A Graphical User Interface for Formal Specifications

A Master’s paper in

Computer Science

By

Xiaowen Chen

©1999 Xiaowen Chen

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

February 1999
Abstract

In software development, specification errors can cause waste of resources, budget overrun or complete failure of the development. In an effort to reduce such risk in software development, formal and model-based specification languages have been developed and applied to software development. One such specification language is SPECS-C++, which is used to specify C++ classes. A large subset of SPECS-C++ can be executed by translating specifications to constraint programs. In this work, a graphical user interface to the execution technique was developed using the TCL/TK programming language. This graphical user interface provides users a convenient tool for constructing test cases, modifying specifications, and validating original or modified specifications. In this sense, this work enhances the execution technique. The current work can be easily extended to accommodate more complex test cases and specifications.
TABLE OF CONTENTS

ABSTRACT ................................................................. i
TABLE OF CONTENTS .................................................. ii
LIST OF FIGURES ......................................................... ii
1. INTRODUCTION ......................................................... 1
2. TCL/TK APPLICATION .............................................. 6
3. FEATURES OF THE INTERFACE ................................. 7
4. POSSIBLE APPROACHES IN THE INTERFACE DEVELOPMENT ........ 14
5. CONCLUSION .......................................................... 15
REFERENCES ............................................................... 17
APPENDIX: TCL/TK CODE FOR THE INTERFACE .................. 20

LIST OF FIGURES

Figure 1 The execution of SPECS-C++ specifications. ....................... 5
Figure 2 The interface for the file editor. ...................................... 8
Figure 3 The list of all functions for a selected file. ............................. 9
Figure 4 How results are displayed in the interface. .......................... 11
Figure 5 The updated values of list \( l \). ........................................ 12
Figure 6 The return value after function \( \text{max()} \) is executed. ............ 12
Figure 7 The results after function \( \text{sort()} \) is executed. .................... 13
1. Introduction

It is a well known fact that the capabilities of software have dramatically increased in the past decade. To facilitate more flexibility and more functionality, the software itself is becoming more and more complicated, and as a result, the specification for a software system is more and more likely to contain errors. Moreover, it is difficult to detect all such errors before the software system is implemented. On the other hand, in a software development process, the software is developed according to the software specification, no matter whether it is right or wrong. Therefore, undetected specification errors can have disastrous or even fatal consequences for the whole software development process.

In an effort to reduce such risk in the software specification, many researchers have been trying to develop precise, formal, model-based, and executable specification languages that can be used for the specification of software systems. Such specification languages include VDM [Jon90] [And93], Z [Spi89] [Spi92] [Hay93], Larch [GHG+93] [Lea97], and SPECS-C++ [Bak91] [Col91] [WBL94] [Hav94] and [Gur98].

It should be emphasized that all of the aforementioned languages were originally developed as non-executable specification languages. It is highly desirable that a subset of a non-executable formal specification language can be translated into an executable programming language. Such an executable (in an indirect sense, since it has to be translated first) subset of specification language will make developing and using specifications much easier and more efficient. The critical step is how to translate a formal specification language into some type of executable programming language.
Such translation may have to be done simply by hand, which of course, is of low efficiency and error-prone. Unfortunately, most of the translation work for VDM and Z specifications is currently accomplished by hand [Wal95] [Laz89a] [Laz89b]. In cases where an automated translation exists, only a small subset of the specification language can be translated [Nor90] [O'N89] [O'N92a] [O'N92b]. The technique used for SPECS-C++ can execute a much larger subset of the specification language than the other techniques mentioned. Additionally, this technique could easily be applied to VDM or Z.

The following first describes SPECS-C++, and then the technique used to execute SPECS-C++ specifications. SPECS-C++ is a formal and model-based specification language which is based on the specification language SPECS [Bak91] [Col91] and is used to specify the interfaces of C++ classes. Simple types such as integer, real, character, enumerated, string, and boolean as well as complex types such as tuple, set, sequence, and object can be defined and used in SPECS-C++. For example, object types can be used in SPECS-C++ to model aliasing and mutation at the specification level.

Implementation functions and procedures in SPECS-C++ are specified using a pre-condition (which specifies what must be true for these operations to execute correctly when they are called) and a post-condition (which specifies what is guaranteed to be true when these operations terminate, assuming the pre-condition was satisfied). The pre- and post-conditions are written as first-order predicate logic assertions over the model types.

Model-based specifications can specify procedures (operations that modify their
parameters). When the value of a parameter object (e.g., $v$) changes from the **pre-state** (the system state before the procedure is called) to the **post-state** (the system state after the procedure is called), the pre-state value is decorated with a "^", i.e., $v^\uparrow$, and the post-state value is decorated with a "'", i.e., $v'$.  

