HIGH-LEVEL SPECIFICATION OF GRAPHICAL USER INTERFACES

MUTHURAMAN MUTHU

High-level Specification of Graphical User Interfaces

by

Muthuraman Muthu

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Memorial University of Newfoundland

September 1997

St. John's Canada

Abstract

Recent studies have shown that users of GUIs make fewer mistakes, feel less frustrated, suffer less fatigue and are more able to learn for themselves about the operation of new packages than users of non-graphical or character-based user interfaces. On the other hand, other surveys on user interface programming show that developing a GUI is a very complex task, that in today's applications a considerable amount of resources (time and effort) are devoted to the user interface portion both in the development phase and in the maintenance phase.

This report discusses currently available toolkits and specification methods that facilitate the development of a GUI in an efficient way and compares their merits and demerits. A new solution is then proposed by developing a high-level specification language for interfaces. The proposed approach is implemented using Java/JavaCompilerCompiler (JavaCC). A simple application is also presented.

Key Words: Graphical User Interfaces, Toolkits, Interface Specification Languages, User Interface Design.

Acknowledgment

First I would like to thank my supervisor Dr.Wlodek Zuberek for his thoughtful and patient guidance which he provided from the project's inception to its completion. I would like to thank Michael Rayment for his time and patience in answering a myriad of weird questions at all times. Also I would like to thank Michael Rendell for his useful suggestions on the implementation part, Nolan White for fixing up the system related problems in no time and Elaine Boone for reviewing the early drafts.

I would also like to thank my family and friends and others who have contributed to my research and thesis. My special thanks go to my friends, Janny Rodriguez and L.Srikanth who introduced me to the art of programming when I was working in Wipro Infotech Ltd., Bangalore.

Finally, my sincere thanks to the developers of freely available tools such as Java, JavaCC, LaTeX and other related tools which contributed significantly to the successful completion of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A user interface is the means by which the user communicates with an application and an application with the user. This interface is often the most important feature on which the success of the system depends. An interface which is difficult to use will, at best, result in a high level of user errors; at worst, it will cause the software system to be discarded, irrespective of its functionality.

It should be noted that developers of application software such as personal systems, stock control or order entry systems, typically dedicate a considerable amount of the program code to the implementation of the graphical user interface. Thus the time and cost incurred in the development of the user interface can be very significant [1].

1.1 Graphical user interfaces

Graphical user interfaces (GUIs) have brought quantifiable benefits to users and organizations that rely on software products. Recent studies [2] [3] have shown that users of GUIs make fewer mistakes, feel less frustrated, suffer less fatigue and are more able to learn for themselves about the operation of new packages than users of non-graphical or character-based user interfaces.

From a software designer's point of view, however, GUIs are more difficult to design than character-based interfaces [4]. The user's interaction with the GUI is more complex because it is based on principles of direct manipulation[1] [5] and concurrent user's access to multiple windows, icons, menus and input devices. A character-based interface normally only allows the user sequential access: first view a menu, then make a selection, then view the next screen, then enter the data. With the character-based interfaces, the user interface can be designed in such a way that the user will undertake a task in a predefined sequence. In the case of the GUI, many actions are allowed on interface objects and the user will decide which actions to take and in what order.

On the other hand, GUIs must be designed with care in order to avoid the problems caused by poor GUI design, which include reduced user productivity, unacceptable learning times and unacceptable error levels; all these factors leading to frustration and again potential rejection of the system by the user.

[1] The ability to see and point to menus and icons rather than to remember and type written commands.

1.2 User interface tools

Graphical user interfaces by and large bring considerable amount of benefits to the people and organization using it. But the price to be paid to achieve those benefits is pretty high because of the amount of effort they demand. This is where the need for graphical user interface tools comes in to picture. *Graphical User Interface Tools* are tools that provide programming support for implementing interactive systems [6]. The advantages of such tools can be classified into two main groups:

I. The quality of interface is improved. This is because:

- Design can be rapidly prototyped and implemented, possibly even before the application code is written.
- If any bugs are discovered during testing phase of the application code, they can be corrected easily using the tools.
- There can be multiple user interfaces for the same application.
- Different applications are more likely to have consistent user interfaces if they are created using the same user interface tool.

II. The user interface code is easier and more economical to create and maintain when compared to developing it without any GUI development tools. This is because:

- Interface specification can be represented, validated and evaluated more easily and more thoroughly.
- There is less code to write because much is generated by tools.
- There is better modularization due to the separation of the user interface component from the application. This should allow the user interface to

change without affecting the application, and a large class of changes to the application is possible without affecting the user interface.

- The level of programming expertise of the interface designer and implementors can be lower, because the tools hide much of the complexities of the underlying system.
- The reliability of the user interface will be higher, since the code for the user interface is created automatically from a higher level specification.
- It will be easier to port an application to different hardware and software environments since the device dependencies are isolated in the user interface tool.

In general, the tools might help to:

- *design* the interface given a specification of the end users' task,
- *implement* the interface given a specification of the design,
- *evaluate* the interface after it is designed and *propose* improvements, or at least provide information to allow the designer to evaluate the interface,
- *create* easy-to-use interfaces,
- *allow* the designer to rapidly investigate different designs,
- *allow* non-programmers to design and implement user interfaces,
- *allow* the end user to customize the interface, and
- *provide* portability.

The above specifys the characterizing features of the quality of any user interface tool. They can be used to evaluate the various tools to see how many features they

do support. Naturally, no tool will be able to help with everything; and different user interface designers may put different emphasis on the different features.

1.3 The purpose of this project

This project is an effort towards the rapid development of Graphical User Interfaces (GUIs) by specifying them in a high-level interface specification language, ISL. This high-level specification is translated into source code of a prototyping language which is then compiled to produce the actual GUI.

1.4 Brief overview of remaining chapters

The remainder of this thesis is structured in the following manner. Chapter 2 gives a brief description of the components of a graphical user interface and discusses the different specification styles. Chapter 3 introduces the Interface Specification Language (ISL) and explains the different widgets and features supported by ISL. Chapter 4 presents the available parser generators and then describes the implemented *translator*, which takes ISL as its input and generates the intermediate code for the user interface. Chapter 5 gives a detailed example of how to use ISL for a specific application and describes the implementation of the interface for this application. The final chapter contains concluding remarks, including a summary of the advantages of the proposed approach, restrictions of the implementation and directions for future research. Appendix A gives the specification of ISL in JavaCC. Appendix B gives the complete specification of user interface in ISL for a circuit

simulator application. Appendix C gives the actual Java code for the customized **canvas** object. Appendix D provides the instructions on how to use the ISL, GUI generator and the related files.

Chapter 2

Graphical user interfaces

The user interface is the part of the software system which gets the input data from the user and displays the output from an application program. The following section deals with the different components of the user interface and then describes the *toolkits* and *specification languages* available for user interface design.

2.1 Components of the graphical user interfaces

As shown in Figure 2.1, user interface components can be subdivided into three layers:

- the *windowing system,*
- the *toolkit,* and
- *high-level tools.*

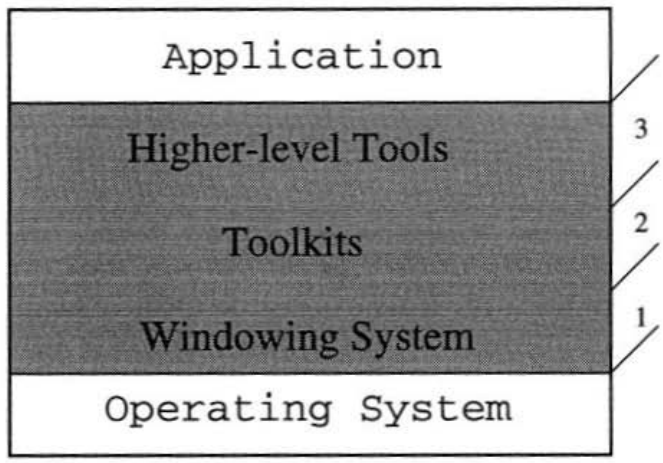

Figure 2.1: The components of user interface.

2.2 Windowing system

The *windowing system* is the lowest of the three layers of the user interface components and closely interacts with the underlying operating system. The "windowing system" supports the subdivision of the screen into different (usually rectangular) regions, called *windows*. Thereby it helps the user to monitor and control different applications by separating them physically into different parts of one or more display screens. The X system divides the window functionality into two layers: the *windowing system,* which is the functional or programming interface, and the *window manager* which is the user interface. Thus the "windowing system" provides procedures that allow the application to draw pictures on the screen and as well get input from the user, while the "window manager" allows the end user to move around windows, and is responsible for displaying the title lines, borders and icons for the windows. The X windowing system solved the problem of portability between different windowing systems by providing the hardware-independent interface to windows. However, many systems use the terminology "window manager"

to refer to both layers; for example, systems such as the Macintosh and Microsoft Windows do not separate the two layers.

2.3 Toolkits

Toolkits basically use the functionality provided by the underlying windowing system and provide a layer of abstraction. Hence the programmer need not worry much about the intrinsics of the underlying operating system and the windowing system. Toolkits are just a library of *widgets* that are available to application programs. A *widget*[1] is a GUI object with a particular appearance and behavior and is usually activated (e.g., "clicked") by mouse by the user to input some values. Typically, widgets in toolkits include menus, buttons, scroll bars, text input fields, etc. The user must, however, take into consideration the trivial things like the position of the widget, the size of the widgets, etc.

2.3.1 Toolkit types

Toolkits come in two basic varieties. The most conventional one is simply a collection of procedures that can be called by application programs. An example of this style includes the SunTools toolkit for the SunView windowing system [7]. The other variety uses an object-oriented programming style which makes it easier for the designer to customize the interaction techniques. Examples include InterViews [8], Xt [9], Tk [10] and AWT [11].

[1] widgets and objects are used interchangeably in this thesis

A natural way to visualize widgets is in units of objects since the menus and buttons on the screen seem like individual objects. They can handle some of the chores that otherwise would be left to the programmer (such as refresh operations). Another advantage is that it is easier to create custom widgets (by sub-classing an existing widget).

The usual way that object-oriented toolkits interface with application programs is through the use of *call-back* procedures. *Call-back* procedures are defined by the application programmer and are invoked when a widget is operated by the end user. For example, the programmer may supply a procedure to be called when the user selects a menu item.

2.3.2 Advantages and disadvantages of toolkits

Toolkits improve the consistency among the applications by making their interfaces appear and behave similarly to the other user interfaces created using the same toolkit. This is the first and foremost of the *eight golden rules of dialogue design* defined by Shneiderman [5]. Another inherent advantage of this approach is that each application does not have to re-write the standard functions, such as menus and other widgets.

On the other hand, a problem with toolkits is that the styles of interaction are usually limited to those provided by the tools. Another problem with toolkits is that they are often difficult to use since they may contain hundreds of procedures, and it is often not clear how to use the procedures to create a desired interface.

2.4 Higher level tools

Programming at the toolkit level can be very difficult. Hence, in their place, higher level tools that simplify the user interface software production process are desirable. These tools come in a variety of forms. One important way that they can be classified is by *how* the designer specifies what the interface should be. Some tools require that the programmer use a special-purpose language, others provide an application framework to guide the programming. While some automatically generate the interface from a high-level model or specification, others allow the interface to be designed interactively with the help of a visual programming environment (interface builders).

2.4.1 State transition networks

Since many parts of the user interface involve handling a sequence of input events, it is natural to implement the interface by using a state transition network to code the interface. A transition network consists of a set of states, with arcs outgoing from each state labeled with the input tokens that will cause a transition to the next state. In addition to input tokens, calls to application procedures and the output to be displayed can also be associated with each arc. In 1968, Newman [12] implemented a simple tool using finite state machines which handled textual input. This was apparently the first user interface tool. Many of the assumptions and techniques used in modern systems were present in Newman's different languages for defining the user interface and its semantics.

State diagram tools are most useful for creating user interfaces where the in-

terface has a large number of modes (each state is considered a mode here). For example, state diagrams are useful for describing the operation of low-level widgets or the overall global flow of an application. However, most highly-interactive systems attempt to be mostly "mode-free" which means that at each point the user has a wide variety of choices of what to do next. This requires a large number of arcs out of each state, so state diagram tools have not been successful for these interfaces.

Another problem with the large number of arcs out of each state is that it can be very confusing for complex interfaces, since this can become a "maze of transitions" and are difficult to follow.

Jacob [13] invented a new formalism, which is a combination of state diagrams with a form of event languages, to exploit the advantages of the state transition diagrams.

Transition networks have been thoroughly researched, but have not proven particularly successful or useful in either the research or commercial approach.

2.4.2 Context-free grammars

Grammar-based systems are based on parser generators used in compiler development systems. For example, the designer might specify the user interface syntax using some form of Backus-Naur Form (BNF). Examples of grammar-based systems are Syngraph [14] and parsers built using the YACC and LEX tools.

