Black-Box Software Testing using Module Interface Specifications

Ruth F. Abraham
McMaster University, Hamilton, Ontario, Canada, L8S 4K1

1 Introduction

Throughout the lifecycle of a software project, many different testing techniques can and should be used. At a high level, the various methods can be categorized as either *random testing*, in which the test data is selected at random, or *systematic testing*, in which the selection of input data follows guidelines based on assumptions about the nature of faults [10]. There are two major techniques for the systematic selection of test data: *black-box* testing and *clear-box* testing. In black-box testing, the program implementation is ignored and the module internal design remains hidden. Test cases are chosen solely based on information contained in specification documents. This is contrary to clear-box testing, in which the specification is ignored and tests are chosen using detailed knowledge of the module's source code and internal data structure [7]. These two techniques complement each other and are usually both employed in the testing of any large scale project.

In practice, exhaustive black-box testing would involve more test cases than one could feasibly consider. It is necessary to choose a manageable set of test cases that will maximize fault detection. As highlighted in [9], the difficulties encountered when using black-box methods may be largely due to the fact that there is no "standard" notation for the specifications of the software to be tested. A module interface specification is often non-existent, or if present, it may be incomplete or imprecise. It has been hypothesized that the use of natural language in specifications may hinder the formulation of precise descriptions of programs; formal language descriptions may overcome this problem.

The Software Engineering Research Group at McMaster University, in its effort to improve the quality of software documentation, advocates the use of tabular expressions to represent the mathematical functions and relations necessary to document computer systems. The *Table-Tool System* is intended to provide software developers with a practical tool for producing and using better documentation. It will accomplish this by providing services, such as table input, output formatting and table comparison [4]. As it is the group's mandate to use the methods it develops, the Table-Tool System will itself be formally documented. The Table Holder, which will store tabular expressions without interpreting them, is the first "building-block" of the system. The interface to this module is formally specified using the *Trace Assertion Method* [6].

This report describes black-box testing with interface specifications, using the Table Holder interface as an example. Section 2 gives an overview of the specification method and Section 3 describes the testing techniques and tools used. Section 4 describes the testing process. The limitations of this method and ideas for future work are discussed in Section 5, followed by conclusions in Section 6. This report uses the Table Holder testing only as an illustration of this method. A complete test report is contained in [1].

2 Trace Specifications

2.1 Module Interface Specification

A *module* is a group of programs that share access to a "private" data structure. Each instantiation of this data structure is an *object* of that module. The set of assumptions that an external program can make about a module is called a *module interface*, and the interface belongs to the module which provides the services. The *module interface specification* treats the module as a black-box in that it does not reveal the module's secret (i.e. private data structures). It instead specifies an object by its externally observable behaviour, identifying all programs that can be called from outside the module (*access-programs*) and describing their visible effects [5]. The module specification does not include how the module is, or should be implemented. The Table Holder module interface specification is documented using the trace assertion method.

2.2 Trace Assertion Method

The *trace assertion method* [6] provides a means for producing a black-box description of an object that is precise enough to support systematic analysis. The externally observable aspects of a module are *events*, which are invocations of the module's access-programs. An object is modelled by a finite state machine where the input is represented by event expressions, and the output is represented by the output variables. A *trace* is a finite sequence of event expressions (i.e. program calls) separated by ".", the string concatenation operator [8].

A trace provides a complete history of the externally visible behaviour of an object, including all events which affect it and all outputs that it produces. An empty sequence of events is denoted by "_", the *empty trace*. An object can only have a fixed number of distinct states, therefore many traces denote the same state. Equivalent traces have identical output and identical future behaviour, and a single trace, called the *canonical trace*, is chosen to represent each equivalence class [8]. Every trace is equivalent to a single canonical trace.

In the trace assertion method, an extension table is given for each access-program. The table describes an extension function which maps single element extensions of canonical traces to the equivalent canonical trace. For each canonical trace, the output values must also be specified. The domain of each extension table is the canonical trace set.

2.3 Document Structure

A *trace module interface specification* is a document which uses the trace assertion method to specify, for each canonical trace and extension combination, the behaviour of the module. In the specification, access-programs are described using tables. The four sections of the module interface specification document, as discussed in [2], are Syntax, Canonical Traces, Equivalences, and Values.

2.3.1 Syntax Section

The Syntax section contains an access-program table which specifies the type of the arguments and return values for each access-program. The input-output characteristics of arguments are represented by the descriptors “O”, “V”, and “N” [2]. Each argument may have one or more descriptors, to be interpreted as follows:

- “O” indicates an output argument. A name must be provided, and the value associated with this name may be changed.
- “V” indicates an input argument. A value must be provided, or a name from which a value can be obtained. The value can not change unless also marked “O”.
- “N” indicates name-dependent behaviour. A name must be provided, and the value associated with this name is irrelevant unless also marked “V”. The value can not change unless also marked “O”.

2.3.2 Canonical Traces Section

This section defines a predicate “canonical”, which is the characteristic predicate of the set of all canonical traces.

2.3.3 Equivalences Section

The Equivalences section contains extension tables which define the extension function for each access-program. The left column of each table contains predicates that partition the domain of canonical traces into mutually exclusive rows (i.e. for a given set of input data, only one row in each table can evaluate to “true”). The expression in the right column is the equivalent canonical trace, or an error token if this is undesired behaviour for the module [6].

2.3.4 Values Section

A table in the Values section specifies how output values are returned by access-programs. The *value* of an object is defined as the canonical trace which is equivalent to the history of the object [2]. The output value may be the return value of the program or the value of a particular argument. A function for the output value is listed for each applicable access-program. This may be in the form of an *auxiliary function*, which is defined for canonical traces using the table format described in section 2.3.3.

2.4 Error Cases

A trace specification must include error detection, yet need not specify how an error is handled by the implementation. An error occurs when the single element extension of a canonical trace defines behaviour that is undesired, in which case the trace is illegal. *Legal traces* are those for which the module is expected to be useful. *Illegal traces* contain events that the user of the module is supposed to avoid [6].

Output values may not be defined for illegal traces. Even though illegal traces will not occur if the module is used correctly, it is a good practice to require a module to give an error indication if the trace is illegal. This is usually communicated by the presence of an error token in the Equivalences tables (e.g. %error-name%). Where errors are detected, if no equivalent canonical trace is given, then it is assumed that the state of the object does not change.

3 Testing Tool

A testing tool has been developed which accepts traces as input, performs the required program calls, and monitors the state of the object. The tool provides a trace interpreter that reads traces and makes calls to the Table Holder interface (the code is contained in a library called `libt.a`). An introduction to the Table Holder modules is given in Appendix A. The state of the object should be known at all times in order to determine the equivalence of different traces and to detect undesired side effects. This section provides a description of the testing tool, with a depiction of the tool given in Figure 1, “Testing Tool”, on page 5.

