OBJECT TREE - A GRAPHICAL USER INTERFACE FOR EDITING SPECIFICATION OBJECTS

A Master’s Paper in Computer Science

By
Yiching Cheng

© 2001 Yiching Cheng

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science

April 2001
TABLE OF CONTENTS

ACKNOWLEDGEMENT i

ABSTRACT ii

1. INTRODUCTION 1

2. BACKGROUND 3

   2.1 Formal Specifications 6
   2.2 Model-Based Specifications 6
   2.3 SPECS-C++ 7
   2.4 Object Tree 8

3. DESIGN 10

   3.1 Tasks/Features 10
   3.2 Contents 11
   3.3 Operations 15
   3.4 Rules 16
   3.5 Implementations 17

4. DISCUSSION 19

5. CONCLUSIONS AND RECOMMENDATIONS 19

REFERENCES 21
ACKNOWLEDGEMENT

I am indebted to a number of people. First and foremost, I would like to express my
gratitude to Dr. T. Wahls for his endless advice and guidance during my study at
Penn State Harrisburg. Thanks are also expressed to the committee members: Dr. T. Bui, Dr. L.
Null, and Dr. P. Naumov for their interest in this study and their reviews on the manuscript.

The financial support and the encouragement of Mr. T. Hammaker from IBM Corporation
at Mechanicsburg, PA are gratefully acknowledged. However, the opinions expressed in this
report are solely of the writer and in no way reflect the views of the sponsor.

Finally, I would like to thank my husband and children for their patience, endurance, love,
and support throughout my entire study.
ABSTRACT

Formal specifications are much more useful in practice if they can be executed. In this study, we describe a graphical user interface, called Object Tree, that can display complex object values resulting from executing specifications and can also be used to edit such objects. This new interface is user friendly, as it can be easily used to edit complex specification objects for constructing test cases (for use as input when executing specifications). In other words, Object Tree is a nested object editor for displaying nested objects, and thus facilitates the execution of specifications.

The proposed Object Tree interface has a unique combination of features not available in other interfaces for executing formal specifications. It represents objects graphically using a tree structure that is appealing and easy to read and understand. It also allows users to construct trees with nodes of object values compactly. Thus, Object Tree is a helpful tool that makes executing specifications much easier and more efficient, and so widens the possibilities for the use of formal specifications in practice.
1. INTRODUCTION

Software is rapidly becoming more and more complicated, mainly because more flexibility and more features must be added to software to meet the needs of industry and users. As such, the specifications of such systems are becoming larger and more complex, and so more prone to errors and ambiguity. To avoid failure and budget overrun, valid and verifiable specifications of software should be developed. However, this is still a great challenge for today’s large-system software developers. The present study proposes an improved specification interface model that makes the execution of specifications much more effective and efficient.

Programmers can compile and execute programs to receive immediate response about their content. At present, there are few tools available to developers for getting immediate responses about the content of their specifications. The inability to receive immediate response is definitely a barrier to the use of formal specifications. Receiving immediate responses makes the specification more understandable, and consequently easier to use. In effect, it makes learning and using formal specification techniques more cost effective. One possible method for achieving immediate responses and increasing cost effectiveness is to execute formal specifications.

Thus, it is highly desirable that a subset of a non-executable formal specification language can be translated into an executable programming language. By definition, a formal specification is a precise, unambiguous specification of the functionality of a software system. Using an executable subset of a formal specification language will make developing and using specifications much easier and more efficient [Gurski 1995]. The critical problem is how to translate a formal specification language into some type of executable programming language.
This problem is addressed in [Wahls et al. 2000].

An interface capable of editing objects can facilitate the execution of formal specifications. For a user to edit advanced and complex objects, a good way is to simply click the mouse. The only problem here is how one can make the interface used for executing specifications more dynamic for editing complex test cases. Alternatively, once one has generated a large number of test values for a specification, how does one evaluate or edit the values? Certainly, it would be unnecessary and difficult to do this manually. Our solution is to develop a graphical user interface called Object Tree which serves as a convenient tool for constructing complex objects. Once such tool is available, one can then edit objects, as he/she desires.

