A Graphical User Interface for Editing Formal Specification Objects

A Master’s Paper in
Computer Science
By
Duanqing Wu

©2001 Duanqing Wu

Submitted in Partial Fulfillment
Of the Requirements
For the Degree of

Master of Science

July 2001
# TABLE OF CONTENTS

TABLE OF CONTENTS ............................................................................................................. i  
LIST OF FIGURES .................................................................................................................. i  
ACKNOWLEDGMENTS .............................................................................................................. ii  
I.  INTRODUCTION ................................................................................................................... 1  
II.  DESIGN AND IMPLEMENTATION ..................................................................................... 3  
   1. USER INTERFACE DESIGN ............................................................................................ 3  
      1.1 USER INTERFACE ..................................................................................................... 3  
      1.2 TREE HIERARCHY .................................................................................................... 5  
      1.3 OPERATIONS .......................................................................................................... 5  
      1.4 ALIASES .................................................................................................................. 7  
      1.5 UNDO/REDO ............................................................................................................ 9  
      1.6 RULES .................................................................................................................... 9  
   2. TCL/TK ............................................................................................................................. 10  
   3. OBJECT CONVERSION .................................................................................................. 11  
III. RELATED WORK .............................................................................................................. 12  
IV.  FUTURE WORK AND CONCLUSION ............................................................................. 13  
REFERENCES .......................................................................................................................... 15  

# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The architecture of the system for executing SPECS-C++ specifications.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Object Editor.</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>The Add dialog window.</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>The Update dialog window.</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>An example of adding an alias.</td>
<td>8</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

My wife, Li, has been with me through many trying times in my graduate career, and supporting me through this process has to be one of her greatest challenges. She has always had, and continues to have, my heartfelt thanks and undying love. Our two children, Nicholas and Jacqueline, are a tremendous inspiration, in spite of many almost sleepless nights since they were born.

I extend my deepest gratitude to Dr. Timothy Wahls for serving as my advisor, for providing me this opportunity to work with him, and for his time and patience to guide me throughout this work.

I would also like to express my appreciation to Drs. Thang N. Bui, Linda M. Null, and Pavel Naumov for serving as my committee members and for their critiques on the manuscript.
I. Introduction

The standard software development cycle of analysis, design, and implementation is a multiphase approach intended to maintain disciplined and systematic project control. Specifications are formed during the analysis and design phases after user requirements are gathered and analyzed. They define all the user requirements of the software system to be implemented. They also serve as an important communication channel between users and developers. As software developers strive to provide more flexibility and more functionality, software is becoming more and more complicated. As a result, specifications play an increasingly important role in software development.

A formal specification is a precise and unambiguous specification of the functionality of a software system [1]. It employs mathematical notations to describe in a precise way the characteristics of the system. It provides a high level of abstraction by specifying what the system must do without the details of how it is to be done. A formal specification can provide a great deal of insight into a system. Thus, developing a formal specification can improve the quality of software.

As the implementation of a software system is carried out according to its specifications, specifications that do not correctly capture user requirements are of little benefit, and this problem is considered as one of the most serious in software development [2]. Therefore, the ability to validate specifications before any implementation is carried out is critical and can have a dramatic impact on system development costs.

With a formal specification, there are a number of techniques available for validation, including review, proof, and execution. Executable formal specifications are of great interest since one problem in software development is that the mapping from a specification to its implementation is complex. There is no automated procedure to transform analysis into design, and verifying program correctness is not trivial. The lack of executable specifications leads to long delays in evaluating whether the direction chosen by the specifier for development is right. If a formal specification can be executed, then the specification can be validated against informal requirements and can be used as a prototype and test oracle. Executable specifications allow demonstration of the behavior of a software system before it is actually implemented. Errors can be corrected immediately without incurring costly redevelopment. Requirements that are
initially unclear can be clarified and completed by hands-on experience with executable specifications. Although execution of formal specifications is still somewhat controversial \[2, 3, 4\], the ability to execute formal specifications provides many desired benefits that outweigh the disadvantages. Much research has been conducted on executing specifications written in formal, model-based specification languages, such as VDM \[5,6\], Z \[7\], B \[8\], JML \[9\] and SPECS-C++ \[10\], although these languages were originally designed to be non-executable.