Grammar-based tools, like state diagrams, are not appropriate for specifying highly-interactive interfaces since they are oriented to batch processing of strings with complex syntactic structures. These systems are best for textual command

languages, and have been mostly abandoned for specification of user-interfaces by researchers and commercial developers.

2.4.3 Event languages

In this kind of specification system, the inputs are considered to be "events" that are sent to individual event handlers. Each handler will have a condition clause that determines what types of events it will handle, and when it is active. The body of the handler can generate (next) events, change the internal state of the system, or call application routines.

The ALGAE system [15] uses an event language which is an extension of Pascal. The user interface is programmed as a set of small event handlers which ALGAE compiles into conventional code. The HyperTalk language that is part of HyperCard for the Apple Macintosh can also be considered an event language.

The advantages of event languages are that they can handle multiple input devices active at the same time, and it is straightforward to support non-modal interfaces where the user can operate any widget or object at any point of time. The main disadvantage is that it can be very difficult to create the correct code, since the flow of control is not localized and small changes in one part can affect many different pieces of the program. It is also typically difficult for the designer to understand the code once it reaches a non-trivial size.

2.4.4 Declarative languages

Another approach is to define a language that is *declarative* (stating what should happen) rather than procedural (how to make it happen). Cousin [16] and OpenDialogue [17] both allow the designer to specify user interfaces in this manner. The user interfaces supported include textfields, menus and buttons. There are also graphic output areas that the application can use in whatever manner desired. The application program is connected to the user interface through "variables" which can be set and accessed by both the user interface and the application program.

The layout description languages constitute another class of declarative languages that comes with many toolkits. For example, Motif's User Interface Language (UIL) allows the layout of widgets to be defined. Since the UIL is interpreted when an application starts, users can (in theory) edit the UIL code to customize the interface. UIL is not a complete language, however, in the sense that the designer must still write C code for many parts of the interface, including any areas containing dynamic graphics and any widgets that change.

The advantage of using a declarative language is that the user interface designer does not have to worry about the time sequence of events and can concentrate on the information that needs to be passed back and forth.

The disadvantage is that only certain types of interfaces can be provided in this way, and the rest must be programmed manually. The kinds of interactions available are preprogrammed and fixed. In particular, these systems provide no support for such things as dragging graphical objects, rubber-band lines, drawing new graphical objects, or even dynamically changing the items in a menu based on the application mode or context. However, these languages are now proving

successful as intermediate languages describing the layout of widgets (such as UIL) that are generated by interactive tools.

2.4.5 Constraint languages

Constraints are relationships that are declared once and then maintained automatically by the system. For example, the designer can specify that the color of the rectangle is constrained to be the value of a slider and then the system will automatically update the rectangle if the slider is moved.

A number of user interface tools allow the programmer to use constraints to define the user interface. NoPump [18] and Penguims [19] allow constraints to be defined using spreadsheet-like interfaces.

The advantage of constraints is that they are a natural way to express many kinds of relationships that arise frequently in user interfaces. For example, that lines should stay attached to boxes, that labels should stay centered within boxes, etc.

However, a disadvantage of constraints is that they require a sophisticated runtime system to solve them efficiently. Another problem is that they can be difficult to debug when specified incorrectly since it can be difficult to trace the cause and consequences of changing values.

2.4.6 Database interfaces

Major database vendors such as Oracle provide tools which allow designers to define the user interface for accessing and setting data. Often these tools include inter-

active forms editors, which are essentially interface builders, and special database languages like Structure Query Languages (SQL). The main disadvantage of these kinds of packages is that they are fine-tuned more towards database applications rather than a general purpose application.

2.4.7 Visual programming

Another approach to user interface design is by way of using visual programming languages based on the hypothesis that two-dimensional visual languages are easier to learn than one-dimensional textual languages. Many approaches to using visual programming to specify user interface have been investigated. The user interface is usually constructed directly by laying out pre-built widgets, in the style of interface builders. Using a visual language seems to make it easier for novice programmers, but large programs still suffer from the familiar "maze of wire" problem.

2.4.8 Summary of different approaches

In summary, there have been many different types of languages that have been designed for specifying user interfaces. The major problem with all these approaches is that they can only be used by professional programmers, as they need to know a great deal about user interface design. Unfortunately, quite often this is not the case.

This work is an effort to bridge the gap between the reality and the present situation in user interface design. A high level specification language is proposed, in which the user can specify the user interface. The specification is then translated

to a high level language, which is compiled to get the final GUI.

The apparent advantage of this approach is that many of the technicalities of the user interface can be hidden from the GUI programmer, and the interface can be designed and/or modified quickly without much effort or resources.

2.5 Other considerations

The proliferation of GUIs such as Toolbox, X11/Motif, X11/OpenLook or NextStep on different operating platforms creates a perplexing problem for the developer. This problem becomes a real issue when the application program is aimed at targeting more than one operating system. The most apparent solution would be separating the application component from the user interface component thereby creating a user interface abstraction.

The primary goal in designing a user interface abstraction is that the amount of effort to retarget the user interface component must be much less than that required to implement it from scratch. Another potential benefit of the abstraction is that maintenance of the code will be much easier.

An application using a specific GUI should have the same appearance and operating characteristics as the other so-called "native" applications. Native applications are the ones that define the "look-and-feel" of a software system. By providing consistency the user can apply techniques already learned with one application to other applications on the same machine. Conversion of the application to a new GUI must be done carefully, otherwise an application created for one system will have a "foreign" appearance or feel on subsequent systems and will probably not

gain wide acceptance.

Performance is another important issue which needs to be considered when the GUI is developed. The development cost in terms of time, resources and overhead should also be minimal.

Chapter 3

Specification of user interfaces

The simplest and most convenient method, from the user's point of view, is to develop the user interface automatically from a high level specification. In order to specify an interface in a high-level language, first this specification language must be defined.

3.1 Interface components

To produce the interface component with a minimal amount of programming, an overall structure of the component and its interrelations need to be introduced. In general, a user interface consists of graphical objects, implicit or explicit information about the change of control windows, semantic actions and results of these actions. *Graphical Objects* consist of windows, dialog boxes, menus, buttons and other items that are directly visible to the user. *Change of focus* defines how the interface will

change on inputs from the user or values returned by the application. *Semantic actions* are the operations that drive the applications. These actions are associated with the events that the user might generate. When an event occurs, a semantic action associated with it is performed. These actions can take the form of calls to user supplied functions, execution of programs or files [20]. This allows the user to pass data to the application and allows the application to output the results to a file or to present the results on a display.

3.2 Interface Specification Language (ISL)

It is assumed that the specification of the user interface is a sequence of interface object descriptions. In the proposed specification language, the supported objects include buttons, labels, edit texts, lists, checkboxes, menus and windows. Each of the objects has a list of attributes like *name* of the object, *action* to be performed on selecting the object, default values, etc. Attributes are uniformly specified as pairs:

<attribute_name> = <attribute_value>

In this document, attribute_value is also referred to as *value* in some places. The definition of the specification is:

<specification> ::= START_ISL <object_list> END_ISL

<object_list> ::= <object> | <object_list> <object>

<object> ::= <identifier> (<list>);

<list> ::= <list_a> | <list_i>

<list_i> ::= <identifier> | <list_i>, <identifier>

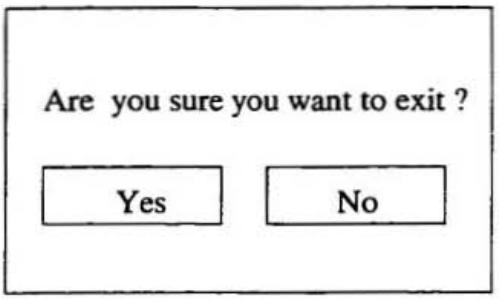

Figure 3.1: An example information dialog window.

<list_a> ::= <pair> | <list_a>, <pair>

<pair> ::= <attribute_name> = <attribute_value>

Any sequence of specification begins with the keyword **START_ISL**, followed by the actual specification of the interface and ends with the keyword **END_ISL**. The <object_list> in the above specification can be recursively expanded to any number of <object> definitions. In turn, <object> can be recursively expanded to any number of <identifier>s or <pair>s. Since any GUI is a collection of series of interface objects, they can be defined using <object>s. The finer details of the objects are defined using the <pair> specifications which are nothing but pairs of attribute names and their corresponding values.

Comments can be included in the specification file in C++ format, i.e. both "\\" and "/* */" formats can be used, but nested comments are not permitted.

For example, the window shown in Figure 3.1 can be defined using ISL as follows:

```
window : info(label = question, button = yes, button = no);
label : question(name = "Are you sure you want to exit ? ");
button :yes(name = "Yes", action = exit);
button :no (name = "No", action = hide);
```

The above object specification is the expansion of <object_list> into <object>

four times. As mentioned in Appendix A, "window", "label" and "button" are some of the objects supported by ISL. The "window" object is in turn expanded to three attribute pairs. As shown above the attribute name again can be an object name like "label", "button" or just simple attributes like "name" or "value" or can be the special attribute "action". If the attribute name is a valid object name, then it can further be expanded recursively. If the attribute name is "action", then it will result in a call to the function named that attribute value. In the above example, for button named "Yes", action is "exit". This will result in a call to the function named "exit".

3.3 Objects and their attributes

This section gives the complete description of all the objects and the details of their attributes which are supported by the current implementation of the ISL. It should be noted that ISL has been designed to facilitate easy addition of new objects to the existing basic objects with very little modification of the source file.

3.3.1 window

Any specific application can have only one **window** object. This is the main container object which contains all the other objects.

Attributes

Special attributes supported by the **window** object are:

- *preferred_size:* specifies the preferred size of the application window, which can always be enlarged or reduced at run time. It is always advisable to specify this attribute. The attribute has two arguments: one for x and one for y coordinate and both indicate the number of pixels.

- *layout:* specifies the arrangement of the other objects within this container object. It can take "flow", "grid" or "border" as its value. These arguments are explained in detail below.

Other attributes include *menu, canvas, button, label, textfield, panel, checkbox* and *checkboxgroup* which are explained in detail below.

Typical usage:

```
window: mainwindow(preferred_size = (500, 400), layout = flow,
        menu = mainmenu, canvas = graph_canvas, button = exit);
```

3.3.2 menu

This **menu** object creates a new menu bar.

This is the only object which takes arguments as a list rather than as argument pairs.

Typical usage:

```
menu : mainmenu(file, analysis, about);
```

It is mandatory that each item in the list must have a corresponding **menuitem** specification, i.e., the above specification must be followed by three **menuitem** specifications, one for each "file", "analysis" and "about".

3.3.3 menuitem

menuitem objects behave like **buttons.**

Attributes

Attributes supported by this object are:

- *submenuitem:* specifies the individual menu item name.
- *name:* specifies the string to be displayed on the menu item.
- *action:* specifies the action to be performed when that particular menuitem is clicked. This action can be a call to a function or it can in turn pop up another subwindow.

Typical usage:

```
menuitem : analysis(name = "Analysis",
           submenuitem = ac_analysis, name = "AC Analysis",
                                      action = showaaw,
           submenuitem = dc_analysis, name = "DC Analysis",
                                      action = showdaw,
           submenuitem = tr_analysis, name = "Transient",
                                      action = showtaw);
```

3.3.4 label

Label object is provided for displaying text in the GUI.

Attributes

- *name:* specifies the string to be displayed on the object.

Typical usage:

```
label : sourcetype(name = "Source Type");
```

3.3.5 button

A button object is a simple control that generates an action event when it is "clicked".

Attributes

- *name:* specifies the string to be displayed on the button.
- *action:* specifies the action to be performed when the button is selected by the user. Normally it will be an invocation of a function.
- *lang*: specifies the language in which the procedure has been implemented. The default is "Java." If its value is "native", then it implies a call to a procedure written in some other language, like C/C++.

Typical usage:

```
button : cancel(name = "Cancel", action = hide);
```

3.3.6 subwindow

Subwindow is a dialog box that pops up when the user selects a button or a menu item. It is similar to **window** object, and the only difference is that an application can have any number of **subwindows.**

Attributes

- *layout*: specifies the arrangement of other objects within this container object. It takes "flow", "border" or "grid" as its value.

The other attributes include the objects namely *panel, button, label, textfield, checkboxgroup, checkbox* and *canvas.*

Typical usage:

```
subwindow: showaaw(layout = border, panel = toppanel,
                   canvas = center_canvas, panel = bottompnl);
```

3.3.7 panel

Like **window** or **subwindow**, it is also a container class, within which other objects can be placed. Hence it also has the **layout** attribute.

Attributes

- *layout*: specifies the arrangement of other objects within this container object as in **subwindow.** It can take "flow", "grid" or "border" as its value.
- *location*: since the **panel** object itself can be inside a container object, this attribute specifies the location within the container. It can take "north", "south", "west" or "east" as its value.