Object Tree is an application of a technique for editing SPECS-C++ specification objects. SPECS-C++ [Baker 1991; Coleman 1991; Wahls et al. 1994; Haerdink 1994; Gurski 2001] is a formal, model-based, interface specification language for C++ classes. SPECS-C++ models classes through discrete structures (e.g., sets, sequences, tuples, strings, and objects) as well as several primitive types (e.g., integer, real, character). The specification of class member and friend functions is done using pre- and post-condition assertions. These assertions are written in first-order predicate calculus over the abstract class instance model. As such, SPECS-C++ is related to other formal, model-based specification languages like the Vienna Development Method (VDM, based on sets and relations) [Jones 1990; Andrews et al. 1993], Z (based on set theory) [Spivey 1989 & 1992; Hayes 1993] and the Larch family of specification languages (based on logic) [Guttag et al. 1993; Leavens 1997], which have been proposed by a number of researchers as described in [Alagar and Periyasamy 1998].

Object Tree can display nested objects, and allows users to develop or maintain the
nested object hierarchy as a tree. By “object”, we mean a container for a value (i.e., a memory location, l-value, or a cell), and not an instance of a class. An instance of a class is modeled as a tuple composed of the data members of a class. An instance is not an object, although it can be the value of an object. A nested object is an object contained within the value of another object. Nested objects are useful for modeling aliasing and mutation at the specification level.

2. BACKGROUND

It should be recognized that several specification languages, formal methods and interface tools have been developed by different research groups. This study builds on the work done by Chen and Wahls [Chen and Wahls 2001]. As shown in Figure 1, the execution of a SPECS-C++ specification is done by translating it to an Agents Kernel Language (AKL) [Janson and Haridi 1994] program. Then, the AKL interpreter is used to execute the program, and the results of this execution are displayed by the interface. AKL is used because it is based on logic and concurrent constraint programming, and so works well for executing specifications written using logical assertions that constrain possible values for specification objects.
Chen and Wahls’s interface was developed for the SPECS-C++ execution system. It possesses the following features:

- The interface is easy to use, as it does not require users to directly interact with the AKL interpreter when running the translated specifications.
- The interface has an integrated editor for developing and modifying specifications.
- The interface supports developing and maintaining many data values that can be used as test cases for specifications.
Object Tree is an extension of the interface developed by Chen and Wahls [Chen and Wahls 2001]. The role of Object Tree within software development is shown in Figure 2. Object Tree uses an explicit representation of nodes and stores various values that consist of objects, tuples, sets, strings, and sequences. Organizing the essential material to explore SPECS-C++ specification objects in Object Tree presents a real challenge. In this study, the architecture of the system for executing SPECS-C++ specification objects is modified, as shown in Figure 1. The main objectives of Object Tree are to represent specification objects graphically and to allow users to apply this tool to construct and update specification objects.
2.1 Formal Specifications

Formal specifications are used to specify procedures. From the perspective of a procedure, a software system has two states: the pre-state, and the post-state. The former is the system state before the procedure is called, while the latter is the system state after the procedure is executed. A procedure specification typically defines the post-state in terms of the pre-state.

A formal specification is usually done at the ADT (Abstract Data Type) level, in which the behavior of each operation on the data type is specified. A formal specification of an ADT should ensure that [Wahls 1998]:

- Implementers of the ADT know exactly how each operation should act.
- Implementers of other parts of a system understand precisely how to use each operation provided by the ADT.

2.2 Model-Based Specifications

Model-based specification languages are formal specification languages that describe the behavior of programming language functions and procedures in terms of an underlying mathematical model. Among the many model-based specification languages that have been developed, VDM [Jones 1990; Andrews et al. 1993] and Z [Spivey 1989 & 1992; Hayes 1993] are the most prominent. Model-based specification languages typically include primitive types, such as integers and characters, as well as more structured types such as finite sets, sequences, tuples and functions. All model-based specification languages use pre- and post-conditions written over values of some/all of these model types.

A pre-condition specifies what must be true for the operation to execute correctly when it
is called. A post-condition specifies what is guaranteed to be true when the operation terminates, assuming the pre-condition was satisfied. The prime notation (') denotes post-state values in the post-condition, while the ^ denotes pre-state values in either the pre- or post-condition. These conditions are written as first-order predicate logic assertions over the model types [Gurski 1995].

2.3 SPECS-C++

Object Tree is an editor for objects specified in the SPECS-C++ specification language. SPECS-C++ is a model-based specification language that is designed for specifying the interfaces of C++ classes.