3.1 Program Variables

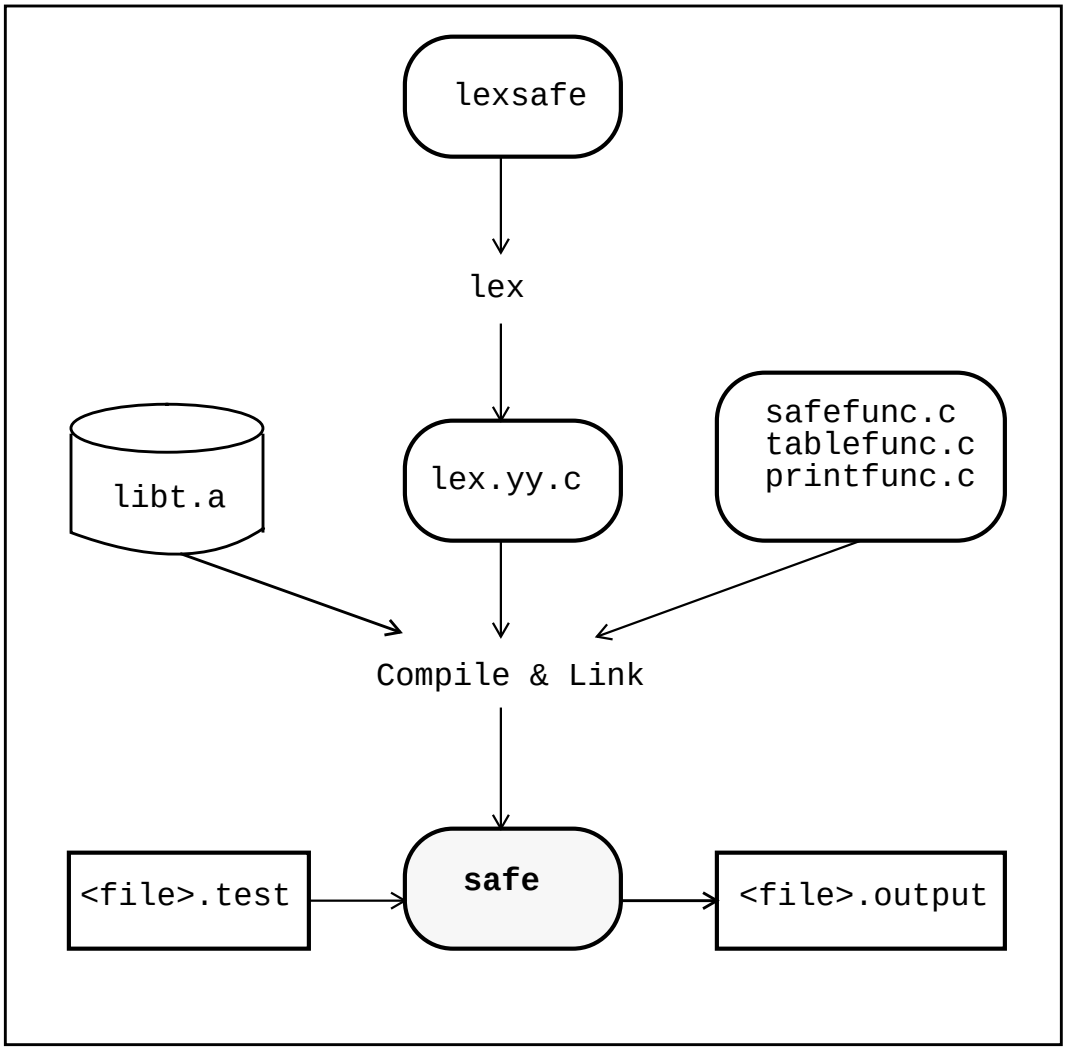
In the testing environment, access-program arguments are recognized as being either integers or strings. The strings are members of a predefined set of variable names belonging to each module. These variables may have a value associated with them, which is stored in a table, using the variable name as a key into this table. Arguments of type

`int` are not able to be represented by a variable name in this environment. Integer arguments must be actual integer values.

In order that the values of all variables be retained throughout each trace, a lookup table was created to store the variable names with their reference. The algorithm used is a hash search derived from one detailed in [3]. Each entry in the hash table is a C structure containing four elements: a string denoting the variable name, a tag denoting the type of user defined variable (either Index, Path, Name, Expn or Tag), a pointer to the variable itself, being a union of all necessary types, and a pointer to the next entry in the chain. Two major routines are used to manipulate the table entries:

- `install(s, t, u)` records the name `s`, the tag `t`, and the value `u` in the table.
- `lookup(s)` searches for the variable name `s` in the table, returning either a pointer to where it was found, or the value `NULL` if string `s` has not yet been installed.

FIGURE 1. Testing Tool



A variable name will not be recognized as correct input by the testing tool until it is installed in the lookup table (similar to declaring a variable name in C code). The declarations of hash table functions and structures are contained in `tablefunc.h` as in Appendix B, and the function code is in the file `tablefunc.c`, as included in Appendix C.

3.2 Printing Tools

Rudimentary printing tools have been developed for each of the four modules. The purpose of these print functions is simply to display the state of an object, with minimal emphasis on format. Since the Table Holder interface is the only means used to gain access to the object, only those values returned by access-programs are printed. During testing, it is useful to display an object's contents after an event which may change the state of the object occurs (i.e. arguments having the descriptor "O"). In special cases, such as copy programs, both objects may be printed before and after the copy is executed, to ensure that no undesired side effects occur. The source code file `printfunc.c` is contained in Appendix C.

3.3 Lexical Parser

The Unix lexical analysis program generator, `lex`, was chosen for the initial stage of reading the traces from a text file. The C program generated by `lex` searches the input stream for regular expressions described in a file, in this case a file named `lexsafe`. If an expression match is made, the corresponding C action specified in `lexsafe` is taken. The two categories of expressions searched for are: 1) administrative keywords; and 2) access-program names.

3.3.1 Administrative Keywords

When the special keywords listed in `lexsafe` are recognized, the actions taken mainly control the scope of variable names. For example, the keyword `START` initializes all hash table entries to `NULL`, and then installs the Tag names in the table for later lookup. Each of the four interface modules has two keywords associated with it:

- `INIT*` installs a predefined set of variable names in the lookup table,
- `REM*` removes the variable names from the scope of the testing tool,

where "`*`" represents the initial letter of the module name. In the current version there are five variables installed per module. For example, whenever the keyword `INITP` is recognized, the strings `p1`, `p2`, `p3`, `p4`, and `p5` are installed in the table as Path variables with an initial value of `NULL.` The keyword `REMP` will remove these variables from the table. Other keywords act to display test case numbering and any unrecognized text.

3.3.2 Access-Program Names

When an access-program name is recognized, the specified action is to call an intermediate program which is passed a text string including the access-program name and the argument list. Intermediate programs are necessary to ease debugging (otherwise all actions would be contained in one function), and to simplify modifications to individual actions by bypassing the `lex` program generation stage. The name of an intermediate program is very similar to its corresponding access-program's name, with the exception of a leading "r" and the omission of all capital letters (i.e. an access-program named `SampAccProg()` would have an intermediate program called `rsampaccprog()`).

The tasks performed by an intermediate program are as follows:

- parse the recognized string into a program name and its arguments.
- check that variable names are valid (i.e. installed in the hash table).
- call the access-program.
- print a copy of the program call, all return values, and any other information about the object in question that may prove helpful in testing the behaviour of the access-program.

Intermediate programs check input data to insure that the traces are syntactically correct. The source code for the intermediate programs are contained in the file `safefunc.c`, as in Appendix C.

3.4 Test Case Format

The testing program is called `safe`, and can be run interactively, but for the purposes of formal testing the test cases are grouped into files and used as input to `safe`. In the test case file, a trace is written as a list of event expressions (program name and arguments) separated by the "." symbol. The last event of any given trace is the access-program being tested, as it is desired to observe how the occurrence of this event changes the state of the object.

The use of administrative keywords controls the scope of variables. When testing the Index module, for example, it is important to reinstall the Index variable names after every test case to ensure that the initial state of the module be identical for every trace. Often, values belonging to a module other than the module being tested are needed. If these values are used throughout a particular group of tests, and the Syntax table in the specification marks the arguments as simply "V", then it is necessary to install and assign values to the variable names only at the beginning of the test file. If, however, an access-

program argument belonging to another module is marked “NO” or “VO”, then it is necessary to reinstall that module’s variable names between traces as well.

The test cases are stored in files named `<file>.test`, and the test results stored in `<file>.output`, where `<file>` is a name that indicates which module, or even access-program, is being tested. Each test case is uniquely numbered within a file. The output results include a listing of all program calls with their relevant output information (as printed by the intermediate programs), as well as all error messages received from the code and the operating system. At present, no automated checker exists to validate the results, therefore the expected results must be calculated from the specification and used to manually check the test results. The purpose is not simply to identify defects in the code, but to identify all discrepancies between the specification and the code. These differences should be resolved in order that the interface is accurately documented.

4 Testing Process

This section describes the testing strategy employed when using the testing tool. The module interface specifications are followed to select the data for test cases. The specifications are also used to determine whether test cases pass or fail. Examples of the type of discrepancies found through this testing process are discussed at the end of this section.

4.1 Test Case Selection

Some guidelines have been followed when selecting input data for testing each access-program. The trace specification provides all the information needed to generate most test cases. Each access-program is tested using it as a single-element extension to canonical traces. The group of canonical traces, together with their single element extensions (the access-program being tested) should contain all of the following cases:

- the input trace as the empty trace.
- the input trace as every major form of the canonical trace set.
- input which is invalid according to the specification but which is valid according to the code’s access-program declaration alone.
- each condition row in the Equivalences table is explicitly tested (the specification is written such that any given row will describe at most one error condition).
- within each condition row, various ranges and subsets of input data are tested, with concentrated testing around boundary values.

