GENERATING A TEST ORACLE FROM PROGRAM DOCUMENTATION

By
DENNIS PETERS, B. ENG.

A Thesis

Submitted to the School of Graduate Studies

in partial fulfilment of the requirements

for the degree of

Master of Engineering

McMaster University

© Copyright by Dennis Keith Peters, April, 1995

MASTER OF ENGINEERING(1995) McMASTER UNIVERSITY
(Computer)

TITLE: Generating a Test Oracle From Program Documentation

AUTHOR: Dennis Keith Peters, B. Eng. (Memorial University of Newfoundland)

SUPERVISOR: Dr. David L. Parnas

NUMBER OF PAGES: xii, 97

Abstract

Software testing involves execution of a program under test using some fault revealing input data and examination of the output to determine success or failure. A fundamental assumption of this testing is that there is some mechanism, an *oracle*, that will determine whether or not the results of a test execution are correct. In practice, this is often done by comparing the output, either automatically or manually, to some pre-calculated, presumably correct, output [39]. However, if the program is formally documented it is possible to use the specification to determine the success or failure of a test execution, as in [1], for example. This thesis discusses the development of a prototype tool that automatically generates a test oracle from formal program documentation.

In [25], [27] and [28] Parnas et al. advocate the use of a relational model for documenting the intended behaviour of programs. In this method, tabular expressions are used to improve readability so that formal documentation can replace conventional documentation. Relations are described by giving their characteristic predicate in terms of the values of concrete program variables. This documentation method has the advantage that the characteristic predicate can be used as a test oracle—it must be evaluated for each test execution (input and output) to assign pass or fail. This form of documentation is used for generating an oracle.

The design of a test oracle and a tool that can be used to generate an oracle are discussed in this thesis.

Acknowledgements

I would like to express my sincere appreciation for the assistance and guidance of Dr. David L. Parnas in the preparation of this thesis.

Thoughtful comments from Dr. Jeffery I. Zucker and Dr. Arne Maus have helped to clarify the ideas expressed in this work. The efforts of Brian Smith and Jonathan Bosloy, both of Newbridge Networks Corporation, made it possible for me validate my methods using industrial software. Ruth Abraham and Doris Burns both helped to improve the quality of this work through their careful proofreading and constructive criticism.

Finally, I gratefully acknowledge the financial assistance received from the Natural Sciences and Engineering Research Council (NSERC) and the Telecommunications Research Institute of Ontario (TRIO).

Table of Contents

Abstract iii

Acknowledgements iv

Table of Contents v

List of Figures ix

List of Tables x

List of Acronyms xi

1 Introduction 1
- 1.1 Purpose 2
- 1.2 Scope 3
- 1.3 Related Work 4
- 1.4 Outline of This Thesis 6

2 Notation and Terminology 7
- 2.1 Predicate Logic 7
  - 2.1.1 Notational Conveniences 8
  - 2.1.2 Quantified Expressions 8
  - 2.1.3 Inductively Defined Predicates 9
- 2.2 Tabular Expressions 10
  - 2.2.1 Syntax of Grids and Tables 10
  - 2.2.2 Semantics of Tables 11
  - 2.2.3 Expressions 12
- 2.3 Relational Specification 12
  - 2.3.1 Limited Domain Relations 13
- 2.4 Program Variables and State Descriptions 14
  - 2.4.1 Before and After Value 15
- 2.5 Functional Testing 15
- 2.6 Test Oracle 16
- 2.7 Test Harness 17

3 Program Documentation Method 19
- 3.1 Primitives 19

3.1.1 Data Types 20
3.1.2 Primitive Functions..20
3.1.3 Primitive Predicates 20
3.2 Documentation Components..21
3.2.1 Constants..21
3.2.2 Variables..21
3.2.3 Program Specifications 22
3.2.4 Auxiliary Predicate Definitions 22
3.2.5 Auxiliary Function Definitions 23
3.2.6 Inductively Defined Predicate Definitions..23
3.2.7 User Definitions 23
3.3 Sample Program Documentation 24

4 Oracle Design 25
4.1 Programming Language..25
4.2 Interface 25
4.3 Internal Design Overview 27
4.3.1 Expression Implementation 28
4.4 Scalar Expressions 29
4.4.1 Logical Operators 29
4.4.2 Primitive Relations 30
4.4.3 Inductively Defined Predicates 30
4.4.4 Quantification..32
4.5 Tabular Expressions 33
4.6 Auxiliary Predicates and Functions 34
4.7 Compilation and Execution 35

5 Test Oracle Generator Design..37
5.1 Requirements 37
5.1.1 Assumptions..37
5.1.2 User Interface..37
5.1.3 Input Format 38
5.1.4 Anticipated Changes 38
5.2 Module Decomposition..39
5.2.1 User Interface (TOG_main.c) 39
5.2.2 Specification Interface 40
5.2.2.1 Specification File (TOG_spec.c)..40
5.2.2.2 Constants (TOG_const.c)..43
5.2.2.3 Variables (TOG_vars.c) 44
5.2.2.4 Applications (TOG_applic.c)..44
5.2.2.5 Inductively Defined Predicates (TOG_indPred.c) 45

5.2.2.6 Table Holder (libtblhold.a) ... 46
5.2.3 Oracle Generation ... 46
5.2.3.1 Oracle Structure (TOG_oracle.c) ... 46
5.2.3.2 Expression (TOG_expn.c) ... 46
5.2.3.3 Code (TOG_code.c) ... 47
5.2.3.4 Context (TOG_context.c) ... 49
5.2.3.5 Procedures (TOG_procedures.c) ... 50
5.2.4 Output ... 50
5.2.4.1 File Output (TOG_output.c) ... 51
5.2.4.2 Line Buffer (TOG_line.c) ... 51
5.2.5 Utility Module ... 52
5.2.5.1 Id Table (idTable.c) ... 52
5.2.5.2 Name Table (nameTable.c) ... 52
5.2.6 Status Reporting ... 53
5.2.6.1 Error Token (TOG_error.c) ... 53
5.2.6.2 Message Logging (sw_error.c) ... 53
5.3 Algorithm Overview ... 54
5.3.1 Expression Coding ... 54

6 Trial Application ... 55
6.1 Program Overview ... 55
6.2 Test Procedure ... 55
6.3 Testing Results ... 56
6.4 Discussion ... 58
6.4.1 Specification Faults ... 58
6.4.2 Test Harness Construction ... 59
6.4.3 Non-Testable Properties ... 59
6.4.4 Oracle Generation ... 60

