is picked from a drawing of the list. (See figure 7-2.) It is expected that later versions of the interface will make more use of picks than the prototype.

The system maintains some current objects, including a current tool, a current task and a current context. The current objects are used as defaults in the various commands. In some cases, this means selection of the parameters requires only a button push while in others, it

DISPLAY_NAME	BOOLEAN		
	TRUE		
	FALSE		
ORIENTATION	ORIENTATION		
	HORIZONTAL		
	VERTICAL		
RANDOM			
LIST_LABEL	ENTER STRING		
PAGING_PERMITTED	BOOLEAN		
	TRUE		
	FALSE		
BACKGROUND_COLOR	HUE	ENTER REAL	0.000
	SATURATION	ENTER REAL	0.000
	VALUE	ENTER REAL	0.000
NUMBERED	BOOLEAN		
	TRUE		
	FALSE		
LABELLED	BOOLEAN		
	TRUE		
	FALSE		
GENERATE_ITEMS	BOOLEAN		
	TRUE		
	FALSE		
LIST			
	DONE		

Figure 7-1: Form for creating list tool instances

ADD SYMBOL
REMOVE SYMBOL
CHANGE TITLE
ITEM LIST

Figure 7-2: Picking a position within a list

means that the current is included in a form until the user changes it. Reference to an object other than a current changes the current for that kind of object.

CHAPTER VIII

Output

The output from GUIDE falls into several classes. First, the design which is being created can be stored for later modification or extension. Second, the system creates a file for the designer containing error information. Third, a journal file is created for each GUIDE session. Finally, upon request, the system will generate the code and data necessary to integrate the interface with the designer's action routines.

8.1. Data structures

The designer may request that the entire interface under construction be stored in a file. This allows the designer to create the interface in several sessions or to edit the prototype at a later date. When the designer exits the system, there is a reminder to store the prototype in this manner. The details of storing the data structures are discussed in section 9.7.

8.2. Error file

The error file is used to inform the designer of various potential problems in his design. Prior to generating the interface, the design is checked for several kinds of consistency, and any inconsistencies are reported in the error file. In addition, the error file is used for creating prototype headers for any routines used in the design which are neither built into Pascal or GUIDE nor exist in the environment file provided by the designer.

The design is checked for any tasks which contain no tools, any contexts which contain no tasks, any task/tool pairs which are missing conversion routines, and specification of a start context. In addition, a list of objects which have been created, but never used, is printed in the file.

The prototype headers contain the actual name of the routine and a parameter list containing the actual types of the parameters along with a dummy name for them. If desired, the designer can take such a header and, with a little editing, use it as the header for the action routine. It is hoped that such a facility will aid in rapid correction of mismatches between action routines and interface specifications.

8.3. Journal file

Each time it is used, GUIDE will create a journal file, containing a complete record of the user's session. This file will be used to implement the *undo* commands. It can also be used to recreate a terminal session, by replaying all of the commands which were executed. This facility has not been included in the prototype implementation.

8.4. Interface

The major output of the UIMS generator system is the Pascal code and data, which when combined with the designer's action routines and the library of tool routines, will constitute the designer's system. The code includes context code, which determines the control path, task routines, which include the interface with the action routines, tool routines, code to handle expressions and conditions, and code for determining at any time what tool and task have been invoked.

The code generated by GUIDE follows good programming style. It shows structure by indentation, contains comments and uses blank lines for readability. The examples shown in the following sections contain the code exactly as it is generated by GUIDE, except where a code line was too long to fit onto a single line. Such lines have been broken intelligently.

8.4.1. Context Code

Every context will be translated into a procedure. Entering a context, therefore, corresponds to invocation of a procedure. Since contexts will sometimes be stacked when a new context is invoked and sometimes not, we do not want context procedures to invoke each other directly. A special procedure is used to invoke the various procedures, consisting primarily of a large CASE statement. Each context procedure is assigned a code number, which is used to invoke it. Also, each procedure returns, as a parameter, the code number for the procedure which to be invoked next. The mechanism for stacking contexts takes advantage of this structure.

Each context procedure has three sections. The first section draws the context. The second section awaits and processes input and invokes any action routines. The third section evaluates the various conditions in light of the task used and determines the next context. The third section, also, handles any processing necessary to push and pop contexts onto the context stack and save and restore variables.

Figure 8-1 shows the procedure generated for the main context in the layout system.

8.4.2. Task routines

Two routines are generated for each task, one to draw it and one to process input. The task drawing routines invoke routines to set up and draw each of the tools contained in the task. Figure 8-2 shows the procedure generated to draw the task used to get the room width in the layout system.

The processing routine for each task receives as parameter a code for the tool which was actually used. Using this, it invokes the appropriate conversion routine. Once the task value has been computed, the routine determines which action routine, if any, to invoke. If any parameter contexts have been specified, they are invoked, automatically stacking the current context. Finally, the task routine invokes the action routine, computing the actual parameters in the call. Figure 8-3 shows the procedure generated for the `choose_command` task in the layout system. This task contains the menu for the application and is associated with the application routines which update the room data structure.

8.4.3. Tool Routines

For each tool, two routines are generated. The first makes the calls necessary to set up the tool and draw it. This routine invokes an appropriate set-up routine, if there is one. If the task has a default value, this routine invokes the appropriate conversion routine to compute the default tool value. Then, it installs the tool window in the window manager and invokes the specified drawing routine for the tool. Figure 8-4 shows the code generated to set up and draw the keyboard tool used to get the room width in the layout system.

The second procedure generated for each tool is used to echo and process input to that tool. It first invokes the echo routine for the tool; then, if a tool use has actually been completed, it invokes the processing routine. It is possible to use a tool in such a way that the system does not signal that the tool input should be processed. For example, if a list is paged and the user selects the page message, it is necessary to echo the tool by displaying the next page, but the input should not be processed and passed to the task. A boolean variable is used to indicate whether or not to process the input. The various echo routines must set this variable appropriately. The context procedures continue to await and process input values until this variable indicates that a valid tool use has been completed. Figure 8-5 contains the procedure generated to echo and process the width keyboard tool in the layout system.

```

[global]procedure main_context(var next_context:integer);
(* this procedure implements context main_context *)

var
  upic_loc:location_type;
  error_pos,prompt_pos:point;
  error_vport:location_type;
  time:real;
  eclass:nametype;
  enum:integer;
  this_context:integer;
  task_used:integer;

  WINDOWCONTROLLERTOOL_loc:location_type;
  layout_menu_loc:location_type;
  exittool_loc:location_type;
  tool140_loc:location_type;
begin
  begin_bupdt;
  delall;
  newframe;
  (* prepare viewport to device table *)
  dispose_run_hash_table(vtd_hash_table);
  (* drawing pictures in class title_class *)
  (* draw picture title_pic *)
  upic_loc[top]:= 8.10021E-01;
  upic_loc[bottom]:= 7.54349E-01;
  upic_loc[rightdim]:= 6.79061E-01;
  upic_loc[leftdim]:= 6.52313E-04;
  add_or_update_window('title_pic',upic_loc, TRUE, TRUE,1);
  DRAW_TITLE(cur_room);

  (* drawing pictures in class room_class *)
  (* draw picture room_pic *)
  upic_loc[top]:= 7.54349E-01;
  upic_loc[bottom]:= 2.78358E-03;
  upic_loc[rightdim]:= 6.79061E-01;
  upic_loc[leftdim]:= 6.52313E-04;
  add_or_update_window('room_pic',upic_loc, TRUE, TRUE,1);
  DRAW_ROOM(cur_room,set_window_params.window_used);

  WINDOWCONTROLLERTOOL_loc[top]:= 9.99304E-01;
  WINDOWCONTROLLERTOOL_loc[bottom]:= 7.54349E-01;
  WINDOWCONTROLLERTOOL_loc[rightdim]:= 9.94781E-01;
  WINDOWCONTROLLERTOOL_loc[leftdim]:= 6.81670E-01;
  draw_task_set_window_params(WINDOWCONTROLLERTOOL_loc,
                              false);

```

Figure 8-1: Procedure generated for main layout context

```

layout_menu_loc[top]:= 7.54349E-01;
layout_menu_loc[bottom]:= 2.78358E-03;
layout_menu_loc[rightdim]:= 9.97391E-01;
layout_menu_loc[leftdim]:= 6.79061E-01;
draw_task_choose_command(layout_menu_loc,false);
exittool_loc[top]:= 1.00000E+00;
exittool_loc[bottom]:= 9.50000E-01;
exittool_loc[rightdim]:= 1.25000E-01;
exittool_loc[leftdim]:= 0.00000E+00;
draw_task_exittask(exittool_loc,false);

if help_flag
then begin
    draw_task_get_help_object(tool140_loc,false);
end; (* if help_flag *)

end_bupdt;
make_pic_current;
(* prepare for error drawing *)
error_pos.x:= 1.00000E-01;
error_pos.y:= 1.00000E-01;
error_vport[leftdim]:= 0.00000E+00;
error_vport[rightdim]:= 1.00000E+00;
error_vport[bottom]:= 1.00000E+00;
error_vport[top]:= 8.00000E-01;

(* clear flag for computing defaults *)
context_first_drawing:=false;

(* wait for something to happen *)
process_complete:=false;

while not process_complete do
begin
    clear_errors;
    await_event(time,eclass,enum);
    process_event(eclass,enum,next_context,task_used,
        process_complete);
    draw_first_error(error_pos,error_vport);
    report_all_errors(output);
end;

```

Figure 8-1, continued

```

(* save context value to see if re-drawing *)
this_context:=next_context;

(* determine next context *)
if task_used = 49
then begin
end; (* set_window_params*)
if task_used = 123
then begin
end; (* choose_command*)
if task_used = 55
then begin
begin
(* popping previous context from stack *)
next_context:=0;
end
end; (* exittask*)
(* see whether re-drawing *)
if next_context<>this_context
then context_first_drawing:=true;

end;

```

Figure 8-1, concluded

```

[global]procedure draw_task_get_width(widthtool_loc:
location_type;has_val:boolean);
(* drawing for task get_width *)

begin
if context_first_drawing
then begin
get_width:=cur_room.room_size.x;
has_val:=true;
end;
(* set-up and draw tool width tool *)
set_up_and_draw_widthtool(widthtool_loc,get_width,has_val);
end; (* draw_task_get_width *)

```

Figure 8-2: Procedure to draw a task

8.4.4. Expression and Condition Code

GUIDE generates a function for each expression defined in the interface, except for those which are used only as parameters and contain only application variables. The function evaluates the expression and returns the value. A separate function is necessary for each expression due to type considerations. The layout system generates no non-trivial expression

```

[global]procedure process_choose_command(tool_id:integer);
(* processing for task choose_command
   invoked by tool with id tool_id *)

var t:runtime_tool_ptr;

begin
  (* convert tool data to task value *)
  t:=find_tool(tool_id); (* get tool record *)
  case tool_id of
    124:PASS_ALONG_STRING(t^.this_menu^.current_value,
                          choose_command);
  end; (* case *)

  (* now invoke action routine, if any*)
  if check_last_error=0
  then begin
    if eqnms(choose_command,'add symbol')
    then begin
      begin
        invoke_context(244);
      end;
      begin
        ADD_SYMBOL(cur_room,
                  choose_symbol,
                  choose_location,
                  choose_angle)
      end;
    end
  else if eqnms(choose_command,'remove symbol')
  then begin
    begin
      invoke_context(289);
    end;
    begin
      DELETE_SYMBOL(cur_room,
                   pick_symbol)
    end;
  end
end

```

Figure 8-3: Procedure to process a task

```

else if eqrms(choose_command,'change title')
then begin
  begin
    invoke_context(317);
  end;
  begin
    NEW_TITLE(cur_room,
      get_title)
  end;
end
else if eqrms(choose_command,'change room size')
then begin
  begin
    invoke_context(520);
  end;
  begin
    UPDATE_ROOM_SIZE(cur_room,
      get_width,
      get_length)
  end;
end;
end; (* if no errors occurred *)
end; (* choose_command *)

```

Figure 8-3, concluded

```

[global]procedure set_up_and_draw_widthhtool(tool_loc:
  ion_type;taskval:REAL;has_val:boolean);
(* invoke set_up and draw routines for toolwidthhtool *)

var t:runtime_tool_ptr;
    prompt_pos:point;

begin
  (* look up this tool's record *)
  t:=find_tool(195);
  set_up_keyboard(t,'widthhtool');
  if has_val
  then begin
    CONVERT_REAL_TO_STRING(get_width,
      t^.this_keyboard^.current_value);
  end; (* then *)
  add_or_update_window('widthhtool',tool_loc, TRUE, TRUE,0);
  DRAW_KEYBOARD(t^.this_keyboard);

end; (* set_up_and_draw_widthhtool *)

```

Figure 8-4: Procedure to set up and draw a tool

functions. Figure 8-6 shows one of the expressions functions generated for the GUIDE

```

[global]procedure widthtool(var process_complete:boolean);
(* processing for tool widthtool *)

var t:runtime_tool_ptr;

begin
  t:=find_tool(195);
  ECHO_KEYBOARD(t^.this_keyboard);

  if process_complete
    then PROCESS_KEYBOARD(t^.this_keyboard);

end; (* widthtool *)

```

Figure 8-5: Procedure to echo and process input to a tool

interface. When an expression is to be used in GUIDE-generated code, a reference to the appropriate function is generated. This deals with the possibility that an expression may refer to other expressions. Generating expressions in line in this case would be extremely difficult. Parameter expressions containing only application variables are generated in line, however, to allow for the possibility of variable parameters.

```

[global]function compute_expression6577:BOOLEAN;
(* compute expression expression6577 *)

begin
  compute_expression6577:=
    eqnms(choose_command,'create/edit tool instance') and
    tool_chosen(current_design.current_tool,pot_tool)
end; (* compute_expression6577 *)

```

Figure 8-6: Function to compute an expression value

In addition, there are a number of places where conditions appear but the evaluation of these conditions occurs in the GUIDE runtime system, rather than being generated by GUIDE. An example of this is the evaluation of a condition to determine whether to include a menu item in the menu about to be drawn. The call to evaluate such a condition comes from the set-up routines for menus. In order to handle these cases, a function is generated which takes a code and evaluates the condition to which the code refers. The evaluation function simply invokes the appropriate expression function and returns the boolean result of the condition. The layout system contains only one condition evaluated by the runtime system. This condition is of a type that is evaluated in-line rather than by a separate expression function. Figure 8-7 shows the condition evaluation function generated.

```

[global]function evaluate_condition(condition_code:integer):
                                boolean;
(* evaluate the condition with code condition_code
   and return the result *)

begin
  case condition_code of
    508:evaluate_condition:=cur_room.contains<>nil;
  end; (* case *)
end; (* evaluate_condition *)

```

Figure 8-7: Condition evaluation function generated by GUIDE

8.4.5. Window controller extremes evaluation

For each window controller instance, it is necessary to compute the current world extremes each time the controller is drawn. The designer provides expressions for these extremes. A routine is generated which evaluates these expressions (either directly or by calling the appropriate functions). The routine contains a case statement based on the tool code and then, for each window controller instance, performs the appropriate evaluations. Figure 8-8 shows this routine for the layout system, which contains a single window controller instance.

```

[global]procedure evaluate_windcont_extremes(wc_code:integer;
      var xtremes:real_dim_array);
(* evaluate the extremes for the window controller
   with tool code wc_code *)

begin
  case wc_code of
    46:begin
      xtremes[leftdim]:=compute_expression80;
      xtremes[rightdim]:=cur_room.room_size.x;
      xtremes[top]:=cur_room.room_size.y;
      xtremes[bottom]:=compute_expression69;
    end;
  end; (* case *)
end; (* evaluate_windcont_extremes *)

```

Figure 8-8: Procedure to compute window controller extremes

8.4.6. Invocation Routines

Four routines are used to invoke other routines: one invokes a context based on a code; another invokes the routine to draw a specified task; a third invokes tool processing routines; and a fourth invokes task processing routines. The context invocation procedure is necessary to avoid stacking up the calls to context procedures on the Pascal stack as discussed above. The routine takes a context code as parameter and consists of a loop which terminates when the context code is zero. Inside the loop, a case statement invokes the context referred to by the code. Figure 8-9 contains the context invocation procedure generated by GUIDE for the layout system. The routine to invoke task drawing is needed to handle forms. Drawing of a form uses this routine to invoke the routines used to draw the individual fields of the form. The tool invocation routine is called after an event has occurred to invoke the routine which echoes and processes the tool to which the event applies. The task invocation routines is called after a tool has actually been used to invoke the routine to process the appropriate task.

```
[global]procedure invoke_context(code:integer);
(* invoke the context procedure indicated by code.
   loop as long as code returned is not 0 *)

begin
  context_first_drawing:=true;
  while code<>0 do
    (* invoke specified routine *)
    case code of
      520:size_context(code);
      317:chg_context(code);
      289:del_context(code);
      244:add_context(code);
      184:start_context(code);
      146:main_context(code);
    end; (* case *)

    (* don't re-compute defaults on pop *)
    context_first_drawing:=false
  end; (* invoke_context *)
```

Figure 8-9: Context invocation procedure

8.4.7. Task Lookup Code

The last item of code produced by the interface system takes a tool code and determines what task contains the specified tool. The task lookup procedure returns a code for the task which is then used to determine which task processing routine to invoke. Figure 8-10 shows the task look-up procedure generated for the layout system. In this example, each task contains only a single tool.

```
[global]procedure look_up_task(tool_code:integer;
                               var task_code:integer);
(* look up the task containing tool tool_code *)

begin
  case tool_code of
    309:task_code:=308;
    272:task_code:=271;
    263:task_code:=262;
    234:task_code:=233;
    216:task_code:=215;
    207:task_code:=206;
    203:task_code:=200;
    195:task_code:=192;
    188:task_code:=185;
    124:task_code:=123;
    106:task_code:=109;
    88:task_code:=91;
    67:task_code:=70;
    63:task_code:=61;
    52:task_code:=55;
    46:task_code:=49;
  end;
end; (* look_up_task *)
```

Figure 8-10: Task look-up code generated for the layout system

8.4.8. Environment

GUIDE generates a file containing the headers for all of the above routines. This file constitutes the environment for the interface. The file containing the code described in the above sections inherits this environment. This avoids the necessity of generating code in the order required by the Pascal compiler.