- error causing data is only given to the access-program being tested. The preceding program calls contained in the trace are only given what is thought to be “correct” data, therefore isolating the condition being tested.
- if it becomes apparent that an error exists, extra test cases are selected for this condition to determine the extent of the discrepancy.

4.2 Pass Criteria

A test is successful if the object’s behaviour is the same as the behaviour documented in the trace specification. The expected behaviour for each test is manually determined from the specification, and the test is said to either pass or fail based on a comparison of the actual and expected results. A failure indicates a discrepancy between the specification and the code. However, there are often assumptions made about what is correct behaviour when an error occurs or when invalid data is given. The following criteria are used to determine success or failure:

legal traces

- the state of the object must be identical to the corresponding equivalent canonical trace.
- all arguments that should not change must retain the same value.
- all specified return values must be equal to that determined from the Values section.

illegal traces

- for all conditions that give error tokens, the user must be notified that an error occurred.
- error messages should preferably refer to the access-program in which the error occurred, but reference to a lower level routine is also accepted.
- the state of the object should remain unchanged after an error occurs, according to the specification for this project.

4.3 Typical Discrepancies

This section gives an overview of the general types of discrepancies found as a result of the black-box testing of the Table Holder interface, many of which are common to more than one module. The complete test report is contained in [1]. The following discrepancies may have been more difficult to detect if the interface was not formally specified.

- Programs that destroy an object are equivalent to the empty trace in the specification, yet destroyed variables may still be valid program arguments in the code.
- Programs that copy objects are designed differently in the specification and the code. The specification places no restrictions on the variable to be copied into, whereas the

code requires that it have the same type, the same Tag, and (where applicable) the same length as the value being copied into it.

- Programs that get a value do not behave as specified for the case where the value requested has not yet been set. The code often returns the value 0 or -1, whereas the specification either raises an error token or is not yet defined for this condition. If the code exits for this error, testing identified which programs still required a method to determine whether the value had been set.
- The use of tags is initially very confusing to a user. The specification has three different kinds of tags, whereas the code has only one named Tag, which is the union of the previous three. Valid input for an argument of type Tag can only be one of the three sets of tags, but input values are checked for validity by the access-program, not the compiler.
- Maximum values are not consistently represented. The specification has one maximum value which the code does not enforce. The code has maximum limits on five values, but these are not documented in the specification.
- An object of type `Name` contains specific information on the attributes of an `Expn` object. According to the specification, an expression must follow these restrictions. The code stores but does not apply this information to the expression.
- A well-defined object is one that has a value assigned to all of its elements. Using not well-defined indices as arguments for some access-programs is an error in the specification. The code allows the use of not well-defined indices in almost all cases, often causing either immediately invalid results, or later errors in subsequent calls to other access-programs.
- The specification often checks that an argument (not of type Tag) has the correct Tag associated with it. The code for CreateExpn does not ensure that its Index argument has the correct Tag, causing spurious errors in the subsequent use of that expression. The existence of a close link is identified between two “independent” modules.
- Code design has not been properly implemented in programs when Segmentation Fault errors occur for valid input. Access-programs in one module do not interact correctly, as changing the order of independent programs calls causes such errors.

5 Lessons Learned

The purpose of this testing exercise was not solely to discover faults in the Table Holder code and interface specification, but also to learn more about the use of trace specifications in black-box testing. This section discusses the lessons learned through using this method, and suggests some future improvements for the testing tool.

5.1 Limitations of Method

- A module implementation may handle errors by displaying an error message and exiting to the operating system. This method of handling errors is incompatible with black-box testing using trace specifications, as there is no way to check the state of an object after an error occurs. In the specification, an error token indicates an illegal trace, and the Table Holder interface specification also indicates that the state of the object should remain unchanged after an error occurs. The Table Holder code handles most errors by exiting with an error message.
- It is helpful to have some working knowledge of what the modules are intended to be used for. Too much knowledge may hinder a literal understanding of the specification, as one might make broad assumptions. However, it is useful to be able to make note of cases in which the code and the specification agree, but may both be wrong.
- Ideally the specification is completed prior to the implementation, but circumstances may dictate that the specification and the code for a project be written in parallel. For historical reasons, there were cases in this software project where the code and the specification had equally good and complete, but different, designs for an access-program. Here, testing is given the job of discovering major design differences that should have already been resolved, instead of discovering more subtle discrepancies.
- The use of the printing tools to aid testing is somewhat limiting, as the tools use the access-programs that are being tested. A defect may remain hidden, or a discrepancy may be attributed to the wrong access-program, due to the use of “non-tested” functions to display the state of an object.
- In testing the Table Holder interface, special input data had to be chosen for programs that printed expressions, otherwise the printing routine might have stopped execution. One access-program necessary to completely print an expression will crash if the information is not set, yet there is no program to check whether or not the information is set before calling it.
- The tester must use caution when describing a discrepancy not to hypothesize about the cause of the error. Since a black-box tester has no knowledge of the implementation, it is more helpful to state what is known about the conditions in which an error occurs, rather than to theorize about its probable cause.

5.2 Future Ideas for Testing Tool

- In the `lex` file, `lexsafe`, it was necessary to have a rule to recognize each access-program name. The use of the Unix parsing tool, `yacc` (yet another compiler-compiler), together with `lex`, would have simplified this process by searching instead for a grammar rule, i.e., access-program name with arguments.

- When the code handles errors by exiting, the testing process is greatly hindered. A test group has to be rerun after every error case occurs, along with careful provision that global variables for that group are reset to the correct values. This could be avoided by adding a higher-level program that calls the testing program and continues execution even after exit routines are called.
- Presently, integer values must be input directly as integers. It might prove helpful to have the capacity for integers to be represented by variables.
- The capacity might be added to have programs as arguments to other programs recognized, although this is not commonly used in trace specification notation. This would require `lex` to recognize nested patterns.
- Trace specifications often use traces as arguments, but this is not yet supported by the testing tool. The access-program calls that the trace represents must always be explicitly present.
- The choice of variables names might become more accessible to the tester, allowing quick name changes and names which carry more significance.
- The use of `NULL` is permitted in the code for “don’t care” arguments and empty paths, but it is not supported by the testing tool.

6 Conclusions

The use of trace specifications in black-box testing provides a means of systematically testing for both error conditions and the correct usage of a module. No knowledge of the implementation is needed, as the behaviour of the module is completely and precisely detailed by the interface specification. This means a more accurate analysis of the code’s behaviour can be performed than would be possible without specifications documents, because a standard exists against which to test.

The trace specification is written in such a way that possible error ranges are already identified. By testing to the specification, fewer test cases were needed to discover these errors.

Testing of the Table Holder interface did not include exhaustive black-box testing, yet an average of about 15 test cases per access-program did uncover more than 50 discrepancies between the Table Holder code and interface specification. A design review is currently in process, and a comparison of the results of this black-box testing exercise with the results of an independent clear-box inspection reveals that many faults were common to both reports. However, many discrepancies were discovered by only one of the two methods, showing a clear need for both, and also showing the importance of the use of trace specifications for accurate software testing.

Acknowledgements

I am very grateful to Yabo Wang for sharing his knowledge of the trace assertion method, to Dennis Peters for his suggestions on implementing the testing tool, to Philip Kelly for his helpful comments on software testing, and to David Parnas for the opportunity to work with the Software Engineering Research Group at McMaster University. This work was supported by the Government of Ontario through the Telecommunications Research Institute of Ontario (TRIO).

References

1. Abraham, R.F., "Table Holder Interface Black-Box Testing Report", *CRL Report No. 268*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, April 1993.

2. Iglewski, M., Madey, J., Parnas, D.L. and Kelly, P., "Documentation Paradigms (A Progress Report)", *CRL Technical Report*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada; *in preparation*

3. Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1978.

4. Parnas, D.L. and Bharadwaj, K., "The Table-Tool System", *CRL Technical Report*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada; *in preparation*

5. Parnas, D.L. and Madey, J., "Functional Documentation for Computer Systems Engineering. (Version 2)", *CRL Report No. 237*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, September 1991.