7 Discussion and Conclusion ... 63
7.1 Applications for This Work ... 63
7.2 Limitations of the Method ... 64
7.3 Future Work ... 66
7.4 Conclusions ... 68

Appendix A - Hash Module Documentation ... 71
A.1 Introduction ... 72
A.2 Internal Design Documentation ... 72
A.2.1 Informal Description ... 72
A.2.2 User Definitions ... 73
A.2.2.1 Constants ... 73

A.2.2.2 Data Structures 73
A.2.2.3 Hash Tables 73
A.2.3 Program Functions 74
A.2.3.1 HashAdd 74
A.2.3.2 HashRemove 75
A.2.3.3 HashFind 76
A.2.4 Auxiliary Predicate Definitions 76
A.2.5 Auxiliary Function Definitions 78
A.3 Hash Module Code 79
A.3.1 hash.c 79
A.3.2 hash.h 85
A.3.3 stuff.h 86

Appendix B - TOG Input File Format 89
B.1 Format Description 90
B.1.1 Constants 90
B.1.2 Variables 90
B.1.3 Program Specification 91
B.1.4 Auxiliary Predicate Definitions 91
B.1.5 Auxiliary Function Definitions 91
B.1.6 Inductively Defined Predicate Definitions 92
B.1.7 Built-in Functions 92
B.1.8 User Definitions 92
B.2 Formal Grammar 93

References 95

List of Figures

FIGURE 1 - Sample Test Harness Flowchart..36
FIGURE 2 - First Level Decomposition Module Uses Relation....................................40

List of Tables

TABLE 1 - Find Program Specification 24
TABLE 2 - Oracle Access Programs 27
TABLE 3 - Logical Operator Conversions 29
TABLE 4 - Module Uses Relation....................41
TABLE 5 - Specification File Module Access Programs 42
TABLE 6 - Constants Module Access Programs....................44
TABLE 7 - Variables Module Access Programs 44
TABLE 8 - Applications Module Access Programs....................45
TABLE 9 - Inductively Defined Predicates Module Access Programs....................45
TABLE 10 - Oracle Structure Module Access Program....................46
TABLE 11 - Expression Module Access Program 47
TABLE 12 - Code Module Access Programs....................47
TABLE 13 - Context Module Access Programs....................49
TABLE 14 - Procedures Module Access Programs....................50
TABLE 15 - File Output Module Access Programs 51
TABLE 16 - Line Buffer Module Access Programs....................51
TABLE 17 - Id Table Module Access Programs 52
TABLE 18 - Name Table Module Access Programs....................52
TABLE 19 - Error Token Module Access Programs....................53
TABLE 20 - Message Logging Module Access Programs....................54
TABLE 21 - Test Suite Descriptions 57
TABLE 22 - Summary of Hash Module Test Results....................57
TABLE 23 - HashAdd Program Description....................74
TABLE 24 - HashRemove Program Description 75
TABLE 25 - HashFind Program Description 76
TABLE 26 - Formal Grammar Symbols....................94

List of Acronyms

ADT	Abstract Data Type
ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
FSM	Finite State Machine
IDP	Inductively Defined Predicate
PUT	Program Under Test
TH	Table Holder
TES	Test Execution Summary
TOG	Test Oracle Generator
TTS	Table Tool System

1 Introduction

As software becomes pervasive in our society, its correct behaviour becomes increasingly critical to the safety and well-being of people and businesses. Consequently, there is an increasing need for the application of strict engineering discipline to the development of software systems. The Software Engineering Research Group at McMaster University seeks to address this need by developing techniques and tools to facilitate the production of software design documentation that is 1) clear enough to be read and understood by both ‘domain experts’ and programmers with a minimum of special training, 2) complete and precise enough to allow thorough analysis, both manually and mechanically and 3) suitable for use as a specification from which to produce an acceptable program. The use of tabular expressions to represent relations [29], and hence program specifications, is one of the cornerstones of these techniques.

Experience with producing tabular expressions for program documentation has shown that existing documentation tools are not well suited to this purpose—the creation and editing of tables is a time consuming process and more time is spent concentrating on the format of the table than on its content. To help overcome this problem, a set of computer programs is being developed, which together are known as the Table Tool System (TTS). (For a list of acronyms used in this thesis see page xi.) This system will provide one set of tools for editing the content of such documentation and another set for editing its format. Other tools in the system will be used to analyse the documentation for various purposes. This work describes a TTS tool which is used in the analysis of program documentation for the purpose of software testing.

1.1 Purpose

Although it is well known that “program testing can be a very effective way to show the presence of bugs (faults), but it is hopelessly inadequate for showing their absence”[7], it is widely agreed that testing is an important step in the software development process. It has also been observed that such testing is time consuming and costly—as much as 50% of the development costs for a project can be attributed to testing—and is itself error prone [20], [33], [36]. It seems natural, therefore, that any set of tools intended to improve the software development process will include tools to aid testing.

One fundamental assumption, known as the *oracle assumption*, of software testing research and practice is that there is some mechanism, an *oracle*, that will determine if the output from a program is correct [39]. In many cases this mechanism is a manual comparison of the test output with some, previously determined, ‘expected output’, which can be time consuming, tedious and error prone. There are also many cases where it is very difficult to determine the expected output, e.g. where the properties of the desired result are known, but not its value, as may frequently be the case in numerical calculations.

If a program has been formally specified, it should be possible to use the specification as an oracle, so the expected output need not be given by the user. This is particularly useful if the formal documentation is of a form that can be read and understood by both domain experts and programmers. Such documentation can be reviewed by the domain experts to ensure that the specified behaviour is correct and then used to communicate their intentions to the programmers. Generating an oracle from this documentation allows us to ensure that the documentation and program are consistent.

The purpose of this work is to develop a prototype automated Test Oracle Generator (TOG) tool that, given a relational program specification [25] using tabular expres-

sions [29], will produce a program that will act as an oracle. This oracle program will take as input an input, output pair from the program under test and will return ***true*** if the pair satisfies the relation described by the specification, or ***false*** if it does not.

1.2 Scope

Testing is defined by the IEEE as

> "The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results."[2]

In this thesis, only testing that consists of exercising executable components of the software system is considered. Testing is a possible method of *verification* of software components, but the latter term is not used in this thesis since its meaning is more broad. Neither the selection of appropriate tests for a component nor the effectiveness of those tests is discussed, as these issues are not relevant to the oracle generation problem. Readers are referred to [40] for a discussion of some of these other issues and a good survey of the relevant literature.