8.4.9. Data

GUIDE generates certain data items for use at runtime. A help library file is generated, as well as information regarding which help messages to print when help is requested. In addition, data is stored describing the tool instances created for this application.

The GUIDE help system uses the VAX/VMS help system [VAX-11 82] so all help messages must be stored in a help library file. The VAX/VMS help system allows multiple levels of help for any object and is, therefore, useful in providing the two levels of help used in GUIDE.

In order to look up a help message using the VAX/VMS system, one or more keys indicating which message is desired must be provided. GUIDE creates these keys and stores them in the library file. In addition, since any object may have more than one help specified and distinguished by conditions, it is necessary to store data which indicates what conditions to evaluate when help is requested, and which key to use when conditions are found to be true.

The GUIDE runtime system requires certain data about the tools used in any interface. For example, the runtime system needs to know the names of a menu's items, the orientation of the menu, whether or not it is paged and so forth. GUIDE stores this information for each tool in a file read at runtime.

CHAPTER IX

Implementation

This chapter describes details of the implementation of GUIDE which do not affect the design of the system, but are otherwise significant. Discussed are the organization of GUIDE, the structure of GUIDE, the symbol tables used to maintain objects within GUIDE, the algorithm used to read and write designs, the algorithm used for copying structures, the problems of mapping tools to CORE logical devices, and the parser used in GUIDE.

9.1. The organization of GUIDE

GUIDE has been implemented in Pascal on a VAX-11/785 running VMS. The code output of GUIDE is also in Pascal. The data output of GUIDE consists of two text files. GUIDE uses an implementation of the CORE graphics system [Stluka 82] for graphical input and output and assumes the existence of a local window manager which also uses CORE. As much as possible, dependencies on the above have been localized.

The code implementing GUIDE is divided into a number of groups:

- the code manipulating the GUIDE data structures;
- the routines which generate the interface code;
- the parser code;
- the code for checking the design for inconsistencies;
- the message handling package;
- the conversion routines used in the interactive version of GUIDE;
- the routines which draw the GUIDE data structures for the interactive version;
- the code which implements the tools (*i.e.*, the toolkit);
- the code which handles input from CORE and determines what tool was used.
This includes code which handles mapping of tools to devices (see section 9.9).

A change in the target language for GUIDE-generated interfaces would require changes

only in the code generation routines and the parser. Within the parser group, only the semantic routines for type-checking would require modification. In addition, new parse tables would need to be generated. No other routines would need to be changed provided the operating system could handle linkage of routines in different languages.

Similarly, use of a different window manager would affect only the input-handling routines and the toolkit.

Changing the graphics package used would be more serious, requiring modifications in the code generation routines, the input-handling code, the toolkit, the routines which draw GUIDE data structures and possibly the conversion routines. Since the tools provided by GUIDE are based on the set of devices provided by CORE, some re-evaluation of the set of tools provided might also be desirable.

GUIDE depends on VMS only for its help system and for the environment capability provided by VAX Pascal.

9.2. The structure of GUIDE

This section diagrammatically shows the structure of GUIDE, the relationships among its components and the relationships with application code.

Figure 9-1 shows the files which are input to the GUIDE data structure routines at or near the beginning of a session. The tool kit and task kit files are read only when the designer indicates commencement of a new design. Figure 9-2 shows the files which can be created by GUIDE. Figure 9-3 shows the flow of data during a GUIDE session. GUIDE runtime services include the window manager and CORE, and is diagrammed in figure 9-4. It should be noted that the interaction between the GUIDE data structure code and the runtime services is through the message package, the window manager and CORE. There is no interaction between the GUIDE data structure code and the tool code or tool to device mapping code.

Figure 9-5 shows processing using system services which must occur following generation by GUIDE of an interface.

Figure 9-6 shows the structure of an application package using a GUIDE interface. Since GUIDE is such an application, the similarity of this figure to figure 9-3 is expected.

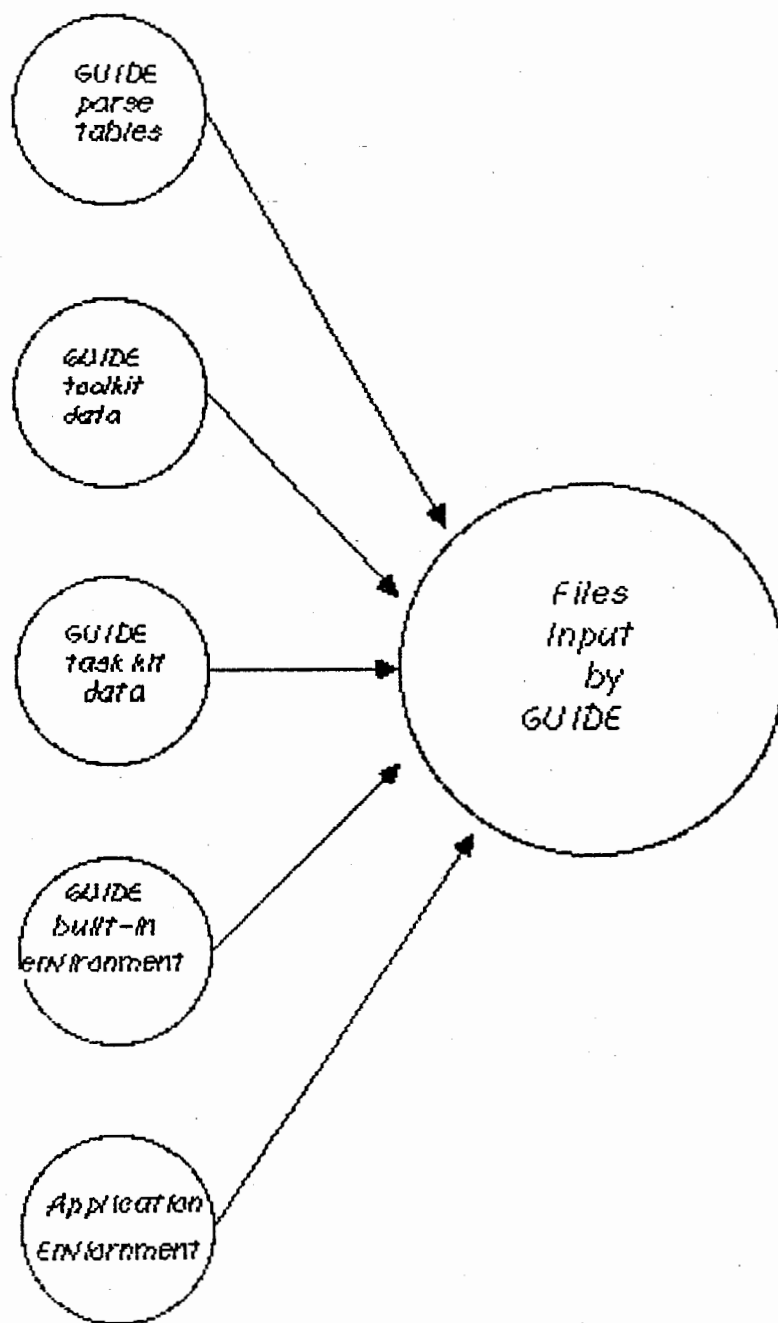


Figure 9-1: Files input by GUIDE

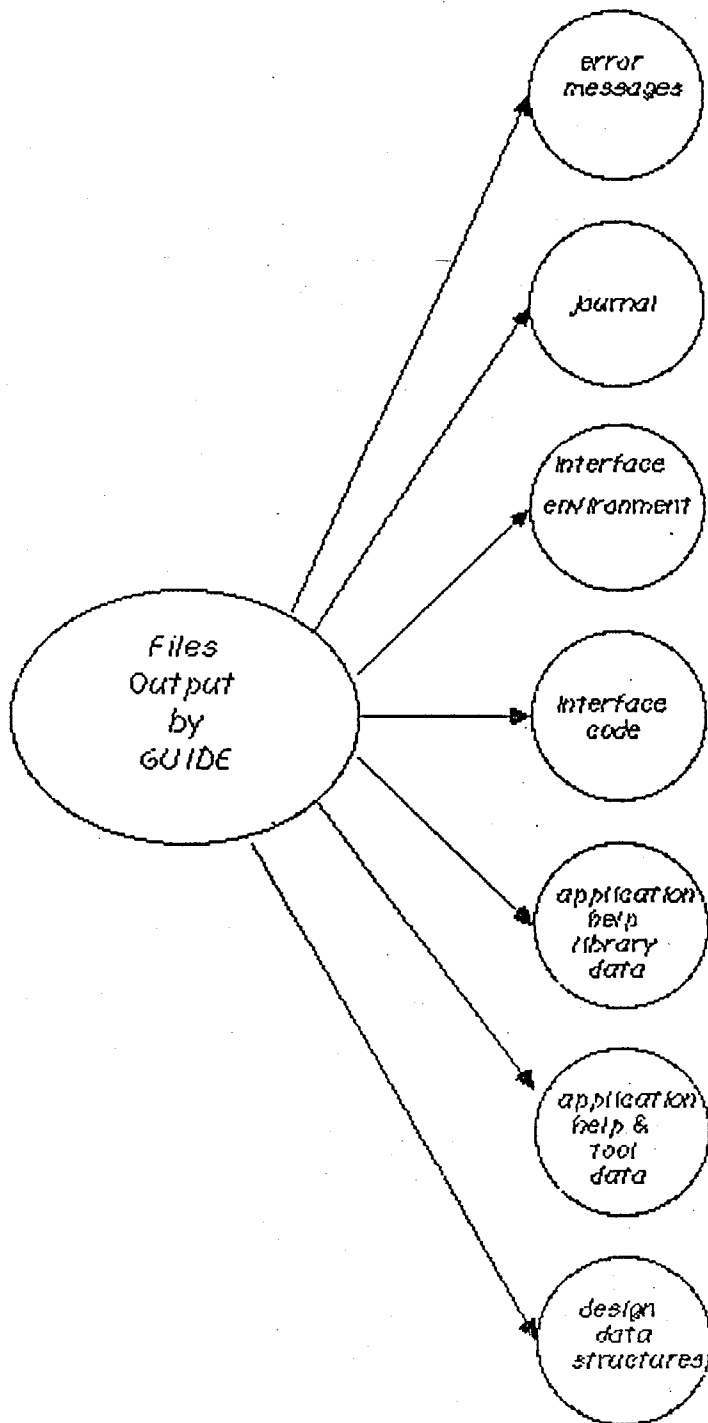


Figure 9-2: Files output by GUIDE

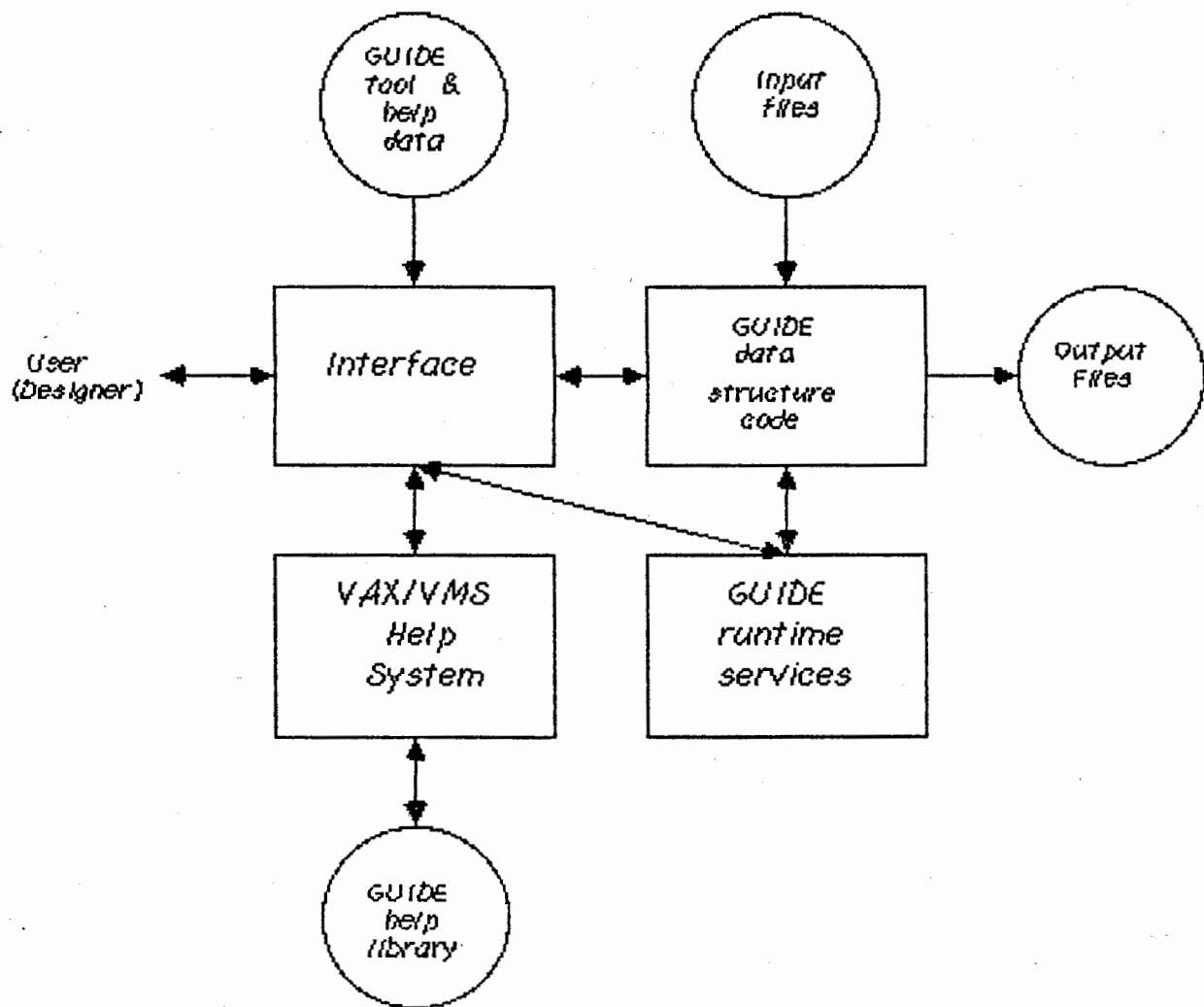


Figure 9-3: Data flow in GUIDE

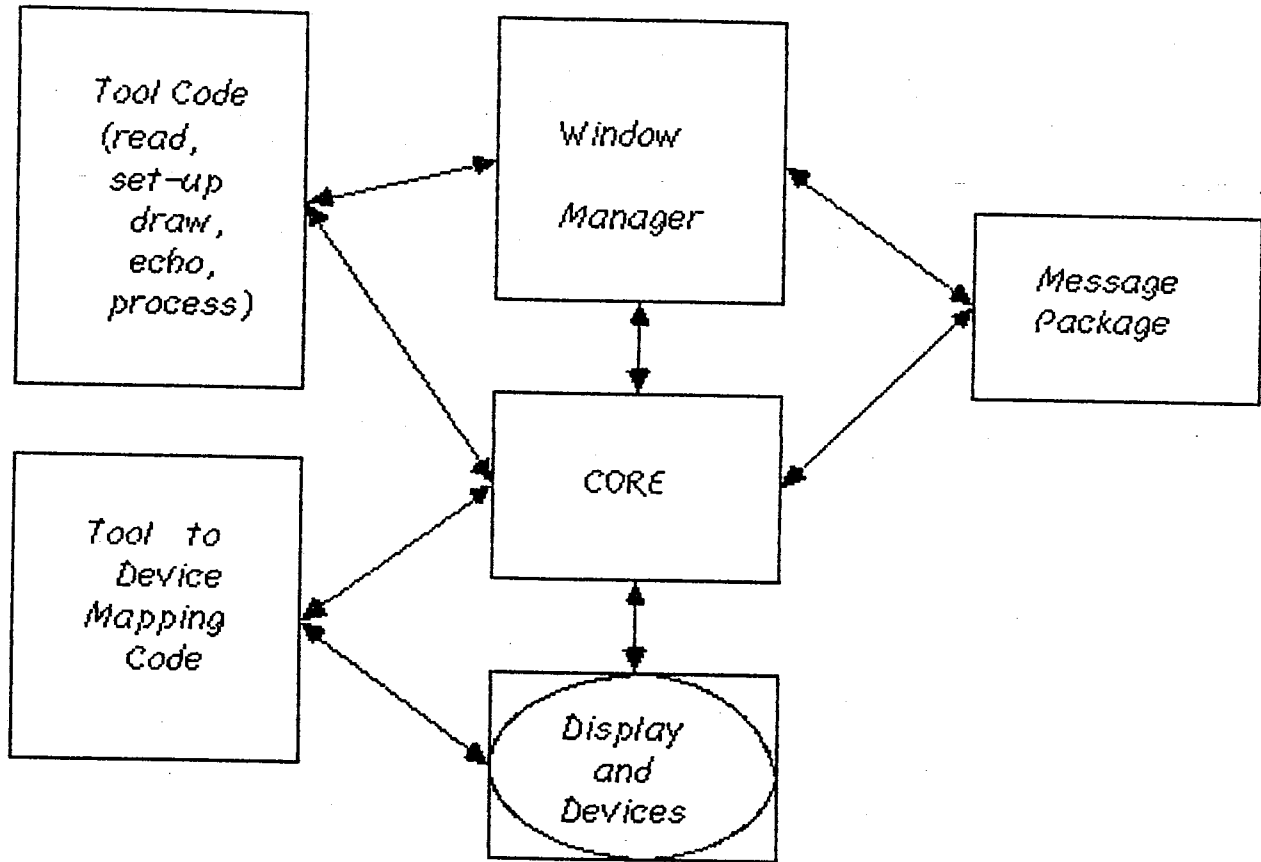


Figure 9-4: GUIDE runtime services

9.3. Choice of a graphics package

CORE was chosen as the underlying graphics package because of the availability of a local implementation. In most areas, the choice of CORE caused no serious problems in implementing the toolkit. As noted in chapter 3, it was necessary to associate CORE's sampled devices with buttons to make event tools. Since this capability is provided by CORE, it was not a serious problem.