A typical example of a SPECS-C++ specification [WLB99] is given below, in which a class List is defined. Although the SPECS-C++ language appears to be very similar to the C++ language, a significant characteristic of the SPECS-C++ language is that the logical assertions used to specify a function are enclosed by a prefix "/*" and a suffix "*/". This syntax in the SPECS-C++ language is exactly the same as the syntax of comments in the C++ language. The purpose for imposing such a syntax in the SPECS-C++ language is to allow the specification and the actual class declaration to appear in the file, without causing any possible confusion. The specification file can be used as a C++ header file.

class List {
    /* model */
    domains
        sequence of int listtype;
    data members
        listtype tl;
    abstract functions
        define sorted(listtype l) as bool such that
            result = (\forall int i [ 1 <= i < length(l) =>
            l[i] <= l[i + 1] ]); 
        define smax(listtype l) as int such that
            \exists int i [ i \in domain(l)
            \forall \forall int j [ j \in domain(l) => l[i] >= l[j] ]
            result = i ]; 
    /* */
    public:
        List();
        /* modifies: self */
        post: tl' = <> */
void insert(int i);
/* modifies: self
   post: tl' = <i> || tl^ */

void sort();
/* modifies: self
   post: range(tl^) = range(tl') \/\ sorted(tl') */

int max();
/* post: result = tl^[smax(tl^)] */

A SPECS-C++ specification is composed of two major parts: the model, and the operations. The model in a SPECS-C++ specification is built upon the simple and the complex types described previously and generally has four sections: the domains section, the data members section, the constraint section and the abstract functions section.

In the example given above, the domain section defines a type of sequence of integer; the data members section declares an abstract data member \( tl \) with the same type defined in the domains section; there is no constraints section for this particular example; and the abstract functions section contains two functions, function \( sorted \) to test whether a sequences of integers is sorted, and function \( smax \) to find the index of the largest integer in an integer list.

A SPECS-C++ specifications is executed by first translating it into Agents Kernel Language (AKL) [JH94], and then the resulting AKL program is executed. AKL is based on concurrent constraint programming and works well as a translation target. Features of AKL include:

- a simple syntax
- clean support for constraint programming
• efficient propagation of constraints
• backtracking for finding alternative solutions

The procedure for the execution of SPECS-C++ can be summarized as follows [WLB99]:

a. the SPECS-C++ specification and code for testing the specification are translated into a syntax tree by using standard compiler techniques
b. an AKL program is generated from the syntax tree
c. the AKL program and a library of AKL agents that implement the built-in operators of SPECS-C++ and some addition functionality are loaded, and the program is executed

![Figure 1. The execution of SPECS-C++ specifications.](image)

In this work, we developed a general graphical user interface for the C++ class specification validation system. The class validation system was developed using the SPECS-C++ specification language [WLB99], and the interface was developed using the TCL/TK scripting language [Ous94] [Wel97]. The purpose of this work is to develop a user friendly graphical interface that is concise, convenient, and compact.
Through this graphical interface, a user can quickly get useful insight into the specification, construct test cases for system, and display the test results immediately. Figure 1 shows the execution of SPECS-C++ specifications.

2. TCL/TK Application

TCL/TK is a programming system that has very useful graphical interface facilities. TCL is the basic programming language, and TK is a tool kit of widgets (similar to other graphical user interface tool kits such as Xlib, Xview and Motif).

TCL is a string-based language, as it was originally developed to be a reusable command language. The language has only a small number of fundamental constructs and relatively little syntax. As a result, it is relatively simple and easy to learn and to use. Features of TCL include:

- a number of built-in interactive tools
- syntax for grouping and substitution which can be used with programming variables and nested calls
- extended functionality which can be implemented using the C language and integrated into standard TCL

As a tool kit, TK provides a set of TCL commands that create and manipulate widgets. A widget is a window in a graphical user interface that has a particular appearance and behavior. Types of widgets include buttons, scrollbars, menus, text windows, listboxes, labels, entries, and canvases.

The main TCL/TK program is called wish, which stands for windowing shell. With
wish, graphical applications can be created and implemented. TCL scripts can be written to be portable among UNIX, Windows, and the Macintosh; however, the details about getting started are different on these systems. For example, on UNIX, the main TCL/TK program is wish; on Windows, it is wish.exe; and on Macintosh, it is Wish.

In this work, we used TCL/TK to develop our graphical user interface, because it is powerful, simple, and relatively easy to apply.

3. Features of the Interface

Generally speaking, for software development, a comprehensive, clear, and practical specification has to be given before an efficient procedure can actually be implemented. However, for our development of a graphical interface for the specification of C++ classes (SPECS-C++), several obstacles prevented us from giving unambiguous procedures for this development at the beginning. Although we had general ideas for the capabilities of our final product, it was hard to imagine all the advantages and the disadvantages for different approaches before implementing and testing at least some of the possible approaches. As a result, we accomplished our development under a general guideline without constraining ourselves to any fixed, pre-arranged procedures.

There are several aspects of our graphical user interface development. For the file list, we use a pull down menu to manage all header files with the filename extension .h, so that a user can easily locate the header files that are needed, and focus his attention on these header files. By managing header files in this manner, users are unlikely to be
distracted by irrelevant information.

Figure 2. The interface for the file editor.

Once a user selects a file, the file editor can be activated right away. Under such a file editor, a user can do most tasks that he expects to be able to do under a popular commercial editor, such as Microsoft Word. A user can easily modify the specification file using this editor. The interface for the file editor is given in Figure 2.