Since the documentation used in this work uses "before/after specifications" (see [31]), it is only suitable for specifying, and hence generating oracles for, programs for which the behaviour of interest can be described in terms of the program initial and final states, i.e., the program must terminate and it must be possible to determine the success or failure of an execution from its initial and final states. Only such terminating programs are considered in this work. If a program is intended not to terminate, some terminating subprograms (e.g. the body of an infinite loop) could be documented and tested using these methods.

Although the methods discussed in this work are applicable to programs written in a wide variety of programming languages, the prototype tool developed to illustrate these techniques is only suitable for those written in ‘C’.

1.3 Related Work

Much of the research aimed at reducing the cost and improving the effectiveness of software testing has concentrated on the judicious selection of test cases [11], [12], [20], [21], while other effort has gone into developing tools that either help generate, maintain and track the testing documentation (e.g. test plans, test cases, expected output or stub and driver routines) or execute tests in simulated environments [5], [14], [22], [23], [36]. Both of these areas of research are complimentary to, but quite distinct from, the work described in this thesis.

Several authors have described tools which can be used to compare the results of a test with some pre-defined ‘correct’ data. In [22], Panzl describes three systems that verify the values of program variables against test cases described using a formal test language. Another system, described by Hamlet in [14], tests a program using a list of input, output pairs which have been supplied as part of the program code. All of these systems require that the user provide the expected output, which may be difficult to obtain. Also, they can only compare for equality of expected and actual output, and hence relational specifications, which may accept more than one possible output for a given input, cannot be used. For example, the program specified in Table 1 on page 24 is required to indicate the location of the value of x in the array B, if one exists. If that value occurs in B in more than one place, then it is sufficient that the program indicate any one of these. Systems such as those described by Panzl or Hamlet would consider some of these occurrences to be invalid.

The latter limitation is partially overcome by the “program testing assistant” described by Chapman in [5]. This system allows the user to specify ‘success criteria’ (e.g. equal, set-equal, isomorphic, etc.) which are used when comparing actual and expected output. This system, however, must record the input and output from previous executions of the program to be used as test cases, so it is only useful if the user at one time had a version of the program that was considered to be correct.

Other systems, such as ANNA [18] and APP [35], allow program code to be annotated with assertions which are evaluated as the code is executed. If these assertions are sufficiently detailed and correctly placed so as to form a specification of the program, which is not the intention of APP, then they can be used as an oracle. However, since the annotations used in these systems are written as specially denoted comments in the program source code, they do not lend themselves well to analysis or review separate from the implementation, such as by non-programmer “domain experts”. Such analysis is one of the intended purposes of the documentation techniques presented in this thesis, so it is important that the documentation be distinct from the program.

In [37], Stocks and Carrington discuss deriving ‘oracle templates’, which describe a set of acceptable outputs for a given set of test cases, from model-based specifications (i.e. those that model the system as a finite state machine with transitions) using the Z notation. In [34], Richardson et al. advocate the derivation of oracles from formal models and specifications. Both papers suggest that the oracle could be automatically generated, but neither discusses the problems of actually producing an oracle procedure.

Other authors have discussed producing oracles for abstract data types (ADTs) that are specified using algebraic specifications, e.g. [1],[4], [10] or ‘trace’ specifications [38]. These specification techniques address a different problem from those used in this

work in that they are required to document the intended properties of an ADT which is implemented by a group of programs, whereas the techniques used in this work are used to describe the effect of a single program on some concrete data structure. The oracle problem is, therefore, different as well—ADT oracles must check that the specified ADT properties hold, whereas program oracles need only check that the data structure has been modified in the specified manner.

1.4 Outline of This Thesis

Chapter 2 defines the terminology that is used in this thesis. Chapter 3 describes the content and format of the type of program specification to be used for generating a test oracle and Chapter 4 discusses the design of the oracle itself. The design of the Test Oracle Generator is discussed in Chapter 5, and Chapter 6 presents the results of these methods when applied to the testing of some code from an industrial application. Chapter 7 discusses the applications and limitations of this work and draws some conclusions.

2 Notation and Terminology

The notation and terminology used in this thesis is defined in this section.

2.1 Predicate Logic

The predicate logic used in this work is based on that described in [30], which differs from traditional logic in that it allows the use of partial functions but ensures that all predicates are total. Axioms and rules of inference for similar logics are discussed in [6] and [9]. The following terminology is adopted from [30].

Function application

A *function application* is a function name together with its actual arguments, which are terms. The usual notation will be used for denoting function applications (i.e. ‘f(x)’, ‘G(x, y, z)’, ‘x+5’, etc.).

Term

A *term* is a constant, a variable or a function application.

Primitive relation

A small set of relations (e.g. <, >, =, etc.) are defined as being *primitive relations*. The value of a primitive relation is defined in the usual way with the addition that it is ***false***, by definition, if one or more of its argument terms is a function application with argument values outside the function’s domain. For example, if F and G are functions and the value of x is not in the domain of F then “F(x) > G(x)”, “F(x) < G(x)” and “F(x) = G(x)” are all ***false*** (assuming that ‘>’, ‘<’ and ‘=’ are primitive relations). Note that “F(x) = F(x)”, which in many other logics is equivalent to ***true*** by the “axiom of reflexivity”, is also ***false*** in the case where x is not in the domain of F.

Note also that this set of primitive relations does not normally include the negation of other primitive relations. For example, '≠' is not defined as a primitive relation since the value of "$f_1(x) \neq f_2(x)$" is not the same as "$\neg(f_1(x) = f_2(x))$"—the latter is ***true*** where x is outside the domain of either f_1 or f_2, whereas the former would be ***false*** (if '≠' is defined as a primitive).

Predicate expression

A *predicate expression* is either a primitive relation or a string of the form $(\forall x, P)$, (P), $P \wedge Q$, $P \vee Q$ or $\neg P$, where P and Q represent predicate expressions and x is a variable, known as the *index variable* of the quantification, which is said to be *bound* within the predicate expression in which it occurs (i.e. inside the outer-most pair of parentheses). These predicates are defined in the usual way as described in [30].