The only area in which our CORE implementation was poorly suited to the desired task

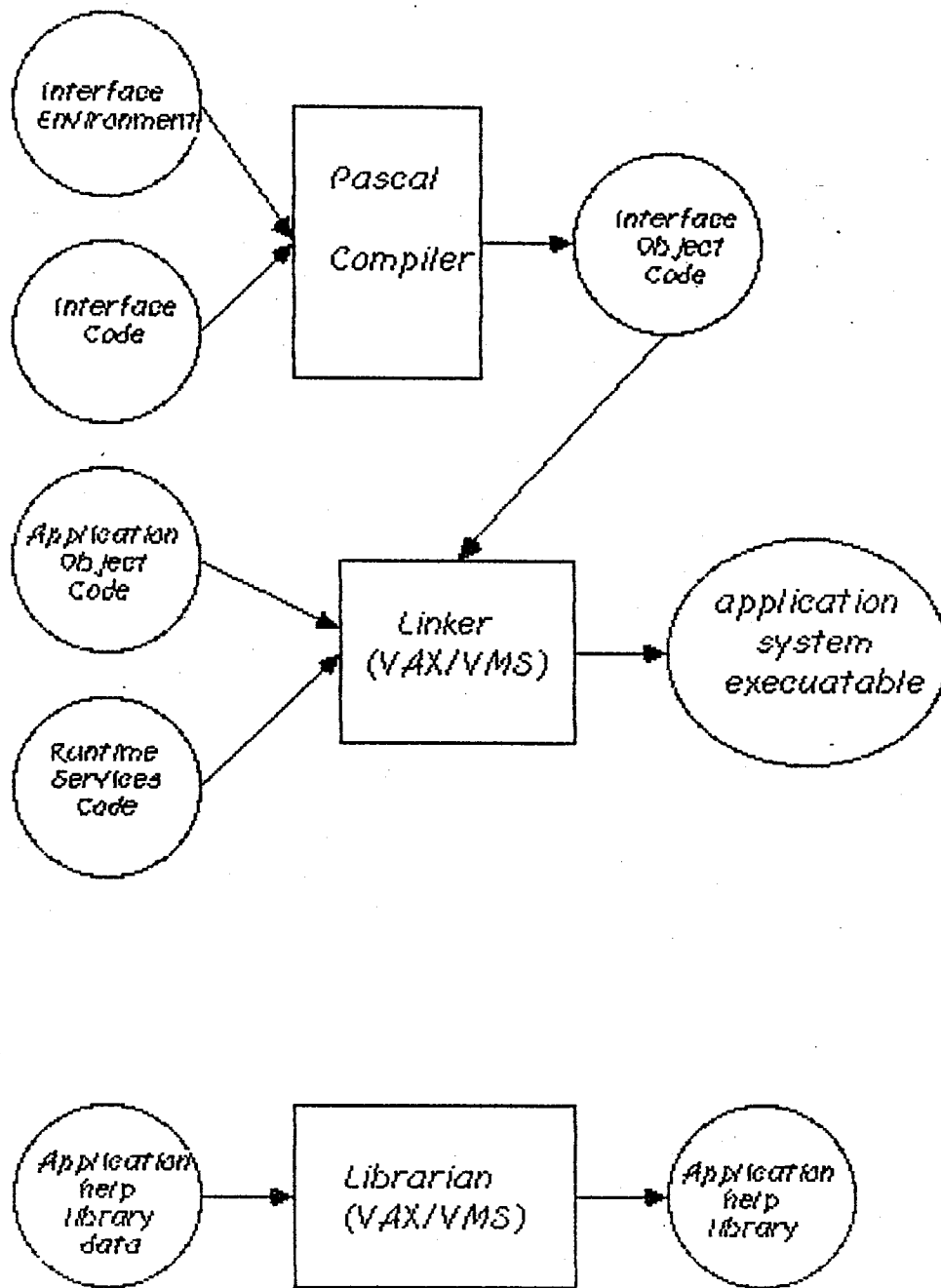


Figure 9-5: Processing after using GUIDE

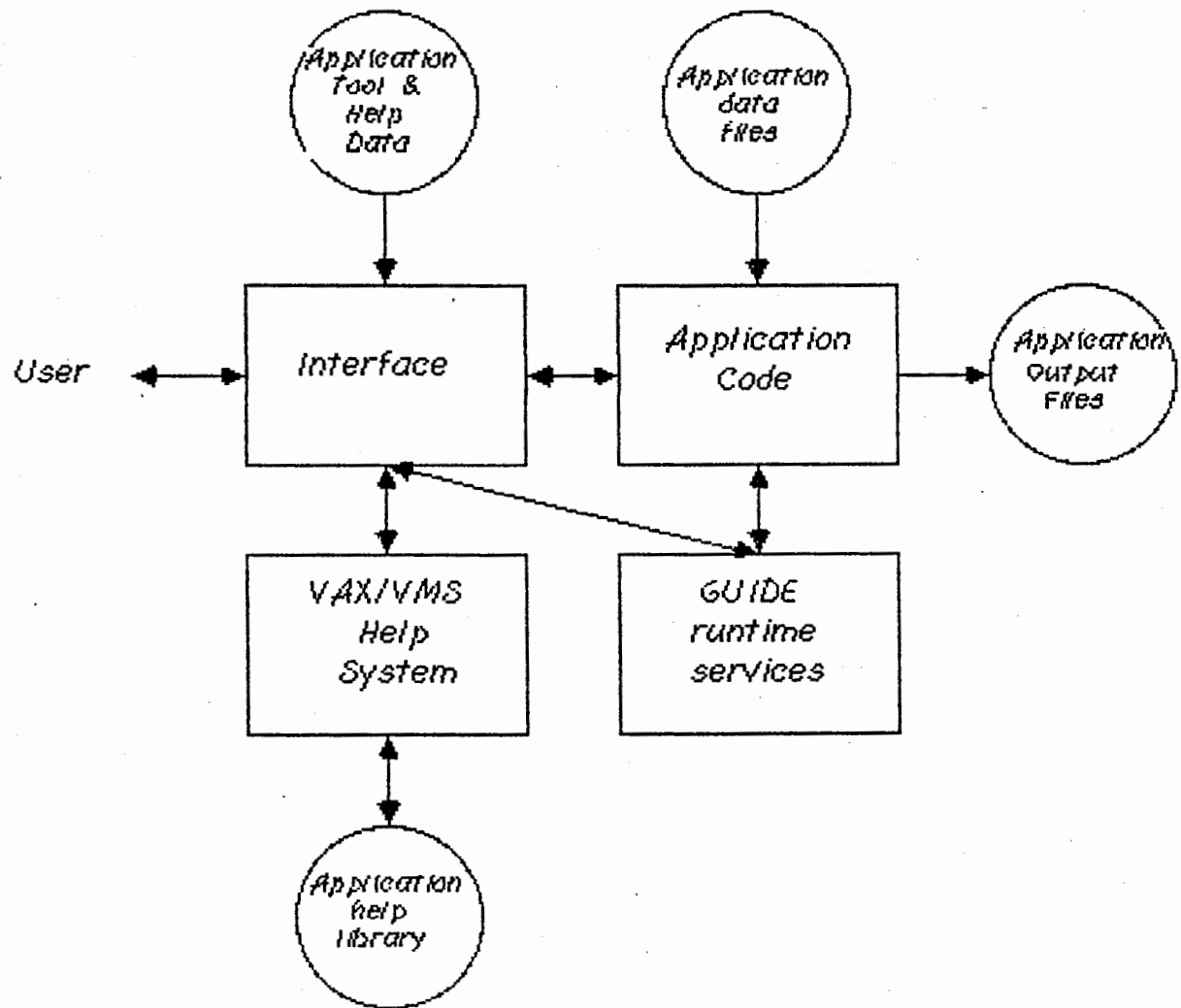


Figure 9-6: Structure of a GUIDE-generated application

was that of providing alternative echoes for pick and locator based tools. A complete CORE system would handle this better. However, GKS [Hopgood 83] seems better suited in this respect. The provision of this capability would allow the use of tools with dragging.

9.4. Representation of the Design

The design being constructed is represented by a collection of Pascal records. A single record represents the design itself; this record references other records representing the objects contained in the design. The entire collection of record types used to represent the design together with the routines which operate on them can be viewed as an abstract data structure.

Pascal records are used to represent each type of object defined in GUIDE. In addition, a number of record types are used to provide links between these GUIDE objects. Figure 9-7 shows a subset of the records used to define the user-defined picture portion of contexts.

A number of operations have been defined on the bulk of the objects. Almost every record type representing a GUIDE object has routines to create instances of the object type, to destroy instances and to update the fields of the instances. A few types have multiple update routines with each handling a subset of the fields. This approach is used when the set of fields naturally divides into unrelated subsets.

Additionally, routines exist implementing linkage operations. There are many routines to add one object to another or to a list of objects and to remove an object from another or from a list. Figure 9-8 shows some of the operations which can be performed on the records defined in figure 9-7.

9.5. The Symbol Tables

GUIDE uses several symbol tables. The first, the internal table, contains almost every dynamic record created in GUIDE, and is used for identifying objects when they are picked from the screen. The second table, the input table, is used in the input process for designs. (See section 9.7.) The third table is the declared symbol table. The table contains information about all of the named objects in GUIDE. It also contains descriptions of the identifiers defined in the application environment, as well as those built into Pascal and those defined in CORE and GUIDE which are available to the designer for use in expressions. This table is used to check types in expressions and for parameter matching.

All symbol tables in GUIDE are represented by hash tables. A generalized hash record is used, which contains the name being hashed, the type of record hashed and a pointer to the actual record being hashed. Pascal's variant record structure makes this organization economical.

```

context = record
    internal_name:string;
    already_written:boolean;
    context_name:string;
    context_code:integer; (*for generation*)
    context_help:help_node_ptr;
    context_prompt:prompt_node_ptr;
    context_td:task_decision_ptr;
    context_upic_classes:class_to_context_ptr;
    error_pos:point;
    error_vport:location_type;
    containing_controls:context_contained_ptr;
    next_context:context_ptr;
    case is_param_context:boolean of
        true:(param_tasks:param_task_list_ptr;
            param_context_lists:
                param_context_contained_ptr)
    end;
end;

class_to_context = record
    internal_name:string;
    already_written:boolean;
    contained_class:upic_class_ptr;
    containing_context:context_ptr;
    next_ctc:class_to_context_ptr
end;

upic_class = record
    internal_name:string;
    already_written:boolean;
    upic_class_name:string;
    class_contained_in:class_contained_ptr;
    next_upic_class:upic_class_ptr;
    case static_list:boolean of
        true:(upic_list:upic_ptr);
        false:(upic_list_head:string;
            get_next_upic:string;
            draw_upic_in_class:string;
            upic_type:ident_ptr)
    end;
end;

class_contained = record
    internal_name:string;
    already_written:boolean;
    containing_context:context_ptr;
    containing_ctc:class_to_context_ptr;
    contained_class:upic_class_ptr;
    next_cc:class_contained_ptr
end;

```

Figure 9-7: Records defining user-defined picture portion of contexts

```

upic = record
    internal_name:string;
    already_written:boolean;
    upic_name:string;
    is_fixed,
    is_permanent:boolean;
    upic_border_type:integer;
    upic_draw:fixed_action_ptr;
    next_upic:upic_ptr;
    upic_viewports:upic_to_vport_ptr;
    containing_upic_class:upic_class_ptr
end;

upic_to_vport = record
    internal_name:string;
    already_written:boolean;
    containing_context:context_ptr;
    upic_vport:vport_ptr;
    containing_upic:upic_ptr;
    next_utv:upic_to_vport_ptr
    (* for free list *)
end;

```

Figure 9-7, concluded

A common set of routines is used to add to, delete from and search in any of the tables. Routines for each table invoke the common routines, making the common code transparent to the routines which use the various hash tables.

9.6. Memory Management

Throughout GUIDE, many records are created and destroyed dynamically as the user manipulates a design. In order to improve efficiency and avoid the pitfalls of actually releasing memory, GUIDE maintains a free list of each type of dynamic record which appears in the system. Individual NEW and DISPOSE routines for each type maintain the lists, creating new records only when none of the appropriate type exist [Jones 82].

```

(***** module class2ctx *****)
procedure add_class_to_context(upc:upic_class_ptr;
                               c:context_ptr);extern;
procedure remove_class_from_context(upc:upic_class_ptr;
                                     c:context_ptr);extern;
procedure set_class_contained(upc:upic_class_ptr;
                              c:context_ptr;
                              ctc:class_to_context_ptr);
                              extern;

(***** module contexts *****)
procedure create_context(var p:context_ptr);extern;
procedure destroy_context(var c:context_ptr);extern;
procedure update_context(var c:context_ptr;name:string;
                        td_list:task_decision_ptr;
                        class_list:class_to_context_ptr;
                        errpos:point;
                        errvp:location_type);extern;
procedure make_context_param(c:context_ptr);extern;
procedure make_context_nonparam(c:context_ptr);extern;
procedure make_context_current(var contextp:context_ptr;
                               name:string);extern;
procedure set_context_pcl_contained(c:context_ptr;
                                    ctl:context_to_list_ptr);
                                    extern;

(***** module upics *****)
procedure position_upic(var up:upic_ptr;loc:location_type;
                       in_context:context_ptr);extern;
procedure assign_upic_vport(v:vport_ptr;up:upic_ptr;
                            c:context_ptr);extern;
procedure create_upic(var up:upic_ptr);extern;
procedure destroy_upic(var up:upic_ptr);extern;
procedure update_upic(var up:upic_ptr;name:string;
                     fixed,perm:boolean;border:integer;
                     draw:fixed_action_ptr);extern;
procedure remove_upic_from_vport(u:upic_ptr;
                                 v:vport_ptr);extern;
procedure make_upic_current(var upicp:upic_ptr;
                            name:string);extern;

```

Figure 9-8: Operations on context records

```

(***** module upicclass *****)
procedure create_upic_class(var upc:upic_class_ptr);extern;
procedure destroy_upic_class(var upc:upic_class_ptr);extern;
procedure add_upic_to_class(up:upic_ptr;
                           upc:upic_class_ptr);extern;
procedure remove_upic_from_class(up:upic_ptr;
                                upc:upic_class_ptr);extern;
procedure update_upic_class(var upc:upic_class_ptr;
                           name:string;
                           class_is_static:boolean;
                           class_list_head,class_get_next,
                           class_draw:string;
                           class_type:string);extern;
procedure make_upic_class_current(var upic_classp:
                                upic_class_ptr;
                                name:string);extern;

```

Figure 9-8, concluded

9.7. Reading and Writing Designs

There are two problems which arise in attempting to store the design which has been constructed in GUIDE. The first is that the design contains a large number of pointers, which will have no meaning if stored and read in. The second is that the graph formed by the design is non-hierarchical and may, in fact, contain cycles.