Another feature in this interface is that all functions in a header file are automatically extracted and listed in a separate box. Obviously, this feature can help a
user to get a quick summary of the functionality from a class. Furthermore, if a user changes or adds a function to a header file, he can easily get the updated list for all functions in a class by simply using the built-in command *Save and Reload* in the editor. This command also rechecks and recompiles the specification so that the new version is ready to run. Figure 3 shows the list of all functions of an example specification.

![Figure 3](image)

**Figure 3.** The list of all functions for a selected file.

A user can pick any function from the list that is automatically summarized from a file by simply double clicking the function name. This feature is helpful for
understanding the specification of the class. The interface will then list the class name that is used for such a function, and a user can declare an object accordingly. Also a user needs to declare variable names that are necessary for calling such a function and assign values to these variables. The interface will provide default values for these variables, and a user can change such default values by adding or removing values of variables from this default list. This will give a user the flexibility to build a customized default value list to meet his own needs. Such a default value list will be saved into a file with the same file name with the header file and a file name extension .dat. Thereafter, each time the interface is activated and a header file is selected, the corresponding default value list file is automatically loaded. Figure 4 illustrates how the results are displayed in the interface.

Moreover, a user can actually run the function by giving the required input, with the results shown beside all information regarding the specification file. If a user accidentally specifies a wrong input, there will be an error message shown when the function is executed. Such an error message can provide detailed information on the error type. When a function is executed, all information such as variable values and function calls is converted to AKL before being written (via a pipeline) to the AKL interpreter, and the running results are sent back through the same pipeline. Each time the selected function is executed, the updated values of the input objects are shown sequentially in the Result box, where a blank line is added between results obtained from two successive runs for better visibility. This provides a log for the whole running history, so that a user can have a clear summary for all cases that have already been
tested. In fact, a user can even clear the result box or copy selected results to the default value list for further testing. More importantly, the results of a step can be used as the input for the next step by simply checking the button *Update variable Value*. When this button is checked, variables in the *variable value* window are automatically updated to reflect the result of running an operation. If the function returns a value, for example, the maximum element value of a list, such a value will be shown in the *Return Value* box.

![Figure 4. How results are displayed in the interface.](image)

For example, a user can use the class "List" to define a list. First of all, a user needs
Figure 5. The updated values of list $l$.

Figure 6. The return value after function max() is executed.
to declare a list, and then elements can be added to the list. In Figure 5, a user defines a list \( l \), then add four elements 3, 1, 4 and 2 to the list \( l \). The results are displayed sequentially in the Result box as \( l=(<3>) \), \( l=(<1, 3>) \), \( l=(<4, 1, 3>) \), \( l=(<2, 4, 1, 3>) \).

After a list is constructed, some more advanced functions can be incorporated as well. In this example, when we execute the function max(), this function returns the maximum element value in the list \( l \). As shown in Figure 6, the updated input object list \( l \) (which did not change after the execution) is displayed in the Result box. Meanwhile, the returned maximum element value of the list \( l \) is displayed in the Return Value box.

Figure 7. The results after function sort() is executed.
Finally, we give an input object, a list \( l = (3, 2, 4, 1) \), and select the function \texttt{sort()}\). After this function is executed on \( l \), we have the updated list \( l = (1, 2, 3, 4) \), which is shown in the \textit{Result} box in Figure 7.

4. Possible Approaches in the Interface Development

There are other possible approaches that may be applied to each stage of our interface development. In the process of this interface development, some of these options were tested and compared. Our goal in this work was to build an effective, dependable, and user-friendly interface.

For example, to list all file names and functions, we can use a list box or a pull down menu. If a list box is used, it stays on the screen even after a selection is made. If a pull down menu is used, it disappears after a selection is made. We encourage a user to focus his attention on functions in the file he selects. In order to complement this motive, we used a pull down menu to list file names, so a user would not be distracted by a long file list all the time. On the other hand, we wanted to give a user a clear summary of all functions in a file as well as an easy access to explore all the functions. Therefore, a list box was used to list all functions in a file.

To declare variables in a function, we originally considered using the entry widget and letting the program count the number of variables needed. Thereafter, the number of lines and the number of class names will be displayed. But after a further investigation, we found that such an idea would impose severe constraints on the interface, since there might be some additional variables that could not be counted by
the program. By removing this constraint and using the text widget, a user can declare as many variables as the program requires.

5. Conclusion

In this work, we developed a graphical user interface using the TCL/TK programming system. This interface provides a convenient tool for the SPECS-C++ specification validation system. Users can use this graphical interface to construct test cases, run the specification, and display the results. Users can also use this interface to modify the specification files and validate these modified results immediately. In this sense, the graphical user interface developed in this work enhances the execution technique and makes the validation process itself much easier.

There is some potential improvement that can be achieved in the future. For example, the current interface deals with a single result and single returned value. The result is displayed in Result box and the returned value is shown in the Return Value box. This is not a problem for a deterministic specification. However, for a nondeterministic specification, multiple values (states) may result; or executing such a specification may cause the execution technique to go into an infinite loop. As a result, some modifications on the user interface should be made in order to deal with such situations. A simple and straightforward remedy for this may be to add a control which gives a user the flexibility to choose single or multiple results, and add some type of separators, such as semicolon, between two returned values.