2.1.1 Notational Conveniences

The following equivalencies allow expressions to be written in the customary manner:

$$(P \Rightarrow Q) \equiv ((\neg P) \vee Q) \tag{EQ 1}$$

$$(\exists x, P) \equiv (\neg(\forall x, \neg P)) \tag{EQ 2}$$

2.1.2 Quantified Expressions

For oracle generation, quantification must be restricted to a finite set, which is characterized by an inductively defined predicate (see below) so that it can be automatically generated. This is accomplished by permitting only the following forms of quantified expressions, where P(x) is an inductively defined predicate and Q(x) is any predicate expression of a permitted form:

Universal: $(\forall x, P(x) \Rightarrow Q(x))$
Existential: $(\exists x, P(x) \wedge Q(x))$

2.1.3 Inductively Defined Predicates

An *inductively defined predicate* (IDP), P, on <type> is defined as the characteristic predicate of a set, S, which is formed in the following way. Given a triple, {I, G, Q}, where:

I is an enumerated finite set of elements of <type>,
G is a function, G: <type> → <type>,
Q is a predicate on <type>, and

$$\forall x \in I, (\exists m, ((\forall j, (0 < j < m \Rightarrow G^j(x) \in \text{dom}(G))) \wedge \neg Q(G^m(x)))). \quad \text{(EQ 3)}$$

S is the least set formed by the following rules:

1. all elements of I are in S
2. $\forall x \in S\ [Q(x) \Rightarrow G(x) \in S]$.

This least set can be constructed by the following inductive steps:

1. $S_0 = I$
2. $S_{n+1} = S_n \cup \{G(x) \mid x \in S_n \wedge Q(x)\}$.[1]

It can be proven that $\exists N, S_{N+1} = S_N$. (In fact, we can take N = the largest element from the set $\{m \mid \exists x \in I, [(\forall j, [0 < j < m \Rightarrow (G^j(x) \in \text{dom}(G) \wedge Q(G^j(x)))]) \wedge \neg Q(G^m(x))]\}$, i.e. the set of m's from EQ 3, above.) Thus $S = S_N$ and S is finite.

A predicate, P(x), is inductively defined by providing appropriate definitions for I, G and Q. For example, the characteristic predicate of the set of integers from MIN to MAX, inclusive, is inductively defined by: $I \equiv \{MIN\}$, $G(x) \equiv x+1$ and $Q(x) \equiv x < MAX$.

Note that P(x) is equivalent to

$$(x \in I \vee (\exists y, (P(y) \wedge Q(y) \wedge (x = G(y))))).$$

---

1. Note that an efficient algorithm for constructing this set would only consider the elements of S_n that are not in S_{n-1} at each step.

2.2 Tabular Expressions

In [29], Parnas describes a method for representing mathematical functions and relations using multi-dimensional tables called *tabular expressions*. These are equivalent to, but often easier to read and understand than, expressions written in a more traditional manner. Tabular expressions are particularly well suited to describing conditional relations of the forms that frequently occur in program specifications. This sub-section gives a brief summary of [29].

A tabular expression is constructed recursively from conventional (scalar) expressions and grids. A *scalar expression* is either a term or a predicate expression as described in Section 2.1.

2.2.1 Syntax of Grids and Tables

A *grid*, G, of *dimensionality* (i.e. the number of dimensions) dim(G), is an indexed set such that the index set is a set of dim(G)-tuples which are the Cartesian product of the sets:

$$\{1, 2, \ldots, \mathrm{len}_1(G)\}, \{1, 2, \ldots, \mathrm{len}_2(G)\}, \ldots, \{1, 2, \ldots, \mathrm{len}_{\dim(G)}(G)\},$$

where $\mathrm{len}_i(G)$ is the length of G in its ith dimension. *shape*(G) is a tuple of length dim(G) whose i^{th} element is $\mathrm{len}_i(G)$. G_I denotes the element (cell) of G with index I, where I is a member of the index set of G.

If I is an n-tuple, "I_j" (for $1 \leq j \leq n$) denotes the jth element of I and "$I|j$" denotes the (n-1)-tuple formed by removing the jth element from I.

A *table*, T, consists of a main grid, G, and coordinate header grids, C_1, C_2, …, $C_{\dim(G)}$ such that $\mathrm{shape}(C_i) = \mathrm{len}_i(G)$. In the remainder of this thesis coordinate header

grids will be referred to as header grids or simply headers. $C_{i,j}$ denotes the jth element of C_i.

Note also that dim(T) = dim(G), $len_i(T) = len_i(G)$ and $T_I = G_I$

2.2.2 Semantics of Tables

In this thesis, a table is classified as being either a *normal, inverted* or *vector* table.

A normal table contains predicate expressions in all of the cells of the header grids. For a normal table, T, the *selected cell* is a cell of the main grid with index, I, such that the value of the conjunction of C_{i,I_i} for $1 \leq i \leq \dim(T)$ is ***true***. If no such cell exists then there is no selected cell. If more than one such cell exists then the table is not well defined.

An inverted table, T, contains predicate expressions in the cells of C_1, C_3, …, $C_{\dim(T)}$ as well as in the cells of the main grid, G. The selected cell of T is a cell of C_2 with index, I_2, such that the conjunction of C_{i,I_i} for i = 1, 3, 4, …, dim(T) and G_I is ***true***. If no such cell exists then there is no selected cell. If more than one such cell exists then the table is not well defined.

Normal and inverted tables can be either *function* or *predicate* tables. A function table contains a term in all cells that could be the selected cell (i.e. cells of the main grid for normal tables or cells of C_2 for inverted tables). A predicate table contains predicate expressions in all cells that could be the selected cell.

The value of the function described by a function table is the value of the term in the selected cell, if such a cell exists, otherwise it is undefined. The value of the predicate

expression described by a predicate table is the value of the predicate expression in the selected cell, if such a cell exists, otherwise it is ***false***.

A vector table contains predicate expressions in the cells of C_2, C_3, …, $C_{dim(T)}$ and strings of the form "x_i |" or "x_i =", where x_i is a variable, in C_1. The selected cells of T are the cells of the main grid with any index, I, such that the value of the conjunction of C_{i, I_i} for $2 \le i \le dim(T)$ is ***true***. Note that these cells will form a column of the main grid—they will have $I_1 = 1, 2, \ldots, len_1(G)$. If no such cells exist then there are no selected cells.

In this work, a vector table is always interpreted as a predicate table. The value of the table is the value of the conjunction of the expressions formed from the selected cells in the following manner: for cells, G_I, for which the corresponding C_1 cell, C_{1, I_1}, is of the form "x_i |" the expression is simply the predicate expression in G_I. For cells for which the corresponding C_1 cell is of the form "x_i =" the expression is the predicate expression "$x_i = G_I$". If there are no selected cells then the table value is ***false***.

2.2.3 Expressions

A function table can be considered to be a function application and is thus a term in the sense of [30], and a predicate table can be considered to be a predicate expression. Any term or predicate expression is an expression and may appear as an element of a grid.