Pointer values are generally represented as memory addresses. Each time a program is run, it may be assigned a different memory area. There is no way of knowing where a particular record is stored. Therefore, storing and retrieving pointer values will normally result in errors when attempting to refer to the linked data structures.

When the data structures are linear, that is, each record points to and is pointed to by a single record, this difficulty can be overcome by storing the records in their linked order. Then, on input, it is known that each record should be made to point to the record which follows it in the file. Even such structures as trees can be stored easily and retrieved by storing them in a fixed order and using some representation for an empty record or the end of a list.

The problem becomes serious when the data structures to be stored do not form a simple hierarchy. If two records may point to the same record or if pointers may form a cycle in which it is possible to return to a record, a method is needed to determine which records have already been processed so as to avoid infinite recursion. Figure 9-9 shows a group of records in which these problems occur. The data structures shown are a simplified version of those used to represent menus in GUIDE.


```

menu_instance = record
    menu_label:string;
    display_type:orientation_type;
    paging_allowed:boolean;
    page_size:integer;
    page_message:string;
    omitted_items_display:omission_type;
    background_color:color_type;
    maintain_history:boolean;

    start_item:menu_item_ptr;
    next_menu:menu_instance_ptr
end;

menu_item_list = record
    include_item:condition_ptr;
    item_to_use:menu_item_ptr;
    next_menu_item_list:menu_item_list_ptr
end;

menu_item = record
    item_code:integer;
    follow_list:menu_item_list_ptr;
    containing_menu:menu_instance_ptr;
    text_string:string;
    text_color:color_type;
    next_menu_item:menu_item_ptr;
end; (* menu_item *)

```

Figure 9-9: Some sample data structures

The solution used in GUIDE is based on a marking technique more commonly used in garbage collection [Knuth 73]. Each record is marked when it is stored and is stored only once. Pointers are replaced in the output file by a string reference to the associated record. On input, a symbol table of these strings is built and used to re-create the pointer references. The order of output ensures that input can be completed in one pass. The marking technique is also used for other traversals of the design data structure and its components.

9.7.1. Overview of the solution

The method for storing these data structures requires two additional fields in each record, an internal name (*internal_name:string*) and a boolean to indicate whether the record has been written yet (*already_written:boolean*). Figure 9-10 shows the sample data structures with these additional fields.

```

menu_instance = record
    internal_name:string;
    already_written:boolean;
    menu_label:string;
    display_type:orientation_type;
    paging_allowed:boolean;
    page_size:integer;
    page_message:string;
    omitted_items_display:omission_type;
    background_color:color_type;
    maintain_history:boolean;

    start_item:menu_item_ptr;
    next_menu:menu_instance_ptr
end;

menu_item_list = record
    internal_name:string;
    already_written:boolean;
    include_item:condition_ptr;
    item_to_use:menu_item_ptr;
    next_menu_item_list:menu_item_list_ptr
end;

menu_item = record
    internal_name:string;
    already_written:boolean;
    item_code:integer;
    follow_list:menu_item_list_ptr;
    containing_menu:menu_instance_ptr;
    text_string:string;
    text_color:color_type;
    next_menu_item:menu_item_ptr;
end; (* menu_item *)

```

Figure 9-10: Sample data structures with additional fields

On output, the *already_written* field controls recursion. When a record is reached which has already been written, recursion terminates. Each pointer is replaced with the internal name of the record to which it points.

On input, the internal name of each record is used as a key to a symbol table. After reading a record, the symbol table is searched for the internal name of each pointer field. If it is found, the pointer is made to point to the associated record. If not, that record is the next one to be read from the file.

9.7.2. The Symbol Table

Each record to be output must have a unique internal name. It is a simple matter to generate these when records are created. At least for the input phase, these names must be stored in a symbol table. In GUIDE, they are stored in the input symbol table mentioned in section 9.5. When the input process has been completed, this table can be cleared.

9.7.3. The Output Phase

The output section of the program is designed so that it is possible to save the same design more than once in a session, whether into different versions of the same file or separate files. Output is performed in two passes. Two procedures are needed for each record type, one for each pass.

The first pass starts with the design record, and visits each record in the design recursively, marking it as *unwritten* (i.e., *already_written:=false*). When a record is visited which has already been marked, no recursive calls are made. The routines implementing this phase for the sample structures shown above are in figure 9-11.

In the second pass, the records are visited in the same order. The fields, including the internal name, of each record are written. Any time a pointer field is encountered, the *internal_name* of the record to which it points is written. If a pointer is nil, a pre-determined string (e.g., 'nil') is written. The record is then marked as *written*, (i.e., the *already_written* field becomes true). Then, each pointer field is examined, and the routine for writing the type of record to which it points is invoked with the pointer value as parameter. When the pointer is nil or the record has already been marked, the procedure terminates without doing anything. Figure 9-12 shows the procedures for this phase.

It should be noted that the criterion for determining whether a record has been visited is different in the two passes. In the first pass, a record has been visited if it has *already_written=false*. In the second pass, it has already been visited if *already_written=true*. In order to ensure that the first pass does, in fact, visit all records, all newly-created records have *already_written* set to true.

```

procedure clear_menu_instance(p:menu_instance_ptr);
(* procedure to clear a record of type menu_instance *)

begin
  if p<>nil
    then with p^ do
      if already_written
        then begin
          already_written:=false;
          clear_menu_item(start_item);
          clear_menu_instance(next_menu);
        end; (* with *)
    end; (* clear_menu_instance *)

procedure clear_menu_item_list(p:menu_item_list_ptr);
(* procedure to clear a record of type menu_item_list *)

begin
  if p<>nil
    then with p^ do
      if already_written
        then begin
          already_written:=false;
          clear_condition(include_item);
          clear_menu_item(item_to_use);
          clear_menu_item_list(next_menu_item_list);
        end; (* with *)
    end; (* clear_menu_item_list *)

procedure clear_menu_item(p:menu_item_ptr);
(* procedure to clear a record of type menu_item *)

begin
  if p<>nil
    then with p^ do
      if already_written
        then begin
          already_written:=false;
          clear_menu_item_list(follow_list);
          clear_menu_instance(containing_menu);
          clear_menu_item(next_menu_item);
        end; (* with *)
    end; (* clear_menu_item *)

```

Figure 9-11: Clear routines for sample data structures

```

procedure write_menu_instance(var outfile:text;
                             p:menu_instance_ptr);
(* procedure to write out a record of type menu_instance *)

begin
  if p<>nil
    then with p^ do
      if not already_written
        then begin
          writeln(outfile,internal_name);
          writeln(outfile,menu_label);
          write_orientation_type(outfile,display_type);
          writeln(outfile,paging_allowed);
          writeln(outfile,page_size);
          writeln(outfile,page_message);
          write_omission_type(outfile,
                              omitted_items_display);
          write_color_type(outfile,background_color);
          writeln(outfile,maintain_history);
          if start_item = nil
            then write_nil(outfile)
            else writeln(outfile,
                        start_item^.internal_name);
          if next_menu = nil
            then write_nil(outfile)
            else writeln(outfile,
                        next_menu^.internal_name);

          (* set flag *)
          already_written:=true;

          (* now write subparts *)
          write_menu_item(outfile,start_item);
          write_menu_instance(outfile,next_menu);
        end; (* with *)
    end; (* write_menu_instance *)

```

Figure 9-12: Output routines for sample data structures

9.7.4. The Input Phase

The input section uses one procedure for each record type. Each procedure reads in the fields of the record in the order in which they were written. The strings representing the pointer fields are stored in local variables. After all data for this record has been read in, the record is added to the symbol table.

The next step is repeated for each pointer field in the record. If the local variable for the

```

procedure write_menu_item_list(var outfile:text;
                               p:menu_item_list_ptr);
(* procedure to write out a record of type menu_item_list *)

begin
  if p<>nil
    then with p^ do
      if not already_written
        then begin
          writeln(outfile,internal_name);
          if include_item = nil
            then write_nil(outfile)
            else writeln(outfile,
                        include_item^.internal_name);
          if item_to_use = nil
            then write_nil(outfile)
            else writeln(outfile,
                        item_to_use^.internal_name);
          if next_menu_item_list = nil
            then write_nil(outfile)
            else writeln(outfile,
                        next_menu_item_list^.internal_name);

          (* set flag *)
          already_written:=true;

          (* now write subparts *)
          write_condition(outfile,include_item);
          write_menu_item(outfile,item_to_use);
          write_menu_item_list(outfile,
                                next_menu_item_list);
        end; (* with *)
    end; (* write_menu_item_list *)

```

Figure 9-12, continued

```

procedure write_menu_item(var outfile:text;p:menu_item_ptr);
(* procedure to write out a record of type menu_item *)

begin
  if p<>nil
    then with p^ do
      if not already_written
        then begin
          writeln(outfile,internal_name);
          writeln(outfile,item_code);
          if follow_list = nil
            then write_nil(outfile)
            else writeln(outfile,
              follow_list^.internal_name);
          if containing_menu = nil
            then write_nil(outfile)
            else writeln(outfile,
              containing_menu^.internal_name);
          writeln(outfile,text_string);
          write_color_type(outfile,text_color);
          if next_menu_item = nil
            then write_nil(outfile)
            else writeln(outfile,
              next_menu_item^.internal_name);

          (* set flag *)
          already_written:=true;

          (* now write subparts *)
          write_menu_item_list(outfile,follow_list);
          write_menu_instance(outfile,containing_menu);
          write_menu_item(outfile,next_menu_item);
        end; (* with *)
    end; (* write_menu_item *)

```

Figure 9-12, concluded

field is not the predetermined nil value, the symbol table is searched for the variable value. If it is found, then the desired record has already been read in and the pointer is set to point to the record. If it is not found, the procedure to read in a record of the appropriate type is invoked.

As long as input starts with the same record as output, the data will be requested in exactly the same order as it was written out. The procedures for input of the sample data structures are shown in figure 9-13.

```

procedure read_menu_instance(var infile:text;
                             var p:menu_instance_ptr);
(* procedure to read out a record of type menu_instance *)
var hp:hash_ptr;
    int_name:string;
    start_item_name:string;
    next_menu_name:string;
begin
    new_mi(p);
    with p^ do
    begin
        readln(infile,int_name);
        readln(infile,menu_label);
        read_orientation_type(infile,display_type);
        read_boolean(infile,paging_allowed);
        readln(infile,page_size);
        readln(infile,page_message);
        read_omission_type(infile,omitted_items_display);
        read_color_type(infile,background_color);
        read_boolean(infile,maintain_history);
        readln(infile,start_item_name);
        readln(infile,next_menu_name);

        (* add to symbol table *)
        add_mi_to_input(p,int_name);

        (* now read sub-structures *)
        if is_nil_string(start_item_name)
        then start_item:=nil
        else begin
            search_input(start_item_name, hp);
            if hp=nil
            then read_menu_item(infile,start_item)
            else if hp^.hashed_rec =menu_item_rec
            then start_item:=hp^.hashed_menu_item
            else set_error(1,'name is wrong type',
                          'read_menu_instance');
        end;
    end;
end;

```

Figure 9-13: Input routines for sample data structures


```

    if is_nil_string(next_menu_name)
    then next_menu:=nil
    else begin
        search_input(next_menu_name, hp);
        if hp=nil
        then read_menu_instance(infile, next_menu)
        else if hp^.hashed_rec =mi_rec
            then next_menu:=hp^.hashed_mi
            else set_error(1, 'name is wrong type',
                          'read_menu_instance');
        end;
    end; (* with *)
end; (* read_menu_instance *)

procedure read_menu_item_list(var infile:text;
                              var p:menu_item_list_ptr);
(* procedure to read out a record of type menu_item_list *)

var hp:hash_ptr;
    int_name:string;
    include_item_name:string;
    item_to_use_name:string;
    next_menu_item_list_name:string;

begin
    new_mil(p);
    with p^ do
    begin
        readln(infile, int_name);
        readln(infile, include_item_name);
        readln(infile, item_to_use_name);
        readln(infile, next_menu_item_list_name);

        (* add to symbol table *)
        add_mil_to_input(p, int_name);

        (* now read sub-structures *)
        if is_nil_string(include_item_name)
        then include_item:=nil
        else begin
            search_input(include_item_name, hp);
            if hp=nil
            then read_condition(infile, include_item)
            else if hp^.hashed_rec =condition_rec
                then include_item:=hp^.hashed_condition
                else set_error(1, 'name is wrong type',
                              'read_menu_item_list');
            end;
        end;
    end;
end;

```

Figure 9-13, continued

```

if is_nil_string(item_to_use_name)
  then item_to_use:=nil
  else begin
    search_input(item_to_use_name, hp);
    if hp=nil
      then read_menu_item(infile, item_to_use)
      else if hp^.hashed_rec = menu_item_rec
        then item_to_use:=hp^.hashed_menu_item
        else set_error(1, 'name is wrong type',
          'read_menu_item_list');
    end;

if is_nil_string(next_menu_item_list_name)
  then next_menu_item_list:=nil
  else begin
    search_input(next_menu_item_list_name, hp);
    if hp=nil
      then read_menu_item_list(infile,
        next_menu_item_list)
      else if hp^.hashed_rec = mil_rec
        then next_menu_item_list:=hp^.hashed_mil
        else set_error(1, 'name is wrong type',
          'read_menu_item_list');
    end;

  end; (* with *)
end; (* read_menu_item_list *)

procedure read_menu_item(var infile:text; var p:menu_item_ptr);
(* procedure to read out a record of type menu_item *)

var hp:hash_ptr;
  int_name:string;
  follow_list_name:string;
  containing_menu_name:string;
  next_menu_item_name:string;

begin
  new_menu_item(p);
  with p^ do
    begin
      readln(infile, int_name);
      readln(infile, item_code);
      readln(infile, follow_list_name);
      readln(infile, containing_menu_name);
      readln(infile, text_string);
      read_color_type(infile, predef_text_color);
      readln(infile, next_menu_item_name);
    end;
  end;
end;

```

Figure 9-13, continued

```

(* add to symbol table *)
add_menu_item_to_input(p,int_name);

(* now read sub-structures *)
if is_nil_string(follow_list_name)
then follow_list:=nil
else begin
  search_input(follow_list_name, hp);
  if hp=nil
  then read_menu_item_list(infile,follow_list)
  else if hp^.hashed_rec =mil_rec
  then follow_list:=hp^.hashed_mil
  else set_error(1,'name is wrong type',
                 'read_menu_item');
end;

if is_nil_string(containing_menu_name)
then containing_menu:=nil
else begin
  search_input(containing_menu_name, hp);
  if hp=nil
  then read_menu_instance(infile,containing_menu)
  else if hp^.hashed_rec =mi_rec
  then containing_menu:=hp^.hashed_mi
  else set_error(1,'name is wrong type',
                 'read_menu_item');
end;

if is_nil_string(next_menu_item_name)
then next_menu_item:=nil
else begin
  search_input(next_menu_item_name, hp);
  if hp=nil
  then read_menu_item(infile,next_menu_item)
  else if hp^.hashed_rec =menu_item_rec
  then next_menu_item:=hp^.hashed_menu_item
  else set_error(1,'name is wrong type',
                 'read_menu_item');
end;

end; (* with *)
end; (* read_menu_item *)

```

Figure 9-13, concluded

9.7.5. Problems

In GUIDE, the internal names of the records have significance other than just for input/output of the data structures. Also, it is possible to read in data structures while some already exist (*e.g.*, to merge two structures). In this case, some names may be repeated.

The solution GUIDE uses for these problems is to use the stored internal names only for reading in the data and then, as part of the input process, to assign new internal names to the records being read. Two different symbol tables are used, one for input only and one for the names actually in use when the input begins. All the nodes in the input table can be released as soon as the input is completed. Since the stored internal name for each record is needed only long enough to store the record in the symbol table, it is stored in a local variable.

Another problem which occurs is that some nodes may not be part of the design data structure. This happens because the designer may create an object and then choose not to add it to the design or may end a session without completing his design. This problem is solved by using the symbol table of all records which is maintained by GUIDE. After traversing the design data structure, on each output pass, we traverse the symbol table, looking for records which have not been appropriately marked. In the first pass, we mark them as *unwritten*. In the second pass, we write them out, preceded by a string identifying the record type. As in the design data structure, we follow all pointers and write out the associated records. Only those records written directly from the symbol table need to be preceded by their types.

In the input phase, after the design data structure has been read in, we check to see whether anything remains in the file. If so, there are records not contained in the design data structure. The type is read and used to determine which input procedure to invoke, continuing in this manner until the end of the file.

9.8. Copying Data Structures

In order to simplify the process of creating a design, it is desirable to be able to copy a structure which the designer has already created. However, the meaning of "copying" a structure varies depending both on the structure to be copied and whether the structure is being copied alone or as part of a larger structure. For example, copying a condition alone suggests making a duplicate of the condition record with the same expression, whereas, copying a condition as part of a decision suggests making a reference to the existing condition record.

Problems also arise because the data structures contain a number of records whose sole function is linkage among the records which actually contain the design data. When copying a structure, clearly it is necessary to make the new copies of such records refer to the new copies of the actual data structures rather than the original structures being copied.