6. Parnas, D.L. and Wang, Y., "The Trace Assertion Method of Module Interface Specification", *Technical Report 89-261*, Telecommunications Research Institute of Ontario (TRIO), C&IS, Queen's University, Kingston, ON, Canada, October 1989.

7. Schach, S.R., *Software Engineering*, Aksen Associates Inc., Boston, MA, USA, 1990.

8. Wang, Y. and Parnas, D.L., "Trace Rewriting Systems", *CRL Report No. 247*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, October 1992.

9. Woit, D.M., "An Analysis of Black-Box Testing Techniques", *CRL Report No. 245*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, May 1992.

10. Woit, D.M., "Realistic Expectations of Random Testing", *CRL Report No. 24*6, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, ON, Canada, May 1992.

Appendix A - Introduction to the Table Holder Interface

TABLE 1. Index Module Syntax

Program	Value	Arg#1	Arg#2	Arg#3
CreateIndex		<Index>:NO	<IType>:V	<int>
DestroyIndex		<Index>:VO		
CopyIndex		<Index>:V	<Index>:NO	
AssignEltIndex		<Index>:VO	<int>	<int>
GetEltIndex	<int>	<Index>:V	<int>	
GetTagIndex	<IType>	<Index>:V		
GetLenIndex	<int>	<Index>:V		
IncIndex	<bool>	<Index>:VO	<int>	<Index>:V

TABLE 2. Path Module Syntax

Program	Value	Arg#1	Arg#2	Arg#3
CreatePath		<Path>:NO		
DestroyPath		<Path>:VO		
AssignIndexPath		<Path>:VO	<int>	<Index>:V
InsertIndexPath		<Path>:VO	<int>	<Index>:V
DeleteIndexPath		<Path>:VO		
GetLenPath	<int>	<Path>:V		
GetITagEltPath	<IType>	<Path>:V	<int>	
GetLenIndexPath	<int>	<Path>:V	<int>	
GetIndexPath		<Index>:NO	<Path>:V	<int>

The Table Holder is comprised of four modules; *Index*, *Path*, *Name*, and *Expn*, with between eight and thirteen access-programs each. The Syntax tables from each of the module interface specifications are shown in Table 1 through Table 4. It is known that lower level routines are called by these modules, but these routines are not formally specified and are not to be explicitly black-box tested.

The Table-Tool system is being implemented in the C programming language using the Sun-Unix operating system. The Table Holder code is contained in a library called `libt.a`. For the purposes of interfacing to the Table Holder, the header files of the four modules are available to the programmer. The header files are used to validate argument types against function prototypes, but the actual data structures are declared elsewhere. Some object types can be imported into other modules; the Expn module takes arguments of type Index, Path and Name. Another user defined variable type is the enumerated

variable type *Tag*, which can take any of fourteen values. Upon creation, a variable of type Index, Name or Expn is assigned a Tag in order to distinguish among different classes of these objects. The three kinds of Tags are denoted in the specification as *IType*, *NType*, and *EType*.

TABLE 3. Name Module Syntax

Program	Value	Arg#1	Arg#2	Arg#3
CreateName		<Name>:NO	<NType>:V	
DestroyName		<Name>:VO		
GetTagName	<NType>	<Name>:V		
ConToName		<Name>:NO	<NType>:V	<int>
ConFromName	<int>	<Name>:V	<NType>:V	
CopyName		<Name>:V	<Name>:NO	
SetArityName		<Name>:VO	<int>	
SetNoArityName		<Name>:VO		
GetArityName	<int>	<Name>:V		
SetTypedName		<Name>:VO	<bool>	
CheckTypedName	<bool>	<Name>:V		
AssignArgTypeName		<Name>:VO	<int>	<EType>:V
GetArgTypeName		<EType>:NO	<Name>:V	<int>

TABLE 4. Expn Module Syntax

Program	Value	Arg#1	Arg#2	Arg#3	Arg#4
CreateExpn		<Expn>:NO	<EType>:V	<Index>:V	
DestroyExpn		<Expn>:VO			
GetTagSubExpn	<EType>	<Expn>:V	<Path>:V		
CopySubExpn		<Expn>:NV	<Path>:V	<Expn>:NO	<Path>:V
EqualSubExpn	<bool>	<Expn>:V	<Path>:V	<Expn>:V	<Path>:V
ConToAtom		<Expn>:NO	<Path>:V	<EType>:V	<int>
ConFromAtom	<int>	<Expn>:V	<Path>:V	<EType>:V	
AssignNameApp		<Expn>:VO	<Path>:V	<Name>:V	
GetNameApp		<Name>:VO	<Expn>:V	<Path>:V	
GetArityApp	<int>	<Expn>:V	<Path>:V		
GetDimGrid	<int>	<Expn>:V	<Path>:V		
GetShapeGrid		<Index>:VO	<Expn>:V	<Path>:V	
GetNGridsTable	<int>	<Expn>:V	<Path>:V		

Appendix B - Testing Tool Header Files

tablefunc.h

```
#ifndef _tablefunc_h_
#define _tablefunc_h_
#include “types.h”
#include “index.h”
#include “path.h”
#include “name.h”
#include “expn.h”

#define STRSIZE 10
#define HASHSIZE 100

static struct nlist *hashtab[HASHSIZE];

typedef enum {
    TAG, INDEX, PATH, NAME, EXPN
} ttag;

union thitypes {
    Tag ta;
    Index in;
    Path pa;
    Name na;
    Expn ex;
};

struct nlist {
    char token[STRSIZE];
    ttag token_tag;
    union thitypes info;
    struct nlist *next;
};

extern int hash(char *);
extern struct nlist *lookup(char *);
extern struct nlist *install(char *, ttag, union thitypes *);
extern void remove(char *);
extern void starthash();
extern void endhash();

#endif
```

printfunc.h

```
#ifndef _printfunc_h_
#define _printfunc_h_
#include “types.h”
#include “index.h”
#include “path.h”
#include “name.h”
#include “expn.h”

void printtag(Tag T);
void printIndex(Index I);
void printpath(Path P);
void printname(Name N);
void printtypes(Types T);
void printexpn(Expn E, Path P);

#endif
```

safefunc.h

```
#ifndef _safefunc_h_
#define _safefunc_h_
#include “types.h”
#include “index.h”
#include “path.h”
#include “name.h”
#include “tablefunc.h”

void rcreatename(char text[STRSIZE]);
void rdestroyname(char text[STRSIZE]);
void rgettagname(char text[STRSIZE]);
void rcontoname(char text[STRSIZE]);
void rconfromname(char text[STRSIZE]);
void rcopyname(char text[STRSIZE]);
void rsetarityname(char text[STRSIZE]);
void rsetnoarityname(char text[STRSIZE]);
void rgetarityname(char text[STRSIZE]);
void rsettypedname(char text[STRSIZE]);
void rchecktypedname(char text[STRSIZE]);
void rassignargtypename(char text[STRSIZE]);
void rgetargtypename(char text[STRSIZE]);
void rcreateindex(char text[STRSIZE]);
void rdestroyindex(char text[STRSIZE]);
void rcopyindex(char text[STRSIZE]);
void rassigneltindex(char text[STRSIZE]);
void rgeteltindex(char text[STRSIZE]);
void rgettagindex(char text[STRSIZE]);
void rgetlenindex(char text[STRSIZE]);
void rincindex(char text[STRSIZE]);
void rcreatepath(char text[STRSIZE]);
void rdestroypath(char text[STRSIZE]);
void rassignindexpath(char text[STRSIZE]);
void rinsertindexpath(char text[STRSIZE]);
void rdeleteindexpath(char text[STRSIZE]);
void rgetlenpath(char text[STRSIZE]);
void rgetitageltpath(char text[STRSIZE]);
void rgetlenindexpath(char text[STRSIZE]);
void rgetindexpath(char text[STRSIZE]);
void rcreateexpn(char text[STRSIZE]);
void rdestroyexpn(char text[STRSIZE]);
void rgettagsubexpn(char text[STRSIZE]);
void rcopysubexpn(char text[STRSIZE]);
void requalsubexpn(char test[STRSIZE]);
void rcontoatom(char text[STRSIZE]);
void rconfromatom(char text[STRSIZE]);
void rassignnameapp(char text[STRSIZE]);
void rgetnameapp(char text[STRSIZE]);
void rgetarityapp(char text[STRSIZE]);
void rgetdimgrid(char text[STRSIZE]);
void rgetshapegrid(char text[STRSIZE]);
void rgetngridstable(char text[STRSIZE]);

#endif
```