2.3 Relational Specification

As discussed in [8], [19] and [31], among others, a digital computer can be viewed as a finite state machine (FSM)—it consists of a finite set of memory and bulk storage locations and input and output registers, each of which is itself a finite state machine. The state of a computer is the combination of the state of all of its component FSMs.

For the purpose of discussing programs, the following terminology is adopted from [28] and [31]: An *execution* is a (possibly infinite) sequence of states of the machine, the first of which is known as its *initial* or *starting* state. If an execution is finite then it is said to be a *terminating execution*, and the last state is known as its *final* or *stopping* state. A *program*, P, is a mechanism for establishing a pattern of state changes of the machine and hence denotes a set of executions—the possible sequences of states established by that program—sometimes called the *executions* of P.

Frequently, when specifying a program, the specifier does not want to restrict the intermediate states that the machine might be in during execution (i.e. the algorithm used by the program) but only requires that the stopping state be correct for each starting state. In these cases only the initial and final states of terminating executions are of interest. A pair of states that are the initial and final states of a terminating execution are referred to as an *execution summary*. Recalling that a binary relation is a set of pairs, clearly the set of acceptable execution summaries for a program can be described using a relation.

2.3.1 Limited Domain Relations

In [25], [26], [28] and [31] Parnas et al. describes the use of Limited Domain Relations (LD-relations) to specify programs. An *LD-relation*, L, is a pair (R_L, C_L) where R_L is an ordinary relation and C_L is a subset of the domain of R_L, known as the *competence set*. The domain and characteristic predicate of L are the domain and characteristic predicate of R_L.

An LD-relation, L, can be used to specify a program by letting R_L be the set of acceptable start state, stop state pairs (i.e. execution summaries) and C_L be the set of starting states for which the program must terminate. A program, P, is said to *satisfy* a specification, L, if and only if

- when started in any state, x, if P terminates, it does so in a state, y, such that <x, y> is an element of R_L, and
- for all starting states, x, in C_L, P will always terminate.

Note that if a starting state $x \notin \text{domain}(R_L)$ then P cannot terminate such that P satisfies L.

In the case of a deterministic program, R_L is a function. In the case where C_L is the domain of R_L (always for deterministic programs) C_L need not be given.

In this thesis a program is assumed to be specified by an LD-relation, which is referred to as the *specification relation*. If the competence set is not given, it is assumed to be the domain of R_L.

2.4 Program Variables and State Descriptions

As described in [19], a computer can be considered to be sub-divided into smaller FSMs some of which are referred to as *program variables*. A *type* can be associated with each program variable to denote its number of possible states and the abstraction used to describe these states. The suitably abstract description of the state of a program variable is known as its *value*. For example, if we say a program variable is of ANSI C type `int` then it must have at least 2^{16} possible states which are typically represented by the integer numbers between -32766 and 32767.

In the context of documentation for a program or set of programs, however, only a very small percentage of the possible program variables are typically relevant. The *data structure* of a program or set of programs is defined as being the set of program variables whose values affect, or are affected by the program(s) (i.e. the memory locations and registers that are used by the program). In this thesis, the term *state* is assumed to refer to the

state of the data structure with respect to a particular program or set of programs. A *state description* is a tuple giving the value of each program variable in the data structure.

A *program variable name* is a string of characters used to represent a program variable in a program text (i.e. code). Abusing the notation slightly, “the value of x”, where “x” is a program variable name, is used to refer to the value, in some state, of the program variable represented by x.

2.4.1 Before and After Value

The following convention for denoting the value of program variables before and after a program is executed is adopted from [28] and [15]:

> Let P be a program and x_i, …, x_k be the names of the program variables in the data structure of P. Then
>
> - “$'x_i$” (to be read “x_i before”) denotes the value of x_i in the initial state of an execution of P and
> - “x_i'” (to be read “x_i after”) denotes the value of x_i in the final state of an execution of P.

For the purposes of interpretation of a specification, x_i' and $'x_i$ are different terms.

2.5 Functional Testing

Functional testing of a program involves executing the program under test (PUT) using some ‘test data’ and examining the output data to verify the program behaviour [16]. This work considers only functional testing, referred to simply as *testing*.

For the purpose of this work a *test case*, X, is a description of a starting state for a program. A *test execution summary* (TES) is a pair of state descriptions, <X, Y>, the first

of which is a test case and the second is a description of the state in which the program terminated after having been started in X. A TES can be said to *pass* with respect to a relational specification if the TES is an element of the specification relation, otherwise it is said to *fail* with respect to that specification.

In [4], Bernot et al. discuss the need to develop a set of hypotheses, H, which express the relationship between the pass or failure of a series of TESes and the correctness of the program. In the case of a deterministic program and exhaustive testing (i.e. every starting state is a test case), clearly H is ‘if all TESes pass then the program is correct, otherwise it is not’. In practice, however, exhaustive testing is rarely practical. Also, in the case of non-deterministic programs, it is impossible to reach such conclusions since any test case can describe the initial state of several different execution summaries.

As stated in Section 1.2, the selection of a set of test cases is not considered in this work. Since the hypothesis set is, in general, a function of these test cases, it will also not be discussed. Interested readers are referred to [4] for a further discussion of this topic.

2.6 Test Oracle

In [16, p.43], Howden describes an *oracle* as a function which, given a program, P, can determine, for each input, x, if the output from P is the same as the output from a ‘correct’ version of P.

Consistent with this, in the context of this work, an *oracle* is a program which, given a TES, will determine if it passes or fails with respect to the specification from which the oracle was derived by evaluating the characteristic predicate of the specification relation—if it evaluates to ***true***, then that TES passes, otherwise it fails. Note that such an oracle does not require the existence of a ‘correct’ version of P.

2.7 Test Harness

Practical program testing typically involves executing the PUT for many different test cases and verifying the results using the test oracle. This can be done using a program known as a *test harness* which may partially simulate the environment in which the PUT is designed to be used, and may also perform such tasks as collecting statistics on the number of failed tests, etc.

3 Program Documentation Method

The documentation which is the input to the TOG is design documentation for a single program (procedure), i.e. it describes the intended behaviour of a program in terms of its effect on the actual data structure. This is distinct from *module interface documentation* which describes the externally observable behaviour of a module without reference to the data structure used in their implementation (see [27], [28] and [38]). (A *module* is a group of programs which are designed and implemented as a single work assignment. Typically they implement an abstract data type or encapsulate a design decision, e.g. algorithm or external device interface.) This chapter describes the program documentation method used in this work, which is based on that described in [28] and has the following desirable properties.