A solution to these problems has been worked out, providing an "intelligent" copy facility for all of the records contained in GUIDE. For any record field, there are four possible actions which can be taken when copying the record, depending upon the circumstances of the copy and the type of the field. First, a field value can be duplicated. This is done with most scalar fields. Second, a pointer reference can be copied, making the new record refer to the same record as the original. Third, a structure which is referenced by pointer can be copied via this facility. Last, an entirely new value can be inserted. This is done, for example, with the *internal_name* fields discussed in section 9.7. In some cases, the new value to be inserted is passed as a parameter to the copy routine. This allows, for example, for a tool being copied as part of a task copy operation to point to the new task copy as its *containing_task*.

The copying of menu structures is particularly complicated due to the generality of the data structures used. The algorithm for copying a menu instance borrows from the input/output method discussed in section 9.7. The menu data structure is written to a temporary file and read back in. Reading the structure in has the side effect of creating a duplicate data structure. The output routines are sufficiently modified to take into account the distinctions among the different kinds of fields, and allow those changes necessary to make a usable copy.

9.9. Tool/Logical Device Mappings

One of the problems which arises in a UIMS is mapping tools onto logical input devices. In particular, many tools may use the same logical device. Even within a single context, more than one tool may use some device. For example, a context may contain several keyboard tools, all of which use the same keyboard. So, a method is needed to determine which tool is intended when a device is used.

The first thing to note is that there is no problem when the device used is a pick device. The location of the pick determines which tool is intended. This observation is used in the solution for other devices. The user is required to pick the intended tool prior to using it.

A current tool is maintained for each physical device. When the user picks a point, the window manger determines what viewport it is in. If the viewport is that of a non-pick-based

tool, that tool becomes the current tool for the device on which it is based. Otherwise, the pick is processed as an input to the appropriate tool. When a device other than a pick is used, its current tool is looked up and the input is processed by that tool.

It is felt that the small amount of work necessary to make the appropriate tool current is far outweighed by the convenience of being able to include multiple tools based on the same device in a context.

9.10. The parser

The parser which is used by GUIDE to check expressions is based on a parser generator system of Gallier [Schimpf 81, Gallier 85]. The generator produces tables for an LR(1) parser. The kernel of this system was modified to allow the use of multiple sets of parse tables and to allow the parser to be invoked from other programs.

The parser is used by GUIDE for three distinct tasks:

- reading in the application environment to set up the symbol table;
- checking expressions entered as conditions or parameters for syntactic and type correctness;
- checking restore list expressions for syntactic and type correctness.

Each of the three requires a different grammar and therefore a different set of parse tables. Semantic routines had to be added for each case to perform the appropriate actions on the GUIDE data structures.

The original parser was a stand-alone program which took an input file and parsed it. Since it was necessary to be able to call the parser repeatedly and to continue processing after parsing, the program was modified to allow its use as a set of procedures.

CHAPTER X

Conclusions

This chapter discusses experience with GUIDE. Included are problems which have arisen and ways in which the system's behavior was different from that which was expected. It also describes some of the future directions which could be taken with GUIDE. Some of the future directions mentioned are improvements to the implementation while others are modifications of the design of GUIDE, possibly requiring further research.

10.1. Unavailable Features

There are a number of features included in GUIDE's design which have not yet been implemented. In addition, a number of features were implemented, but are not accessible from the prototype interface. The next two sections briefly describe each of these features.

10.1.1. Features not yet implemented

The following are part of the design of GUIDE, but have not been included in the initial implementation.

- Journal file - No facility for creating a record of a GUIDE user's session has been included.
- Undo - Since this command is expected to be based on the journal file, it has not been included. When it is implemented, it should be added as a task-consequence.
- Abort - This command also has not been implemented. Inclusion of an abort facility, which would halt the current parameter input process without calling the specified action routine, seems to require the addition of a single global variable to the GUIDE-generated interface. This variable, which would be set to true when the abort task is used, would control continuation of the parameter context list and the call to the action. Like "undo", "abort" is a task-consequence.
- Variations on Tools - Thus far, a single configuration has been implemented for each tool type. For lists, menus and forms, only vertical orientation has been

provided. Horizontal potentiometers have been implemented. Where possible, the prototype interface defaults to these characteristics.

Among the other tool features not yet implemented are the use of icons, generation of lists, hierarchical lists, generation of menu items, command-parser type menus and different cursor types for picks and locators.

- Re-reading the environment file - No provision has been made to allow GUIDE to re-read the application environment file in the course of a session to allow re-checking of procedure headers. This feature will not be useful until access to the editor from GUIDE is provided (see section 10.1.2).

10.1.2. Inaccessible Features

The following features have been implemented, but are not accessible from the GUIDE interface. In most cases, appropriate commands need to be added to the interface.

- Copying data structures - Routines to copy all GUIDE data structures in the manner described in section 9.8 exist. However, the prototype interface contains no commands providing access to these routines.
- Help - Due to a problem in accessing help libraries from Pascal, help has not been integrated with VMS Help. Therefore, the help task-consequence has not been included in the interface. However, the designer is able to create and modify help messages and to include the help task-consequence in the application interface. Therefore, when help is fully integrated either with VMS Help or with an alternative system, applications should be able to provide help immediately.
- Designer-defined Tool Routines - The prototype interface does not provide the capability for a designer to specify the draw, echo and process routines for a tool.
- Naming of Expressions and Conditions - No provision has been made in the interface for specifying names for expressions and conditions.
- Invocation of the Editor - No provision is made in the interface for invoking the editor as a sub-process.

10.2. Problems with defaults

Providing defaults at the task level has entailed several unforeseen problems. The first problem which arose was the interaction between computation of defaults and the stacking of contexts. The original algorithm called for each task's default value to be computed at the start of any context containing that task. However, if a context is stacked and later popped, it is undesirable to recompute the default value when re-entering the original context, since the task value may have been changed by the contexts which were visited in the interim. Note that this case is different than the case of a context which is entered, exited without stacking and later entered again. In this latter case, the default must be recomputed.

It was, therefore, necessary to find a method of determining, upon entry to a context, whether the context is being drawn for the first time of this use or whether it is simply being re-drawn. Since it is also undesirable to recompute defaults when the same context is used repeatedly, this algorithm also distinguishes between entry to a new context and re-entry to the context just used.

This algorithm has some undesirable consequences in the prototype GUIDE interface. The form which is used for the design itself uses default values drawn from the current design. This form is drawn in the main GUIDE context and therefore has its defaults computed soon after entry to GUIDE. If the designer then reads in a design, the current design is updated. However, the design form is not modified since the defaults are computed only the first time the context is drawn. This problem has not been resolved.

A second problem with the default computation algorithm arises when a form is given a default. In this case, it is possible that the termination task for the form will appear even if the values in some fields are not appropriate.

10.3. Conversion Routines

In the interface to GUIDE, the conversion routines from tools to tasks acquired more importance than was expected. Rather than just performing a type conversion, many of these routines actually create new records and fill in some values. This is due primarily to the use of keyboard tools for specifying objects which are referenced by pointers. For example, when a designer wants to create or edit a tool, he must enter the tool name. This name is then looked up in the symbol table. If a tool with that name exists, the task is assigned a pointer to it. If no such tool exists, it must be created for the task to have a usable value.

A second area in which conversions have taken on more importance than anticipated is in the conversion of form tools to their respective tasks. Again, these task values are generally pointers and the conversion routines must create a record, if necessary, and then assign the contained values to the respective fields of the record.

10.4. Experience with GUIDE

To date, GUIDE has been used to provide interfaces for GUIDE and for the layout system. The interface to GUIDE was generated using GUIDE as a subroutine package. The program which builds the necessary data structure and generates the interface is approximately 4,400 lines, most of them calls to GUIDE routines. The interface main program generated by GUIDE for itself is about 25,000 lines.

The interface for the layout system was designed interactively using the above interface. It took about 4 hours of real time to build the design. The resulting program is about 1,700 lines.

10.5. Future Directions

10.5.1. Expression Functions

One way in which GUIDE's efficiency could be improved is to optimize generation of expression functions. That is, prior to generating the functions, the list of expressions should be analyzed and a single function generated for those expressions which are identical. Care must be taken not to modify the design data structure in this process as it may be changed and expressions which were identical in one version may not be in a later version.

10.5.2. Tool Process Routines

GUIDE currently generates a routine for each tool instance which invokes the appropriate echo and process routines for the tool based on its type. It would be more economical to use one routine for each tool type and have the particular tool instance as a parameter. At such time as designer-specified routines are permitted for echoing and processing, any tools which have such routines specified would require distinct routines to make those calls.

10.5.3. Additional use of conditions

One way in which GUIDE's generality can be increased is to allow the designer to specify conditions regulating the inclusion of tools in tasks. In this way, the set of tools in any given task may vary according to the user and the state of the system. This would allow the designer to make some judgments about which tools are appropriate in certain situations and about whether certain tools are useful to some users.

Along the same lines, conditions could be added to user-defined pictures, determining their inclusion in their associated classes. In this way, the designer could restrict the view of the application as he sees fit.

10.5.4. Changes to Default Task Values

Experience with GUIDE has shown that allowing a single default value for each task leads to creation of a very large number of tasks. Several modifications would improve this situation. First, defaults for tasks could be specified with respect to contexts and forms. That is, a designer could indicate that a specific task is to have a certain default value in a specific context or in all appearances of a specified form. This is similar to the way in which locations for tools are specified.

Another way in which defaults could be modified is to add conditions to defaults so that the designer can specify the circumstances under which a task is to receive some default value.

It is unclear whether one of the above changes is sufficient to solve the problems with defaulting or whether it would be desirable to implement both.

10.5.5. Default Device Selection

When only one tool in a context uses a particular logical device, the current tool for that device should be set to that tool automatically upon entry to the context.

It would also be possible to allow the designer to specify an initial tool for each device for each context. This would allow the user to use a more common tool without the necessity of picking it first.

10.5.6. Device Specification

Currently, the tools are implemented for use with one particular set of graphics devices. It would be desirable to add specification of the display type to GUIDE and have the tools be drawn according to the type of display in use. This would be particularly useful when no graphics display is available and users must work with a video display terminal only.

10.5.7. Use of a database

The code which manipulates the GUIDE data structures is extremely redundant due to the restrictions of Pascal. It would be very useful to use a data base management system to handle this part of GUIDE and then to re-write the remaining code to make calls to the DBMS. This would also provide an opportunity to test the simultaneous use of a DBMS and a UIMS. It is unknown whether a DBMS can provide efficient enough access to the data to give reasonable response time in GUIDE.

10.5.8. Maintaining Session Status in the User Profile

The user profile can be made more useful by including in it information about the status of the user from session to session. The ability to store session status would increase the power of default task values, particularly if conditions were added as suggested in section 10.5.4. It also would allow decisions to be based on previous activity by the user, not just that in the current session.

While this can be done at present in a roundabout way, a modification of the user profile would make inclusion of session status a simple matter transparent to the application user. At present, any modifications to user profile field values come from the tasks with which they are associated. For the profile to be updated, therefore, the form containing these tasks must be drawn, and used by the application user.

To include session status, we need to introduce a new type of profile field, one which takes its value directly from an expression. Then, the designer can indicate that these fields are to be evaluated and the profile stored prior to ending an application session.

APPENDIX A

Entities built into GUIDE

This appendix contains the declarations of objects built into GUIDE for use by the designer.

```
module builtins;

type text = file of char;
tool_types = (no_tool, form_tool, list_tool, menu_tool,
              pwl_tool, pot_tool, windcont_tool,
              keyboard_tool, pick_tool, vwb_tool,
              lwb_tool, button_tool);

string=varying[32] of char;
long_string=varying[255] of char;

point = record
    x,y:real
end;

viewp_dim = (top,bottom,rightdim,leftdim);
real_dim_array = array[viewp_dim] of real;
boolean_dim_array = array[viewp_dim] of boolean;
integer_dim_array = array[viewp_dim] of integer;

location_type = array[viewp_dim] of real;

adjustment = (shrink,expand,shift,lock,extend);

orientation_type = (horizontal,vertical,random);

termination_type = (implied,explicit);

color_type = array[1..3] of real;

list_item_types = (textstring,list_icon);

two_d_pos = array[1..4] of real;

omission_type = (halftone,leave_space,omit);
```

```

menu_item_types = (predef_text, icon_item, gen_text);

potentiometer_types = (round_pot, horiz_pot, vert_pot);

help_types = (tool_help, task_help, context_help);

window_params = record
    xtremes:location_type;
    window_used:location_type
end;

(* the following declaration will allow the user
   to create a user profile
   without getting errors regarding task type *)

user_profile_type = record
    dummy_up_field:integer;
    (* to placate parser *)
end; (* user_profile_type *)

function ord(i:scalartype):integer;extern;
function pred(i:scalartype):scalartype;extern;
function succ(i:scalartype):scalartype;extern;
function chr(i:integer):char;extern;
function odd(i:integer):boolean;extern;
function abs(a:scalartype):scalartype;extern;
function sqr(a:scalartype):scalartype;extern;
function trunc(a:real):integer;extern;
function round(a:real):integer;extern;
function sqrt(a:real):real;extern;
function arctan(a:real):real;extern;
function cos(a:real):real;extern;
function sin(a:real):real;extern;
function exp(a:real):real;extern;
function ln(a:real):real;extern;

(***** TOOLKIT ROUTINES *****)

procedure draw_form;extern;
procedure echo_form;extern;
procedure process_form;extern;
procedure draw_list;extern;
procedure echo_list;extern;
procedure process_list;extern;
procedure draw_menu;extern;
procedure echo_menu;extern;
procedure process_menu;extern;
procedure draw_window_controller;extern;
procedure echo_window_controller;extern;
procedure process_window_controller;extern;
procedure draw_pick_with_locator;extern;
procedure echo_pick_with_locator;extern;
procedure process_pick_with_locator;extern;

```

```

procedure draw_potentiometer;extern;
procedure echo_potentiometer;extern;
procedure process_potentiometer;extern;
procedure draw_pick;extern;
procedure echo_pick;extern;
procedure process_pick;extern;
procedure draw_valuator_with_button;extern;
procedure echo_valuator_with_button;extern;
procedure process_valuator_with_button;extern;
procedure draw_locator_with_button;extern;
procedure echo_locator_with_button;extern;
procedure process_locator_with_button;extern;
procedure draw_button;extern;
procedure echo_button;extern;
procedure process_button;extern;
procedure draw_keyboard;extern;
procedure echo_keyboard;extern;
procedure process_keyboard;extern;

(***** HELP ROUTINES *****)
(***** module get_helps *****)
procedure get_help(htype:help_types;obj_code:integer);extern;
procedure get_tool_help(object:integer);extern;
procedure get_task_help(object:integer);extern;
procedure get_context_help(object:integer);extern;

(***** module dummy *****)
procedure dummy_routine(seg:string;pickid:integer;
                        obj:integer);extern;

(***** module pass_along_data *****)
procedure pass_along_string(instring:string;
                            var outstring:string);extern;

(***** module set_help_mode *****)
procedure set_help_mode(inst:string;var outst:string);extern;

(***** FOR WINDOW CONTROLLER TASK *****)
(***** module window_conversions *****)
procedure convert_windcont_to_task(wc_vport,
                                   wc_extremes:location_type;
                                   var wp>window_params);
                                   extern;
procedure pass_locs_to_windcont(wp>window_params;
                                var wc_vport,
                                wc_extremes:location_type);
                                extern;

```

```
(***** routines for user profiles *****)
(* note that these are not the correct
   headers for these routines. The actual
   headers must be generated by guide. These
   will prevent the user from
   getting errors when he creates a user
   profile while using guide. *)
procedure convert_to_profile_task;extern;
procedure convert_to_profile_tool;extern;
end.
```


APPENDIX B

The Layout System - An Example

This appendix shows a complete example of the use of GUIDE. The example chosen is the layout system from Foley and van Dam [Foley 82]. The first section contains the environment file which specifies the data structures, procedures and functions of the layout system. The second section traces through the creation of a design using GUIDE, and shows the result.