A SPECS-C++ specification of a class has two major parts: the model, and the specification of the operations. The former is a mathematically precise description of the instances of the class, while the latter consists of the first order predicate logic assertions over the model defining the behavior of each operation. The model typically contains domains (type definitions), data members, constraints (the invariant for the class), and abstract functions. The data members and abstract functions provide the vocabulary for specifying procedures [Wahls et al. 2000].

SPECS-C++ uses the following notation for logical operations [Wahls 1998]:

- /\ - logical “and”
- \\ / - logical “or”
- = > - logical implication
- ! – logical negation
- \forall – universal quantification
• \exists – existential quantification

Though SPECS-C++ does look quite similar to C++, one distinction is that the logical assertions employed to specify a member function are enclosed by a prefix of “/*” and a suffix of “*/”, i.e., the assertions are C-style comments. The parts of the specification that are not commented out are legal C++ syntax. This syntax allows the specification and the actual class declaration to appear in the file without causing confusion or ambiguity. C++-style comments are used for usual (English) comments in the specification. The specification file can be used as a C++ header file.

A representative example of a SPECS-C++ specification is provided in Figure 3, where a class ObjSetTable is defined. The data structure defined by this class allows sets of objects to be stored and retrieved using integer keys. The example specification includes a constructor ObjSetTable for initializing instances of the class. Function AddObj adds an object to the set of objects associated with the key. Function RetrieveVals retrieves an object set from an ObjSetTable for a given key. Boolean functions SameObj and SameVals check if the set of objects and the set of values are the same, respectively, for the two given keys.

2.4 Object Tree

As shown in Figure 1, the function of Object Tree is to edit SPECS-C++ objects. The edited SPECS-C++ objects will be returned to the interface [Chen and Wahls 2001]. In other words, Object Tree can facilitate the execution of SPECS-C++ specifications by displaying and editing complex objects. Other benefits of Object Tree include:

(1) Creating and manipulating objects quickly.
(2) Adding, updating, or deleting nodes in a simple, interactive way.

(3) Displaying and constructing nested object values that would be difficult to edit and
display in a text-based system.

template <class Elem> class ObjSetTable {
   /* model
      domains
         set of Elem& osettype;
         tuple (int key, osettype os) elemtype;
         set of elemtype estype;
      data members
         estype OSet;
      constraints
         \forall elemtype e1 [e1 \in OSet =>
            \forall elemtype e2 [e2 \in OSet =>
               (e1 != e2 => e1.key != e2.key)]]
      abstract functions
         define isin(int i, estype es) as bool such that
            result = tobool(\exists elemtype e [e \in es \&\& e.key = i]);
         define getset(int i, estype es) as osettype such that
            \exists elemtype e [e \in es \&\& e.key = i \&\& result = e.os]
            \& (!isin(i, es) \&\& result = {});
         define getelemvals(osettype es) as set of Elem such that
            (es = {} \&\& result = {})
            \& (\exists Elem& eref [eref \in es
               \&\& result = {eref^} \union getelemvals(es - {eref})));
       // note that the following function is partial -- it is only
       // defined if i is a defined key in the table
         define getelem(int i, estype es) as elemtype such that
            result \in es \&\& result.key = i;
   */
   public:
      ObjSetTable();
      /* modifies: self
         post: OSet' = {} */
      void AddObj(int key, Elem& val);
      /* modifies: self
         post: (isin(key, OSet^) => OSet' = (OSet^ - {getelem(key, OSet^)})
            \union {{key, getset(key, OSet^)
            \union {val}}})
            \& (!isin(key, OSet^) => OSet' = OSet^ \union {{key, {val}}})
      */
      osettype RetrieveVals(int key);
      /* post: result = getset(key, OSet^) */
      bool SameObjs(int key1, int key2);
      /* post: result = tobool(getset(key1, OSet^) = getset(key2, OSet^)) */
      bool SameVals(int key1, int key2);
/* post: result = tobool(getelemvals(getset(key1, OSet^))
   = getelemvals(getset(key2, OSet^))) */

Figure 3: ObjSetTable - an example of a SPECS-C++ specification

(4) Detecting errors in input values.

(5) Building compact, but easy to read trees on the computer screen.

Object Tree is an efficient graphical user interface (GUI) for editing and displaying complex object values.