Appendix C - Testing Tool Source Code

lexsafe

```
%p 4500
%e 1500
%n 700
    #include <stdio.h>
    #include “tablefunc.h”
    #include “safefunc.h”
    Index i1, i2, i3, i4, i5;
    Path p1, p2, p3, p4, p5;
    Name n1, n2, n3, n4, n5;
    Expn e1, e2, e3, e4, e5;
%%
“START”     starthash();
“END”       endhash();
“INITI”     {install(“i1”, INDEX, (union thitypes *) &i1);
            lookup(“i1”)->info.in = NULL;
            install(“i2”, INDEX, (union thitypes *) &i2);
            lookup(“i2”)->info.in = NULL;
            install(“i3”, INDEX, (union thitypes *) &i3);
            lookup(“i3”)->info.in = NULL;
            install(“i4”, INDEX, (union thitypes *) &i4);
            lookup(“i4”)->info.in = NULL;
            install(“i5”, INDEX, (union thitypes *) &i5);
            lookup(“i5”)->info.in = NULL;}
“REMI”      {remove(“i1”);
            remove(“i2”);
            remove(“i3”);
            remove(“i4”);
            remove(“i5”);}
“INITP”     {install(“p1”, PATH, (union thitypes *) &p1);
            lookup(“p1”)->info.in = NULL;
            install(“p2”, PATH, (union thitypes *) &p2);
            lookup(“p2”)->info.in = NULL;
            install(“p3”, PATH, (union thitypes *) &p3);
            lookup(“p3”)->info.in = NULL;
            install(“p4”, PATH, (union thitypes *) &p4);
            lookup(“p4”)->info.in = NULL;
            install(“p5”, PATH, (union thitypes *) &p5);
            lookup(“p5”)->info.in = NULL;}
“REMP”      {remove(“p1”);
            remove(“p2”);
            remove(“p3”);
            remove(“p4”);
            remove(“p5”);}
“INITN”     {install(“n1”, NAME, (union thitypes *) &n1);
            lookup(“n1”)->info.na = NULL;
            install(“n2”, NAME, (union thitypes *) &n2);
            lookup(“n2”)->info.na = NULL;
            install(“n3”, NAME, (union thitypes *) &n3);
            lookup(“n3”)->info.na = NULL;
            install(“n4”, NAME, (union thitypes *) &n4);
            lookup(“n4”)->info.na = NULL;
            install(“n5”, NAME, (union thitypes *) &n5);
            lookup(“n5”)->info.na = NULL;}
“REMN”      {remove(“n1”);
            remove(“n2”);
            remove(“n3”);
            remove(“n4”);
            remove(“n5”);}
“INITE”     {install(“e1”, EXPN, (union thitypes *) &e1);
            lookup(“e1”)->info.ex = NULL;
            install(“e2”, EXPN, (union thitypes *) &e2);
            lookup(“e2”)->info.ex = NULL;
            install(“e3”, EXPN, (union thitypes *) &e3);
            lookup(“e3”)->info.ex = NULL;
            install(“e4”, EXPN, (union thitypes *) &e4);
            lookup(“e4”)->info.ex = NULL;
            install(“e5”, EXPN, (union thitypes *) &e5);
            lookup(“e5”)->info.ex = NULL;}
“REME”      {remove(“e1”);
            remove(“e2”);
            remove(“e3”);
            remove(“e4”);
            remove(“e5”);}
“CreateIndex(&”[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
            rcreateindex(yytext);}
“DestroyIndex(“[^\.\)]+”)” {
            rdestroyindex(yytext);}
“CopyIndex(“[^\.\)]+”,”[^\.\)]+”)” {
            rcopyindex(yytext);}
```

```
“AssignEltIndex(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rassigneltindex(yytext);}
“GetEltIndex(“[^\.\)]+”,”[^\.\)]+”)” {
          rgeteltindex(yytext);}
“GetTagIndex(“[^\.\)]+”)” {
          rgettagindex(yytext);}
“GetLenIndex(“[^\.\)]+”)” {
          rgetlenindex(yytext);}
“IncIndex(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rincindex(yytext);}
“CreatePath(&”[^\.\)]+”)” {
          rcreatepath(yytext);}
“DestroyPath(“[^\.\)]+”)” {
          rdestroypath(yytext);}
“AssignIndexPath(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rassignindexpath(yytext);}
“InsertIndexPath(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rinsertindexpath(yytext);}
“DeleteIndexPath(“[^\.\)]+”,”[^\.\)]+”)” {
          rdeleteindexpath(yytext)}
“GetLenPath(“[^\.\)]+”)” {
          rgetlenpath(yytext);}
“GetITagEltPath(“[^\.\)]+”,”[^\.\)]+”)” {
          rgetitageltpath(yytext);}
“GetLenIndexPath(“[^\.\)]+”,”[^\.\)]+”)” {
          rgetlenindexpath(yytext);}
“GetIndexPath(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rgetindexpath(yytext);}
“.”       printf(“\n”);
“#”[0-9]+ {printf(“%s “, yytext)}
[A-Za-z0-9]+ printf(“unrecognized text: %s\n”, yytext);
“CreateName(&”[^\.\)]+”,”[^\.\)]+”)” {
          rcreatename(yytext);}
“DestroyName(“[^\.\)]+”)” {
          rdestroyname(yytext);}
“GetTagName(“[^\.\)]+”)” {
          rgettagname(yytext);}
“ConToName(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rcontoname(yytext);}
“ConFromName(“[^\.\)]+”,”[^\.\)]+”)” {
          rconfromname(yytext);}
“CopyName(“[^\.\)]+”,”[^\.\)]+”)” {
          rcopyname(yytext);}
“SetArityName(“[^\.\)]+”,”[^\.\)]+”)” {
          rsetarityname(yytext);}
“SetNoArityName(“[^\.\)]+”)” {
          rsetnoarityname(yytext);}
“GetArityName(“[^\.\)]+”)” {
          rgetarityname(yytext);}
“SetTypedName(“[^\.\)]+”,”[^\.\)]+”)” {
          rsettypedname(yytext);}
“CheckTypedName(“[^\.\)]+”)” {
          rchecktypedname(yytext);}
“AssignArgTypeName(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rassignargtypename(yytext);}
“GetArgTypeName(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rgetargtypename(yytext);}
“CreateExpn(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rcreateexpn(yytext);}
“DestroyExpn(“[^\.\)]+”)” {
          rdestroyexpn(yytext);}
“GetTagSubExpn(“[^\.\)]+”,”[^\.\)]+”)” {
          rgettagsubexpn(yytext);}
“CopySubExpn(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rcopysubexpn(yytext);}
“EqualSubExpn(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          requalsubexpn(yytext);}
“ConToAtom(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rcontoatom(yytext);}
“ConFromAtom(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rconfromatom(yytext);}
“AssignNameApp(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rassignnameapp(yytext);}
“GetNameApp(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rgetnameapp(yytext);}
“GetArityApp(“[^\.\)]+”,”[^\.\)]+”)” {
          rgetarityapp(yytext);}
“GetDimGrid(“[^\.\)]+”,”[^\.\)]+”)” {
          rgetdimgrid(yytext);}
“GetShapeGrid(“[^\.\)]+”,”[^\.\)]+”,”[^\.\)]+”)” {
          rgetshapegrid(yytext);}
“GetNGridsTable(“[^\.\)]+”,”[^\.\)]+”)” {
          rgetngridstable(yytext);}
%%
```