Apart from this, it can have all the other objects that can be placed in a container namely *button , label* etc.

Typical usage:

```
subwindow : showaaw(layout = border, panel = toppanel);
panel : toppanel(location = north, label = label1, textfield = tf1);
```

3.3.8 textfield

textfield is an object which allows the editing of a single line of text.

Attributes

- *value:* specifies the "default value" to be displayed on the screen.

Typical usage:

```
textfield : tf1(value = 1000);
```

3.3.9 checkbox

checkbox object can be used if the user wants to have a boolean variable to be displayed on the interface.

Attributes

- *name*: specifies the text to be displayed on the screen.
- *value*: specifies the state of the object, which can either be "true" or "false".

Typical usage:

```
checkbox : xb1(name = "Linear", value = true);
checkbox : xb2(name = "Octal",  value = false);
checkbox : xb3(name = "Decimal",value = true);
```

Note: There is no connection between `xb1`, `xb2` and `xb3`. All three act independent of each other. So, they can all be true, or all be false or they can take any other possible combination of values.

3.3.10 checkboxgroup

Unlike **checkbox** object which acts independently, the **checkboxgroup** object is used to create a multiple-exclusion scope for a set of choices. For example, creating a **checkboxgroup** buttons with the same checkboxgroup object means that only one of those checkbox buttons will be allowed to be "on" at a time.

Attributes

- *name:* specifies the string to be displayed in front of the checkbox button on the screen.
- *value:* specifies the boolean state of the button.

Typical usage:

```
checkboxgroup : xbg(name = "Linear", value = false,
                name = "Octal", value = false,
                name = "Decimal",value = true);
```

Note: Only one of the checkbox buttons can be true. If the user selects some other checkbox button, then that button's state will become true and changes the other to false.

3.3.11 canvas

This is the object to be used if the user wants to plot graphs or draw figures. Since the **canvas** object needs more information, like an array of data to be plotted and

algorithm to scale the graph to fit the canvas and the data arrays, it is implemented as a separate Java file[1]. Whenever the user wants to use this object, the implemented **canvas** class is instantiated instead of using the standard Java canvas widget.

Attributes

- *location:* since **canvas** is an object which can be placed within any other container objects, it has this attribute for placement within the container.
- *name:* specifies the name of the canvas.

Typical usage:

```
canvas : graph_canvas(location = center, name = "Graph");
```

3.4 Geometry managers

Widgets do not determine their own size and location on the screen. This function is carried out by *geometry managers*. Each geometry manager implements a particular style of layout. Given a collection of objects to manage and some controlling information about how to arrange them, a geometry manager assigns a size and location to each object.

[1]see Appendix C for more information

3.4.1 Flow layout

In order to arrange the set of objects in a horizontal row, the layout has to be specified as "flow". The window manager will then position the widgets so that they abut but do not overlap. If the user changes the size of the containing window, then the window manager will adjust the position of the widgets automatically to accommodate the new dimensions.

3.4.2 Border layout

The other supported layout type is "border". This layout will arrange the widgets into position using the directions namely "north", "south", "east", "west" and "center".

When border layout is used a location parameter must be specified. The default interpretation of the location is "center".

The "north", "south", "east" and "west" components get laid out according to their preferred sizes and the constraints of the container's size. The "center" component will get any space left over.

3.4.3 Grid layout

This type creates a grid layout with specified rows and columns, which are obtained as parameters from the user.

Typical usage:

```
panel : newpanel(layout = grid(2,2));
```

The above specification will arrange the widgets in the panel with two elements in each row.

3.5 Examples

This section gives a brief overview of how to arrange widgets within a container object like **window, subwindow, panel, etc.**

3.5.1 More on layouts

The following set of ISL specifications gives a general idea of how different layouts work.

```
window : main_window(layout = flow, preferred_size=(200,100),
         button = one, button = two, button = three, button = four,
         button = exit);
button : one(name="One", action= fun_one);
button : two(name="Two", action= fun_two);
button : three(name="Three", action= fun_three);
button : four(name = "Four", action = fun_four);
button : exit(name="Exit", action= fun_exit);
```

The above specification with the *layout* attribute value of "flow" will generate a window as shown in Figure 3.2.

The size of the window can be altered any time and the arrangement of the widgets within the window depends on the layout type. Since the *layout* type is "flow", when the window size is increased to 250 (from 200), all the buttons will get aligned in the same line as shown in Figure 3.3.

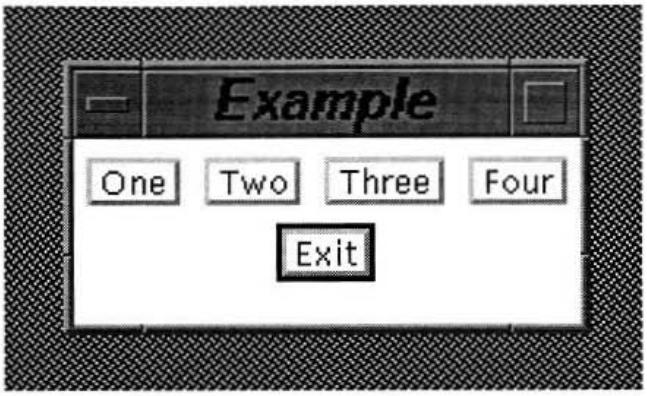

Figure 3.2: Flow layout.

If the *layout* attribute of the **window** object is changed from "flow" to "grid" as shown below, then this will alter the appearence as shown in Figure 3.4.

```
window : main_window(layout = grid(2,3), preferred_size=(200,100),
         button = one, button = two, button = three, button = four,
         button = exit);
```

If we want to change the arrangement completely then the *border* layout can be used. A sample specification is given below and arrangement of the widgets will be as shown in Figure 3.5.

Note: It is not enough just to specify the layout type to be the "border". The location of each widget within the container has to be specified so that they can be placed suitably. Another obvious restriction with this layout type is that at the most it can accomodate only five objects. To overcome this problem, a series of panels can be used, where each panel can accomodate up to five objects.

```
window : main_window(layout = border, preferred_size=(200,100),
         button = one, button = two, button = three, button = four,
         button = exit);
button : one(location = north, name="One", action= fun_one);
```

```
button : two(location = south,  name="Two", action= fun_two);
button : three(location = west, name="Three", action= fun_three);
button : four(location = east, name = "Four", action= fun_four);
button : exit(location = center,name="Exit", action= fun_exit);
```

In the above example specifications, only the **button** is used to keep the example simple. For the same reason, only the **window** object is used for placing the other widgets. In those specifications, the **button** object can be replaced by any other widget and the **window** can be replaced by another container object like **subwindow, panel** etc. More detailed examples are given in Chapter 4 and Chapter 5.

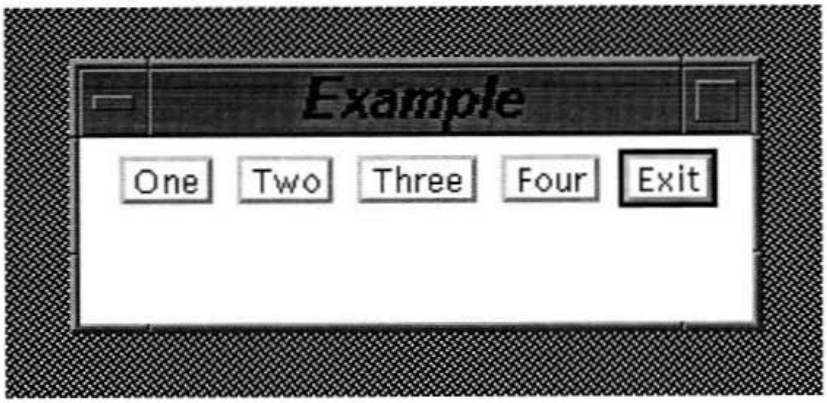

Figure 3.3: Flow layout.

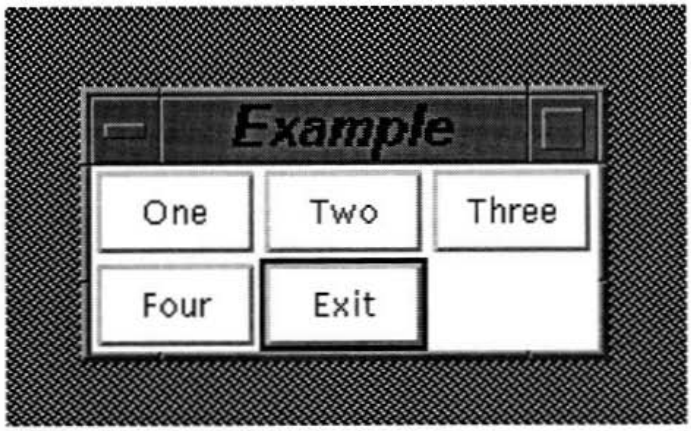

Figure 3.4: Grid layout.

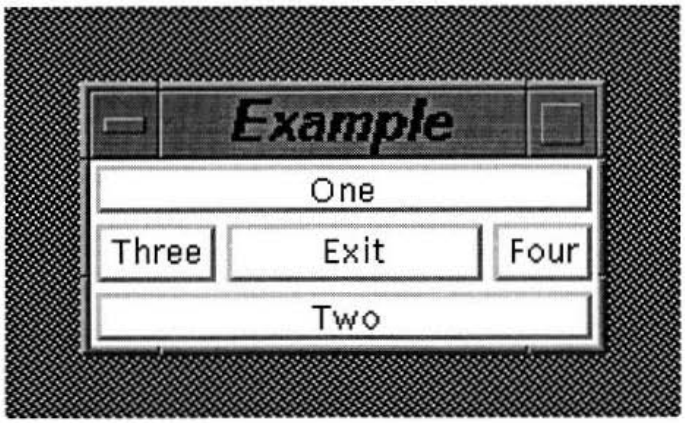

Figure 3.5: Border layout.

Chapter 4

Translation

ISL specifications described in the previous chapter serves no purpose unless it is converted into a format suitable for execution. The approach that has been taken in this work is a two-stage translation process. The user input (ISL specifications) is translated into intermediate code (using the implemented translator) in the first stage. In the next stage, the intermediate language's compiler is used to compile the code into a format suitable for execution. This chapter gives more information on the above process.

4.1 Two-stage translation

As the first stage of the two stage translation process, the ISL specification is given as input to the translator as shown in Figure 4.1. The decision to be made at this point is: what should be the intermediate language for the translator and what are

the important characteristics of the intermediate language ? The available options are: either to generate a high-level language code and then compile it using that language's compiler to get the final GUI or to directly generate the executable code for the GUI from ISL.

There are several advantages of translating the specification into a high-level language and then using that high-level language's compiler to compile it to get the final GUI.

First, if the high-level specification has to be translated directly to a machine code or assembly language for the target hardware, then a thorough knowledge of the machine's architecture is required (code generation is concerned with the choice of machine instructions, allocation of machine registers, addressing, interfacing with the operating system and so on).

Second, in order to produce faster or more compact code, the code generator should include some form of code improvement or code optimization. If the intermediate language is a high-level language, the compiler of this language will take care of these optimization issues.

Third, in the case of using an intermediate language, the code generated is easy to understand. This is important during the debugging process as it is possible to see immediately what the code generator is doing. So correcting any unexpected behavior during the development phase is easier.

Finally, a high-level intermediate language provides platform independence because there is no tight coupling between the code generated and the machine architecture. Hence portability issues can be handled in an elegant way.

For the implementation purpose, to make the job simpler, it has been decided

to generate a high-level language code as the output by the translator. The entire process is represented in Figure 4.1.