B.1. Environment File for the Layout System

This appendix contains the environment file provided to GUIDE for the layout system.

```
[inherit ('guidedir:commonenv.pen'),
 environment ('layoutenv.pex')]

module layout_environment;

type
  symbol_type = (window, door, desk, chair, partition, divider,
                 bookcase, floorplant);

  symbol_ptr = ^symbol;
  symbol = record
    symb_type:symbol_type;
    movable:boolean;
    symb_size:point;
  end;

  symbols = array[symbol_type] of symbol_ptr;
  (* represents symbol descriptions *)
  symbol_file = file of symbol;

  symbol_posn_ptr = ^symbol_posn;
  symbol_posn = record
    symb_name:string; (* for segment name *)
    symb:symbol_ptr;
    position:point;
    orientation:real;
    next_symbol_posn:symbol_posn_ptr
  end;
```

```

room = record
    title:string;
    room_size:point;
    contains:symbol_posn_ptr;
end;

user_type=(architect,designer);

var
    symbol_list:[external]symbols;
    cur_room:[external]room;
    next_unique_number:[external]integer;
    start_window_params:[external]window_params;

(* now external declaration for routines *)
(***** module add_symbol *****)
procedure add_symbol(var r:room;stype:symbol_type;pos:point;
    orient:real);extern;

(***** module delete_symbol *****)
procedure delete_symbol(var r:room;symname:string);extern;

(***** module draw_room *****)
procedure draw_room(r:room;room_window:location_type);extern;

(***** module draw_symbols *****)
procedure draw_window(sname:string;orientation:real);extern;
procedure draw_door(sname:string;orientation:real);extern;
procedure draw_desk(sname:string;orientation:real);extern;
procedure draw_chair(sname:string;orientation:real);extern;
procedure draw_partition(sname:string;
    orientation:real);extern;
procedure draw_divider(sname:string;orientation:real);extern;
procedure draw_bookcase(sname:string;orientation:real);extern;
procedure draw_floorplant(sname:string;
    orientation:real);extern;

(***** module draw_title *****)
procedure draw_title(r:room);extern;

(***** module equal_names *****)
function EqNms(s1,s2:string):boolean;extern;

(***** module initialize_layout *****)
procedure initialize_layout;extern;

(***** module make_unique_name *****)
procedure make_unique_name(var name:string);extern;

(***** module new_title *****)
procedure new_title(var r:room;titl:string);extern;

```


B.2. Using GUIDE to create a design

This section traces through the creation of a design for the layout system using GUIDE, and shows what the resulting interface looks like. It should be noted that many of the operations can be performed in different orders, that is, the organization shown below is one of many possibilities.

Prior to beginning with GUIDE, it is a good idea to determine exactly what is going into the design. In the case of the layout system, we want to implement a set of commands operating on a data structure representing a room. The data structures for the room and the routines which operate upon the structures are shown in Appendix B.1.

The commands which we want to implement are: add symbol, remove symbol, change title, change room size and change the window into the world. We note immediately that changing the window into the world can be implemented by the window controller tool. The other four commands will go into a menu. We also notice that removing a symbol has meaning only if the room contains any symbols. Therefore, we will include that command in the menu only if room contents are not empty.

Each of the commands in the menu will be associated with a routine which operates on the room. We must determine what the parameters for each routine are and decide what kinds of tools to use to determine their values.

The "add symbol" command will cause the "add_symbol" routine to be invoked. This routine takes four parameters: a room, a symbol_type, a point and a real number representing the orientation. The application designer has already determined that operations should take place on a "current room", called cur_room, so the user of the layout system does not need to specify a value for that parameter. For the symbol_type, a list of possible types seems to be an ideal representation. The point at which to position the new symbol should be picked by the user from the current room. A potentiometer seems an appropriate way to select an orientation. We also need the "exit task" to allow the user to indicate that all values are correct and the action should be invoked. Therefore, for the "add symbol" command, we will need a parameter context which contains:

- a task to select the symbol type (using a list tool);
- a task to choose the position (using a locator with button tool);
- a task to enter the orientation (using a potentiometer tool);
- a drawing of the current room;

- a drawing of the title of the room;
- a window controller to allow the user to move around in the room;
- the exit task.

The "remove symbol" command will cause the "delete_symbol" routine to be invoked. This routine takes two parameters: a room and the name of a symbol. Again, the room will be the "current room", so we need only to allow the user to specify the symbol to be removed. This should be done with a pick tool. To allow the user to pick a symbol from the room, we will need a drawing of the room and a window controller to allow the user to move around in case the symbol to be deleted is not in the current window into the room. For the "remove symbol" command, we need a parameter context containing:

- a task to pick a symbol (using a pick tool);
- a drawing of the current room;
- a drawing of the room title;
- a window controller;
- the exit task.

The "change title" command invokes the "update_room" routine. This routine takes three parameters: a room, a string for the new title and a point for the new room size. Again, we use the "current room" for the room parameter. Since the size of the room is not changing, we will specify the size field of the current room for the point parameter. (The ability to use application variables in expressions is necessary here.) The only task needed to get parameter values is one which reads in the new title. Clearly, a keyboard tool is called for. For uniformity with the rest of the system, we will also include the window controller, and drawings of the room and title. So, the parameter context for the "change title" command will contain:

- a task to read in the new title (using a keyboard tool);
- a window controller;
- a drawing of the room;
- a drawing of the title;
- the exit task.

The "change room size" will invoke the "update_room_size" routine. This routine requires a room and two reals representing the width and length. Keyboard tools seem appropriate for the width and length entry. As before, we want to include drawings of the room and title and a window controller. The parameter context for the "update_room_size" routine will contain:

- a task to read the new width (using a keyboard tool);

- a task to read the new length (using a keyboard tool);
- a drawing of the room;
- a drawing of the title;
- a window controller;
- the exit task.

The main context for the layout system will allow the user access to any of the five commands plus a view of the current state of the room. It must contain:

- a task to choose a command (using a menu);
- a window controller (for the change window command);
- a drawing of the current room;
- a drawing of the title of the room;
- the exit task (to leave the system).

Prior to entering the main context, the user should be able to initialize the room by providing a title and dimensions for the room. A form containing tasks for the title, width and length will do this. This form can contain the tasks described above for these functions. The only other thing needed in this form is a button tool to indicate completion of the form. The start context for the layout system contains:

- a task containing a form to initialize the system

At this point, we can summarize the tasks and user-defined picture classes needed for the layout system. Ten tasks are needed:

1. choose_command containing the menu;
2. get_title containing a keyboard to read the title;
3. get_width containing a keyboard to read the width;
4. get_length containing a keyboard to read the length;
5. start_layout containing a form for the title, width and length;
6. donetask containing a button to indicate completion of the start_layout form;
7. choose_symbol containing a list of symbols;
8. choose_location containing a locator with button;
9. choose_angle containing a potentiometer;
10. pick_symbol containing a pick tool.

In addition, two tasks built into GUIDE will be used, "exit" and "set_window_parameters", the latter containing the window controller.

Two user-defined picture classes must be defined:

1. room_class containing a user-defined picture for drawing the room;
2. title_class containing a user-defined picture for drawing the title.

The layout system will contain six contexts:

1. start_context - to read in initial data;
2. main_context - containing the main menu;
3. add_context - to get parameters for "add_symbol";
4. del_context - to get parameters for "delete_symbol";
5. chg_context - to get parameters for "update_room";
6. size_context - to get parameters for "update_room_size".

We are now ready to begin using GUIDE. When GUIDE is started, it presents a form (figure B-1) requesting the name of the environment file, both the compiled and Pascal versions, the name of the application and whether or not this is a new design. We choose each field in turn, entering the appropriate values and indicating that this is a new design. When we are satisfied with the values that have been entered, we choose the "done" field. GUIDE then reads in the environment, creating a symbol table and, since we have indicated a new design, reads in the tool and task kits.

We are now presented with the main GUIDE context (figure B-2). We choose to start by creating the various tasks needed, beginning with the "choose_command" task. Therefore, we choose the "create/edit task" item in the main menu. A parameter context requests the name of the task to be edited.

This brings us to a task creation context (figure B-3). In this context, we first fill in the form, for now entering only the type ("string") of the task and leaving the default task name which has been provided. Before we can do any more at the task level, we must create the appropriate tool, in this case a menu. Therefore, we choose the "create/edit tool" command.

After specifying the name of the tool ("layout_menu"), we enter a tool creation context (figure B-4). Again, the first order of business is to fill out the form provided. The name has again defaulted appropriately and, for now, we will omit helps and prompts, leaving only the tool type to be specified. We pick the "menu" item from the list of types, and choose "done" causing the form to be processed. When this is completed, the tool now contains an instance type. Therefore, the menu has been updated (figure B-5) to include the "create/edit instance" item. We select this item to allow us to create the menu itself.

EXIT		ENTER START-UP INFORMATION	
ENVIRONMENT FILE		FILE NAME	
PASCAL ENVIRONMENT FILE FILE NAME			
DESIGN NAME		ENTER IDENTIFIER	
START NEW DESIGN?	BOOLEAN		
	TRUE		
	FALSE		

Figure B-1: Start context for GUIDE

We now enter the menu creation context (figure B-6). We fill in the form describing the menu. Everything except the background color defaults to appropriate values. We fill in the color by selecting each component in turn and typing the appropriate real value.

EXIT

COMMANDS	DESIGN NAME	ENTER IDENTIFIER	LAYOUT
CREATE/EDIT TOOL	START ROUTINE ENTER NAME WITH PARAMS		
CREATE/EDIT TASK			
CREATE/EDIT CONTEXT			
MAKE CONTROL PATH	END ROUTINE ENTER NAME WITH PARAMS		
GENERATE CODE			
STORE DESIGN	START CONTEXT CONTEXT NAME		
READ DESIGN			
CREATE/EDIT USER PICTURE			
CREATE/EDIT USER PICTURE CLASS	DONE		
CREATE/EDIT USER PROFILE			

Figure B-2: Main context for GUIDE

Now we want to define the menu items and their relationships. We choose the "create/edit item" item from the menu. After entering the name of the item, "add symbol", we move to the context for defining menu items (figure B-7). In this context, we complete the form

EXIT		NAME: CHOOSE_COMMAND	
COMMANDS	TASK NAME ENTER STRING CHOOSE_COMMAND		
PREVIOUS MENU	CONDITION ENTER CONDITION		
	MESSAGE ENTER STRING		
CREATE/EDIT TOOL	HELP	FILE ENTER IDENTIFIER	
		DONE	
ADD TOOL	INITIAL VALUE ENTER EXPRESSION		
REMOVE TOOL			
CHANGE CONVEPSION	DEFAULT VALUE ENTER EXPRESSION		
CREATE/EDIT ACTION	TASK VALUE TYPE ENTER IDENTIFIER		
ADD ACTION TO TASK			
REMOVE ACTION FROM TASK	DONE		

Figure B-3: Task creation context for GUIDE

by entering the string to be displayed for this item ("add symbol") and entering the hue, saturation and value for the item color. By choosing "exit", we return to the menu creation context. We want to start a menu list to contain the items being created. To do so, we choose

EXIT	NAME: LAYOUT_MENU	
	TOOL CHARACTERISTICS NAME: FIXED NO PERMANENT 0 HIDDEN NO PROGRAM ROUTINE HAS NO PROCESS ROUTINE	
COMMANDS	TOOL NAME ENTER STRING LAYOUT_MENU	
	HELP	CONDITION ENTER CONDITION
		MESSAGE ENTER STRING
		FILE ENTER IDENTIFIER
PREVIOUS MENU	PROMPT	DONE
		CONDITION ENTER CONDITION
		MESSAGE ENTER STRING
	POSITION TOOL	TOOL TYPE
TOOL TYPES		
FORM		
LIST		
MENU		
CREATE/EDIT TOOL INSTANCE		-- MORE --
		DONE

Figure B-5: Tool creation context for GUIDE with instance command

We create the remaining items in the menu analogously, adding them to the list already begun as they are created. When we add the "remove symbol" item, we specify the condition "cur_room.contents<>nil" as its condition for inclusion. All of the other items have null conditions.

EXIT

*** MENU ***

DISPLAY TYPE IS VERTICAL
 PAGING IS NOT ALLOWED
 PAGE SIZE : 0
 PAGE MESSAGE : NONE
 OMITTED ITEMS NOT DISPLAYED
 HISTORY IS NOT MAINTAINED

		ORIENTATION
		HORIZONTAL
		VERTICAL
		RANDOM
CREATE/EDIT ITEM		BOOLEAN
	PAGING_PERMITTED	TRUE
		FALSE
DESTROY ITEM		OMISSION STYLE
		HALFTONE
ADD ITEM TO LIST	OMITTED_ITEMS_DISPLAY	LEAVE_SPACE
		OMIT
REMOVE ITEM FROM LIST		HUE ENTER REAL 0.000
	BACKGROUND_COLOR	SATURATION ENTER REAL 0.000
		VALUE ENTER REAL 0.000
ADD LIST TO ITEM		BOOLEAN
		TRUE
REMOVE LIST FROM ITEM	MAINTAIN_HISTORY	FALSE
SET/EDIT MENU START		
		DONE

Figure B-6: Menu creation context for GUIDE

When all four menu items have been created and added, we want to make the list of menu items be the follow list for each of the four items. To do so, we choose "add list to item" which takes us to a parameter context (figure B-9). We fill in the appropriate item name and

EXIT

ITEM NAME				ENTER STRING		ADD SYMBOL	
ITEM STRING ENTER STRING							
ITEM COLOR	HUE		ENTER REAL		0.000		
	SATURATION		ENTER REAL		0.000		
	VALUE		ENTER REAL		0.000		
DONE							

Figure B-7: Menu item context for GUIDE

pick any item on the list which we have created to indicate that that list is to be added. Picking "exit" executes the command and returns us to the menu creation context.

EXIT	MENU ITEM	ADD SYMBOL
ENTER CONDITION		
<div></div>		
START NEW MENU ITEM LIST		

Figure B-8: Parameter context for "add item to list"

We also must make this list the "start list" for the menu. We select "set/edit start item" from the menu. In the parameter context, we pick the list of items already created.

EXIT	MENU ITEM	CHANGE ROOM SIZE
------	-----------	------------------

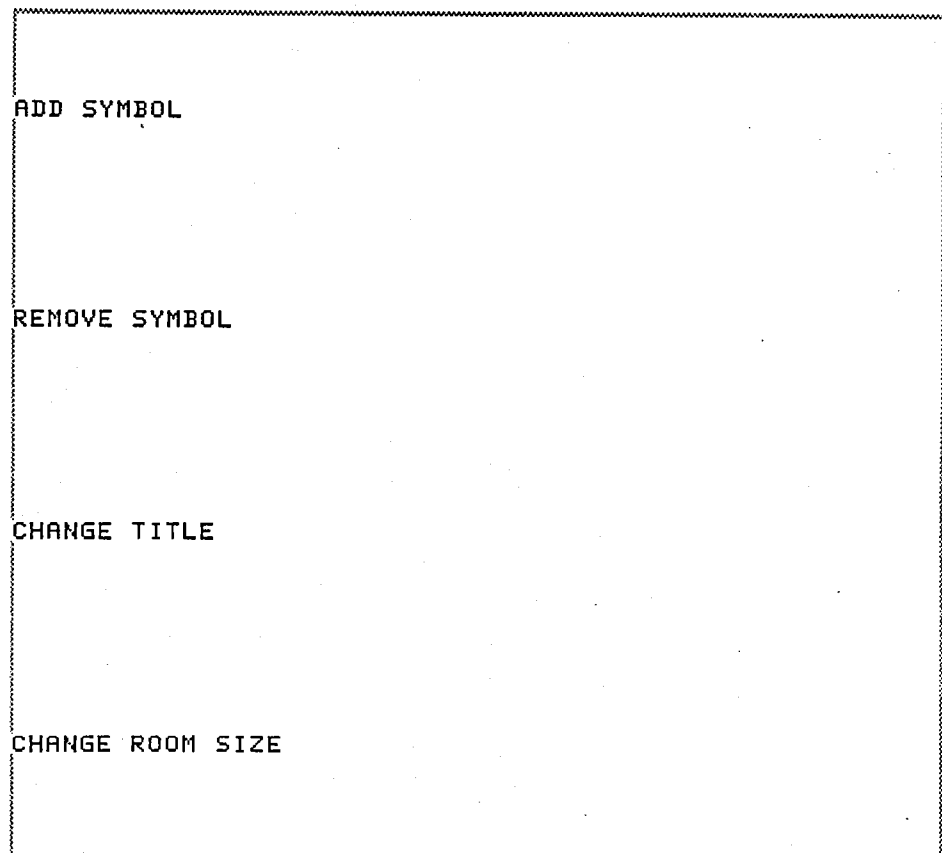


Figure B-9: Parameter context for "add list to item"

At this point, the menu is complete and we choose "exit" to return to the tool creation context. We have nothing more to do here at this point and choose "exit" again to return to task creation. At the task level, we want to add the menu tool just created to the "choose_command"

task under construction. We choose the "add tool" command, which takes us to a parameter context (figure B-10), where we must specify the tool, task and conversion routines. The tool and task have defaulted to the right values, so we choose each of the conversion routines in turn. For each, we enter "pass_along_string", since both the tool and task have string values. Choosing "exit" executes the action routine and returns us to the task creation context. Choosing "exit" again returns us to the main context.

The other tasks are created analogously, although the instance specific portion varies depending on the instance type. (It is actually much simpler in most cases.)

The next thing to do is to create the two user-defined picture classes needed. To do this, we choose the "create/edit user picture class" command. After entering the name of the class ("room_class"), we remain in the main context, but go down one level in the menu (figure B-11).

To create the picture itself, we choose "create/edit user picture". This takes us to a context for creating user-defined pictures (figure B-12). In this context, we complete the form. The first four fields default appropriately, so we need only enter the name and parameters for the drawing routine. We type in "draw_room(cur_room,set_window_params.window_used)". The part of the room to be drawn depends on the value of the window controller task.