3. DESIGN

This study proposes a graphical user interface Object Tree that makes the SPECS-C++ specification execution system easier to use. The Object Tree interface should consist of the following ingredients: tasks/features, contents, operations, rules, and implementations, namely the so-called “Five-Model Approach” [Constantine and Lockwood 1999]. A brief description of these is as follows.

3.1 Tasks/Features

Object Tree possesses the following tasks and features:

- The main task of Object Tree is to edit a set of existing SPECS-C++ specification objects, and to build new ones.
- Object Tree must manipulate a variety of nested SPECS-C++ specification objects.
- Object Tree is a graphical user interface that displays SPECS-C++ specification
objects to the user and increases the user’s understanding of the object’s structure.

- The functionality of the Object Tree will evolve rapidly over time.
- It is easy to extend and customize Object Tree. Object Tree can be refined further by adding more design options.

3.2 Contents

One of the best ways to display specification objects is to utilize a tree hierarchy, as this will make it easy for users to understand the object’s structure and thus edit objects more efficiently. Each tree node represents the value of a specification object, and each child represents a nested (contained) object.

There are two types of objects included in this study: specification objects and tree objects. Tree objects are used to display specification objects graphically. Specification objects are used in SPECS-C++ to model aliasing and mutation at the specification level. **Aliasing** means that an object can have multiple names (i.e., as two different pointers to the same object). **Mutation** occurs when the value held by an object changes between the pre-state and the post-state. An object is a container for an instance of a class or any other value.

The rest of the SPECS-C++ execution system uses specification objects only. Hence, Object Tree must convert specification objects to tree objects for display and editing. After editing, Object Tree must convert tree objects back to specification objects to return them to the rest of the system.

Converting Specification Objects to Tree Objects

11
A specification object is essentially a long list whose contents (specification object sublists) are separated by spaces. Each specification object sublist begins with the left parenthesis “(“ and ends with the right parenthesis “)”, and has two major components: a reference r(id) and a specification object value. A comma separates these components. The id (reference) is a memory location or cell for the object, and the value is the value contained in the object (stored at that location). To convert a specification object to a tree object, a procedure is used to construct two lists. One list contains tree object values and the other contains the child nodes for each reference in the tree object values. Specification object values are converted to tree object values by means of another procedure that uses regular expressions for constructing tree object values. A third procedure is used to combine the two lists as a single list, with tree object values placed first followed by locations and child nodes. This combined list is a tree object sublist. Tree object sublists for the child nodes are combined recursively to form the complete tree object.

A typical example of this process is as follows.

**Specification object:**

This object contains an instance of class ObjSetTable (Figure 3).

```
"(r(0),tuple([set([n(1),set([r(2),r(3)])],tuple([n(2),set([r(3),r(4)])]]),
tuple([n(3),set([n(5)])],tuple([n(5),set([r(2),r(4),r(5)])]]))) (r(2),n(4)) (r(3),n(5))
(r(4),n(4)) (r(5),n(7))"
```

In the above, r(0) is the object at the root of the tree. In our notation, r(0) means object (memory cell) 0. Object r(0) contains a tuple which contains a set of data. The
set contains 4 tuples. Each tuple has two fields – an integer value and a set of objects containing integers.

The converted tree object appears in Figure 4. Textually, this tree object is represented as:

```
“{(1,{},2,{},3,{},{5,{},},) 0 {{4 2 {8} 5 3 {8} 5 3 {8}}
{4 4 {8} 4 2 {8} 4 4 {8} 7 5 {8}}} 56”
```

**Figure 4:** "Object Tree" displaying an object containing an instance of class ObjSetTable (after converting it to a tree object).

There are four major parts in each tree node: The first is the node value (i.e., 
```
((1,{},2,{},3,{},{5,{},},))
```
). The second is the object identifier or location for the node value. The third consists of a list of the node’s children (i.e., 
```
{{4 2 {8} 5 3 {8} 5 3 {8}}
{4 4 {8} 4 2 {8} 4 4 {8} 7 5 {8}}
```
). The fourth is the width of the node’s value when displayed in the tree (i.e., 56). Though each node
has two coordinates, x and y, the width here is calculated based on (horizontal) x coordinates only.

There are two possible scenarios related to the width as follows:

- The first occurs when a node has no children at all. Then, the width of the node’s value is just the number of characters in the value plus some padding to make the tree easy to read.

- The second occurs when a node has children. In this case, the width of the node is the larger of the number of characters in the node (plus padding), and the sum of the widths of the children (again, plus padding).