tablefunc.c

```
/*********************************************************
* tablefunc.c
* Functions to create and use a symbolic lookup table.
* A hash search is conducted on a string, where the hashval
* of the string is a key into the table. Each table entry
* stores the string, a tag, the value associated with the
* string, and a pointer to the next entry.
*
* Ruth Abraham        CRL     93.02.01
*********************************************************/
#include <stdio.h>
#include "tablefunc.h"

/***** hash computes the hashval of a string ************/
int
hash(char *s)
{
    int hashval;

    for (hashval = 0; *s != '\0';)
        hashval += *s++;
    return (hashval % HASHSIZE);
}


/***** lookup determines whether or not a string is located
****** in the hash table, returning a pointer to the nlist */
struct nlist *
lookup(char *s)
{

    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->token) == 0)
            return (np);
    return (NULL);
}


/***** install stores the values associated with a string
****** token, either replacing old values, or creating a
****** new table entry **********************************/
struct nlist *
install(char *token, ttag token_tag, union thitypes * info)
{

    struct nlist *np;
    int hashval;

    if ((np = lookup(token)) == NULL) {
        np = (struct nlist *) malloc(sizeof(struct nlist));
        if (np == NULL) {
            return (NULL);
        }
        strcpy(np->token, token);
        hashval = hash(np->token);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    }
    np->info = *info;
    np->token_tag = token_tag;

    return (np);
}

/***** remove removes a string and its values from the hash
****** table ******************************************/
void
remove(char *s)
{

    struct nlist *np, *prev;
    int hashval = hash(s);

    if (lookup(s) == NULL) {
        printf("%s not installed, cannot remove\n", s);
        return;
    } else {
        prev = hashtab[hashval];
        if (strcmp(s, prev->token) == 0) {
            hashtab[hashval] = prev->next;
            free(prev);
        } else {
```

```
            for (np = prev->next; np != NULL; np = np->next) {
                if (strcmp(s, np->token) == 0) {
                    prev->next = np->next;
                    free(np);
                    return;
                }
                prev = np;
            }
        }
    }
}


/***** starthash initializes all hashtable elements to zero
****** and installs all Tag names in the hash table *******/
void
starthash()
{

    int i = 0;
    Tag t = ConstTag;

    for (i = 0; i < HASHSIZE; i++) {
        hashtab[i] = NULL;/* initialize table */
    }

    install("ConstTag", TAG, (union thitypes *) & t);
    t = VarTag;
    install("VarTag", TAG, (union thitypes *) & t);
    t = FATag;
    install("FATag", TAG, (union thitypes *) & t);
    t = FTableTag;
    install("FTableTag", TAG, (union thitypes *) & t);
    t = PETag;
    install("PETag", TAG, (union thitypes *) & t);
    t = LETag;
    install("LETag", TAG, (union thitypes *) & t);
    t = PdTableTag;
    install("PdTableTag", TAG, (union thitypes *) & t);
    t = GridTag;
    install("GridTag", TAG, (union thitypes *) & t);
    t = FNameTag;
    install("FNameTag", TAG, (union thitypes *) & t);
    t = PdNameTag;
    install("PdNameTag", TAG, (union thitypes *) & t);
    t = LONameTag;
    install("LONameTag", TAG, (union thitypes *) & t);
    t = AIndexTag;
    install("AIndexTag", TAG, (union thitypes *) & t);
    t = GIndexTag;
    install("GIndexTag", TAG, (union thitypes *) & t);
    t = TIndexTag;
    install("TIndexTag", TAG, (union thitypes *) & t);
}


/***** endhash removes all Tag names from the table ******/
void
endhash()
{

    remove("ConstTag");
    remove("VarTag");
    remove("FATag");
    remove("FTableTag");
    remove("PETag");
    remove("LETag");
    remove("PdTableTag");
    remove("GridTag");
    remove("FNameTag");
    remove("PdNameTag");
    remove("LONameTag");
    remove("AIndexTag");
    remove("GIndexTag");
    remove("TIndexTag");

}
```

printfunc.c

```
/*********************************************************
* printfunc.c
* Functions to print types Types, Tag, Index, Path, Name
* and Expn to standard output.
*
* Ruth Abraham     CRL     Created: 93.01.27
*                          Edited: 93.03.01
*********************************************************/
#include <stdio.h>
#include “types.h”
#include “index.h”
#include “path.h”
#include “name.h”
#include “expn.h”

/*********************************************************/
void
printtag(Tag T)
{
    switch (T) {
        case ConstTag: printf(“ConstTag “);
                       return;
        case VarTag:   printf(“VarTag “);
                       return;
        case FATag:    printf(“FATag “);
                       return;
        case FTableTag:printf(“FTableTag “);
                       return;
        case PETag:    printf(“PETag “);
                       return;
        case LETag:    printf(“LETag “);
                       return;
        case PdTableTag:printf(“PdTableTag “);
                       return;
        case GridTag:  printf(“GridTag “);
                       return;
        case FNameTag:printf(“FNameTag “);
                       return;
        case PdNameTag:printf(“PdNameTag “);
                       return;
        case LONameTag:printf(“LONameTag “);
                       return;
        case AIndexTag:printf(“AIndexTag “);
                       return;
        case GIndexTag:printf(“GIndexTag “);
                       return;
        case TIndexTag:printf(“TIndexTag “);
                       return;
        case -1:       printf(“** -1 ** “);
                       return;
        default:       printf(“Unknown Tag “);
                       return;
        }
}

/*********************************************************/
void
printindex(Index I)
{

    int i = 0;

    switch (GetTagIndex(I)) {
    case AIndexTag:
    case TIndexTag:
        printf(“<%d> “, GetEltIndex(I, 1));
        break;
    case GIndexTag:
        printf(“<“);
        for (i = 1; i < GetLenIndex(I); i++) {
            printf(“%d,”, GetEltIndex(I, i));
        }
        printf(“%d> “, GetEltIndex(I, i));
        break;
    default:
        printf(“illegal index tag “);
        break;
    }
}

/*********************************************************/
void
printpath(Path P)
{
```

```
    Index I;
    int i = 0;

    if ((GetLenPath(P) == 0) || (P == NULL)) {
        printf("empty path ");
        return;
    }
    printf("<");
    for (i = 1; i <= (GetLenPath(P)); i++) {
        switch (GetITagEltPath(P, i)) {
        case AIndexTag:
        case TIndexTag:
            CreateIndex(&I, GetITagEltPath(P, i), 1);
            GetIndexPath(I, P, i);
            printindex(I);
            if (i < GetLenPath(P))
                printf(",");
            else
                printf("> ");
            DestroyIndex(I);
            break;
        case GIndexTag:
            CreateIndex(&I, GIndexTag, GetLenIndexPath(P, i));
            GetIndexPath(I, P, i);
            printindex(I);
            if (i < GetLenPath(P))
                printf(",");
            else
                printf("> ");
            DestroyIndex(I);
            break;
        default:
            printf("illegal index tag\n");
            break;
        }
    }
    return;
}

/*********************************************************/
void
printname(Name N)
{

    printtag(GetTagName(N));
    printf("%c %d ", ConFromName(N, GetTagName(N)), GetArityName(N));
    if (CheckTypedName(N)) {
        printf("Type attribute enabled.");
    }
}

/*********************************************************/
void
printtypes(Types T)
{

    printf("{");
    if (TagInT(T, ConstTag))
        printf("ConstTag ");
    if (TagInT(T, VarTag))
        printf("VarTag ");
    if (TagInT(T, FATag))
        printf("FATag ");
    if (TagInT(T, FTableTag))
        printf("FTableTag ");
    if (TagInT(T, PETag))
        printf("PETag ");
    if (TagInT(T, LETag))
        printf("LETag ");
    if (TagInT(T, PdTableTag))
        printf("PdTableTag ");
    if (TagInT(T, GridTag))
        printf("GridTag ");
    if (TagInT(T, FNameTag))
        printf("FNameTag ");
    if (TagInT(T, PdNameTag))
        printf("PdNameTag ");
    if (TagInT(T, LONameTag))
        printf("LONameTag ");
    if (TagInT(T, AIndexTag))
        printf("AIndexTag ");
    if (TagInT(T, GIndexTag))
        printf("GIndexTag ");
    if (TagInT(T, TIndexTag))
        printf("TIndexTag ");
    printf("} ");
```

```
}

/*********************************************************/
void
printexpn(Expn E, Path P)
{

    int i = 0;
    Name N;
    Index I, SHAPE;

    if (P == NULL) {
        CreatePath(&P);
    }
    if (GetTagSubExpn(E, P) != -1) {

        switch (GetTagSubExpn(E, P)) {

        case ConstTag:
            printf("%c ", ConFromAtom(E, P, ConstTag));
            break;
        case VarTag:
            printf("%c ", ConFromAtom(E, P, VarTag));
            break;
        case FATag:
            CreateName(&N, FNameTag);
            GetNameApp(N, E, P);
            printf("%c( ", ConFromName(N, FNameTag));
            for (i = 1; i <= GetArityApp(E, P); i++) {
                CreateIndex(&I, AIndexTag, 1);
                AssignEltIndex(I, i, 1);
                InsertIndexPath(P, (GetLenPath(P) + 1), I);
                printexpn(E, P);
                DestroyIndex(I);
                DeleteIndexPath(P, GetLenPath(P));
            }
            printf(") ");
            DestroyName(N);
            break;
        case PETag:
            CreateName(&N, PdNameTag);
            GetNameApp(N, E, P);
            printf("%c{ ", ConFromName(N, PdNameTag));
            printf("{ ");
            for (i = 1; i <= GetArityApp(E, P); i++) {
                CreateIndex(&I, AIndexTag, 1);
                AssignEltIndex(I, i, 1);
                InsertIndexPath(P, (GetLenPath(P) + 1), I);
                printexpn(E, P);
                DestroyIndex(I);
                DeleteIndexPath(P, GetLenPath(P));
            }
            printf("} ");
            DestroyName(N);
            break;
        case LETag:
            CreateName(&N, LONameTag);
            GetNameApp(N, E, P);
            printf("%c( ", ConFromName(N, LONameTag));
            for (i = 1; i <= GetArityApp(E, P); i++) {
                CreateIndex(&I, AIndexTag, 1);
                AssignEltIndex(I, i, 1);
                InsertIndexPath(P, (GetLenPath(P) + 1), I);
                printexpn(E, P);
                DestroyIndex(I);
                DeleteIndexPath(P, GetLenPath(P));
            }
            printf(") ");
            DestroyName(N);
            break;
        case PdTableTag:
        case FTableTag:
            printf("\n[");
            for (i = 0; i <= (GetNGridsTable(E, P) - 1); i++) {
                CreateIndex(&I, TIndexTag, 1);
                AssignEltIndex(I, i, 1);
                InsertIndexPath(P, (GetLenPath(P) + 1), I);
                printexpn(E, P);
                DestroyIndex(I);
                DeleteIndexPath(P, GetLenPath(P));
            }
            printf("]\n");
            break;
        case GridTag:
            CreateIndex(&SHAPE, GIndexTag, GetDimGrid(E, P));
```

```
            GetShapeGrid(SHAPE, E, P);
            printindex(SHAPE);
            CreateIndex(&I, GIndexTag, GetDimGrid(E, P));
            for (i = 1; i <= GetDimGrid(E, P); i++) {
                AssignEltIndex(I, 1, i);
            }
            printf("[ ");
            do {
                InsertIndexPath(P, (GetLenPath(P) + 1), I);
                printexpn(E, P);
                DeleteIndexPath(P, GetLenPath(P));
            } while (IncIndex(I, 1, SHAPE));
            DestroyIndex(I);
            DestroyIndex(SHAPE);
            printf("]\n");
            break;
        case -1:
            printf("__ ");
            break;
        default:
            printf("ERROR : not an Expression");
            break;
        }
    } else {
        printf("__ ");
    }
}
```