The critical step in executing specifications is translating a formal specification into some type of executable programming language, either automatically or manually (in the cases of VDM and Z). For automated translation, often only a restricted subset of the specification language can be translated or a set of possible values for all variables in the specification must be provided. To overcome these limitations, Wahls developed a technique \[4, 10\] to execute SPECS-C++ specifications by translating them to Agents Kernel Language (AKL) \[11\] and then executing the resulting AKL programs. This technique can execute a much larger subset of the specification language than other methods mentioned above and can be easily extended to executing other object-oriented model-based specification languages such as VDM++ \[12\] and Object-Z \[13\].

To make this technique easier to use, Chen and Wahls developed a graphical user interface (we will call it CW Interface hereafter) \[14\] for the SPECS-C++ specification execution system. This graphical interface provides an integrated editor for developing specifications. It also provides support for constructing and managing test cases, support for executing sequences of member function calls, and ability to use the execution system without knowing the underlying programming language. The interface greatly increases the usefulness of the specification execution system.

As an extension to the CW interface, a graphical interface called Object Tree was developed by Cheng to construct and display specification objects \[15\]. However, Object Tree is limited to work with a single object, which may be a complex one that contains deeply nested objects. We will discuss nested objects fully in Section II. The functionality provided by Object Tree is also limited to some simple operations, for example, a new value or nested node can only be inserted to the end of an object and aliases are not presented in the display. In this paper, we describe an enhanced revision of Object Tree, named Object
Editor, as a new extension to the CW Interface to allow a user to display and edit specification objects. We will show that the Object Editor provides additional functionality to make the SPECS-C++ specification execution system more useful.

II. Design and Implementation

1. User Interface Design

1.1 User Interface

The overall architecture of the specification execution system, including the extension described here, is shown in Figure 1.

![Figure 1. The architecture of the system for executing SPECS-C++ specifications.]

The new system for executing SPECS-C++ specifications now consists of five components:

1) The CW Interface. This is the system interface for users to use the execution system.
2) The Specification Editor. This is a text editor to display and edit the specification files.
3) The Specification Compiler. This is the translator to translate the SPECS-C++ specifications to AKL programs.
4) The AKL Interpreter. This is the interpreter to execute the AKL programs.
5) The Object Editor. This is the interface to display and edit specification objects.

Object Editor is an integral part of the CW Interface, as it interacts directly with the CW Interface. A user can invoke the Object Editor from the CW Interface. When the user finishes editing the objects, Object Editor returns to the CW Interface the edited or original specification objects, depending on the user’s choice.

The appearance of Object Editor was designed to be similar to most windowing applications, in an attempt to make it look familiar to users and thus make it easier to use. Figure 2 shows a screenshot of Object Editor.

![Figure 2. Object Editor.](image-url)
We also use many pop-up dialog windows. These dialogs are used to collect input values from the user, to show an error message, or to confirm a choice made by the user. We make these dialogs modal in order to force an immediate response from the user. As long as such a dialog is present, the user cannot do anything else in the application but must make a certain choice to continue.

1.2 Tree Hierarchy

An object, by our definition, is a container for a value. A nested object is an object contained within the value of another object. For a complex, deeply nested object, it is obviously very difficult to display and edit such an object in a text form.

To better present an object, Cheng used a tree hierarchy in Object Tree [15]. We employ a similar idea so that a specification object is displayed as a tree and all specification objects form a forest.

In our tree structure, each node represents the value of an object. A directed line connecting two nodes represents the relationship between the nodes, with the starting node (parent) as the object whose value contains a nested object, which is the ending node (child). The root of a tree is labeled with the variable name for the root object. All trees of the forest are stacked vertically. Rules that must be followed to draw trees will be discussed in a later section.

We also use the tree hierarchy to model aliasing, which will be discussed in more detail later.