Another possible future enhancement involves nested objects. If an input object is
nested, i.e., one object contains other objects, such as $\text{OBJ}_1(\text{OBJ}_{11}, \text{OBJ}_{12}, \cdots, \text{OBJ}_{n_1})$, there should be way to display such objects clearly. For example, in displaying the results, the $\text{OBJ}_1$ can be followed by all its composing objects $\text{OBJ}_{11}, \text{OBJ}_{12}, \cdots$, and $\text{OBJ}_{n_1}$ below it with certain left indentation. In principle, there should be no obstacle to prevent such an enhancement of the current graphical user interface.

The graphical user interface developed in this work passed various types of tests we have implemented and has proved to be dependable. The interface makes developing and validating formal specifications much faster and easier, and so helps in overcoming some of the limitation of formal specification. This tool can be easily extended to facilitate an enhanced specification validation system that is under development. Thus, this work offers an improvement over current software engineering techniques and can serve as a foundation for future research.
References:


#!/usr/local/bin/wish
# this first section of the script should be executed once (when the
# program is first executed)

proc newer { file1 file2 } {  
# is file1 newer then file2?  
if ![file exists $file2] {  
    return 1  
} else {  
    # Assume file1 exists  
    expr [file mtime $file1] > [file mtime $file2]  
}
}

proc discard { file pat } {  
# throw away text from file until text contains pat  
set ans ""  
while {![string match $pat $ans]} {gets $file ans}
}

proc isdigit {d} {  
    if [string match \[0-9\] $d] {set r 1} else {set r 0}  
    return $r
}

proc replacewild {str} {  
# replace all unbound variables in the string with []  
set result ""  
set oldind 0  
set i 0  
while {$i < [string length $str]} {  
    if [string match _ [string index $str $i]] {  
        append result [string range $str $oldind [expr $i - 1]] "\[\]"  
        incr i  
        while {[isdigit [string index $str $i]]} (incr i)  
        set oldind $i  
    } else {incr i}  
    append result [string range $str $oldind end]  
    return $result
}

# create temp directory  
set id [pid]  
set tmpdir /tmp/toakl/
catch {exec mkdir $tmpdir}
catch {exec chmod 777 $tmpdir}
set tmpdir $tmpdir/$id
catch {exec mkdir $tmpdir}

# open a pipe into an AKL process
if [catch {open | agents r+} pipe] {
    destroy .f
    errordialog "agents executable not installed or not in path"
    exit
}
discard $pipe "All *"

# load the library
puts $pipe "load(speclib). "
flush $pipe
discard $pipe "yes*"

# set up the main window name and variables
wm title . "SPECS-C++"
set senddata empty
set updatevalue ""
set filena 0
set number 0
set filenamel ""
set filename ""

# create a frame to hold the menubar
frame .mbar -relief raised -borderwidth 1
pack .mbar -fill x

# create file menu button and exit button, then place them inside the
# window
menubutton .mbar.file -text "Specification File" -menu .mbar.file.m -
padx 2
menu .mbar.file.m -tearoff true
button .mbar.exit -text "Exit" -command {
    catch {close $pipe}
    catch {exec rm -rf $tmpdir}
    exit
}
pack .mbar.file -side left
pack .mbar.exit -side right

# add commands to the file menu
foreach i [lsort [glob *.h]] {
    .mbar.file.m add command -label "$i" -command {
        set filena [.mbar.file.m entrycget active -label]
do1
        do2
    }
}

# load default values into main window, open file editor and load file
# into the file editor
proc do1 {} {
    global x1 y1 filenamel filena filename number pipe fname tmpdir
    set a [string range $filena 0 [expr [string length $filena]-3]]
    set filename [format "%s.dat" $a]
    set filenamel [format "%s.txt" $a]
    destroy .edit
    .b.value delete 1.0 end
    editpro
    .b.declare delete 1.0 end
    loaddata .b.defaultvalue $filename
    OpenFile .b.declare $filenamel
    .c.label4 configure -text ""
    .c.function.entry5 configure -text ""
    .c.function.label5 configure -text ""
    set number1 $number
    while {$number1>0} {
        destroy .c.frame.left.entry$number1
        destroy .c.frame.right.label$number1
        incr number1 -1
    }
}

# collect the data sent to pipe line, run the function
proc do2 {} {
    global x1 y1 filenamel filena filename number pipe fname tmpdir
    loadoperation
    .d.run configure -state disabled
    destroy .f

    # assume fname is the name of the file (with no .h extension)
    set fname [string range $filena 0 [expr [string length $filena] -3 ]]

    # compile the specification if necessary (.h -> .akl)
    set err 0
    if [newer $fname.h $fname.akl] {
        set err [catch {exec toakl $fname.h $fname.akl >& $tmpdir/_errs}]
        if $err {
            set errfile [open $tmpdir/_errs r]
            set ans [read $errfile]
            if {$ans == ""} {
                set ans "toakl executable not installed properly"
            }
            destroy .f
            errordialog $ans
            close $errfile
        }
    }
    if !$err {
        # compile the specification if necessary (.akl -> .pam)
        if [newer $fname.akl $fname.pam] {
            puts $pipe "fcompile($fname). "
            flush $pipe
            discard $pipe "yes*
        }
    }
}
# load the compiled specification
puts $pipe "load($fname). "
flush $pipe
discard $pipe "yes"
}
}