- It is precise and formal.
- It is clear enough to be read and understood with a minimum of special training.
- Reading a specification neither gives any details about, nor requires any knowledge of, the algorithm used by the program specified.

It is assumed that this documentation is created using other TTS tools and is stored in an appropriate format.

3.1 Primitives

Since specifications are written in terms of the values of the program variables in the data structure, it is convenient to describe these variables and operations on them using the notation of the programming language used for the implementation of the program under test (PUT). This has the clear advantage that programmers and verifiers responsible

for reading the specifications will be familiar with the notation. It also makes it easier to produce tools, such as the TOG, which interact with the specified program or its environment. For this reason, the primitives supported by the TOG are specific to the programming language used. In this thesis, the programming language is C.

3.1.1 Data Types

Since, as discussed in Section 2.4, the value of a program variable is understood in the context of its type, the documentation must give the type of each program variable in the data structure. Also, since functions defined in the documentation must be compared with program variables, they too must be assigned a type. Those types which are supported by the programming language or defined (in the syntax of the programming language) in the ‘user definitions’ section of the documentation are taken to be primitive.

3.1.2 Primitive Functions

Primitive functions are those functions that are assumed to be ‘built in’ to the system and can be used without definition in specifications. Parnas et al. [28] define a *known* program as “one that does not require a specification” (i.e. its specification is assumed to be understood) and an *available* program as one that “exists in a project or system library”. Programs that are available or are made available through declarations in the ‘user definitions’ section, are treated as primitive functions. Primitive functions are assumed to be total (i.e. their domain is the cartesian product of their argument types).

3.1.3 Primitive Predicates

The usual primitive relations, $=$, $>$, $<$, $\geq$ and $\leq$, are defined and used in the standard infix notation style for comparing terms of the same primitive type. Note that these primitive relations are not defined for non-primitive (i.e. abstract) types, so expressions

such as “x' = *myFunc*('x)” (where x is a program variable of non-primitive type) are not permitted. Auxiliary predicates may be defined to evaluate relations such as equality on abstract data types.

3.2 Documentation Components

The documentation consists of: constants, variables, program specifications, auxiliary predicate definitions, auxiliary function definitions, inductive predicate definitions and user definitions. The manner in which these are represented to the user is an implementation detail of the program documentation editing tools—examples given here are for illustration purposes only. The documentation components are described below.

3.2.1 Constants

A constant is any string of symbols that is interpreted as a constant in the syntax of the programming language. For example, in C the following strings are constants: `13`, `TRUE`, `0x2b` and `"A Text String"`.

3.2.2 Variables

In the documentation, strings of characters called *variables* (not a *program variable*, which, as described in Section 2.4, is a FSM) are used to represent either the value of program variables in the initial state or final state of an execution, the value of expressions passed as arguments in auxiliary or inductive definitions (i.e. formal arguments), or as quantification indices. As mentioned in Section 2.4.1, a program variable name, annotated with a single quote (') either before or after, is a variable used to represent the value of that program variable in the initial or final state, respectively. Variables which represent quantification indices are considered to represent a value only where they are bound. A variable that is the same as the name of the program specified by the specification is used to repre-

sent the value returned by the program (e.g. the value of the accumulator register in the final state), if appropriate for the programming language.

All variables must have a type, which is as described in Section 3.1.1. For this prototype oracle generator, the type of a particular variable must be the same throughout the program documentation. (i.e. The same quantification index variable cannot be used to quantify over two different types in different parts of the documentation.)

3.2.3 Program Specifications

A *(relational) program specification*, as illustrated in Table 1 on page 24, consists of three components: (1)The *program invocation* gives the name and type of the program and lists all of its actual argument program variables. (2)The *external variable list* lists all other program variables referred to (by annotated variables with the same name) in the specification relation expression. (3)The *specification relation* defines the LD-relation that specifies the behaviour of the program. It includes expressions that give the characteristic predicates of the domain, competence set and relational components of the LD-relation. Note that, by default, if the competence set is not given then it is taken to be the same as the domain of the specification relation.

3.2.4 Auxiliary Predicate Definitions

Auxiliary predicates can be defined so that complicated or frequently used predicate expressions can be written more concisely in the documentation. The definition of an auxiliary predicate consists of a name, a list of formal argument variables, and a predicate expression written in terms of the formal arguments. When its name, together with a list of actual arguments, is used in the documentation, it is evaluated by substituting the values

represented by the actual arguments for their corresponding formal arguments in the definition predicate expression and evaluating the resulting predicate expression.

3.2.5 Auxiliary Function Definitions

Auxiliary functions can be defined so that complicated or frequently used functions can be written more concisely in the documentation. The definition of an auxiliary function consists of a name, a type, a list of formal argument variables, a term expression and an optional predicate expression, which gives the domain of the function, both written in terms of the formal arguments. When the auxiliary function name, together with a list of actual arguments, is used in the documentation it is evaluated by substituting the values represented by the actual arguments for their corresponding formal arguments in the term expression. If the definition contains a domain expression that does not evaluate to ***true***, the value of the function is undefined, otherwise the value is that described by the expression.

3.2.6 Inductively Defined Predicate Definitions

An *inductively defined predicate* is an auxiliary predicate that is defined inductively as described in Section 2.1.3. Its definition consists of a name, a formal argument variable and the definition components I, G, and Q. ‘I’ is a string which is an enumerated set in the syntax of the programming language (an initial value for an array in the C language) and ‘G’ and ‘Q’ are expressions in terms of the formal argument variable.

3.2.7 User Definitions

A *user definition* is a sequence of text in the syntax of the programming language which is used to declare data structures, functions or symbols that are used in the documentation and are not primitive to the programming language. This is required so that the

basic symbols (e.g. constant names) and operators (e.g. structure element access) which are used in the specification can be understood.

3.3 Sample Program Documentation

Table 1, which is adapted from an example used in [28], specifies a program ‘find’ which searches an integer array ‘B’ for a value given by ‘x’, returns its index in ‘j’ and, using a boolean variable ‘present’, indicates if a match was found.