safefunc.c

```
/*********************************************************
* safefunc.c
* Functions to execute lex recognized commands
* corresponding to THI Expression Modules programs.
* Each function parses a string into a program name and its
* arguments, checks the validity of variable names, calls
* the program, and prints the object or the output value.
*
* Ruth Abraham         CRL     93.02.25
*********************************************************/
#include <stdio.h>
#include "bool.h"
#include "types.h"
#include "index.h"
#include "path.h"
#include "name.h"
#include "printfunc.h"
#include "tablefunc.h"
#include "safefunc.h"

char tstr[STRSIZE], istr[STRSIZE], dstr[STRSIZE];
char estr[STRSIZE], nstr[STRSIZE], pstr[STRSIZE], c;
struct nlist *tokp, *desp, *tagp, *exp, *namp;
int j, k;
bool b;

/********** INDEX module access programs *****************/

void rcreateindex(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CreateIndex(&%s , %s , %d)", istr, tstr, &k);
    if (((tokp = lookup(istr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        CreateIndex(&(tokp->info.in), tagp->info.ta, k);
        printindex(tokp->info.in);
        printtag(tagp->info.ta);
    } else {
        printf("testing error: %s is an illegal tag type\n", tstr);
    }
    return;
}

void rdestroyindex(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "DestroyIndex(%s )", istr);
    if ((tokp = lookup(istr)) != NULL) {
        DestroyIndex(tokp->info.in);
    } else {
        printf("testing error: %s is an illegal index type\n", istr);
    }
    return;
}

void rcopyindex(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CopyIndex(%s , %s )", istr, dstr);
    if (((tokp = lookup(istr)) != NULL) && ((desp = lookup(dstr)) != NULL)) {
```

```
        CopyIndex(tokp->info.in, desp->info.in);
        printindex(tokp->info.in);
        printindex(desp->info.in);
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
    return;
}

void rdestroypath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "DestroyPath(%s )", pstr);
    if ((tokp = lookup(pstr)) != NULL) {
        DestroyPath(tokp->info.pa);
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
    return;
}

void rassignindexpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "AssignIndexPath(%s , %d , %s )", pstr, &j, istr);
    if (((tokp = lookup(pstr)) != NULL) && ((desp = lookup(istr)) != NULL)) {
        AssignIndexPath(tokp->info.pa, j, desp->info.in);
        printpath(tokp->info.pa);
    } else {
        printf("testing error: either path or index is illegal\n");
    }
    return;
}

void rinsertindexpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "InsertIndexPath(%s , %d , %s )", pstr, &j, istr);
    if (((tokp = lookup(pstr)) != NULL) && ((desp = lookup(istr)) != NULL)) {
        InsertIndexPath(tokp->info.pa, j, desp->info.in);
        printpath(tokp->info.pa);
    } else {
        printf("testing error: either path or index is illegal\n");
    }
    return;
}

void rdeleteindexpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "DeleteIndexPath(%s , %d)", pstr, &j);
    if ((tokp = lookup(pstr)) != NULL) {
        DeleteIndexPath(tokp->info.pa, j);
        printpath(tokp->info.pa);
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
    return;
}

void rgetlenpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetLenPath(%s )", pstr);
    if ((tokp = lookup(pstr)) != NULL) {
        printf("%d ", GetLenPath(tokp->info.pa));
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
    return;
}

void rgetitageltpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetITagEltPath(%s , %d)", pstr, &j);
    if ((tokp = lookup(pstr)) != NULL) {
        printtag(GetITagEltPath(tokp->info.pa, j));
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
    return;
}

void rgetlenindexpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetLenIndexPath(%s , %d)", pstr, &j);
    if ((tokp = lookup(pstr)) != NULL) {
        printf("%d ", GetLenIndexPath(tokp->info.pa, j));
    } else {
        printf("testing error: %s is an illegal path\n", pstr);
    }
```

```
    return;
}

void rgetindexpath(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetIndexPath(%s , %s , %d)", istr, pstr, &j);
    if (((tokp = lookup(pstr)) != NULL) && ((desp = lookup(istr)) != NULL)) {
        GetIndexPath(desp->info.in, tokp->info.pa, j);
        printindex(desp->info.in);
    } else {
        printf("testing error: either path or index is illegal\n");
    }
    return;
}

/********** NAME module access programs ******************/

void rcreatename(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CreateName(&%s , %s )", nstr, tstr);
    if (((tokp = lookup(nstr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        CreateName(&(tokp->info.na), tagp->info.ta);
        printname(tokp->info.na);
    } else {
        printf("testing error: either name or tag is illegal");
    }
    return;
}

void rdestroyname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "DestroyName(%s )", nstr);
    if ((tokp = lookup(nstr)) != NULL) {
        DestroyName(tokp->info.na);
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rgettagname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetTagName(%s )", nstr);
    if ((tokp = lookup(nstr)) != NULL) {
        printtag(GetTagName(tokp->info.na));
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rcontoname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "ConToName(%s , %s , %c)", nstr, tstr, &c);
    if (((tokp = lookup(nstr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        ConToName(tokp->info.na, tagp->info.ta, c);
        printname(tokp->info.na);
    } else {
        printf("testing error: either name or tag is illegal");
    }
    return;
}

void rconfromname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "ConFromName(%s , %s )", nstr, tstr);
    if (((tokp = lookup(nstr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        printf("%c ", ConFromName(tokp->info.na, tagp->info.ta));
    } else {
        printf("testing error: either name or tag is illegal");
    }
    return;
}

void rcopyname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CopyName(%s , %s )", nstr, dstr);
    if (((tokp = lookup(nstr)) != NULL) && ((desp = lookup(dstr)) != NULL)) {
        printname(tokp->info.na);
        printname(desp->info.na);
        CopyName(tokp->info.na, desp->info.na);
        printname(tokp->info.na);
        printname(desp->info.na);
    } else {
        printf("testing error: one name type is illegal");
    }
    return;
```

```
}

void rsetarityname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "SetArityName(%s , %d)", nstr, &k);
    if ((tokp = lookup(nstr)) != NULL) {
        SetArityName(tokp->info.na, k);
        printname(tokp->info.na);
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rsetnoarityname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "SetNoArityName(%s )", nstr);
    if ((tokp = lookup(nstr)) != NULL) {
        SetNoArityName(tokp->info.na);
        printname(tokp->info.na);
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rgetarityname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetArityName(%s )", nstr);