To draw a parent’s node value on the screen and to position it at the center above all its child nodes, we need to know the widths of the node’s value and of its children. Because these calculations involve the widths of the children (which recursively are based on the widths of grandchildren and so on), we calculate widths once and store them in the tree. This avoids repeated (inefficient) calculation of the width of the same subtree. The line connecting a parent node and a child node must originate from the correct position in the parent’s node value and point to the child’s node value.

Converting Tree Objects Back to Specification Objects

A tree object list is generally enclosed by curly brackets ({})) and consists of the tree object value, id, children and width in that order. The converting of a tree object into a specification object initiates with leaf nodes. If the child node list of a tree object sublist is not empty, then a recursive procedure is called to convert it. On the other hand, if a tree object sublist has no children, then another procedure is called to construct the specification object
sublist. If the value of the tree object sublist is one of the following three strings: {}, <>, or (),
then it can be directly converted into a specification object sublist, i.e. (r(id),set([])),
(r(id),seq([])), or (r(id),tuple([])). If there is content (such as a comma or space) within one of
the strings for the tree object sublist value, then a third procedure is called to build the
specification object sublist. This procedure analyzes the first character of a value, e.g., {, <, or (, and then tree object sublist value is converted to the specification object sublist, e.g.,
(r(id),set([sublist])), (r(id),seq[sublist]), or (r(id), tuple([sublist])). A fourth recursive procedure
is called to build the specification object sublist for the elements of a set, sequence or tuple value.
In this procedure, each element of the value is extracted, and then each element is analyzed in
more detail. Finally, each element is converted into a specification object sublist, and all sublists
are appended recursively, to form a specification object list.

If we use these algorithms to convert a specification object to a tree object, and then back
to a specification object, we will end up with the same specification object that we started with
(assuming it was not edited).

3.3 Operations

Object Tree provides a selection menu that allows users to choose operations to apply
to a tree. The selection menu consists of the following buttons: Quit, Delete Tree, Original
Tree, Del All, Del Node, Add Node, Update Node, Undo, and Done. Note that Del All,
Del Node, Add Node and Update Node are radio buttons meaning that one of these
buttons must be clicked before executing an action in a tree.

• Quit: to exit from Object Tree.
• Delete Tree: to delete the whole tree from the screen.
• Original Tree: to return to the original tree (used for developmental purposes).
• Del All: to delete the whole tree when any node is clicked.
• Del Node: to delete the subtree rooted at the selected node from the tree.
• Add Node: to add a child to the selected node.
• Update Node: to update (change the value of) the selected node.
• Undo: to undo the previous action.
• Done: to exit and print out the result.

3.4 Rules

The user will be alerted by the sound of a beep and the appearance of an error-message window if a mistake is made. To make the error-message window disappear, one can simply click the “Retry” button in the window and then proceed with re-execution. If the user does not make a mistake at the beginning, then he can edit specification objects with a simple menu selection from Object Tree. In general, the following two rules must be followed:

• Input Value Rule

User input values must be one of the following forms:

Numbers:

Numerical signed or unsigned decimal values.

For example, a number cannot contain an alphabetic character or be empty.

Structured Values:

The structured values are sets, sequences, and tuples. Sets are represented as curly brackets ({{}}) around a list of values (elements of the set). Sequences are
enclosed in angle brackets (<>), and tuples in parentheses. For example, 

\{(1,\langle 2,3,7\rangle),(2,\langle 4,6,7\rangle)\} \) is a set containing two tuples. Each tuple contains an integer and a sequence of integers.

• Operation Rule

If the node is a number, only the Update Node, Del Node or Del All options are applicable. In other words, the “Add-Node” option cannot be used. For example, trying to add a child node to the node whose value is “77” will cause an error.

On the other hand, if the node contains a structured value, only the Add Node, Del Node or Del All options can be used. A child node must be added on the right-hand side. For example, trying to update a node whose value is “\{\}” will cause an error.

3.5 Implementations

Each language has strengths and weaknesses making it appropriate for certain tasks. To build an application, one needs to decide which language will make the most sense for the application. To have a powerful and flexible drawing interface, one can apply the programming language TCL/TK [Harrison 1997] [Ousterhout 1994] [Welch 1997]. TK is a toolkit for programming graphical user interfaces. It provides a set of commands that create and manipulate widgets. A widget is a window in a graphical user interface that has a particular appearance and behavior. TK also has a general-purpose drawing widget called a canvas that allows for the creation of lines, boxes and bitmaps.