TABLE 1 - Find Program Specification

void
find(int B[N], int x, int j, bool present)

external variables:

D$_{\textbf{find}}$ = ***true***

C$_{\textbf{find}}$ = ***true***

R$_{\textbf{find}}$ (,) =

	(∃i, *bRange*(i) ∧ 'B[i] = 'x)	(∀i, *bRange*(i) ⇒ ¬('B[i] = 'x))
j' |	'B[j'] = 'x	***true***
present' =	TRUE	FALSE

∧ *NC*('B, 'x, B', x')

Auxiliary Predicate Definitions

NC(int 'a[], int 'b, int a'[], int b')
 $\stackrel{df}{=}$ (∀i, *bRange*(i) ⇒ 'a[i] = a'[i]) ∧ ('b = b')

Inductively Defined Predicates (see Section 2.1.3)

bRange(int i)
 $\stackrel{df}{=}$ I = {0}, G(i) = i+1, Q(i) = i<(N-1)

User Definitions

```
#include "defs.h"

#define N 10   /* Size of array to search */
```

4 Oracle Design

This chapter describes the interface and internal design of the oracle that will be the output of the TOG. The design is illustrated using specific examples from an oracle prototype, which was manually produced for the simple 'find' program specification given in Section 3.3.

4.1 Programming Language

The example programs to be tested (see Appendix A) are written using the C programming language, and hence the primitives used in the specification are C style. To simplify the oracle generation process and the interface to the test harness, the oracle is implemented using C and C++. This decision should not be seen as a significant feature of the design—if the intended application were different, the oracle design could be translated with little significant change.

4.2 Interface

The interface to the oracle is a set of three boolean valued programs: `inRelation`, `inCompSet` and `inDomain`. `initOracle` is an initialization program which should be called by the test harness once, before the first oracle program is called.

`inRelation` evaluates the characteristic predicate of the relational component of the specification relation. It takes the value of the PUT data structure in the initial state and final state (i.e. the TES) as arguments and can be called by the test harness to evaluate TESes as necessary. It returns `TRUE` if the TES passes, or `FALSE` otherwise.

`inCompSet` and `inDomain` evaluate the characteristic predicates of the competence set and domain, respectively, specified for the specification relation. Their arguments are those values from the start state of the TES which are required to evaluate the characteristic predicates. They return `TRUE` if the test case is in the set, or `FALSE` otherwise. These two programs can be used to avoid executing the PUT using test cases for which either there is no acceptable TES (i.e. the test case is not in the domain) or the PUT may be non-terminating (i.e. the test case is not in the competence set).

An alternative interface design, similar to that used in [35], would be to use the 'debug' information supplied by a compiler to resolve references to the data structure, and to embed code in the PUT to evaluate the oracle predicate at the appropriate points in the execution. While this method seems to lend itself to an elegant test harness design, it is felt that it may also be limiting, and will certainly make the job of the test oracle generator difficult. Also, since it involves modification of the PUT, it introduces the potential for errors being avoided during testing which may appear in the 'released' version (i.e. without the oracle code). If desired, the chosen interface design could be adapted for use in such a manner by embedding calls to the oracle access programs at appropriate locations in the PUT.

In a testing environment, it is often desirable to know in some sense why a test execution fails, so that program (or specification) faults can be easily isolated. Since relational program specifications are used in this work, which may allow several correct stopping states for a particular test case, it is not, in general, possible for an oracle to determine why a TES has failed.

Table 2 gives the syntax of the interface to the prototype oracle access programs. Each row in the table describes the interface to one of the access programs, which is

named in the first column. The second column gives the type of value returned by the program and the subsequent columns give the formal names and types of the program arguments, in the order that they appear. Note that, with the exception of `initOracle`, the actual number and types of arguments for these access programs are specific to the particular PUT and specification ('find' in this case).

TABLE 2 - Oracle Access Programs

Name	Value	Arg. 1	Arg. 2	Arg. 3	Arg. 4	Arg. 5	Arg. 6
initOracle							
inCompSet	bool						
inDomain	bool						
inRelation	bool	int B[]	int x	int B_p[]	int x_p	int j_p	bool present_p

4.3 Internal Design Overview

The oracle can be viewed as a 'compiled' version of the specification in that it is generated by translating the 'source' specification into an executable form (C code). Once it has been 'compiled', it can be executed without reference to the specification from which it was derived. One advantage of this design is that it allows the TOG to use optimization techniques to reduce the time required for oracle execution.

An alternative is to construct the oracle as an 'interpreter' which would represent the specification by data and evaluate it directly for each TES. An advantage of this design is that the oracle generation process is relatively simple, probably involving no code generation (the oracle programs are the same for any specification, only the data they use is dependant on the specification). A disadvantage is that the oracle will need to interpret the semantics of the documentation during evaluation, and so would probably be comparatively slow to execute. This is seen as a significant disadvantage since, in real applications

of this work, speed of oracle execution is much more important than oracle generation simplicity.

4.3.1 Expression Implementation

Each of the oracle access programs (with the exception of `initOracle`) evaluates a predicate expression which may be arbitrarily complex. Since any expression is made up of one or more sub-expressions, the complexity is managed by decomposing each expression into its constituent sub-expressions and implementing each sub-expression individually. In addition to the access programs, the oracle code consists of a set of internal functions and objects, each of which implements a sub-expression and may call other internal functions or object methods.

Since programming languages in general, and C in particular, support basic logical and relational operators (i.e. $\wedge$, $\vee$, $\neg$, $>$, $<$, $=$ etc.), these operators can be used to directly implement some of the expressions. Using these operators it is possible to implement an entire expression as a single C statement by translating it into a purely scalar, quantifier free expression (by expanding quantification to a series of conjunctions or disjunctions and translating tabular expressions into an equivalent disjunction of conjunctions) but for all but the most trivial specification, the resulting C statement would be many lines long. While this would undoubtedly result in an oracle that executes relatively quickly, since there would be none of the overhead associated with loops or function calls, it would require significant effort on the part of the TOG to do the translation and would result in virtually incomprehensible oracle code. For this reason the oracle is implemented using the C logical and relational operators only where they directly represent the operators in the specification (see Sections 4.4.1 and 4.4.2, below).

Another method of implementing expressions uses a class of C++ objects, with a sub-class for each expression type. A particular expression is implemented by instantiating the appropriate objects, which contain references to their sub-expression objects. For expression forms such as tabular expressions, which have complex semantics, this helps to simplify the oracle generation process—the TOG need only translate the expression into the appropriate object constructor. For forms with less complex semantics, however, the gain in simplicity of the TOG does not warrant the overhead of a C++ object.

The code to implement each type of expression is described in the following sections.

4.4 Scalar Expressions

Scalar (i.e. non-tabular) expressions can be translated into equivalent C statements as described below.