# create a frame and inside widgets for function list box
frame .a
label .a.label3 -text "Function:" -anchor w
listbox .a.operation -relief sunken -borderwidth 2 -xscrollcommand {.a.xscroll2 set} -yscrollcommand {.a.yscroll2 set} -bg white
scrollbar .a.yscroll2 -command ".a.operation yview" -orient vertical
scrollbar .a.xscroll2 -command ".a.operation xview" -orient horizontal

# place all the widgets inside the frame .a on the screen
grid .a.label3 -row 0 -column 0 -sticky nsew
grid .a.operation -row 1 -column 0 -sticky nsew
grid .a.yscroll2 -row 1 -column 1 -sticky nsew
grid .a.xscroll2 -row 2 -column 0 -sticky nsew

# create a frame and inside widgets for variable declaration, variable # value and default variable value
frame .b
label .b.label1 -text "Variable Declaration:" -anchor w
text .b.declare -bg white -xscrollcommand {.b.xscroll1 set} -yscrollcommand {.b.yscroll1 set} -width 15 -height 6
scrollbar .b.yscroll1 -command ".b.declare yview" -orient vertical
scrollbar .b.xscroll1 -command ".b.declare xview" -orient horizontal
label .b.label2 -text "Variable Value: " -anchor w
text .b.value -bg white -xscrollcommand {.b.xscroll2 set} -yscrollcommand {.b.yscroll2 set} -width 15 -height 6 -wrap none
scrollbar .b.yscroll2 -command ".b.value yview" -orient vertical
scrollbar .b.xscroll2 -command ".b.value xview" -orient horizontal
label .b.label -text "Default Variable Value:" -anchor w
listbox .b.defaultvalue -relief sunken -xscrollcommand {.b.xscroll set} -yscrollcommand {.b.yscroll set} -bg white
scrollbar .b.yscroll -command ".b.defaultvalue yview" -orient vertical
scrollbar .b.xscroll -command ".b.defaultvalue xview" -orient horizontal
button .b.remove -text "Remove Default Variable Value" -command {remove } -borderwidth 1

# place all the widgets inside the frame .b on the screen
grid .b.label1 -row 0 -column 0 -sticky nsew
grid .b.declare -row 1 -column 0 -sticky nsew
grid .b.yscroll1 -row 1 -column 1 -sticky nsew
grid .b.xscroll1 -row 2 -column 0 -sticky nsew
grid .b.label2 -row 3 -column 0 -sticky nsew
grid .b.value -row 4 -column 0 -sticky nsew
grid .b.yscroll2 -row 4 -column 1 -sticky nsew
grid .b.xscroll2 -row 5 -column 0 -sticky nsew
grid .b.label -row 6 -column 0 -sticky nsew
grid .b.defaultvalue -row 7 -column 0 -sticky nsew
grid .b.yscroll -row 7 -column 1 -sticky nsew
grid .b.xscroll -row 8 -column 0 -sticky nsew
grid .b.remove -row 9 -column 0 -pady 3

# create a frame and inside widgets for function call, result and # return value display
frame .c
label .c.label3 -text "Function:" -anchor w
label .c.label4 -bg white -anchor w
label .c.label1 -text "Function Call:" -anchor w
frame .c.function
entry .c.function.entry5 -bg white -textvariable data2 -width 7
label .c.function.label5 -bg white -anchor w -width 14
frame .c.frame -width 28 -height 110
frame .c.frame.left
frame .c.frame.right
label .c.label2 -text "Result:" -anchor w
listbox .c.result -width 12 -height 10 -bg white -xscrollcommand {c.xscroll2 set} -yscrollcommand {.c.yscroll2 set}
scrollbar .c.yscroll2 -command ".c.result yview" -orient vertical
scrollbar .c.xscroll2 -command ".c.result xview" -orient horizontal
label .c.label5 -text "Return Value:" -anchor w
entry .c.return -xscrollcommand {c.xscroll3 set} -bg white -textvariable returnvalue
scrollbar .c.xscroll3 -command ".c.return xview" -orient horizontal

# place all the widgets inside the frame .c on the screen
pack .c.frame.left .c.frame.right -side left -in .c.frame -padx 1
pack .c.function.entry5 .c.function.label5 -side left -in .c.function
grid .c.label3 -row 0 -column 0 -sticky nsew
grid .c.label4 -row 1 -column 0 -sticky nsew
grid .c.label1 -row 2 -column 0 -sticky nsew
grid .c.function -row 3 -column 0 -sticky nsew
grid .c.frame -row 4 -column 0 -sticky nsew
grid .c.label2 -row 5 -column 0 -sticky nsew
grid .c.result -row 6 -column 0 -sticky nsew
grid .c.yscroll2 -row 6 -column 1 -sticky nsew
grid .c.xscroll2 -row 7 -column 0 -sticky nsew
grid .c.label5 -row 8 -column 0 -sticky nsew
grid .c.return -row 9 -column 0 -sticky nsew
grid .c.xscroll3 -row 10 -column 0 -sticky nsew