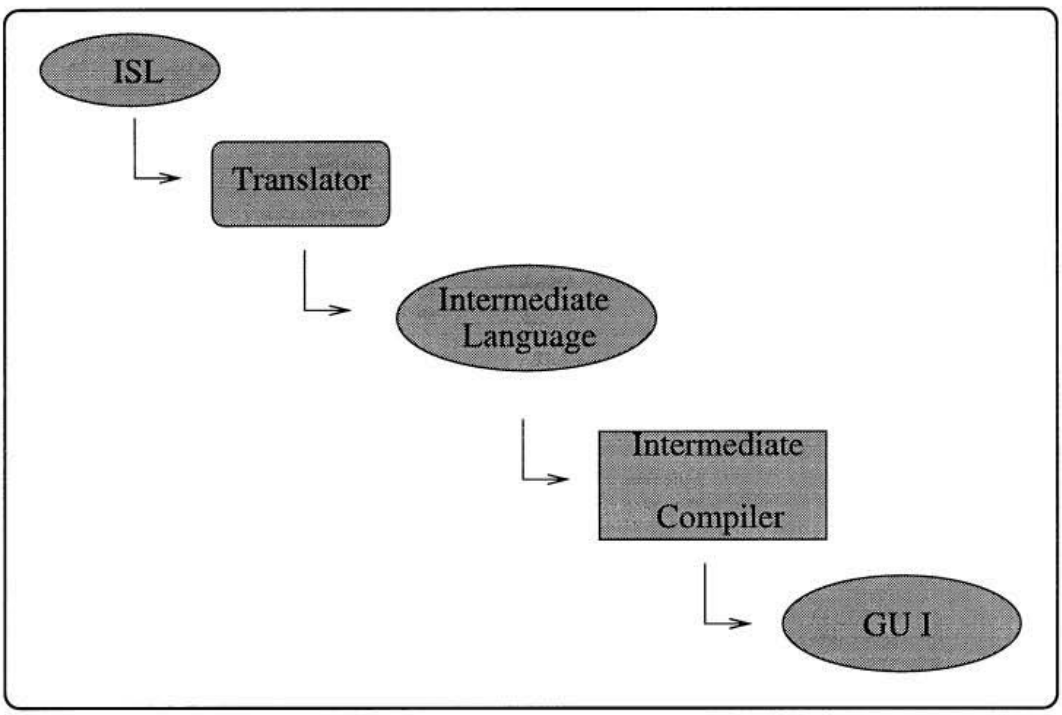

Figure 4.1: Processing interface specification.

The process of translation of ISL specification can be divided very broadly into two steps:

1. The analysis of the source program.
2. The synthesis of the object program.

In a typical compiler, the analysis step consists of three phases: lexical analysis, syntax analysis and semantic analysis. The synthesis step is simply the *code generation phase.*

4.2 Compiler writing tools

Rather than writing a new scanner and parser to process the ISL specifications, the available tools are analyzed to select the tool which could be used for translation.

4.2.1 Lex and Yacc

The most popular compiler writing tools are **lex** and **yacc. Lex** is a tool for generating *lexical analyzers.* **Yacc** is a general-purpose parser generator that converts an LALR(1) grammar into a table-driven C language parser for this grammar. **Yacc** has several shortcomings, including that it cannot accept extended BNF grammars, and requires that separate lexical and syntactic descriptions be maintained and be consistent. It provides only minimal support for error recovery [25].

4.2.2 Flex and Bison

The Free Software Foundation's GNU project supports an "improved" version of *lex* and *yacc* called *flex* and *bison*, for use on Unix and other non-Unix platforms. They have a better error correction and error detection facility when compared to its predecessors.

4.2.3 PCCTS

The Purdue Compiler Construction Tool Set (**PCCTS**) is another compiler writing toolkit. Two components of **PCCTS,** namely DFA and ANTLR [1], provide similar

[1] ANother Tool for Language Recognition.

functions as **lex** and **yacc.** However ANTLR accepts LL(k) grammars as opposed to the LALR(1) grammars used by **yacc.** The code that **PCCTS** generates is much more readable than the code generated by **yacc,** and ANTLR output consists of recursive C/C++ functions. Diagnosing errors in the grammar specifications is comparatively easier since the code is in a more readable form. The main problem is that the symbol table generated is inefficient and so it grows rather large when processing a big collection of files. It also suffers from macro redefinition and memory management problems.

4.2.4 JavaCC

JavaCC is a parser generator written in Java, which is customizable and generates parser in the Java language. By default, JavaCC generates an LL(1) parser. However, there may be portions of the grammar that are not LL(1). JavaCC offers the capabilities of syntactic and semantic lookahead to resolve parsing conflicts locally at these points, i.e., the parser is LL(k) only at such points, but remains LL(1) everywhere else for better performance.

Advantages and disadvantages

JavaCC is much easier to use than the tools discussed above. The way JavaCC handles grammars is much more straightforward than LALR parsers. JavaCC generates a top-down parser and has a detailed error reporting facility whereas bottom-up parsers like **yacc** are non-intuitive and have a poor error reporting facility. Also, JavaCC's error messages suggest likely corrections. JavaCC comes with an algo-

rithm to aid in inserting the necessary lookahead information and it also supports "infinite lookahead." The main concern with **JavaCC** is that the generated parsers are not as fast as in the case of **yacc.**

JavaCC can be easily customized to generate Java code (or for that matter, any code) when required.

Since JavaCC is easily customizable, supports the code generation feature, and has a detailed error reporting facility, JavaCC has been selected for the implementation of the ISL.

4.3 High-level intermediate language

As mentioned in the previous sections, generating a high-level language code eliminates many complications for the implementation purpose. But literally hundreds of high-level programming languages are available for developers to solve problems in specific areas. For the implementation one high-level language has to be selected. The options considered for this project were only Tcl/Tk and Java, because tools developed in other languages would be more cumbersome and difficult to port to other platforms.

4.3.1 Tcl/Tk

As a scripting language, Tcl is similar to UNIX shell languages like Bourne Shell (sh) and the C Shell (csh). It provides enough constructs (variables, control flow, and procedures) to build complex scripts that assemble existing programs into a new tool tailored for a particular need [24]. As a script based approach to the user

interface programming, it has the following benefits:

- Development is fast because of the rapid turnaround; there is no waiting for long compilations.
- The Tcl commands provide a higher-level interface to X.
- The core set of Tk widgets is often sufficient for most of the user interface needs. Furthermore, it is also possible to write custom Tk widgets in C, if required.

4.3.2 Java

Java language environment, on the other hand, creates an extremely attractive middle ground between very high-level, portable, slow scripting languages and very low level, fast but non-portable, compiled languages. Java provides a level of performance that's entirely adequate for all but the most computationally intensive applications. The other advantages are:

- Java is a simple language. Java omits many rarely used, poorly understood, confusing features of C++.
- Java has automatic garbage collection, thereby simplifying the task of memory management.
- Java is object-oriented. It facilitates clean definition of interfaces and makes it possible to provide reusable "software ICs".
- Java is robust. Java puts a lot of emphasis on early checking for possible problems. It is strongly typed, hence will not allow automatic coercion of one data type to another. The single most important difference between Java

and C/C++ is that Java does not use explicit pointers which eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting. Java programs also cannot gain unauthorized access to memory, which can happen in C/C++.

- Static typing. Dynamic languages like Lisp, Tcl and Smalltalk are often used for prototyping, for they do not force decisions to be made early. But Java forces choices to be made early because it has static typing. Along with these choices comes a lot of assistance: any call to invalid functions will be checked at the compilation time and not delayed till run time.

- Architecture neutral. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC, Apple Macintosh, and different flavors of Unix. With Java, the same version of the application runs on all platforms without any modification. Java compiler makes this possible by generating *bytecode*[2].

- Java is portable. Being architecture neutral increases the portability by a very high degree, but there's more to being portable than just architectural neutral. Unlike C and C++, there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point representation.

- Java is interpreted. The performance of the interpreted bytecode is usually more than adequate. There are situations where higher performance is re-

[2]The Java compiler compiles the source code into a bytecode i.e., each part of the source code is reduced to a sequence of bytes that represents instructions to a virtual machine; virtual because those sequence are not specific to any microprocessor.

quired. In such situations, the bytecode can be translated to machine code for the particular CPU in which the application will be running. In such cases, the performance is almost indistinguishable from native C or C++.

- Java supports multi-threading. Java has a set of synchronization primitives that are based on the widely used monitor and condition variable paradigm introduced by Hoare [23]. Hence Java has a better interactive responsiveness and real-time behavior.
- Other benefits include an extensive networking facility and security for network applications.

Because of many of these features, Java has been selected as the intermediate language.

4.4 Implementation details

The translator is implemented in JavaCC. Appendix A gives the complete specification of ISL for JavaCC excluding the code generation part. When the input file (GUI.jack [3]) is compiled using JavaCC, it results in generation of three new files:

- GUI.java: the parser,
- GUITokenManager.java: the lexical analyzer, and
- GUIConstants.java: a bunch of internal constants

Apart from the above three files that are generated for every input grammar, three more files are generated once for use with all input grammars, namely:

[3] JavaCC was formerly known as Jack

- ASCII_UCodeESC_CharStream.java: an ASCII stream reader to process unicode sequence,
- Token.java: the type specification for the "Token" class, and
- ParseError.java: error handling file.

Invoking JavaCC on the input grammar for ISL results in the generation of the above files. Then the generated Java files are compiled using the Java compiler. This results in the generation of the *translator* for ISL.

The generated translator takes the specifications in ISL as input and generates a set of Java files. The generated files have to be compiled using the Java compiler to get the "actual" GUI.

4.4.1 Code generation process

Once the input specification is scanned and parsed, the complete information required to generate the intermediate code is gathered in the internal data structures of the translator. For any input specification, there will be one main file (main class) and zero or more auxiliary files (auxiliary classes), depending on the specification. For every **window** object there is a main file and for every **subwindow** object there is an auxiliary file.

In the code generation process, first the main file is generated. Within that file, the user interface component declarations are generated in the order of ISL specifications. If the **window** object has the *preferred_size* attribute set, then the standard preferredSize() method[4] is generated. The *translator's* internal data

[4]In Java all functions are known as methods

structure (x_val and y_val) contains the argument values for the preferredSize() method.

Then the constructor[5] for the main class is generated. Main class is basically a "container" object, which contains other interface components. Within the main class constructor, all the interface objects are defined by instantiating the corresponding Java objects and are laid suitably within the containers based on the container's *layout* attributes.

The application must eventually react to the user input or user events, such as input from the keyboard or a pointing device such as a mouse. There are two common models that are used to support the handling of input events by the program. Either the application program can continuously poll all the input devices to check for any events or the events generated by the interface components can be queued for processing. The latter approach is used in Java to handle the events. An "action" method is generated by the translator . This method has an entry for all the potential interface objects which might generate any events.

If the *action* is "hide" or "exit", the translator automatically generates a call to standard Java methods hide() or System.exit(0), respectively. If the *action* is anything else, then the complete function has to be specified.

Then, the "main" method is generated which basically instantiates the main class and displays the main window on the screen. From that point onwards, the event-handler takes charge of the complete application by processing the incoming events.

[5]Constructor is a member function that is executed automatically whenever an object is created, in order to initialize the internal data structures of that object.

After the generation of the main file, the auxiliary files are generated in succession. The major difference between the main file and auxiliary file is that there is no "main" method for the auxiliary files. Except for that, the rest of the code generation process remains the same. The following sections show the correspondence between the elements of ISL and the Java code generated.

4.4.2 Sample translation

This section gives a complete example of interface specification using ISL for a small section of an application, which gets the user name and SIN number in the main window. The main window also has two more buttons: "Exit" and "More Info", as shown in Figure 4.2.

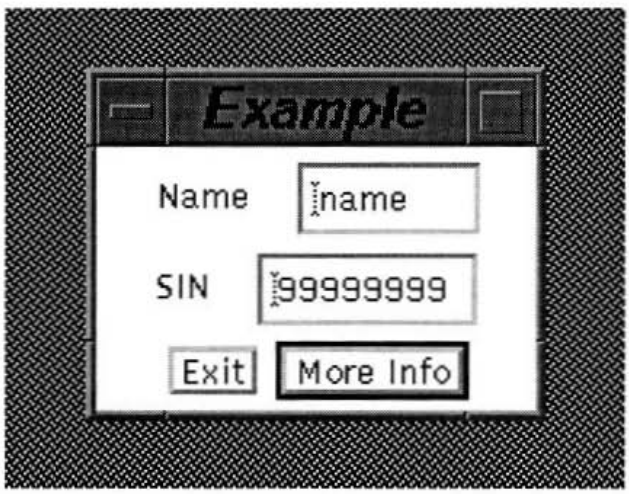

Figure 4.2: Main window.

When the "Exit" button is clicked, the application terminates; when "More Info" button is clicked, another dialog window appears on the screen, with three "radio buttons" to select the age group as shown in Figure 4.3.

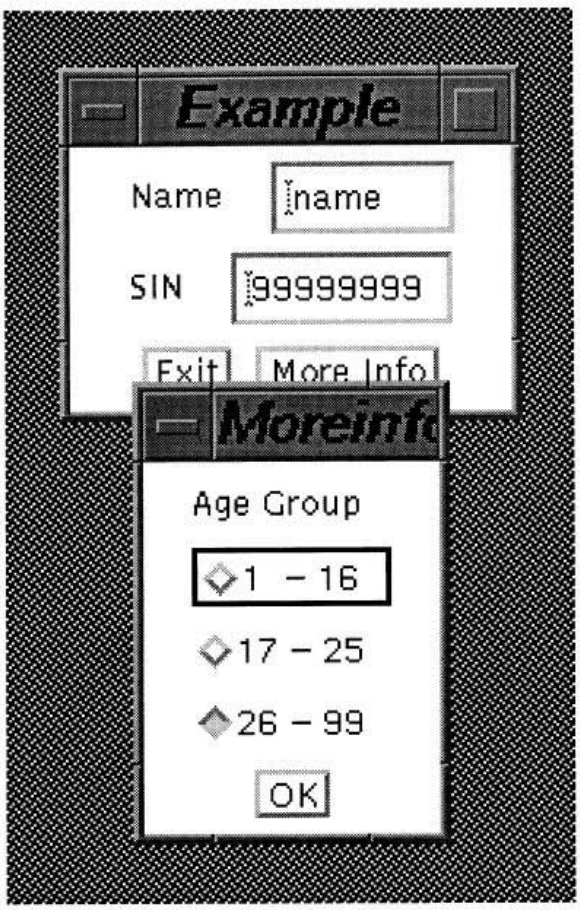

Figure 4.3: Dialog window on selecting "More Info" button.