    if ((tokp = lookup(nstr)) != NULL) {
        printf("%d ", GetArityName(tokp->info.na));
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rsettypedname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "SetTypedName(%s , %s )", nstr, dstr);
    if ((tokp = lookup(nstr)) != NULL) {
        if (strcmp(dstr, "TRUE") == 0) {
            b = TRUE;
        } else if (strcmp(dstr, "FALSE") == 0) {
            b = FALSE;
        } else {
            printf("testing error: %s is illegal bool type\n", dstr);
            return;
        }
        SetTypedName(tokp->info.na, b);
        printname(tokp->info.na);
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rchecktypedname(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CheckTypedName(%s )", nstr);
    if ((tokp = lookup(nstr)) != NULL) {
        if (CheckTypedName(tokp->info.na)) {
            printf("TRUE ");
        } else {
            printf("FALSE ");
        }
        printname(tokp->info.na);
    } else {
        printf("testing error: %s is an illegal name\n", nstr);
    }
    return;
}

void rassignargtypename(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "AssignArgTypeName(%s , %d , %s )", nstr, &k, tstr);
    if (((tokp = lookup(nstr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        printname(tokp->info.na);
    } else {
        printf("testing error: either name or tag is illegal");
    }
    return;
}

void rgetargtypename(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetArgTypeName(%s , %s , %d)", tstr, nstr, &k);
```

```
    if (((tokp = lookup(nstr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        printname(tokp->info.na);
    } else {
        printf("testing error: either name or tag is illegal");
    }
    return;
}

/********** EXPN module access programs ******************/

void rcreateexpn(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CreateExpn(&%s , %s , %s )", estr, tstr, istr);
    if (((exp = lookup(estr)) != NULL) && ((tagp = lookup(tstr)) != NULL) &&
     ((tokp = lookup(istr)) != NULL)) {
        CreateExpn(&exp->info.ex, tagp->info.ta, tokp->info.in);
        printtag(tagp->info.ta);
        printindex(tokp->info.in);
        printtag(GetTagSubExpn(exp->info.ex, NULL));
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rdestroyexpn(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "DestroyExpn(%s )", estr);
    if ((exp = lookup(estr)) != NULL) {
        DestroyExpn(exp->info.ex);
    } else {
        printf("testing error: %s is an illegal expn\n", estr);
    }
    return;
}

void rgettagsubexpn(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetTagSubExpn(%s , %s )", estr, pstr);
    if ((exp = lookup(estr)) != NULL) {
        desp = lookup(pstr);
        printtag(GetTagSubExpn(exp->info.ex, desp->info.pa));
        printexpn(exp->info.ex, desp->info.pa);
    } else {
        printf("testing error: %s is an illegal expn\n", estr);
    }
    return;
}

void rcopysubexpn(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "CopySubExpn( %s , %s , %s , %s )", estr, pstr, istr, dstr);
    if (((exp = lookup(estr)) != NULL) && ((tokp = lookup(istr)) != NULL) &&
     ((tagp = lookup(dstr)) != NULL)) {
        desp = lookup(pstr);
        /*
         * printexpn(exp->info.ex, desp->info.pa);
         * printexpn(tokp->info.ex, NULL);
         */
        CopySubExpn(exp->info.ex, desp->info.pa, tokp->info.ex, tagp->info.pa);
        /*
         * printexpn(exp->info.ex, desp->info.pa);
         * printexpn(tokp->info.ex, NULL);
         */
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void requalsubexpn(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "EqualSubExpn( %s , %s , %s , %s )", estr, pstr, istr, dstr);
    if (((exp = lookup(estr)) != NULL) && ((tokp = lookup(istr)) != NULL)) {
        desp = lookup(pstr);
        tagp = lookup(dstr);
        if (EqualSubExpn(exp->info.ex, desp->info.pa, tokp->info.ex, tagp->info.pa)) {
            printf("TRUE\n");
        } else {
            printf("FALSE\n");
        }
        printexpn(exp->info.ex, desp->info.pa);
        printexpn(tokp->info.ex, tagp->info.pa);
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
```

```
}

void rcontoatom(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "ConToAtom( %s , %s , %s , %c)", estr, pstr, tstr, &c);
    if (((exp = lookup(estr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        desp = lookup(pstr);
        ConToAtom(exp->info.ex, desp->info.pa, tagp->info.ta, c);
        printf("%c ", ConFromAtom(exp->info.ex, desp->info.pa, tagp->info.ta));
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rconfromatom(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "ConFromAtom( %s , %s , %s )", estr, pstr, tstr);
    if (((exp = lookup(estr)) != NULL) && ((tagp = lookup(tstr)) != NULL)) {
        desp = lookup(pstr);
        printf(".%c. ", ConFromAtom(exp->info.ex, desp->info.pa, tagp->info.ta));
        printexpn(exp->info.ex, desp->info.pa);
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rassignnameapp(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "AssignNameApp( %s , %s , %s )", estr, pstr, nstr);
    if (((exp = lookup(estr)) != NULL) && ((namp = lookup(nstr)) != NULL)) {
        desp = lookup(pstr);
        AssignNameApp(exp->info.ex, desp->info.pa, namp->info.na);
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rgetnameapp(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetNameApp( %s , %s , %s )", nstr, estr, pstr);
    if (((exp = lookup(estr)) != NULL) && ((namp = lookup(nstr)) != NULL)) {
        desp = lookup(pstr);
        GetNameApp(namp->info.na, exp->info.ex, desp->info.pa);
        printname(namp->info.na);
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rgetarityapp(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetArityApp( %s , %s )", estr, pstr);
    if ((exp = lookup(estr)) != NULL) {
        desp = lookup(pstr);
        printf("arity = %d ", GetArityApp(exp->info.ex, desp->info.pa));
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rgetdimgrid(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetDimGrid( %s , %s )", estr, pstr);
    if ((exp = lookup(estr)) != NULL) {
        desp = lookup(pstr);
        printf("%d ", GetDimGrid(exp->info.ex, desp->info.pa));
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}

void rgetshapegrid(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetShapeGrid( %s , %s , %s )", istr, estr, pstr);
    if (((exp = lookup(estr)) != NULL) && ((tokp = lookup(istr)) != NULL)) {
        desp = lookup(pstr);
        GetShapeGrid(tokp->info.in, exp->info.ex, desp->info.pa);
        printindex(tokp->info.in);
    } else {
        printf("testing error: illegal argument input\n");
    }
```

```
    return;
}

void rgetngridstable(char text[STRSIZE])
{   printf("%s ", text);
    sscanf(text, "GetNGridsTable( %s , %s )", estr, pstr);
    if ((exp = lookup(estr)) != NULL) {
        desp = lookup(pstr);
        printf("%d ", GetNGridsTable(exp->info.ex, desp->info.pa));
    } else {
        printf("testing error: illegal argument input\n");
    }
    return;
}
```