# create a frame and inside widgets for additional buttons
frame .d
button .d.run -text "Run" -state disabled -command runcommand
button .d.clear -text "Clear" -command ".c.result delete 0 end"
button .d.add2 -text "Add It to Default Variable" -command {
adddata [.c.result get active]
 .b.defaultvalue insert end [.c.result get active]
}
checkbutton .d.addtoinput -text "Update Variable Value" -variable indicator

23
label .d.space

# place all the widgets inside the frame .d on the screen
pack .d.run -pady 90
pack .d.clear .d.add2 .d.addtoinput -pady 5
pack .d.space -pady 30

# place all the frames on the screen
pack .a .b .c .d -side left -fill both -padx 4

# associate keystroke commands with the function list box
bind .a.operation <Double-Button-1> {
    global filena
    set number1 $number
    .d.run configure -state normal
    # distract the function name and each variables from the function
    # put them in the main window
    while {$number1>0} {
        destroy .c.frame.left.entry$number1
        destroy .c.frame.right.label$number1
        incr number1 -1
    }
    set number1 $number
    set f [open $filena "r"]
    while {[gets $f line]>=0} {
        set subline [string trim $line]
        if [string match class* $subline] {
            set subline [string trimright $subline {]]
            set name [string range $subline 6 end ]
        }
    }
} close $f
set operation [.a.operation get active]
.c.label4 configure -text "$operation"
set position [string last "(" $operation]
set operation1 [string range $operation 0 $position]
if [string match {*\ *} $operation1] {
    set position1 [string first " " $operation1]
    incr position1 1
    set operation1 [string range $operation1 $position1 end]
}
.c.function.label5 configure -text ".$operation1"
set number [regsub -all , $operation * temp]
if {$number == 0} {
    set temp1 [string trimright $operation]
    set temp1 [string trimright $temp1 \;]
    set temp1 [string trimright $temp1]
    set temp1 [string trimright $temp1 \})
    set temp1 [string trimright $temp1]
    if [string match {*(} $temp1] {set number -1}
}
set number [expr $number+1]
set number1 $number
while {$number1>0} {
entry .c.frame.left.entry$number1 -bg white -width 17
pack .c.frame.left.entry$number1 -in .c.frame.left -pady 1
label .c.frame.right.label$number1 -text "," -bg white -width 3
pack .c.frame.right.label$number1 -in .c.frame.right -pady 1
incr number1 -1
}

set number1 $number
if {$number == 0} {
    .c.function.label5 configure -text ":$operation1":"
} if {$number1 > 0} {
    bind .c.frame.left.entry$number1 <Return> {focus .c.frame.left.entry$number1}
    .c.frame.right.label1 configure -text "\)
} if {$number1 > 1} {
    bind .c.frame.left.entry$number1 <Return> {
        incr number1 -1
        focus .c.frame.left.entry$number1
    }
}

focus .b.declare

# associate keystroke commands with the variable default value entry
bind .b.defaultvalue <Double-Button-1> {
    focus .b.value
    .b.value insert end [selection get]\n
}

# procedure definitions
# associate keystroke commands with the result display entry
# begin proc runcommand
proc runcommand {} {
    global x1 y1 filename1 filena filename number pipe fname tmpdir
    operation1 updatevalue returnvalue indicator
    destroy .f
    loading "Busy ..."
    update
    SaveFile .b.declare $filename1
    set text1 [string trim [.b.declare get 1.0 end]]
    set text2 [string trim [.b.value get 1.0 end]]
    set senddata $text1$text2
    set data2 [.c.function.entry5 get]
    append senddata $data2\.$operation1
    set number1 $number
    set temp3 ""
    while {$number1 > 0} {
        set temp2 [.c.frame.left.entry$number1 get]
        set temp2 [string trim $temp2]
        set temp3 $temp3$temp2,
        incr number1 -1
    }
    set temp3 [string trim $temp3 ,]
    append senddata $temp3\n;
set querystring $senddata

# copy spec to tempdir
exec cp $fname.h $tmpdir/$fname.h
# add the query to the specification
set tmpfile [open $tmpdir/$fname.h a]
puts $tmpfile $querystring
close $tmpfile

# convert the query to akl
if [catch {exec toakl $tmpdir/$fname.h $tmpdir/$fname.akl 
    >& $tmpdir/_errs}] {
    set errfile [open $tmpdir/_errs r]
    set ans [read $errfile]
    if {$ans == ""} {set ans "toakl executable not installed properly"}
    destroy .f
    errordialog $ans
    close $errfile
} else {
    set pipe2 [open $tmpdir/$fname.akl r]
    gets $pipe2 ans2
    while {![string match "qXXXXXX(*)*" $ans2]} {gets $pipe2 ans2}
    set tmpfile [open query.akl w]
    puts $tmpfile $ans2
    while {![string match "" $ans2]} {
        gets $pipe2 ans2
        puts $tmpfile $ans2
    }
    close $pipe2
    close $tmpfile
    set result ""