4.4.3 Sample specification using ISL

```
START_ISL
window      : mwindow(layout = flow, preferred_size=(180,130),
              label = lbl1, textfield = tf1, label = lbl2,
              textfield = tf2, button = exit, button = more);
label       : lbl1(name= "Name");
textfield   : tf1(value = "name ");
label       : lbl2(name="SIN");
textfield   : tf2(value = 99999999);
button      : exit (name = "Exit", action = quit);
button      : more(name="More Info", action = showmoreinfo);
```

```
subwindow     : showmoreinfo(layout = flow, label = age,
                checkboxgroup = cbg, button = ok);
label         : age(name="Age Group");
checkboxgroup: cbg(item = first_grp, name = "1 - 16", value = true,
                item = sec_grp , name = "17 - 25", value = false,
                item = third_grp , name = "26 - 99", value = false);
button        : ok(name = "OK", action = hide);
END_ISL
```

4.4.4 Generated code

This section gives the code generated, by the translator, for the above specification. There is clear mapping between the ISL specification given in the previous section and the code generated. For example, in the specification, the *layout* of the **window** object is specified as "flow". This generates the following line of code:

```
setLayout(new FlowLayout());
```

The *preferred_size* attribute specification results in the generation of the following lines of code:

```
public Dimension preferredSize() {
return new Dimension(180,130);
}
```

For the ISL specification:

```
button : exit(name = "Exit", action = quit);
```

the code generated is:

```
public boolean action (Event event, Object arg){
   if (event.target == exit){
```

```
        System.exit(0);
        return true;
    }
    return false;
}
```

The action triggered by the "Exit" button is a standard action provided by the Java. To execute some customized action, the ISL has to be specified in the following way:

```
button    : more(name = "More Info", action = showmoreinfo);
subwindow: showmoreinfo(layout = flow, label = age, checkboxgroup = cbg,
           button = ok);
```

The first line of the specification says that the action to be triggered when the "More Info" button is clicked is "showmoreinfo". According to the next line, "showmoreinfo" is a **subwindow** object. It implies, that when the "More Info" button is clicked, the subwindow has to be popped up.

The translator is implemented in such a way that when a new **subwindow** object has to be created, the corresponding code is generated in a separate file and the file name is the same as the action specified, with its first letter capitalized. In this case a new file, "Showmoreinfo.java" is generated. The complete code generated for the example in Section 4.4.2 is:

```
import java.awt.*;
import java.lang.Math;
public class MMainframe1 extends Frame {
    private Showmoreinfo showmoreinfo;
    Label lbl1;
    TextField tf1;
```

```
Label lbl2;
TextField tf2;
Button exit;
Button more;
public Dimension preferredSize() {
    return new Dimension(180,130);
}
public MMainframe1() {
    setBackground(Color.white);
    setLayout(new FlowLayout());
    lbl1 = new Label("Name");
    tf1 = new TextField("name ");
    lbl2 = new Label("SIN");
    tf2 = new TextField("99999999");
    exit = new Button("Exit");
    more = new Button("More Info");
    add(lbl1);
    add(tf1);
    add(lbl2);
    add(tf2);
    add(exit);
    add(more);
}
public boolean action(Event event, Object arg){
    if (event.target == exit){
        System.exit(0);
        return true;
    }
    if (event.target == more){
        if (showmoreinfo == null ){
            showmoreinfo = new Showmoreinfo(this, "Moreinfo");
        }
        showmoreinfo.show();
        return true;
```

```
        }
        return false;
    }
    public static void main(String args[]){
        MMainframe1 mainframe= new MMainframe1();
        mainframe.setTitle("Example");
        mainframe.pack();
        mainframe.show();
    }
}

/* Showmoreinfo.java */

class Showmoreinfo extends Dialog{
    MMainframe1 myparent;
    Label lbl1;
    CheckboxGroup cbg;
    Button ok;
    Showmoreinfo(Frame frame, String title){
        super(frame, title, false);
        setLayout(new FlowLayout());
        lbl1 = new Label("Age Group");
        add(lbl1);
        cbg = new CheckboxGroup();
        add(new Checkbox("1 - 16",cbg,true));
        add(new Checkbox("17 - 25",cbg,false));
        add(new Checkbox("26 - 99",cbg,false));
        ok = new Button("OK");
        add(ok);
        pack();
    }

    public boolean action(Event event, Object arg){
        if (event.target == ok){
```

```
            hide();
            return true;
        }
        return false;
    }
}
```

4.4.5 Specification errors

The generated Java code might not be syntactically correct if the ISL specification of the interface is not complete. This kind of error can be fixed easily when the generated code is compiled using the Java compiler. As mentioned earlier this is one of the important advantages of generating a high-level intermediate code.

For example, consider the ISL specification shown below:

```
window : main_window(layout = border, preferred_size=(200,100),
button = one, button = two, button = three, button = four,
         button = exit);
button : one(name="One", action= fun_one);
button : two(name="Two", action= fun_two);
button : three(name="Three", action= fun_three);
button : exit(name="Exit", action= fun_exit);
```

In this specification, the "main_window" contains five buttons namely "one", "two", "three", "four" and "exit", but in the description above, the button "four" is missing. This kind of missing specification can be detected when this specification file is processed by the translator. Sometimes exactly the opposite can also happen, i.e., button four might have been specified completely but its entry in the "main_window" object might be missing. The above mentioned errors and typos

can also be detected and corrected easily when this specification file is processed by the translator.

4.4.6 Advanced features

Sometimes a given application cannot be written entirely in Java and in such cases the code must be written in some other language. These special situations might arise due to the following reasons:

- A large amount of working code already exists. Providing a Java layer for that code is easier than porting it all to Java.
- An application must use system-specific features not provided by Java classes.
- The Java environment is not fast enough for time-critical applications and implementation in another language may be more efficient.

To help with these situations, Java supports *native* functions (procedures) written in some local (native) language [26]. The implemented translator completely supports this feature with the help of the attribute called *lang*. The default value for this attribute is "java." It can also take "native" as its value, if the functions are written in some other languages like C/C++. The detailed example given in Chapter 5 uses this option, since the complete application is written in Fortran.

Chapter 5

Application

This chapter gives a detailed example of using ISL to specification of an interface for the circuit simulation package **SPICE-PAC** [21]. SPICE_PAC is an interactive simulation package that is upward compatible with the popular **SPICE-2G** program. SPICE_PAC is a collection of loosely coupled modules with a well-defined interface. Hence it can be used in many different ways for different applications. Typical examples of module functions include reading a circuit description, performing circuit analysis, changing values of some circuit elements or redefining circuit parameters. The operations of the package are thus performed "on demand", as required by a particular application.

In the case of interactive simulation, it is the user who - during a simulation session - selects the order, type and parameters of analyses. The flexible structure of the package makes it possible to combine the same set of "standard" analyses with several input processors accepting different forms of circuit specification. It also allows representation of the results in different ways (binary for further processing,

textual for storing in a file and so on) [21] [22].

5.1 General organization

SPICE-PAC is organized in two major levels of routines: *main* routines and *internal* routines. The main routines constitute the "simulation interface" which includes SPICEA, SPICEB, ..., SPICEY; these main routines perform "simulation primitives", such as reading and processing circuit descriptions (SPICEA), definitions of circuit variables (SPICEB), etc. All circuit analyses (DC,TRANSIENT, AC, NOISE, etc.) are performed by the routine SPICER. Each main routine invokes a number of internal subroutines and functions, which however are "invisible" to users; users need to use only the main routines of the package.

5.2 Analyses and their parameters

This section gives a brief overview of analyses supported by SPICE-PAC, the corresponding main routines and parameters. Of all the main routines, SPICER plays a vital part, as it is called to run any analysis, with its *mode* parameter set to proper value. The parameters of SPICER include:

- *mode* : indicates the specific analysis (e.g., 1-DC Transfer Curve, 2-Transient, 3-AC, 4-Noise, 5-Distortion, 6-Fourier),
- *xtab* : an array parameter which returns independent source values for the DC Transfer Curve analysis, time values for the Transient analysis, frequencies for

the AC, Noise and Distortion analyses, harmonic frequencies for the Fourier analysis,

- *ytab* : an array parameter that returns the results of the DC Transfer Curve, Transient, AC, Noise, Distortion and Fourier analyses;
- *lr* : an integer parameter which indicates the length of the *xtab* array argument and the maximum number of rows of *ytab*,
- *lc* : an integer parameter which if positive, indicates the maximum number of columns of *ytab*, if negative, indicates the total size of *ytab*,
- *ir* : an integer parameter which returns the actual number of "used" rows in the *xtab* and *ytab* arrays, and
- *ic* : an integer parameter which returns the actual number of "used" columns in the *xtab* and *ytab* arrays.

5.2.1 DC transfer curve analysis

SPICED defines the parameters for DC analysis; the parameters include:

- an independent voltage or current source,
- initial value of the source,
- final value of the source, and
- number of steps.

SPICER performs the DC analysis, when the *mode* parameter value is 1.

5.2.2 Transient analysis

SPICET defines the parameters for TRANSIENT analysis; the parameters include:

- initial time for the TRANSIENT analysis,
- final time,
- number of steps,
- maximum step size, and
- initial condition to be used.

SPICER performs the TRANSIENT analysis, when the *mode* parameter value is 2.

5.2.3 AC analysis

SPICEF defines the frequencies for AC analysis; the parameters for this analysis include:

- starting frequency,
- ending frequency,
- number of steps, and
- indicator (logarithmic, arithmetic, etc.).

SPICER performs the AC analysis, when the *mode* parameter value is 3.

5.2.4 Noise analysis

SPICEN defines parameters for NOISE analysis. The parameters include:

- independent voltage or current source,

- output variable which defines the summing point for the equivalent output noise, and
- frequency increment for the Noise analysis.

SPICEF defines the frequencies as mentioned in AC Analysis. SPICER performs the analysis, when the *mode* parameter value is 4.

5.2.5 Distortion analysis

SPICEG defines parameters for DISTORTION analysis. The parameters include:

- output load resistor,
- ratio of distortion to nominal frequencies,
- amplitude of distortion signal frequency,
- reference power level, and
- frequency increment value for distortion analysis.

SPICEF defines the frequencies. SPICER performs the analysis, when the *mode* parameter value is 5.

5.2.6 Fourier analysis

SPICEH defines parameters for FOURIER analysis. The parameters include:

- fundamental frequency,
- number of harmonic components,
- initial time for transient analysis of one period of the fundamental frequency,

- number of steps,
- maximum step size, and
- initial condition.

SPICER performs the analysis, when the *mode* parameter value is 6.

5.2.7 Other analyses

Other analyses supported include DC TRANSFER FUNCTION, AC sensitivity analyses, DC OP-POINT and DC SENSITIVITY analysis. These analyses are not discussed in this thesis.

5.3 Organization of interactive simulator

The organization of an interactive simulator can be outlined as a three-layer structure composed of a "dialogue manager", "command interpreter" and the simulation package [21], as shown in Figure 5.1.

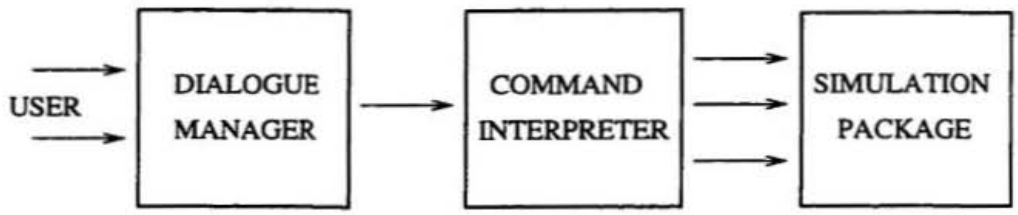

Figure 5.1: Original organization of SPICE-PAC.

"Dialogue manager" mainly organizes the interaction with the user and the "command interpreter" analyzes user-supplied commands and translates them into equivalent sequences of simulation primitives.

The graphical interface replaces the first two blocks by the GUI and so the final representation would be as shown in Figure 5.2.