The tree hierarchy can clearly display the structure of complex nested objects and make editing such objects much easier.

1.3 Operations

The Object Editor supports three major operations to edit a specification object: Insert, Delete and Update. A user can select an editing operation by either clicking a button on the left side of the window or selecting from the Edit menu.

1) Insert. A user can insert a nested object, a value or an alias to an existing object. When Insert is the selected editing operation, a dialog window will appear upon a mouse-click on a displayed object. Figure 3 shows such a window.
This dialog window allows a user to select which type of insertion to perform and what value to insert if Nested Object or Value is selected. A user can also cancel the action. When a user select Alias, the dialog window will disappear and the user can insert an alias by clicking the node to be aliased. This operation will be discussed in more detail later.

The position within the object at the mouse-click point is the position where an item (an alias or a nested object or a value) will be inserted. Therefore, a user can insert to any selected position within an object if that insertion is legal according to certain rules, which will be discussed later.

2) Delete. This operation allows a user to delete an alias or a node. The object at the mouse-click point is the one to delete. If a link (the directed line) is clicked and the linked object (the nested object) has no other links incident to it (i.e., the linked object has in-degree of one), the linked object together with the link will be deleted. When the linked object has other links incident to it in addition to the clicked one, only the clicked link is deleted without affecting other links. This allows removal of an alias without deleting the aliased object. When a node is deleted, either directly or by removal of all links to that node, all the nested objects (children of that node) will also be deleted, unless those nested objects are referenced by other objects (in other words, they have other links connected to them).
3) **Update.** A node can be updated if it contains no other objects and is a number or a boolean value.

This allows a user to change the value of an object. Figure 4 shows the dialog window that will appear for updating.

![Figure 4. The Update dialog window.](image)

A user can undo an operation performed on the object and can also redo an operation that has been undone. More discussion on these two operations will be given later.

If a user decides to exit the Object Editor, he or she has the choice to either save or discard all the modifications made to the object, before returning to the CW Interface. The Done operation will save any modifications and return the modified specification objects, while the Cancel operation will ignore any changes and return the original specification objects.

### 1.4 Aliases

Aliasing refers to the phenomenon that an object can have multiple names. By this, we mean that an object can be referenced by more than one other object. Many different objects can contain references to the same object, and one object can contain multiple references to another. For example, there are two trees (one tree has G as its root and the other has V as its root) shown in Figure 2. The object with the value of 4 in tree “G” is referenced by two objects, the root object of tree “G”, which contains two references to it, and the root object of tree “V”, which contains one reference to it.

To support editing of a nested object with aliases, we have chosen an approach that can accommodate aliasing. In our tree hierarchy, the value of a nested object is displayed with a link connected to it from the parent object. If aliasing exists, the value of a nested object will have multiple links incident to it.
With the **Insert** editing mode selected, if a user wishes to insert an alias, the **Insert** dialog window will disappear and the shape of the mouse pointer will change to a hand. Also, a dashed line originating from the position to which this alias should be inserted together with a help message will attach to and move with the mouse pointer until the user clicks a node to which a new alias will be created, as shown in Figure 5. Right-clicking the mouse will cancel this action before an alias is created.

**Figure 5.** An example of inserting an alias.

With deletion, if a node is deleted, all the aliases of the node will be deleted too. A user can also selectively delete one alias by clicking the link (the directed line) that represents the alias. In this case, only one alias is deleted without affecting the other aliases.
The goal of our approach is to help users understand the structure of nested objects with aliases and to make it easier to edit such objects.

1.5 Undo/Redo

The **Undo** and **Redo** operations allow a user to undo a previous action. Users should have a choice of either continuing their modification of the object or going back to a previous stage – either to undo or redo a change.

Undoing an operation will erase the effect of that operation on the modified object. Undoing an insertion will delete anything just inserted to the object, either an alias or a nested object or a value. Undoing a deletion will insert anything just deleted from the object. Undoing an update will change the new value back to the old value.

We did not set a limit on how many **Undo** operations can be performed. Thus, the user can trace back to every modification that has been done until the original object if the user chooses to do so.