The advantages of the TCL/TK programming system that make it suitable for implementing Object Tree include [TCL Developer Xchange 2001]:

17
• With its TK toolkit, TCL provides facilities for creating graphical user interface (GUIs) in a very simple form but without sacrificing the power of such interfaces.

• With TCL, the implementation or modification of applications is rapid. As a rule of thumb, it is 5-10 times faster as compared to other languages. This is particularly evident if the application involves GUIs, string-handling, or integration of multiple components written in different languages.

• TCL is an interpreted scripting language that can be easily extended and modified.

• With TCL, coordinating existing components and applications and making them work together effectively is an easy task. For example, one can use TCL as a control language for special-purpose hardware and protocols, for adding a GUI or network interface to an application, or for integrating new JAVA applications with C or C++. In the SPECS-C++ execution system, TCL is used to integrate the specification compiler and the AKL interpreter.

• TCL is very easy to learn. In a typical situation, TCL is first applied by more experienced programmers to create a basic family of facilities. The casual programmers then write TCL scripts to customize those facilities, create their own business rules, … etc.

• The community of TCL users and developers is large and helpful. It is a reliable source of ideas, free extensions, applications, and technical support.

A comparison of TCL with other popular scripting languages is shown in Table 1 [TCL Developer Xchange 2001]. Since TCL/TK is powerful, flexible, extensible and fairly easy to apply, it is utilized in this study to develop the graphical user interface Object Tree.
<table>
<thead>
<tr>
<th>Features</th>
<th>Tcl</th>
<th>Perl</th>
<th>Java Script</th>
<th>Visual Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of use</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rapid development</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flexible, quick evolution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Regular expressions</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth of functionality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easily extensible</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embeddable</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy GUIs</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet and Web-enabled</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Enterprise usage</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cross platform</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Internationalization support</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Thread safe</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Database access</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. TCL Comparison Chart

4. DISCUSSION

Available systems for executing formal specifications that have a graphical interface include the B-Toolkit Animator [B-Core 2001], the animator for IFAD VDM-SL [Elmstrom et al. 1994; IFAD 2000], and Schach and Gray’s animator for UML [Gray and Schach 2000]. Most of the published work on executable specifications emphasizes the execution technique and/or the subset of the language that can be executed, except for the last one cited above. Therefore, it is not easy to directly compare the proposed interface model Object Tree with the others. However, none of the interfaces mentioned above have support for displaying and editing nested objects.

One possibility is to compare the B-Toolkit with Object Tree, as both deal with structured data. With the B-Toolkit, a graph can be automatically generated to view the development of specifications and designs. More detail can also be viewed using its zoom option. However,
unlike Object Tree, the B-Toolkit cannot edit specification objects, nor is it capable of representing a nested object using a tree structure.

5. CONCLUSIONS AND RECOMMENDATIONS

The interface model proposed in this study is capable of editing specification object values. It has the unique combination of features not available in other interfaces for executing formal specifications. It represents nested objects graphically using a tree structure that is appealing and easy to read and understand. It also allows users to construct trees with nodes of object values compactly. Hence, Object Tree makes running the AKL interpreter and interpreting the result of executing a specification much easier and more efficient. As described and demonstrated above, the proposed interface model is an improved one as it widens the possibilities for the use of formal specifications in practice.

Some improvements can be added to the interface model developed. For instance, the interface could be modified to allow the reuse of layout information when a tree changes. This will avoid the reconstruction of layout or redrawing of a tree if there is a change only in one subtree [Moen 1990]. Another possible improvement concerns the Add Node option. Presently, a child node can only be added to the rightmost position of the parent node. Object Tree can be revised to permit the addition of a child node at any position within the node.

Though the graphical user interface Object Tree is a convenient tool for constructing and validating specification objects, one still needs human intelligence and thought in order to fully utilize it. In other words, Object Tree may be imagined as a living tree best described in the famous poem [Paz 1988]:
“A tree grew inside my head.

A tree grew in.

Its roots are veins,

Its branches nerves,

thoughts its tangled foliage.

....... 

There, within, inside my head,

the tree speaks.

Come closer – can you hear it?”

REFERENCES