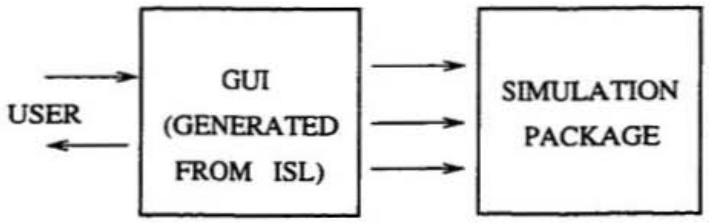

Figure 5.2: Modified organization of SPICE-PAC.

5.4 Presentation of results

SPICE-PAC uses binary representation of information for the purpose of interaction with other packages. In other words, the parameters passed to the package, as well as results returned from the package, are stored in variables and arrays defined in an external "driving" program; it is this external program that must perform all required conversions and all input/output operations. Hence, there are no "printing" or "plotting" facilities built into the package, and the required form of "output" has to be provided by the external "driving" routines. After running any analysis, the result of the analyses are stored in arrays for use by any subsequently called routines.

5.5 Specification of GUI in ISL

Appendix B gives the complete specification of the user interface for SPICE-PAC. The main window for the application is defined as:

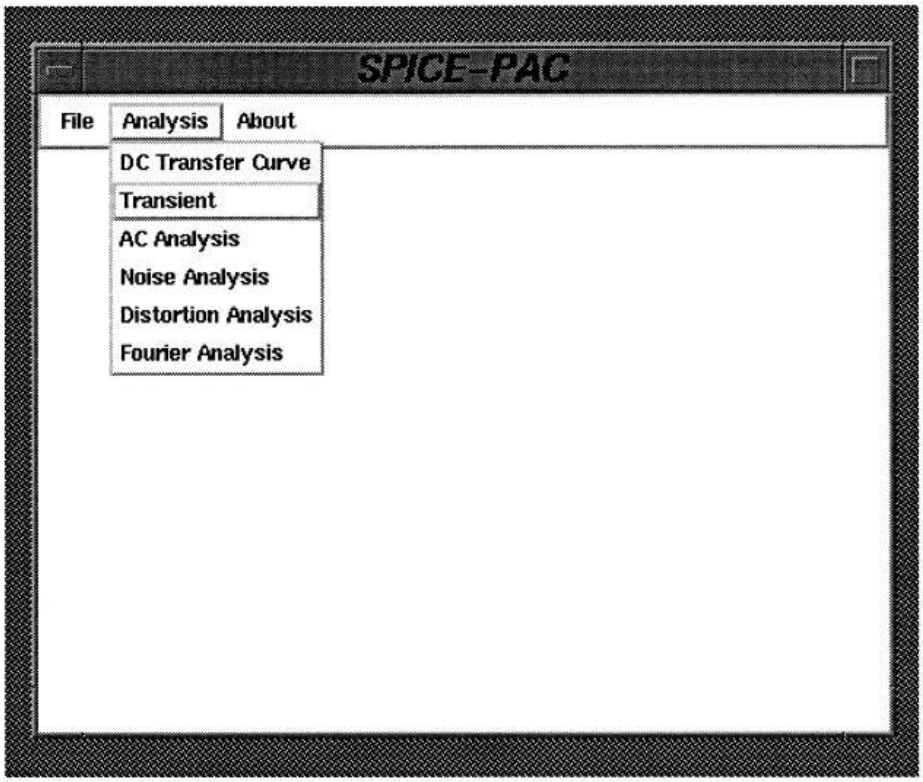

Figure 5.3: Snapshot of main window.

```
window : main_window(layout = border, menu = main_menu);
menu : main_menu(file, analysis, about);
menuitem : file (name = "File",
           submenuitem = quit, name = "Quit",action=function_quit);
menuitem : analysis(name = "Analysis",
           submenuitem = ac_analysis, name = "AC Analysis",
                                              action = showaaw,
           submenuitem = dc_analysis, name = "DC Analysis",
                                              action = showdaw,
           submenuitem = tr_analysis, name = "Transient ",
                                              action = showtaw);
menuitem : about(name = "About",
           submenuitem = help,name = "About this Application",
                                            action = show_about);
```

This specification corresponds to a window shown in Figure 5.3, with the analysis menu popped up.

When Transient analysis is selected, the dialog window corresponding to that analysis is displayed to check and possibly modify the parameters for the analysis. The corresponding specification is shown below:

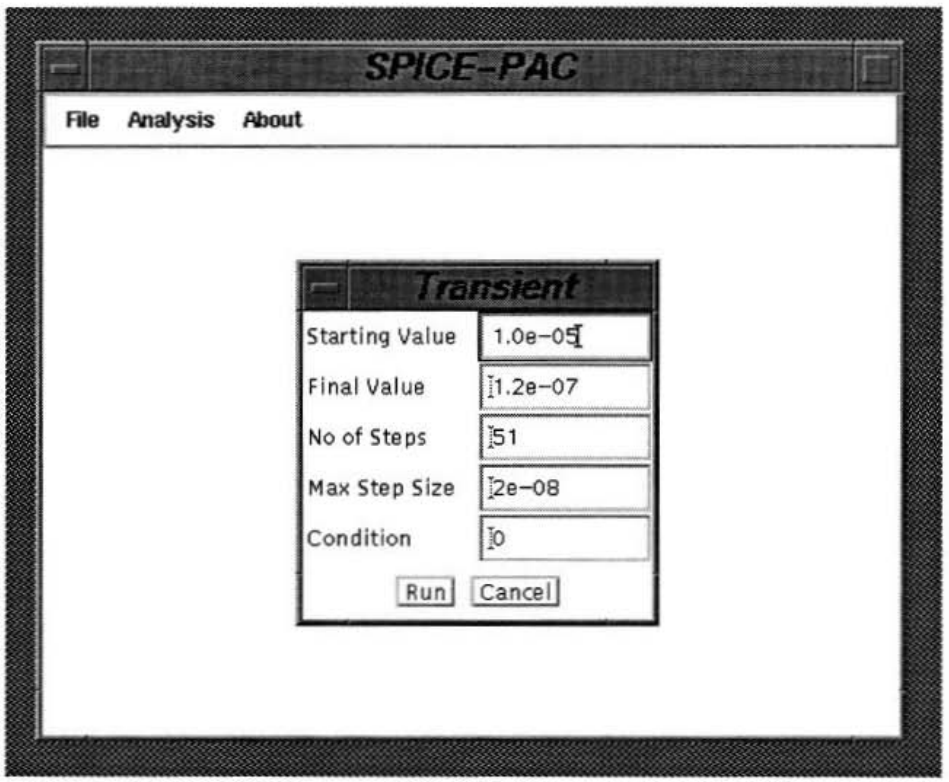

Figure 5.4: Transient dialog window.

```
subwindow : showtaw(layout = border, item = panel6, item = panel7);
panel     : panel6(location = center, layout = grid(5,2),
            item = label60, item = tf60,
            item = label61, item = tf61,
            item = label62, item = tf62,
            item = label63, item = tf63,
            item = label64, item = tf64);
```

```
label     : label60(name = "Starting Value");
```

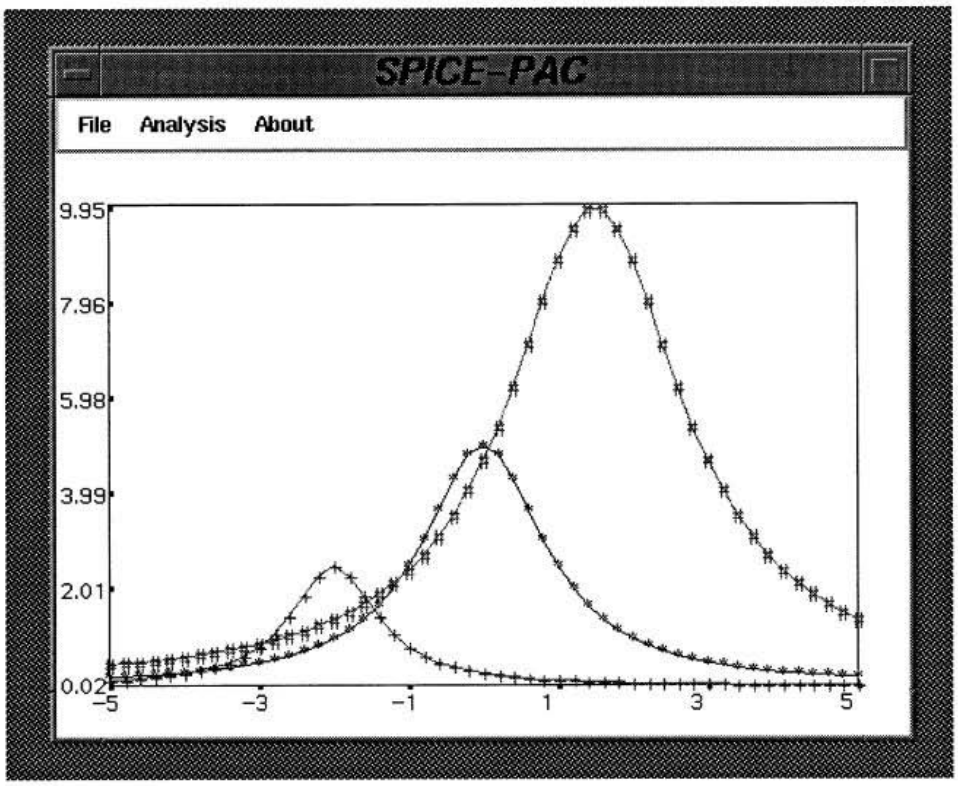

Figure 5.5: Output of transient analysis.

```
textfield : tf60(value = 1.0e-05);
label     : label61(name = "Final Value");
textfield : tf61(value = 1.2e-07);
label     : label62(name = "No of Steps");
textfield : tf62(value = 51);
label     : label63(name = "Max Step Size");
textfield : tf63(value = 2e-08);
label     : label64(name = "Condition");
textfield : tf64(value = 0);
panel     : panel7(location = south, layout = flow,
            item = run, item = cancel);
button    : run(name = "Run", action = action_tr, lang = native);
```

```
button     : cancel(name = "Cancel", action = hide, lang = java);
```

The snapshot for the above specification is shown in Figure 5.4.

Once the input parameters are set, and the "Run" button is clicked, transient analysis is executed and the output is displayed on the screen as shown in Figure 5.5.

Chapter 6

Conclusions

This thesis briefly overviews the currently available tools and techniques for the specification of Graphical User Interfaces. Then it defines a high-level specification language for graphical user interfaces, called ISL. ISL is designed to be as simple as possible, so that any user should be able to design the interface in a short period of time. A translator that converts ISL specification into a GUI is implemented in Java and the translation process is illustrated for a small, hypothetical example and a real application is discussed in greater detail.

6.1 Advantages of the proposed approach

The suggested approach has many advantages which include: the generated interface has a native "look-and-feel", it acts similarly to other user interfaces created by this approach i.e., assures consistency, and reliability of the code generated is

comparatively higher since the generation process is more standardized. The development cost in terms of time and human resources is very small which is one of the most important advantages of this approach, because writing applications that are easily movable to various computer platforms with different user interfaces is a complex task. Since the implementation is in Java, the generated code is platform neutral, and retargetting the GUI to any other platform is very simple. It also implies that maintaining and modification of such GUIs is also simple since the changes have to be made only to the specification and not to the individual copy on each platform.

ISL is also supposed to reduce the interface development effort; ISL is a higher-level specification notation, so its specifications are usually much shorter than equivalent ones in other programming or scriting languages. For example, the specification given in Appendix B is about 80 lines which resulted in generation of about 600 lines of Java code. Other indirect benefits of ISL include ease of modification. Since lines of code written is less, modifications to the existing code can be done without much effort. Moreover, modification of the code does not require any special knowledge of Java or the underlying windowing system since everything is taken care of the translator.

With respect to the potential uses of the GUI generated from ISL specification, the current implementation can be used to generate GUIs for a small or moderate size applications. The main issue in using this approach is the "integration" of the GUI with an application. If the background application has a clear interface for interaction with external applications, the proposed approach can be used seamlessly with that application. On the other hand, if the application requires a specialized

communication and synchronization protocol then the proposed approach may not be suitable.

6.2 Future research

The proposed ISL can be used to specify interface for many general applications as it supports the commonly used set of widgets. Despite the potential benefits mentioned earlier, there are still several improvements which can be made to enhance the functionality, like supporting new widgets, providing special keys features, hidden commands and macro facilities.

More work on the implementation part can be done on the manipulation of graphics. The current status, as it stands, is that when a particular analysis is run the output can be displayed as a graph on a canvas. The whole canvas can be enlarged only by changing the size of the window. Special keys or interface objects can be provided for such manipulations. More support can be provided for comparing results of a particular analysis with different input data, super-imposing one graph over the other for easier comparison, selecting a particular portion of a graph and zooming it for more details, etc.