Choosing "exit" returns us to class creation, where we choose "add user picture". Both the picture and the class default appropriately in the parameter context (figure B-13), so choosing "exit" is sufficient to invoke the action. We must choose "previous menu" from the menu to return to the top menu level.

The class "title_class" is created analogously and we are now ready to begin creating contexts. We choose "create/edit context" from the menu. Again, the name is requested and, after entering it ("main_context"), we find ourselves in a context creation context (figure B-14). The largest viewport contains a picture of the context contents, showing each tool or user-defined picture which has been positioned in this context.

We begin by adding tasks. We select "add task", and are prompted for the task and context. In this way, we add "choose_command" and "set_window_params". To add the exit task, we choose the command "add predef td to context". The parameter context provides a list of predefined tasks (figure B-15), from which we choose "exit". Since the exit tool has a default position in the upper left corner of the context, the context display, upon returning to the context creation context, shows that tool.

EXIT	TOOL NAME	LAYOUT_MENU
	TASK NAME	CHOOSE_COMMAND
	CONVERSION TO TASK ROUTINE	
	CONVERSION TO TOOL ROUTINE	

Figure B-10: Parameter context for "add tool"

At this point, we want to position the tools for "choose_command" and "set_window_params". We choose "create/edit tool", specifying "layout_menu" as the tool to be edited. In the tool creation context (figure B-4), we choose the "position tool" command. We

EXIT

COMMANDS	DESIGN NAME ENTER IDENTIFIER LAYOUT	
PREVIOUS MENU	START ROUTINE ENTER NAME WITH PARAMS	
CREATE/EDIT USER	END ROUTINE ENTER NAME WITH PARAMS	
ADD USER PICTURE	START CONTEXT CONTEXT NAME	
REMOVE USER PICT		DONE

Figure B-11: Main context for GUIDE with picture class commands

enter a parameter context (figure B-16). The tool and context have defaulted appropriately. We choose the prompt for the lower left corner, then pick a point in the context display. The position of the upper right corner is similarly picked. Choosing "exit" returns us to the tool

EXIT

COMMANDS	UPIC NAME ENTER STRING ROOM_PIC	
	FIXED	BOOLEAN
		TRUE
		FALSE
	PERMANENT	BOOLEAN
		TRUE
		FALSE
PREVIOUS MENU	BORDER TYPE INTEGER 1	
POSITION USER PICTURE	DRAW ROUTINE ENTER ROUTINE NAME AND PARAMS	
	DONE	

Figure B-12: User-defined picture creation context for GUIDE

context. We again pick "position tool". This time, we modify the tool name to "window controller tool" and position it in the upper right corner of the context. After "exit" returns us to the tool context, we choose it again to return to the context creation context.

EXIT	PICTURE NAME	ROOM_PIC
	PICTURE CLASS NAME	ROOM_CLASS

Figure B-13: Parameter context for "add user picture"

The user-defined picture classes, `room_class` and `title_class`, are added to the context by choosing "add user picture class" and specifying the appropriate class name for each. (The first time, we can accept the default.) Positioning of the pictures is done by choosing "create/edit

EXIT

COMMANDS	
PREVIOUS MENU	
CREATE/EDIT TOOL	
CREATE/EDIT TASK	
CREATE/EDIT USER PICTURE	
CREATE/EDIT USER PICTURE CLASS	
ADD TASK	
REMOVE TASK	
ADD USER PICTURE CLASS	
REMOVE USER PICTURE CLASS	
ADD PREDEF TD TO CONTEXT	
NAME OF CONTEXT: MAIN_CONTEXT_____	

Figure B-14: Context creation context for GUIDE

user picture" and then choosing "position user picture". Within the positioning context, we operate analogously to positioning of tools.

EXIT	EXIT TASK
	HELP TASK (TOOL)
	HELP TASK (TASK)
	HELP TASK (CONTEXT)

CONTEXT NAME

MAIN_CONTEXT

NAME OF CONTEXT: MAIN_CONTEXT_____

Figure B-15: Parameter context for "add predefined to context"

When we return to the context creation context, we find that this context is now satisfactory. Choosing "exit" returns us to the main context, where we can create the other contexts in the same way.

EXIT	TOOL NAME	LAYOUT_MENU
CONTEXT NAME		MAIN_CONTEXT
UPPER RIGHT	PICK A POINT IN VIEWPORT CONTEXT_CONTENTS	
LOWER LEFT	PICK A POINT IN VIEWPORT CONTEXT_CONTENTS	
TOOL NAME: EXIT TOOL		
NAME OF CONTEXT: MAIN_CONTEXT		

Figure B-16: Parameter context for "position tool"

After all of the contexts have been created, three tasks remain. We must specify a control path from "start_context" to "main_context"; we must specify the action routines to be called from the task "choose_command"; and we also must fill in the form in the main context regarding the design itself.

EXIT TASK NAME CHOOSE_COMMAND
 ENTER CONDITION

START CONTEXT		CONTEXT NAME	MAIN_CONTEXT
END CONTEXT		CONTEXT NAME	MAIN_CONTEXT
STACK IT	BOOLEAN		
	TRUE		
	FALSE		
PUSH/POP	STACK OPS		
	PUSH		
	POP		
RESTORE LIST ENTER ITEMS TO BE RESTORED			
DONE			

Figure B-17: Parameter context for "make control path"

First, we choose to create the control path. We choose "make control path" from the menu and proceed to a parameter context (figure B-17). There, we specify that this decision is to be associated with the "start_layout" task unconditionally. We fill in the form, specify

EXIT		NAME: CHOOSE_COMMAND	
<p>*** MENU ***</p> <p>DISPLAY TYPE IS VERTICAL</p> <p>PAGING IS NOT ALLOWED</p> <p>PAGE SIZE : NONE 0</p> <p>OMITTED ITEMS NOT DISPLAYED</p> <p>HISTORY IS NOT MAINTAINED</p>			
COMMANDS	TASK NAME	ENTER STRING	CHOOSE_COMMAND
		CONDITION	ENTER CONDITION
PREVIOUS MENU	HELP	MESSAGE	ENTER STRING
		FILE	ENTER IDENTIFIER
			DONE
ENTER ACTUAL PARAM LIST	INITIAL VALUE	ENTER EXPRESSION	
REMOVE PARAM LIST FROM ACTION		ENTER EXPRESSION	
DEFINE PARAM COLLECTION	TASK VALUE TYPE	ENTER IDENTIFIER	
CREATE/EDIT CONTEXT		DONE	

Figure B-18: Task creation context with action menu

"start_context" for the start context and "main_context" for the end context. We specify no stacking, then pick "done" to complete the form. Next, we pick "exit" to return to the main context.

EXIT	CONTEXT NAME	START_CONTEXT
START NEW PARAM CONTEXT LIST		

Figure B-19: Parameter context for "Add context to list"

Now, we wish to define the action calls from "choose_command". We select "create/edit task" and specify "choose_command". This leads us to the task creation context (figure B-3), where we select the "create/edit action" item. This item does not change the context, but modifies the menu list (figure B-18).

EXIT

ACTION ROUTINE

GET_CONTEXT_HELP

ENTER CONDITION

CONTEXT NAME : ADD_CONTEXT

Figure B-20: Parameter context for "Add param context to action"

From the menu, we now pick "define param collection". This again modifies the menu without changing context. We now choose "add context to list", moving to a parameter context (figure B-19), where we specify the context "add_context" and choose to start a new param

EXIT	ACTION ROUTINE	ADD_SYMBOL
ENTER STRINGS (COMMAS BETWEEN)		
ENTER CONDITION		
START NEW PARAM DATA RECORD		
CONTAINED IN ACTION : NO ACTION EXISTS		
CONTEXTS:	EXPRESSIONS:	
<div> <div>CONTEXT NAME :</div> <div>ADD_CONTEXT</div> </div>		

Figure B-21: Parameter context for "Enter actual param list"

context list. Returning to the task context via "exit", we choose "add param context to action". A parameter context (figure B-20) allows us to specify the routine, and to pick the context to add. Choosing "exit" moves us to a second parameter context where we indicate starting a new

EXIT	ACTION ROUTINE	ADD_SYMBOL
TASK NAME	CHOOSE_COMMAND	
ENTER CONDITION		

Figure B-22: First parameter context for "Add action to task"

"param_data" record. Choosing "exit" again moves to a third parameter context, in which we can specify the position of the context on the list in the param_data record. In this case, since we are starting a new one, the list is empty and we simply choose "exit" to execute the command.

EXIT	PASCAL OUTPUT FILE NAME	GLY: LAYOUTINT.PAS
	OUTPUT ENVIRONMENT FILE NAME	GLY: LAYOUTINTENV.PAS
	OUTPUT DATA FILE NAME	GLY: LAYOUT.DAT
	OUTPUT HELP FILE NAME	GLY: LAYOUT.HLP

Figure B-23: Parameter context for "Generate code"

Next, we move up a level in the menu, returning to figure B-18. We choose "Enter actual param list", to enter the parameters for this action. In the parameter context (figure B-21), the routine name defaults to "add_symbol". For the string list, we type

ROOM TITLE ENTER TITLE	
ROOM WIDTH ENTER WIDTH 20.000	
ROOM LENGTH ENTER LENGTH 20.000	
	DONE

Figure B-24: Start context for layout system

"cur_room,choose_symbol,choose_location,choose_angle". There is no condition needed and we choose the param_data record which was just created. Choosing "exit" executes the command. We choose "previous menu" to go up a level in the menu, returning us to figure B-3.

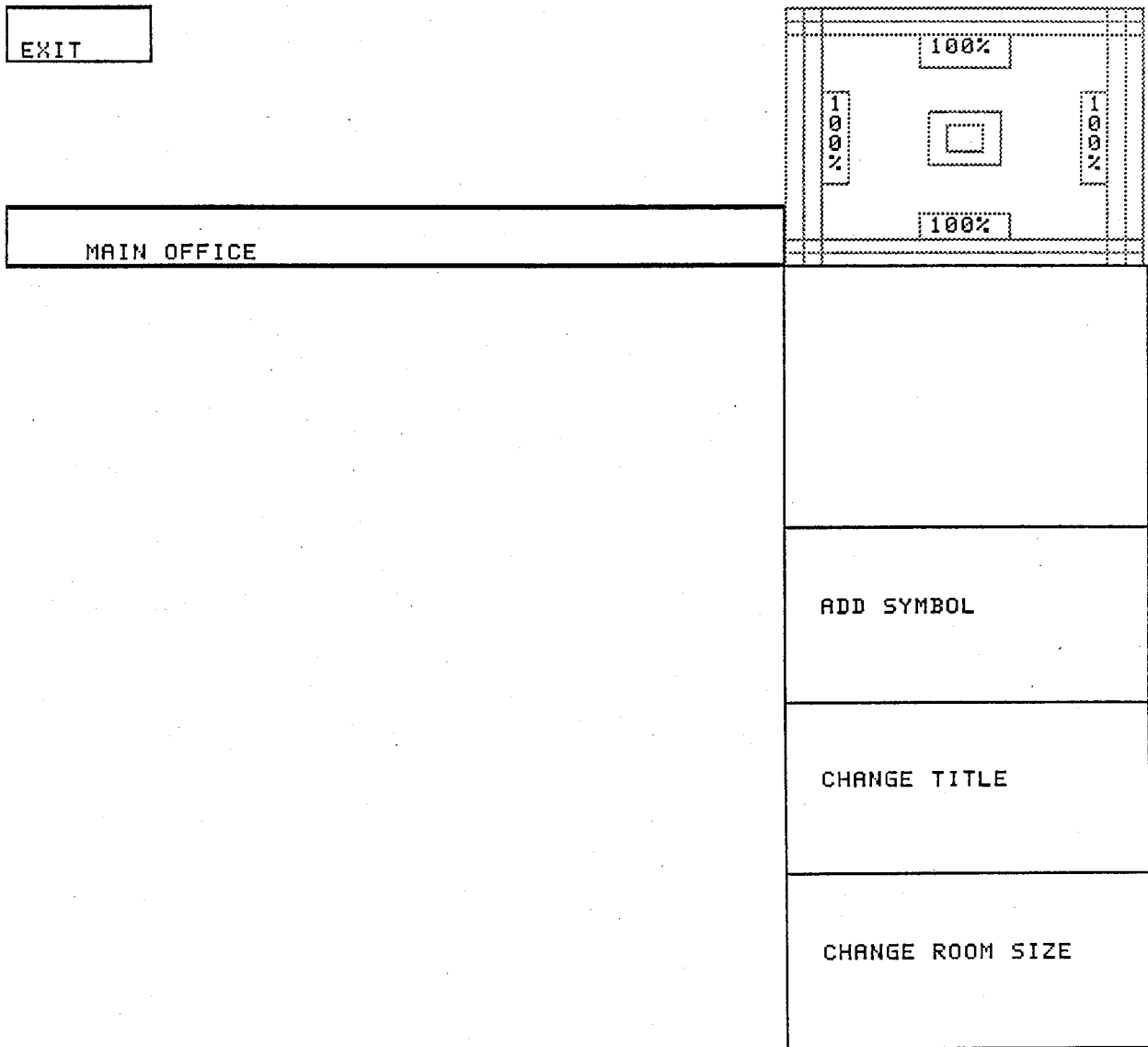


Figure B-25: Main context for layout system

After the "add_symbol" action has been defined, we must add it to the task. We choose "add action to task", and are presented with a parameter context (figure B-22) where we must specify the task, action and a condition. The task and action have defaulted appropriately. For

<div>EXIT</div>		<div>ORIENTATION</div> <div>0.00 359.00</div>			
MAIN OFFICE		PICK A POINT IN VIEWPORT ROOM_PIC			
		SYMBOLS			
		WINDOW			
		DOOR			
		DESK			
		CHAIR			
		PARTITION			
		DIVIDER			
		BOOKCASE			
		FLOORPLANT			

Figure B-26: Parameter context for "add symbol"

the condition, we enter "eqnms(choose_command,'add symbol')", indicating that this action is to be invoked only if the task "choose_command" has the value "add_symbol". When we choose "exit", we are presented with a second parameter context in which we must choose which

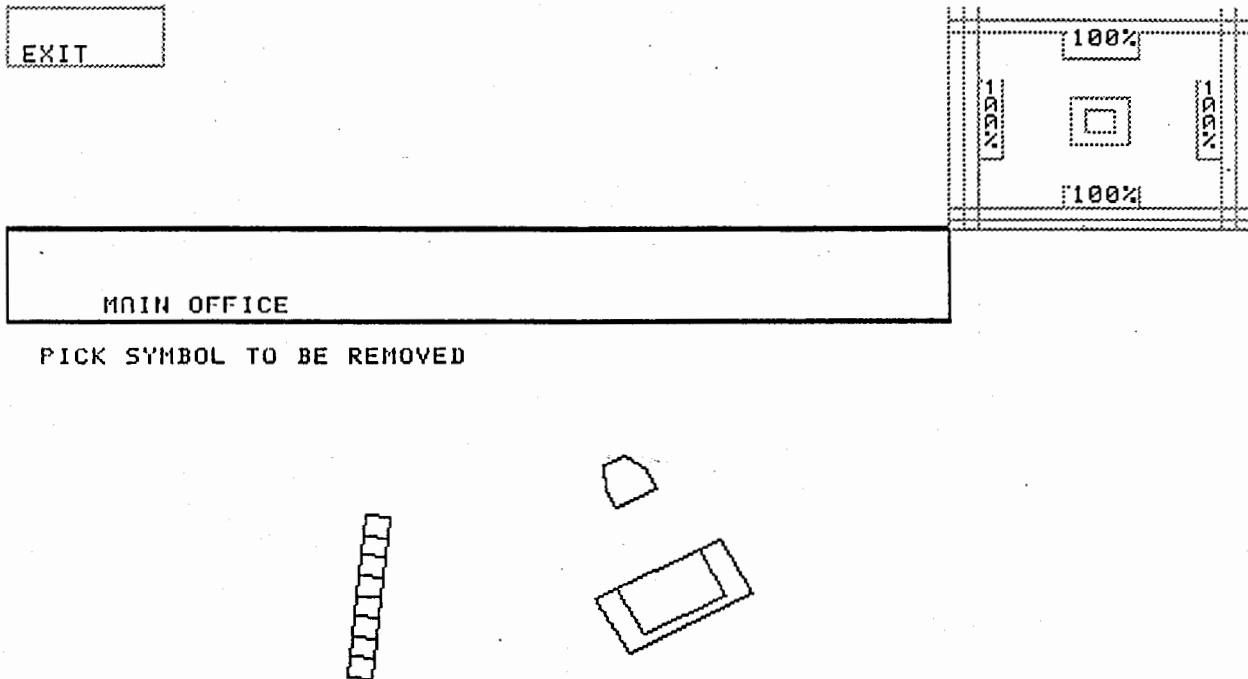


Figure B-27: Parameter context for "delete symbol"

parameter collection and computation methods will be used, and where to place this action on the list of actions for this task. We pick the parameter information which we have created, which is the only set displayed. If we pick no action position, the action is automatically placed at the end of the list for this task. This is suitable, so we select "exit".