The **Redo** operation is used to undo the **Undo** operation. Together with the **Undo** operation, this provides a convenient way for a user to go back and forth to a specific modification and to evaluate such a modification.

1.6 Rules

Certain rules are applied for user input values and for certain operations to ensure the consistency of the system.

1) User input values must be numbers, boolean values, or structured values. The structured values include sets, sequences, tuples, and objects.

2) The **Update** operation can only apply to a node whose value is a number or a boolean. The Update operation cannot apply to a node containing a structured value.

3) The **Insert** operation can only apply to a node containing a structured value. The **Insert** operation cannot apply to a node whose value is a number or a Boolean.

4) The root of a tree cannot be deleted.

If an operation violates any of the above rules, the operation will be canceled and an error message will be shown to indicate the violation.
There are also rules, which are modified according to the tidy-tree rules [16, 17], to draw trees:

1) A parent should be drawn above its children. One exception is allowed in aliasing where a child may be drawn above its parent.

2) Nodes at the same level should lie along a horizontal line.

3) Nodes at all levels should be centered along a vertical line. Nodes at the same level are treated as a single entity when centering.

2. TCL/TK

There are several programming languages that can be used to build a graphical user interface, such as C++, Java, and Visual Basic. Each language has its advantages and disadvantages. The language we chose to build Object Editor is TCL/TK [18, 19].

TCL, standing for “Tool Command Language”, is a simple scripting language for controlling and extending applications. It provides generic programming facilities that are useful for a variety of applications. Its interpreter is implemented as a library of C procedures that can be easily incorporated into applications and each application can extend the core TCL features with additional commands specific to that application.

TK is one of the most useful extensions to TCL. It extends the core TCL facilities with additional commands for building graphical user interfaces, so that user interfaces can be constructed by writing TCL scripts instead of C code. TK is also implemented as a library of C procedures, so it can be used in many different applications. Individual applications can also extend the base TK features with new user-interface elements written in C.

There are many benefits provided by TCL/TK, including:

1) TCL/TK makes it easy for any application to have a powerful scripting language. All that an application needs to do is to implement a few new TCL commands that provide the basic features of that application. Then the application can be linked with the TCL interpreter to produce a full-function scripting language that includes both the commands provided by TCL and those implemented by the application.

2) TCL/TK is easy to learn and thus it can facilitate rapid development.
3) TCL/TK makes an excellent “glue language”. Because it is embeddable, it can be used for many different purposes in many different programs. TCL scripts can be written to combine the features of all the programs.

We chose TCL/TK to take advantage of the above benefits. TCL/TK provides us not only all of the graphical components we need but also the “glue” that combines all components (including those written in other languages) into a single system. The core of the SPECS-C++ execution system is a compiler that translates SPECS-C++ specifications to AKL programs. The AKL interpreter, however, does not provide any interfaces designed for interacting with external programming languages. The CW Interface (developed using TCL/TK) controls the interpreter by using a read/write pipe to simulate the command line interaction of the AKL interpreter with a user. As an extension to the CW Interface, Object Editor is developed using the same programming language as the CW Interface and becomes an integral part of the system.

3. **Object Conversion**

In order to better display specification objects as a tree hierarchy, a concept of tree objects in addition to specification objects was proposed by Cheng and Wahls and used in Object Tree [15]. We use the same concept here with minor modifications.

There are two types of objects in our system: specification objects and tree objects. Specification objects are used in the rest of the SPECS-C++ execution system. A specification object is a list of pairs of a “container” and its “content”. Each pair of container and content is enclosed within a pair of parentheses (“()”) and is separated from other pairs by a space. A container is a reference (represented by r(id)) to a memory location or cell for the object. The content is the value contained in the container, that is, the value stored at that location or the value of the specification object. A container is separated from its content by a comma.