Bibliography

[1] L. Macaulay, *Human-Computer Interaction for Software Designers*, International Thomson Computer Press, 1995.

[2] B. A. Myers and M. B. Rosson, *Survey on User Interface Programming: Human Factors in Computing Systems*, Proceeding of Human Factors in Computing Systems, pp. 195-202, 1992.

[3] B. Laurel, *The Art of Human-Computer Interface Design*, Addison-Wesley, 1990.

[4] B. A. Myers, *State of the Art in User Interface Software Tools*, H. R Hartson and D. Hix, Ed., Advances in Human-Computer Interaction, vol. 4, Ablex Publishing, 1992.

[5] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, 1987.

[6] F. A. Dix, G.J. Abowd and R. Beale, *Human Computer Interaction*, Prentice Hall International, 1993.

[7] Sun Microsystems, *SunWindows Programmers' Guide*, 2250 Gracia Ave., Mtn. View, CA 94043.

[8] M. A. Linton, J. M. Vlissides and P. R. Calder, *Composing User Interfaces with InterViews*, IEEE Computer, vol. 22, no. 2, pp. 8-22, 1989.

[9] J. McCormack and P. Asente, *An Overview of the X Toolkit*, Proceeding of User Interface Software and Technology, Banff, pp. 46-55, 1988.

[10] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1995.

[11] T. Sundsted, *Introduction to the Abstract Windowing Toolkit: A description of Java's user interface toolkit*, Javaworld, 1996.

[12] W. M. Newman, *A System for Interactive Graphical Programming*, AFIPS Spring Joint Computer Conference, pp. 47-54, 1968.

[13] R. J. K. Jacob, *A Specification Language for Direct Manipulation Interfaces*, ACM Transactions on Graphics, vol. 5, no. 4, pp. 283-317, 1986.

[14] D. R. Olsen, Jr. and E. P. Dempsey, *Syngraph: A Graphical User Interface Generator*, Proceeding of SIGGRAPH'83, Detroit, pp. 43-50, 1983.

[15] M. A. Flecchia and R.D. Bergeron, *Specifying Complex Dialogs in ALGAE*, Proceeding of Human Factors in Computing Systems, Toronto, pp. 229-234, 1987.

[16] P. J. Hayes, P. A. Szekely, and R. A. Lerner, *Design Alternatives for User Interface Management Systems Based on Experience with COUSIN*, Proceeding of Human Factors in Computing Systems, San Francisco, pp. 169-175, 1985.

[17] A. J. Schulert, G. T. Rogers, and J. A.Hamilton. *ADM-A Dialogue Manager,* Proceeding of Human Factors in Computing Systems, San Fransisco, pp. 177-183, 1985.

[18] W. Nicholas and C. Lewis, *Spreadsheet-based Interactive Graphics: from Prototype to Tool,* Proceeding of Human Factors in Computing Systems, Seattle, pp. 153-159, 1990.

[19] S. E. Hudson, *User Interface Specification Using an Enhanced Spreadsheet Model,* Technical Report GIT-GVU-93-20, Georgia University of Technology, Graphics, Visualization and Usability Center, Atlanta, Georgia, 1993.

[20] D.D.Cowan, C.M.Durance, E.Giguere and G.M.Pianosi, *CIRL/PIWI: A GUI Toolkit Supporting Retargettability,* Technical Report CS-92-28, University of Waterloo, Department of Computer Science, Waterloo, Canada, 1992.

[21] W. M. Zuberek, *SPICE-PAC version 2G6c - An Overview,* Technical Report #8903, Department of Computer Science, Memorial University of Newfoundland, St.John's, Canada, 1989.

[22] W. M. Zuberek, *SPICE-PAC version 2G6c: User's Guide,* Technical Report #8902, Department of Computer Science , Memorial University of Newfoundland, St.John's, Canada, 1989.

[23] C.A.R. Hoare, *Monitors: An Operating System Structuring Concepts and Communications,* Journal of ACM, vol. 17, no. 10, pp. 549-557, 1974.

[24] B. B. Welch, *Practical Programming in Tcl and Tk,* Prentice Hall, 1997.

[25] J. R. Levine, T. Mason and D. Brown, *lex & yacc*, O'Reily & Associates, 1992.

[26] K. Arnold and J. A. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

Appendix A

This appendix first provides a brief introduction to JavaCC, the parser generator, and then shows the specification for ISL.

Any grammar file for JavaCC starts with the settings for all the options supported by JavaCC. The two settings used here are: LOOKAHEAD and DEBUG[1].

Following the option settings is the Java compilation unit enclosed between "PARSER_BEGIN(*name*)" and "PARSER_END(*name*)". The only constraint on this compilation unit is that it must define a class called *name* - same as the arguments to PARSER_BEGIN and PARSER_END. This is the *name* that is used as the prefix for the Java files generated by the parser generator. In this example, "name" is GUI.

Then the lexical tokens are defined. They can be either simple strings (e.g., "{", "}") or a more complex regular expressions. The first token, named IGNORE_IN_BNF, is a special token. Any tokens read by the parser that match the characters defined in the IGNORE_IN_BNF token are silently discarded. Here this causes the parser to ignore space characters, tabs, and carriage return characters

[1]For more information about these options and other available options please look at the JavaCC home page at www.suntest.com/JavaCC/.

in the input file.

Subsequently the following token definitions define the characters that the grammar will interpret as special characters, GUI objects, reserved words, layout types etc.

After the lexical tokens, list of productions are defined. In JavaCC grammars, nonterminals correspond to methods. Each production defines its left-hand side nonterminal followed by a colon. This is followed by a sequence of declarations and statements within braces (here in all the cases there are no declarations and hence this appears as "{}") and then by a set of expansions (also enclosed in braces).

After the grammar is specified completely, parser can be built by running JavaCC on the input file and compiling the resulting Java files.

ISL grammar

This section gives the grammar specification for ISL, excluding the code generation part.

```
options {
LOOKAHEAD = 1;
DEBUG = true;
}
PARSER_BEGIN(GUI)
public class GUI  {
static void startUp() throws IOException {
GUI parser = new GUI();
parser.Input();
   }
}
```

```
PARSER_END(GUI)
IGNORE_IN_BNF :
{}
{
  " " | "\t" | "\n" | "\r"
| <COMMENT_A: "/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
| <COMMENT_B: "//" (~["\n","\r"])* ("\n"|"\r\n")>
}
TOKEN :   /* special chars */
{}
{ <ASSIGN: "=">
| <COMMA: ",">
| <COLON: ":">
| <SEMICOLON: ";">
| <LPARAN: "(">
| <RPARAN: ")">
| <DOUBLE_QUOTES: "\"" >
}
TOKEN : /* GUI objects */
{}
{ <WINDOW: "window">
| <SUBWINDOW: "subwindow">
| <PANEL: "panel">
| <LABEL: "label">
| <MENU: "menu">
| <MENUITEM: "menuitem">
| <CHECKBOXGROUP: "checkboxgroup">
| <TEXTFIELD: "textfield">
| <BUTTON: "button">
| <CHECKBOX: "checkbox">
| <CANVAS: "canvas">
}
TOKEN :  /* Reserved words */
{}
```

```
{ <START_IDL: "START_IDL">
| <END_IDL: "END_IDL">
| <LAYOUT: "layout">
| <SUBMENUITEM: "submenuitem">
| <PREFERRED_SIZE: "preferred_size">
| <LOCATION: "location">
| <ITEM: "item">
| <NAME: "name">
| <ACTION: "action">
| <RETURN_TYPE: "return_type">
| <VALUE: "value">
| <X_PARAM: "x_param">
| <Y_PARAM: "y_param">
| <VAR: "var">
| <TYPE: "type">
| <VAL: "val">
| <SIZE: "size">
| <LANG: "lang">
| <FUNCTION: "function">
}
TOKEN : /* layout type i.e., flow,grid,border etc */
{}
{ <FLOW: "flow">
| <GRID: "grid">
| <BORDER: "border">
}
TOKEN : /* location type */
{}
{ <NORTH: "north">
| <CENTER: "center">
| <SOUTH: "south">
| <WEST: "west">
| <EAST: "east">
}
```

```
TOKEN : /* function type */
{}
{ <_JAVA: "java">
| <_NATIVE: "native">
}
TOKEN : /* variable type */
{}
{ <INT: "int">
| <DOUBLE: "double">
| <FLOAT : "float">
}
TOKEN :  /* Identifiers */
{}
{ < Id: ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9", "_"])* > }
TOKEN : /* Number */
{}
{ <Number: (["0"-"9"])+
| (["0"-"9"])+ "." (["0"-"9"])+
| "." (["0"-"9"])+
| (["0"-"9"])+ "e" (["+","-"])? (["0"-"9"])+
| (["0"-"9"])+ (".")? (["0"-"9"])+ "e" (["+","-"])? (["0"-"9"])+ >
}
/* Space embeddable String */
TOKEN :
{}
{ <SString: "\"" ["a"-"z","A"-"Z"]
            (["a"-"z","A"-"Z","0"-"9","_", " "])* "\"" > }
void Input() :
{}
{ Start_idl() (Object() <COLON> List() <SEMICOLON>)+ End_idl()  <EOF> }
void Object() :
{}
{ <WINDOW>
| <SUBWINDOW>
```

```
| <MENU> {a.menubar_flag = true;}
| <MENUITEM>
| <PANEL>
| <LABEL>
| <CHECKBOX>
| <CHECKBOXGROUP>
| <CANVAS>
| <TEXTFIELD>
| <BUTTON>
| <VAR>
}
void Layout_type() :
{}
{ <FLOW>
| <GRID>
| <BORDER>
}
void Loc_type() :
{}
{ <NORTH>
|  <CENTER>
|  <SOUTH>
|  <WEST>
|  <EAST>
}
void List() :
{}
{ (<Id> <LPARAN> ( List_i() | List_a() ) <RPARAN> ) }
void List_i() :
{}
{ <Id> ( <COMMA> <Id> )* }
void List_a() :
{}
{ Pair() (<COMMA> Pair()  )* }
```

```
void Pair() :
{}
{ LOOKAHEAD( "preferred_size" ) <PREFERRED_SIZE> <ASSIGN> <LPARAN>
   <Number> <COMMA> <Number> <RPARAN> {a.preferred_size_flag = true;}
| LOOKAHEAD("submenuitem") <SUBMENUITEM> <ASSIGN> <Id>
| LOOKAHEAD("name" )<NAME> <ASSIGN> <SString>
| LOOKAHEAD("value") <VALUE> <ASSIGN>  (<Id>|<Number>|<SString>)
| LOOKAHEAD("layout" "=" "grid" ) <LAYOUT> <ASSIGN> <GRID> <LPARAN>
   <Number> <COMMA> <Number> <RPARAN>
| LOOKAHEAD("layout") <LAYOUT> <ASSIGN>  Layout_type()
| LOOKAHEAD("location") <LOCATION> < ASSIGN> Loc_type()
| LOOKAHEAD("menu") <MENU> <ASSIGN> Attribute_value()
| LOOKAHEAD("canvas") <CANVAS> <ASSIGN> Attribute_value()
| LOOKAHEAD("button") <BUTTON> <ASSIGN> Attribute_value()
| LOOKAHEAD("label") <LABEL> <ASSIGN> Attribute_value()
| LOOKAHEAD("panel") <PANEL> <ASSIGN> Attribute_value()
| LOOKAHEAD("checkbox") <CHECKBOX> <ASSIGN> Attribute_value()
| LOOKAHEAD("checkboxgroup") <CHECKBOXGROUP> <ASSIGN> Attribute_value()
| LOOKAHEAD("textfield") <TEXTFIELD> <ASSIGN> Attribute_value()
| LOOKAHEAD("(") <LPARAN> Pair() <RPARAN>
| LOOKAHEAD("return_type") <RETURN_TYPE> <ASSIGN> Var_type()
| LOOKAHEAD("lang") <LANG> <ASSIGN> Lang_type()
| LOOKAHEAD("type") <TYPE> <ASSIGN> Var_type()
| LOOKAHEAD("size") <SIZE> <ASSIGN> <Number>
| LOOKAHEAD("val") <VAL> <ASSIGN> (<Id>|<Number>)
| LOOKAHEAD("function") <FUNCTION> <ASSIGN> <SString>
| Attribute_name() <ASSIGN> Attribute_value()
}
void Lang_type() :
{}
{ <_JAVA>
| <_NATIVE>
}
void Var_type() :
```

```
{}
{ <INT>
| <FLOAT>
| <DOUBLE>
}
void Attribute_name() :
{}
{ <LAYOUT>
| <LOCATION>
| <SUBMENUITEM>
| <ITEM>
| <NAME>
| <ACTION>
| <X_PARAM>
| <Y_PARAM>
}
void Attribute_value() :
{}
{ <Id> }
void Start_idl() : {
try { startUp() ; } catch (IOException e) { }
}
{ <START_IDL> }
void End_idl() : {
try { closeDown(); } catch (IOException e) { }
}
{ <END_IDL> }
```