# compile and load the query
puts $pipe "compile(query). "
flush $pipe
discard $pipe "yes"
puts $pipe "qXXXXXX(R, S, Foo). "
flush $pipe
gets $pipe store
if [string match "| ?- Error:*" $store] {
    # error generated by akl program
    destroy .f
    errordialog [string range $store 5 end]
    set store ""
    discard $pipe "*Computation aborted*"
} else {
    gets $pipe store
    if [string match "no*" $store] {
        set store ""
    }
    # interpreter returned no - execution was inconsistent
    destroy .f
    errordialog "Error: inconsistent specification."
```python
} else {
    if [string match "*Computation suspended*" $store] {
        set store ""
        # wasn't enough info to run the query
        destroy .f
        errordialog "Error: insufficient information to complete execution"
        discard $pipe "no*"
    } else {
        # query ran successfully, so print the result nicely
        set len [string length $store]
        set len [expr $len - 2]
        set store [string range $store 4 $len]
        gets $pipe result
        set len [string length $result]
        set len [expr $len - 2]
        set result [string range $result 4 $len]
        puts $pipe "speclib.printstore($store). "
        flush $pipe
        gets $pipe ans2
        gets $pipe ans2
        gets $pipe ans2
        gets $pipe ans2
        set store ""
        while {![string match yes* $ans2]} {
            append store $ans2;
            gets $pipe ans2
        }
        puts $pipe "speclib.print($result). "
        flush $pipe
        gets $pipe ans2
        gets $pipe ans2
        gets $pipe ans2
        set result ""
        while {![string match yes* $ans2]} {
            append result $ans2
            gets $pipe ans2
        }
    }
}

# display the variable name and the result in the result box
destroy .f
if {([string length $store]>0)} {
    set temp $store
    set temp1 [string trim [.b.declare get 1.0 end]]
    regsub -all ;; $temp1 " " temp2
    set c 0
    set temp3 ""
    foreach i $temp2 {
        if {$c == 1} {
```
append temp3 $i " \\
incr c -1 \\
else { 
incr c 1 
}
if ($indicator==1) {
.b.value delete 1.0 end
}
# update the input value if necessary
set updatevalue ""
foreach i $temp3 {
    set line [string range $temp 0 [string first ";" $temp]]
    .c.result insert end $i=$line
    if ($indicator==1) {
        .b.value insert end $i=$line\n
    }
    append updatevalue $i=$line\n
    set temp [string range $temp [expr [string first ";" $temp]+1]
end]
# display the return value in the return value box
.c.result insert end " "
c.result see end
set returnvalue $result
}
}
# end proc runcommand

proc loading {message} {
# display a message in a small box on center of the screen
toplevel .f
wm title .f " "
wmg group .f .
label .f.loading -text $message -bd 2
pack .f.loading -padx 35 -pady 25
set x1 [winfo rootx .]
set y1 [winfo rooty .]
set x [expr [winfo width .]/2+$x1-45]
set y [expr [winfo height .]/2+$y1-30]
wmg geometry .f +$x+$y
set s [wmg geometry .f]
set s1 [wmg geometry .]
}

proc errordialog {message} {
# display a error message and error type in a small box on center of # the screen
destroy .e
toplevel .e
wm title .e "ERROR MESSAGE"
set num [string length $message]
incr num 6
label .e.top -relief sunken -text $message -width $num -height 5
button .e.bot -relief raised -bd 1 -text "OK" -command { destroy .e}
pack .e.top .e.bot -pady 3 -ipadx 1 -ipady 1
set x1 [winfo rootx .]
set y1 [winfo rooty .]
set x [expr [winfo width .]/2+$x1-$num]
set y [expr [winfo height .]/2+$y1-6]
wm geometry .e +$x+$y
}

proc adddata {newdata} {
# add a selected value to the end of list box
    global filename
    set f [open $filename a]
    puts $f $newdata
    close $f
}

proc remove {} {
# remove a selected value for listbox
    .b.defaultvalue delete active
    global filename
    set f [open $filename w] set l [.b.defaultvalue get 0 end]
    foreach i $l {
        puts $f $i
    }
    close $f
}

proc loaddata {win filename} {
# load file into list box
    $win delete 0 end
    if [file exists $filename] {
        set f [open $filename r]
        while {gets $f line}>=0} {
            $win insert end $line
        }
        close $f
    }
}

proc loadoperation {} {
# distract and display all the function from a selected file
    global filena
    loading "Loading ...
    update
    .a.operation delete 0 end
    set flag 0
    set flag1 0
    set f [open $filena "r"]
    while {[gets $f line]}>=0} {
        $win insert end $line
    }
    close $f
}
set besubline [string range $subline 0 1]
set endsubline [string range $subline $mm end]

if {([string length $subline]==0) {continue
}
if [string match public: $subline] {
    set flag 1
    set flag1 1
    continue
}
if [regexp {//*} $besubline] {
    set flag1 0
    continue
}
if [regexp {\(//} $besubline] {
    continue
}
if [regexp {\/*} $endsubline] {
    set flag1 1
    continue
}
if [string match private: $subline] {
    set flag 0
    break
}
if {([string match {,*\)*;*} $subline]&&($flag1 == 1)} {
    set flag 0
    break
}
if {($flag == 1)\&\&($flag1 == 1)} {
    .a.operation insert end $subline
}
}
close $f
}

proc editpro {} {
    # a file editor with all kind of functions
    global filena
toplevel .edit
    wm title .edit $filena