Tree objects are used by the Object Editor to graphically display specifications objects. A tree object is a list of its value, its reference, and a sublist of its children. The sublist of children can be empty. If not empty, each child in the sublist has the same structure as its parent, consisting of its own value, its own reference, and a sublist of its own children.
The Object Editor displays only tree objects but handles the conversion between tree objects and specification objects. Specification objects are passed to the Object Editor and converted to tree objects by the Object Editor for displaying and editing. The Object Editor performs all the editing operations on tree objects. When exiting, the Object Editor converts tree objects back to specification objects and returns specification objects to the calling component of the system. The object conversions are transparent to users.

III. Related Work

Object Tree, developed by Cheng in our research group, is another graphical user interface for displaying and editing specification objects. Both Object Tree and Object Editor can display complex specification object values and can also be used to edit such objects. Object Tree, however, displays only one tree representing a single specification object, even though this specification object may contain complex nested objects. Object Tree also fails to model multiple specification objects that interact with each other, such as the situation where they all reference some shared objects. For example, in Figure 2, the object with value 4 is referenced by both tree “G” and tree “V”. Object Tree would display this nested object in tree “G” and tree “V”, but would not show clearly the fact that they reference the same object. Object Tree does not allow inserting a new value or a new nested object to any position within an object; it only allows the new one to be appended to that object. Compared to Object Tree, Object Editor provides a higher degree of integration with the rest of the SPECS-C++ specification execution system. Object Editor is able to display all specification objects passed to it. In other words, it displays the whole forest rather than a single tree. Thus, it can clearly represent the interactions among all the specification objects. Object Editor provides much more functionality than Object Tree: a new value, a new nested object or an alias can be inserted to any position within an object; a single reference to an object (the connecting line in the tree structure) as well as the object itself (a node) can be deleted; and aliasing is better represented and employed for all operations.
Except for Object Tree, it is somewhat difficult to compare the Object Editor with other interfaces for executing specifications, as none of the published work on executable specifications provides a similar graphical user interface for displaying and editing nested objects. Most published work has different focus, such as the execution technique, the subset of the language that can be executed, or semantic foundations for executable specifications. In systems that do provide a graphical interface such as the B-toolkit Animator [20], Schach and Gray’s animator for UML [21], and the animator for IFAD VDM-SL [22], specification objects are not editable directly in the interface.

IV. Future Work and Conclusion

One interesting possibility for future work is improving the efficiency of drawing trees. Object Editor uses an array structure to store certain layout information, such as the length of a node. When trees change, only part of the layout information is recomputed, while other layout information that is not affected by the changes is reused. This is an improvement over Object Tree. However, whenever there is a change, all trees are redrawn. In addition, Object Editor draws trees in a vertically stacked fashion. Drawing trees in a more aesthetical and space-optimized way remains a challenge. One potential improvement is to develop a more efficient algorithm to draw trees so that when there is a change in a tree, only the part affected by the changes will be redrawn. Moen’s algorithm of drawing dynamic trees [23] may be very helpful in this regard.

Object Editor does not allow the Update operation on structured values. Thus, another idea for future work is to implement the Update operation so that the objects with structured values are editable.

Another planned enhancement of our current work is a cleaner integration of Object Editor with the rest of the SPECS-C++ execution system. Currently, a user must exit Object Editor, through either the Done or the Cancel operations, to return to the CW Interface. The enhancement would leave the objects displayed at all times so that a user does not have to exit Object Editor to get back to the CW Interface. The modifications made on the specification objects in Object Editor can be updated and passed to the rest of the system, if so desired.
Object Editor is a graphical user interface for displaying and editing complex specification objects. It includes a number of enhancements of the CW Interface:

- It presents nested objects graphically using a tree hierarchy that is easy to read and understand.
- It provides an integrated editor for conveniently and efficiently constructing and editing nested objects.

These enhancements increase the effectiveness and efficiency of the CW Interface when dealing with nested objects, and therefore make the CW Interface more useful. The CW Interface (including the Object Editor) and the SPECS-C++ execution system will help in overcoming the limitations of formal specifications and increase the usefulness of formal specifications in practice.
REFERENCES

   http://cs.hbg.psu.edu/courses/comp413.taw.f98.