Appendix B

The following is the complete specification of GUI in ISL, as used in Section 5.

```
START_ISL
window      : main_window(layout = border, menu = main_menu,
              canvas = graph_canvas);
menu        : main_menu(file, analysis, about);
menuitem    : file (name = "File",
              submenuitem = quit, name = "Quit",
              action = function_quit);
menuitem    : analysis(name = "Analysis",
              submenuitem = ac_analysis, name= "AC Analysis",
                                         action = showaaw,
              submenuitem = dc_analysis, name= "DC Analysis",
                                         action = showdaw,
              submenuitem = tr_analysis, name= "Transient",
                                         action = showtaw,
              submenuitem = n_analysis, name = "Noise Analysis",
                                        action = shownaw,
              submenuitem = dis_analysis, name = "Distortion Analysis",
                                          action = showdisaw,
              submenuitem = f_analysis, name = "Fourier Analysis",
                                        action = showfaw);
menuitem    : about(name = "About", submenuitem = help,
              name = "About this Application", action = show_about);
```

```
subwindow     : showaaw(layout = border, item =panel0,
                item = panel1, item = panel2);
panel         : panel0(location = north, layout = flow,
                item = label00,  item = cbg);
label         : label00(name = "Source Type");
checkboxgroup: cbg(item = lin, name="Lin", value = false,
                item = oct, name = "Oct", value = false,
                item = dec, name = "Dec", value = true);
panel         : panel1(location = center, layout = flow,
                item = label10, item = start_freq_tf, item = label11,
                item = end_freq_tf);
label         : label10(name = "Start Frequency");
label         : label11(name = "End Frequency");
textfield     : start_freq_tf(value = 100);
textfield     : end_freq_tf(value = 1000);
panel         : panel2(location = south, layout = flow, item = cancel,
                item = run);
button        : cancel(name = "Cancel", action = hide);
button        : run ( name = "Run", action = action_ac, lang= native);
subwindow     : showdaw(layout = border,item = panel3,item = panel4,
                item = panel5);
panel         : panel3(location = north, layout = flow, item = label30,
                item = cbg1);
label         : label30(name = "Source Type");
checkboxgroup: cbg1(item = soucre1, name = "Source1", value = true,
                item = source2, name = "Source2", value = false);
panel         : panel4(location = center,layout = grid(0,1),
                item = label40, item = source_name,
                item = label41, item = source_value,
                item = label42, item = end_value,
                item = label43, item = increment_value);
label         : label40(name = "Source Name");
textfield     : source_name(value = Vin);
label         : label41(name = "Start Value");
```

```
textfield    : start_value(value = 1);
label        : label42(name = "End Value");
textfield    : end_value(value = 5);
label        : label43(name = "Increment Value");
textfield    : increment_value(value = 0.01);
panel        : panel5(location = south,layout = flow, item= cancel,
               item = run);
button       : cancel(name = "Cancel", action = hide_dc);
button       : run(name = "Run DC Analysis", action = action_dc,
               lang = native);
subwindow    : showtaw(layout = border, item = panel6,
               item = panel7);
panel        : panel6(location = center, layout = grid(5,2),
               item = label60, item = tf60, item = label61,
               item = tf61, item = label62, item = tf62,
               item = label63, item = tf63, item = label64,
               item = tf64);
label        : label60(name = "Starting Value");
textfield    : tf60(value = 1.0e-05);
label        : label61(name = "Final Value");
textfield    : tf61(value = 1.2e-07);
label        : label62(name = "No of Steps");
textfield    : tf62(value = 51);
label        : label63(name = "Max Step Size");
textfield    : tf63(value = 2e-08);
label        : label64(name = "Condition");
textfield    : tf64(value = 0);
panel        : panel7(location = south, layout = flow,
               item = run, item = cancel);
button       : run(name = "Run", action = action_tr, lang = native);
button       : cancel(name = "Cancel", action = hide, lang = java);
subwindow    : show_about( layout = flow, item = panel8);
panel        : panel8(location = center,layout = flow, label = myl,
               button= ok);
```

```
label       : myl(name = "SPICE PAC Beta version");
button      : ok(name = "Ok", action = hide);
canvas      : graph_canvas(location = center, name = "GraphCanvas",
              x_param = time, y_param = V1);
END_ISL
```

Appendix C

This appendix gives the complete code for the customized **canvas** object. Though this is fine tuned for SPICE-PAC application, it can be used for other general applications.

The most important part of the implemented Java class is the *paint()* method. This method is called by Java whenever the application (canvas) needs to be painted - when the canvas is initially drawn, when the window containing it is moved, or when another window is moved from over it.

The other methods like setValues, getMax, getMin etc., are used initilize the internal data structures and for other internal data manipulations. Here in this example most of the variables (especially arrays) are fine tuned for SPICE-PAC application.

```
/* Customized  CANVAS object */
import java.awt.*;
import java.lang.Math;
public class  MyCanvas extends Canvas{
   static int count = 0;
   int k=0;
   double ta[] = new double[200];
```

```
    double tb[] = new double[1000];
    double tb1[] = new double[200];
    double tb2[] = new double[200];
    double tb3[] = new double[200];
    int nr=0,nc=0,ir=0,ic=0;
    int nos =0;
    double fv =0, sv = 0;
    int h,w,h1,w1;
    double bta,sta, btb1,stb1, btb2,stb2, btb3,stb3;
    double btb,stb;
    int basex, basey;
    int incrx, incry;
    int NOD; //No.Of.Divisions on x and y axis
    public MyCanvas(int pk,double pta[],double ptb[],double ptb1[],
double ptb2[],double   ptb3[],int pnr,int pnc,int pir,int pic,
double psv, double pfv, int pnos){
    ta = pta;
    tb = ptb;
    nr = pnr;
    nc = pnc;
    ir = pir;
    ic = pic;
    sv = psv;
    fv = pfv;
    nos = pnos;
    }
    public void setValues(int pk,double pta[], double ptb[],
double ptb1[], double ptb2[],double ptb3[], int pnr,int pnc,
int pir, int pic, double psv, double pfv, int pnos){
       k = pk ;
```

```
    ta = pta;
    tb = ptb;
    nr = pnr;
    nc = pnc;
    ir = pir;
    ic = pic;
    sv = psv;
    fv = pfv;
    nos = pnos;
}
/* format the values to be displayed on x and y axes */
public String fmt(Double d, int minWidth){
    String tmp = d.toString();
    if (d.doubleValue() < 0) minWidth++;
    int tmplen = tmp.length();
    if (minWidth < tmplen) tmp = tmp.substring(0,minWidth);
    return tmp;
}
public void paint(Graphics g){
    w = size().width;
    h = size().height;
    w1 = w - 60; //width of the rectangle drawn
    h1 = h - 60; //height of the rectangle drawn
    double tmp = 0;
    int tlx = 30; //top_left_x
    int tly = 30; //top_left_y
    basex = tlx;
    basey = tly;
    int incr = 100;
    double x1,x2;
```

```
double y1,y2;
int int_x1, int_x2;
int int_y1, int_y2;
Color c1 = new Color(0,0,255);
Color c2 = new Color(200,55,100);
Color c3 = new Color(255,0,0);
for (int i = 0; i < ir; i++) {
   for (int j = 0; j < ic; j++){
      if( j == 0) tb1[i] = tb[i+j*nr];
      else if( j == 1) tb2[i] = tb[i+j*nr];
      else if( j == 2) tb3[i] = tb[i+j*nr];
   }
}
bta = getMax(ta);
sta = getMin(ta);
btb1 = getMax(tb1);
stb1 = getMin(tb1);
btb2 = getMax(tb2);
stb2 = getMin(tb2);
btb3 = getMax(tb3);
stb3 = getMin(tb3);
btb = btb1;
else if (btb2 > btb) btb = btb2;
else if (btb3 > btb) btb = btb3;
stb = stb1;
else if (stb2 > stb) stb = stb2;
else if (stb3 > stb) stb = stb3;
g.clearRect(0, 0, w , h);
g.drawRect(tlx, tly, w1 , h1 );
NOD=5; //No. Of Divisions
```

```
 /* draw the markings on the horizontal scale*/
 incrx = (w1)/NOD;
 double xunit =(bta - sta)/NOD;
 double xval = sta;
 for (int i = tlx; i <= w1+tlx; i= i+ incrx){
    StringBuffer x = new StringBuffer();
    g.fillRect(i, (h1+tlx), 3,3);
    Double dval = new Double(xval);
    g.drawString(fmt(dval,8), i-10, (tlx+h1+15));
    xval = xval+xunit;
 }
 /* draw the markings on the vertical scale */
 incry=(int)(h1/NOD);
 double yunit = (btb - stb)/NOD;
 double yval = stb;
 for (int i = h1+tly; i >= tly ; i= i- incry){
    StringBuffer y = new StringBuffer();
    g.fillRect(tlx, i-2, 3,3);
    Double dval = new Double(yval);
    g.drawString(fmt(dval,4), 2, i+5);
    yval = yval + yunit;
}
for(int i = 0; i < ir; i++){
   x1 = ta[i];
   if (i+1 < ir) x2 = ta[i+1];
   else x2 = x1;
   y1 = tb1[i];
   if (i+1 < ir ) y2 = tb1[i+1];
   else y2 = y1;
   g.setColor(c1);
```

```
        int_x1 = (int)Math.rint(transx(x1));
        int_x2 = (int)Math.rint(transx(x2));
        int_y1 = (int)Math.rint(transy(y1));
        int_y2 = (int)Math.rint(transy(y2));
        g.drawLine(int_x1,int_y1,int_x2,int_y2);
        g.drawString("*",int_x1-3,int_y1+6);
        StringBuffer sb = new StringBuffer();
        sb.append(i);
        y1 = tb2[i];
        if (i+1 < ir ) y2 = tb2[i+1];
        else y2 = y1;
        int_y1 = (int)Math.rint(transy(y1));
        int_y2 = (int)Math.rint(transy(y2));
        g.setColor(c2);
        g.drawLine(int_x1,int_y1,int_x2,int_y2);
        g.drawString("#",int_x1-3,int_y1+6);
        y1 = tb3[i];
        if (i+1 < ir) y2 = tb3[i+1];
           else y2 = y1;
           int_y1 = (int)Math.rint(transy(y1));
           int_y2 = (int)Math.rint(transy(y2));
           g.setColor(c3);
           g.drawLine(int_x1,int_y1,int_x2,int_y2);
           g.drawString("+",int_x1-3,int_y1+6);
       }
 }
public double getMax(double ary[]){
   double max =  ary[0];
   for(int i = 0; i < ir; i++){
      if (ary[i] > max) max = ary[i];
```

```
        }
        return max;
    }
    public double getMin(double ary[]){
        double min =  ary[0];
        for(int i = 0; i < ir; i++){
            if (ary[i] < min ) min = ary[i];
        }
        return min;
    }
    double transx(double rawx){
        double ri; //Range Index
        double xrange = bta - sta;
        ri = (rawx - sta)/(xrange/NOD);
        rawx = basex+(ri* incrx);
        return rawx;
    }
    double transy(double rawy){
        double ri; //Range Index
        double yrange = btb - stb;
        ri = (rawy - stb)/(yrange/NOD);
        rawy = ri * incry;
        /* IMPORTANT: convert from canvas to realtime axis */
        rawy = h1+basey - rawy;
        return rawy;
    }
}
```

Appendix D

This appendix provides information related to the use of ISL and the GUI generator. These instructions are valid at the time of writing.

Installation instructions

Installation instructions assume that the target machine is running a recent release of the X Window System (Version 11) on a Unix operating system. Java version 1.0.1 or later and JavaCC version 0.5 or later must also have been installed on the target machine. No special permission is required to use the package.

Extracting the archive file

This distribution includes a single compressed archive file called guigenerator.tar.gz which can be extracted using the command:

$ gzip -dc guigenerator.tar.gz — tar xvf -

This will create a directory called guigenerator and all the relevant files will be placed in that directory. These file and subdirectories are described in Table 1.

Table 1: Files included in the distribution.

Filename	Description
README	A text file containing information on how to use the translator
MyCanvas.java	Java file containing information required to use *canvas* object
GUI.java & GUIa.java	The translator (i.e., scanner, parser and code generator)
Makefile	File to build the translator

Compilation

To generate the GUI after writing the ISL , change to the directory guigenerator and type *make*. If there were no errors during the compiling and linking, required Java classes will be created in the current directory which can then be executed directly using the *java runtime*.

Environment variables and other information

LD_LIBRARY_PATH must be set to the present working directory.

PATH should include the current directory.