EXIT

ENTER TITLE

MAIN OFFICE

MAIN OFFICE

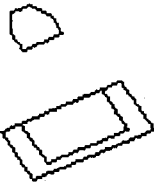
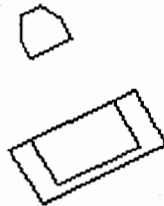


Figure B-28: Parameter context for "change title"

The remaining actions are created and added in the same way, and we return to the main context. Now the only remaining step is to fill out the design form. The name has defaulted to "layout", which we initially specified. For the start routine, we enter "initialize_layout". There is

EXIT	ENTER WIDTH	20.000	
	ENTER LENGTH	20.000	
MAIN OFFICE			



gals = 20.0 parameter context for change size

no end routine, so we leave that field blank. For the start context, we specify "start_context". We then choose "done", allowing the form to be processed.

At this point, the design is complete. We want to store it for later editing and then to generate an interface. We first choose "store design". A parameter context requests a file name. We enter "layout.int" and choose "exit". The design is stored in the file "layout.int".

To generate the interface, we choose the item "generate code". We proceed to a parameter context (figure B-23), where four filenames are requested. We enter each of these and choose "exit", at which time code is generated.

Upon returning to the main menu, we choose "exit" to leave GUIDE.

We now use the Pascal compiler and the linker to create the application system. When this process is complete, we can run the layout system. The six contexts of the layout system are shown in Figures B-24 through B-29.

APPENDIX C

Environment BNF

This appendix shows the BNF which is used to parse the environment file for the application. Angle brackets ('<' and '>') are used to enclose non-terminals. Exclamation points indicate 'or'. A period signals the end of productions for a non-terminal.

```

<PROG> -> <module> .
<module> -> MODULE IDEN ( <ID-LIST> ) ; <BLOCK> END ' .
          ! MODULE IDEN ; <BLOCK> END ' .
          ! [ <env-part> ] MODULE IDEN ( <ID-LIST> ) ;
            <BLOCK> END ' .
          ! [ <env-part> ] MODULE IDEN ; <BLOCK> END ' . .
<env-part> -> <environ> , <inherit>
          ! <inherit> , <environ>
          ! <environ>
          ! <inherit> .
<environ> -> ENVIRONMENT ( <FILE-LIST> ) .
<inherit> -> INHERIT ( <FILE-LIST> ) .
<FILE-LIST> -> <FILE-LIST> , STRING
          ! STRING .
<ID-LIST> -> <ID-LIST> , IDEN
          ! IDEN .
<BLOCK> -> <CONS-PAR> <TYPE-PAR> <VAR-PAR> <PROC-PAR>
          ! <TYPE-PAR> <VAR-PAR> <PROC-PAR>
          ! <CONS-PAR> <VAR-PAR> <PROC-PAR>
          ! <CONS-PAR> <TYPE-PAR> <VAR-PAR>
          ! <CONS-PAR> <TYPE-PAR> <PROC-PAR>
          ! <VAR-PAR> <PROC-PAR>
          ! <TYPE-PAR> <PROC-PAR>
          ! <CONS-PAR> <PROC-PAR>
          ! <TYPE-PAR> <VAR-PAR>
          ! <CONS-PAR> <VAR-PAR>
          ! <CONS-PAR> <TYPE-PAR>
          ! <CONS-PAR>
          ! <TYPE-PAR>
          ! <VAR-PAR>
          ! <PROC-PAR> .

```

```

<CONS-PAR> -> CONST <CON-DCLL> ; .
<CON-DCLL> -> <CON-DCLL> ; <CON-DCL>
! <CON-DCL> .
<CON-DCL> -> IDEN = <CONSTANT> .
<CONSTANT> -> NUMBER
! ADDOP NUMBER
! IDEN
! ADDOP IDEN
! STRING .
<TYPE-PAR> -> TYPE <TYP-DEFL> ; .
<TYP-DEFL> -> <TYP-DEFL> ; <TYPE-DEF>
! <TYPE-DEF> .
<TYPE-DEF> -> IDEN = <TYPE> .
<TYPE> -> <SIMPL-TY>
! <STRUC-TY>
! <POINT-TY> .
<SIMPL-TY> -> <SCALAR>
! <SUBRANGE>
! IDEN .
<SCALAR> -> ( <ID-LIST> ) .
<SUBRANGE> -> <CONSTANT> '... <CONSTANT> .
<STRUC-TY> -> <UNPAC-TY>
! PACKED <UNPAC-TY> .
<UNPAC-TY> -> <ARRAY-TY>
! <RECORD-T>
! <SET-TYP>
! <FILE-TYP> .
<ARRAY-TY> -> ARRAY [ <INDEX-TL> ] OF <TYPE>
! VARYING [ NUMBER ] OF <TYPE> .
<INDEX-TL> -> <INDEX-TL> , <SIMPL-TY>
! <SIMPL-TY> .
<RECORD-T> -> RECORD <FIELD-L> END
! RECORD <REC-SECL> END .
<FIELD-L> -> <REC-SECL> ; <VARIANTP>
! <VARIANTP> .
<REC-SECL> -> <REC-SECL> ; <REC-SEC>
! <REC-SEC>
! <REC-SECL> ;
! ;
! ; <REC-SEC> .
<REC-SEC> -> <ID-LIST> : <TYPE> .
<VARIANTP> -> CASE IDEN : IDEN OF <VARIANTL>
! CASE IDEN OF <VARIANTL> .
<VARIANTL> -> <VARIANTL> ; <VARIANT>
! <VARIANT>
! <VARIANTL> ;
! ; <VARIANT>
! ; .

```



```

<VARIANT> -> <CASE-LAB> : ( <FIELD-L> )
! <CASE-LAB> : ( <REC-SECL> )
! <CASE-LAB> : ( ) .
<CASE-LAB> -> <CASE-LAB> , <CONSTANT>
! <CASE-LAB> , <SUBRANGE>
! <CONSTANT>
! <SUBRANGE> .
<SET-TYP> -> SET OF <SIMPL-TY> .
<FILE-TYP> -> FILE OF <TYPE> .
<POINT-TY> -> ^ IDEN .
<VAR-PAR> -> VAR <VAR-DCLL> ;
! <VAR-PAR> VAR <VAR-DCLL> ; .
<VAR-DCLL> -> <VAR-DCLL> ' ; <VAR-DCL>
! <VAR-DCL> .
<VAR-DCL> -> <ID-LIST> : <TYPE>
! <ID-LIST> : [ EXTERNAL ] <TYPE>
! <ID-LIST> : [ GLOBAL ] <TYPE> .
<PROC-PAR> -> <PROC-PAR> <PRO-FUN> ;
! <PRO-FUN> ;
. <PRO-FUN> -> <PROC-HD> EXTERN
! <FUNC-HD> EXTERN
. <PROC-HD> -> PROCEDURE IDEN ;
! PROCEDURE IDEN ( <FOR-PARL> ) ;
. <FOR-PARL> -> <FOR-PARL> ; <FOR-PAR>
! <FOR-PAR>
. <FOR-PAR> -> <ID-LIST> : <TYPE>
! VAR <ID-LIST> : <TYPE>
! FUNCTION <ID-LIST> : <TYPE>
! PROCEDURE <ID-LIST>
. <FUNC-HD> -> FUNCTION IDEN : <TYPE> ;
! FUNCTION IDEN ( <FOR-PARL> ) : <TYPE> ; .
$

```

References

- [ANSI 85] *Draft American National Standard for the Functional Specification of the Computer Graphics Virtual Device Interface (CG-VDI)*
American National Standards Institute, 1985.
- [Anson 82] Anson, Ed.
The Device Model of Interaction.
Computer Graphics 16(3):107-114, July, 1982.
- [Apple 83] Apple Corp.
Lisa.
Siggraph Video Review (8), October, 1983.
- [Badler 84] Badler, Norman I. and Granor, Tamar E.
The Window Controller.
Transactions on Graphics 3(4):48-51, October, 1984.
- [Barnard 82] Barnard, P., Hammond, N., MacLean, A. and J. Morton.
Learning and Remembering Interactive Commands.
In *Human Factors in Computer Systems*, pages 2-7. Institute for Computer Sciences and Technology - National Bureau of Standards, U.S. Department of Commerce and Washington, D.C. ACM Chapter, Gaithersburg, MD, March, 1982.
- [Bass 85] Bass, Leonard J.
A Generalized User Interface for Applications Programs (II).
Communications of the ACM 28(6):617-627, June, 1985.
- [Black 82] Black, J. and T. Moran.
Learning and Remembering Command Names.
In *Human Factors in Computer Systems*, pages 8-11. Institute for Computer Sciences and Technology - National Bureau of Standards, U.S. Department of Commerce and Washington, D.C. ACM Chapter, Gaithersburg, MD, March, 1982.
- [Bloom 83] Bloom, Douglas A.
A User-Oriented Interface Control (of an Interactive Computer Graphics System).
Master's thesis, University of Pennsylvania, May, 1983.
- [Buxton 83a] Buxton, W., Lamb, M. R., Sherman, D. and K. C. Smith.
Towards a Comprehensive User Interface Management System.
Computer Graphics 17(3):35-42, July, 1983.
- [Buxton 83b] Buxton, William.
Lexical and Pragmatic Considerations of Input Structures.
Computer Graphics 17(1):31-37, January, 1983.

- [Card 82] Card, Stuart K.
User Perceptual Mechanisms in the Search of Computer Command Menus.
In *Human Factors in Computer Systems*, pages 190-196. Institute for
Computer Sciences and Technology - National Bureau of Standards,
U.S. Department of Commerce and Washington, D.C. ACM Chapter,
Gaithersburg, MD, March, 1982.
- [Feldman 82] Feldman, M. and Rogers, G.
Toward the Design and Development of Style-Independent Interactive
Systems.
In *Human Factors in Computer Systems*, pages 111-116. Institute for
Computer Sciences and Technology - National Bureau of Standards,
U.S. Department of Commerce and Washington, D.C. ACM Chapter,
Gaithersburg, MD, March, 1982.
- [Foley 82] Foley, J.D. and van Dam, A.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley, 1982.
- [Gallier 85] Gallier, Jean H., Manion, Frank J. and McEnerney, John.
CISV3: A compiler generator based on attribute evaluation.
Technical Report MS-CIS-85-59, University of Pennsylvania, Department of
Computer and Information Science, 1985.
- [Green 82] Green, Mark.
Towards a User Interface Prototyping System.
In *Graphics Interface '82*, pages 37-45. National Computer Graphics
Association of Canada and Canadian Man-Computer Communications
Society, Toronto, Ontario, May, 1982.
- [Green 83] Green, Mark.
A Catalogue of Graphical Interaction Techniques.
Computer Graphics 17(1):46-52, January, 1983.
- [Green 85a] Green, Mark.
The University of Alberta User Interface Management System.
Computer Graphics 19(3):205-213, July, 1985.
- [Green 85b] Green, Mark.
*The University of Alberta User Interface Management System Design
Principles*.
Human Computer Interaction Project 1, Department of Computer Science,
University of Alberta, July, 1985.
- [GSPC 79] Graphics Standards Committee.
Status Report of the Graphics Standards Committee.
Computer Graphics 13(3), 1979.
- [Hanau 80] Hanau, P. R. and Lenorovitz, D. R.
Prototyping and Simulation Tools for User/Computer Dialogue Design.
Computer Graphics 14(3):271-278, July, 1980.

- [Hartson 84] Hartson, H. R., Johnson, D. H. and Ehrich, R. W.
A Human-Computer Dialogue Management System.
In *Proceedings of Interact 84*, pages 57-61. IFIPS, Imperial College,
London, September, 1984.
- [Hayes 85] Hayes, Philip J., Szekely, Pedro A. and Lerner, Richard A.
Design Alternatives for User Interface Management Systems Based on
Experience with COUSIN.
In *CHI '85 - Human Factors in Computer Systems*, pages 177-183.
ACM/SIGCHI, San Francisco, CA, april, 1985.
- [Heffler 82] Heffler, Michael J.
Description of a Menu Creation and Interpretation System.
Software - Practice and Experience 12:269-281, 1982.
- [Herot 84] Herot, Christopher F.
Graphical User Interfaces.
In Yannis Vassiliou (editor), *Human Factors and Interactive Computer
Systems*, chapter 4, pages 83-103. Ablex, 1984.
- [Hopgood 83] Hopgood, F.R.A., Duce, D.A., Gallop, J.R., and Sutcliffe, D.C.
Introduction to the Graphical Kernel System (GKS).
Academic Press, 1983.
- [Johnson 75] Johnson, S. C.
YACC: Yet another compiler-compiler.
Computer Science Technical Report 32, Bell Labs, 1975.
- [Jones 82] Jones, William B.
Programming Concepts.
Prentice-Hall, 1982.
- [Kamran 83] Kamran, Abid and Feldman, Michael B.
Graphics Programming Independent of Interaction Techniques and Styles.
Computer Graphics 17(1):58-66, January, 1983.
- [Kasik 82] Kasik, David J.
A User Interface Management System.
Computer Graphics 16(3):99-106, July, 1982.
- [Kasik 84] Kasik, David J. and Olsen, Jr, Dan R.
A Taxonomy of User Interface Management.
1984.
- [Knuth 73] Knuth, Donald E.
The Art of Computer Programming, Volume 1 - Fundamental Algorithms.
Addison-Wesley, Reading, MA, 1973.
- [Olsen 83a] Olsen, Dan R. Jr.
Automatic Generation of Interactive Systems.
Computer Graphics 17(1):53-57, January, 1983.
- [Olsen 83b] Olsen, Dan R. Jr. and Dempsey, Elizabeth P.
SYNGRAPH: A Graphical User Interface Generator.
Computer Graphics 17(3):43-50, July, 1983.

- [Roach 82] Roach, J., Hartson, H. R., Ehrich, R., Yuntten, T. and D. Johnson.
DMS: A Comprehensive System for Managing Human-Computer Dialogue.
In *Human Factors in Computer Systems*, pages 102-105. Institute for
Computer Sciences and Technology - National Bureau of Standards,
U.S. Department of Commerce and Washington, D.C. ACM Chapter,
Gaithersburg, MD, March, 1982.
- [Rubel 82] Rubel, Andrew.
Graphic Based Applications - Tools to Fill the Software Gap.
Digital Design, July, 1982.
- [Savage 82] Savage, Ricky E., Habinek, James K., and Thomas W. Barhart.
The Design, Simulation and Evaluation of a Menu-Driven User Interface.
In *Human Factors in Computer Systems*, pages 36-40. Institute for
Computer Sciences and Technology - National Bureau of Standards,
U.S. Department of Commerce and Washington, D.C. ACM Chapter,
Gaithersburg, MD, March, 1982.
- [Schimpf 81] Schimpf, Karl M.
Construction Methods of LR-parser.
Master's thesis, University of Pennsylvani, May, 1981.
- [Schulert 85] Schulert, Andrew J., Rogers, George T. and Hamilton, James A.
ADM - A Dialog Manager.
In *CHI '85 - Human Factors in Computer Systems*, pages 177-183.
ACM/SIGCHI, San Francisco, CA, april, 1985.
- [Seeheim 84] Pfaff, G. and ten Hagen, P. J. W. (editor).
*EUROGRAPHICS-Springer: Seeheim Workshop on User Interface
Management Systems*.
Springer-Verlag, Berlin, 1984.
- [Singh 84] Singh, Baldev, Beatty, John C., Booth, Kellogg S. and Ryman, Rhonda.
A Graphics Editor for Benesh Movement Notation.
Computer Graphics 18(3), July, 1984.
- [Stluka 82] Stluka, F. P., Saunders, B. F., Slayton, P. M. and Badler, N. I.
Overview of the University of Pennsylvania CORE System.
Computer Graphics 16(2):177-186, June, 1982.
- [Symbolics 85] *User's Guide to Symbolics Computers*
Symbolics Corp., Cambridge, MA, 1985.
- [Tanner 84] Tanner, Peter P. and Buxton, William A.S.
Some Issues in Future User Interface Management System Development.
In *EUROGRAPHICS-Springer: Seeheim Workshop on User Interface
Management Systems*. Springer-Verlag, Berlin, 1984.
- [Tesler 81] Tesler, Larry.
The Smalltalk Environment.
Byte 6(8):90-147, August, 1981.
- [Thomas 83] Thomas, James J.
Graphical Input Interaction Techniques Workshop Summary.
Computer Graphics 17(1):5-30, January, 1983.

- [University of North Carolina 81] University of North Carolina Computer Science Department.
The Grip-75 System.
Siggraph Video Review (4), August, 1981.
- [Van den Bos 78] Van den Bos, J.
Definition and Use of Higher-Level Graphics Input Tools.
Computer Graphics 12(3):38-42, August, 1978.
- [VAX-11 82] *VAX-11 Reference Manual*
Digital Equipment Corp., Maynard, MA, 1982.
- [VAX-11 Pascal 82] *VAX-11 Pascal Language Reference Manual*
Digital Equipment Corp., Maynard, MA, 1982.
- [Wong 82] Wong, Peter C. S. and Reid, Eric R.
Flair - User Interface Dialog Design Tool.
Computer Graphics 16(3):87-98, July, 1982.