    # create the text box and scrollbar
text .edit.t -width 80 -height 30 -yscrollcommand {.edit.sb set} -background white
    scrollbar .edit.sb -command {.edit.t yview}

    # create a frame to hold the menubar
    frame .edit.f

    # create the File and Search menubuttons and menus
    menubutton .edit.f.file -text "File" -menu .edit.f.file.filem
    menubutton .edit.f.edit1 -text "Edit" -menu .edit.f.edit1.editm
    menubutton .edit.f.search -text "Search" -menu .edit.f.search.searchm
menu .edit.f.search.searchm -tearoff true
menu .edit.f.file.filem -tearoff true
menu .edit.f.edit1.editm -tearoff true

# add commands to the File menu
.edit.f.file.filem add command -label "Save and Reload" -command {
    SaveFile .edit.t $filena
do2
    update
}
.edit.f.file.filem add command -label "Save (^s)" -command {SaveFile
    .edit.t $filena}
.edit.f.file.filem add command -label "Save as " -command {set
    comman newfile
dialog "New file name:" comman }
.edit.f.file.filem add command -label "Exit (^c)" -command {destroy
    .edit}

# add commands to the Search menu
.edit.f.search.searchm add command -label "Search Forward (^f)" -
    command {
    set comman SearchForward
dialog "Search for:" comman }
.edit.f.search.searchm add command -label "Search Backward (^b)" -
    command {
    set comman SearchBackward
dialog "Search for:" comman }
.edit.f.search.searchm add command -label "Go To" -command {
    set comman Goto
dialog "Go to:" comman }

# add commands to the Edit menu
.edit.f.edit1.editm add command -label "Cut" -command { tk_textCut
    .edit.t}
.edit.f.edit1.editm add command -label "Copy" -command { tk_textCopy
    .edit.t}
.edit.f.edit1.editm add command -label "Paste" -command { tk_textPaste .edit.t}

# place all widgets on the screen
grid .edit.f -row 0 -column 0 -sticky nsew
grid .edit.t -row 2 -column 0 -sticky nsew
grid .edit.sb -row 2 -column 1 -sticky nsew
pack .edit.f.file .edit.f.edit1 .edit.f.search -side left

# associate keystroke commands with the all the entry
bind .edit.t <Control-s> {SaveFile .edit.t $filena}
bind .edit.t <Control-c> {destroy .edit}
bind .edit.t <Control-f> { set comman SearchForward
dialog "Search for:" comman }
bind .edit.t <Control-b> { set comman SearchBackward
dialog "Search for:" comman }
OpenFile .edit.t $filena
}
proc newfile {} {
# save a file to a new name
    set val [.g.frame.right get]
    if {$val != ""} {
        SaveFile .edit.t $val
        destroy .mbar.file.m
        destroy .edit
        menu .mbar.file.m -tearoff true
        foreach i [lsort [glob *.h]] {
            .mbar.file.m add command -label "$i" -command {
                set filena [.mbar.file.m entrycget active -label]
                do1
                do2
            }
        }
        update
    }
}

proc Goto {} {
# move the cursor to given place
    set val [.g.frame.right get]
    if ![catch {.edit.t mark set insert $val.0}] {
        .edit.t see $val.0
    } else {
        errordialog "Illegal line number"
    }
    focus .edit.t
}

proc dialog {message comman} {
# take a input message for file editor
toplevel .g
    wm group .g .edit
    wm title .g ""
    set num [string length $message]
    #incr num 2
    frame .g.frame -height 3
    label .g.frame.left -relief sunken -text $message -width $num
    entry .g.frame.right -width 15
    pack .g.frame.left .g.frame.right -pady 1 -ipadx 1 -ipady 1
    pack .g.frame
    set xl [winfo rootx .edit]
    set yl [winfo rooty .edit]
    incr yl 30
    wm geometry .g +$xl+$yl
    focus .g.frame.right
    bind .g.frame.right <Return> {$comman
        destroy .g}
}

proc OpenFile {win filena} {
# load file to screen
    if [file exists $filena] {

set fileID [open $filena r]
set contents [string trim [read $fileID]]
$win delete 1.0 end
$win insert end $contents
close $fileID
$win mark set insert 1.0
}
focus $win
}

proc SaveFile {win filena} {
# save change to the given file name
set fileID [open $filena w]
set contents [string trim [$win get 1.0 end]]
puts $fileID $contents
close $fileID
focus $win
}

proc SearchForward {} {
# search forward from current point
set val [.g.frame.right get]
set nindex [.edit.t search -forwards $val "insert + 1 chars"]
if {$nindex == ""} {bell} else {
    .edit.t mark set insert $nindex
    .edit.t see $nindex
    focus .edit.t
}
}

proc SearchBackward {} {
# search backward from current point
set val [.g.frame.right get]
set nindex [.edit.t search -backwards $val "insert + 1 chars"]
if {$nindex == ""} {bell} else {
    .edit.t mark set insert $nindex
    .edit.t see $nindex
    focus .edit.t
}
}