4.4.1 Logical Operators

Except when they are the root node of a quantified expression (see Section 4.4.4), logical operators can be directly translated to their C equivalents, as given in Table 3. (P and Q are arbitrary predicate expressions.)

TABLE 3 - Logical Operator Conversions

Logical Operator	C Equivalent
$\neg P$	`!P`
$P \vee Q$	`P || Q`
$P \wedge Q$	`P && Q`
$P \Rightarrow Q$	`(!P) || Q`

Thus the expression which is the definition of the auxiliary predicate NC (see page 24) is implemented in the following procedure (`nc_1` is a procedure which implements the quantified sub-expression):

```
static BOOL
nc(int p_a[N], int p_b, int a_p[N], int b_p)
{
   return(nc_1(p_a, a_p) && (p_b == b_p));
}
```

4.4.2 Primitive Relations

Since the logic used in this work differs from most traditional logics in the definition of primitive relations, the standard programming language relational operators are combined with information about the domain of partial functions. For example, the predicate expression "*guarded_B*('B, j') = x", where *guarded_B* (defined in Section 4.6) is a partial function, is translated into the following code.

```
(guarded_B_domain(j_p) && (guarded_B(p_B, j_p) == x));
```

This translation relies on the fact that C expressions are evaluated from left to right and evaluation stops as soon as the value of the expression is known. So, in the above code, if `guarded_B_domain(j_p)` returns `FALSE` then the right hand side will not be evaluated.

4.4.3 Inductively Defined Predicates

Since inductively defined predicates (IDPs) are intended to be used to characterize sets for quantification purposes, their implementation provides, in addition to the usual predicate expression evaluation operator (i.e. is the predicate ***true*** or ***false*** for a given value), a means to enumerate the set elements. An IDP is implemented in the form of an

(C++) object class which encapsulates the algorithm for determining the next element of the set.

An array is used to represent the ‘I’ component of the IDP definition and two procedures implement the expressions ‘G’ and ‘Q’. For example, the definition for *bRange*(int i) (see Section 3.3) is implemented using the following code.

```
static int bRange_I[] = { 0 };

static int
bRange_G(int i)
{
   return(i+1);
}

static BOOL
bRange_Q(int i)
{
   return(i < (N-1));
}
```

The IDP object classes have three methods: ‘`()`’ (an operator method), `first` and `next`. The method ‘`(e)`’ returns TRUE if e is in the set characterized by the IDP, or FALSE otherwise. `first`, initializes the object’s internal variables and returns the first element of the array representing I. `next` returns the ‘next’ element of the set, as described by the following three cases.

1. If the most recently returned element, say e, is such that Q(e) is ***true***, then G(e) is returned.

2. If Q(e) is ***false*** and there are more elements of I, then the next element of I is returned.

3. Otherwise, then there are no further elements of the set so an element not in the set is returned. (So the `()` operator will return FALSE.)

Thus the enumeration of the set that is characterized by an IDP can be accomplished using the following algorithm (where P is the IDP object).

```
e = P.first();
while (P(e)) {
   process(e);
   e = P.next();
}
```

When an inductively defined predicate is used in an expression, it can be implemented by instantiating an object from the appropriate class, depending on the type of the elements of the set, with the array and procedures corresponding to the IDP definition passed as arguments to the instantiation function. This is illustrated by the object `bRange` of type `IndPred_int` which is used in the quantification example below.

4.4.4 Quantification

Quantifier expressions are implemented using loops that call the appropriate procedures to enumerate the elements of the set characterized by the IDP (see Section 4.4.3, above). The root node of the quantification expression (i.e. the '$\wedge$' for existential or '$\Rightarrow$' for universal) is not implemented as described in Section 4.4.1, but is effected by evaluating its right child expression for only those elements which make the left child expression ***true*** (i.e. the elements of the set characterized by the IDP). To ensure that evaluation is as fast as possible, the loops are designed to terminate as soon as the result of the quantification is known (i.e. the first positive instance for existential quantification, and the first negative instance for universal quantification). Of course, quantification over a large set is inherently a lengthy process.

The quantification "($\exists$i, *bRange*(i) $\wedge$ 'B[i] = 'x)", which is in the first cell of the column header of the table on page 24, is implemented as follows.

```
static BOOL
table_H2_2_1(int p_B[N], int p_x)
{
   IndPred_int bRange(bRange_I, 1,
                      bRange_G,
                      bRange_Q);
   int i;
   BOOL result = TRUE;

   i = bRange.first();
   while (bRange(i) && result) {
      result = !(p_B[i] == p_x);
      i = bRange.next();
   }
   return(!result);
}
```

4.5 Tabular Expressions

Tabular expressions are implemented by instantiating an object of one of several classes of (C++) table objects which implement the various types of tabular expressions (normal, inverted and vector). These table objects encapsulate all knowledge of the semantics of tabular expressions, so the TOG need not have this knowledge and is hence less complicated. The expression in each cell of the table is implemented as a procedure (C function) and a pointer to each of these procedures is stored by the table object. In an attempt to make tabular expression evaluation faster, the table objects evaluate cell expressions as few times as possible and preferentially choose the most recently used cells to evaluate first (if test cases are such that successive cases select the same cell then this will make a test suite run faster).

Table objects have two methods which are used to evaluate the expression: `findCell` determines if the values of the arguments are in the domain of the table (by determining if a selected cell or cells exist), and `value` evaluates the table, returning the value in a `CELL` data structure which is a union of all of the basic data types in C (including `void *`).

An alternative implementation for a tabular expression is to translate it into an equivalent scalar expression and then implement the scalar expression as described in the previous section. This option has the disadvantages that it requires that the TOG have the ability to perform the translation, and it does not allow the above optimizations made possible by the table object design.

4.6 Auxiliary Predicates and Functions

As mentioned in Section 3.2.4 and Section 3.2.5, auxiliary predicates and functions are expressions which are either complicated or used repeatedly. An appropriately typed procedure is used to implement each auxiliary predicate or function definition, with the expression, implemented as described above, forming the body of the procedure. For auxiliary functions for which a domain expression is given, a procedure is produced to implement that expression as well. For example, consider an auxiliary function defined as follows:

int *guarded_B*(int b[], int i)
$\stackrel{\text{df}}{=}$ b[i]
domain: $0 \leq i < N$

This is implemented by the following procedures:

```
static int
guarded_B(int b[], int i)
{
   return(b[i]);
}

static BOOL
guarded_B_domain(int i)
{
   return((0 <= i) && (i < N));
}
```

Appropriate calls to these procedures are used in the code that implements expressions using the auxiliary predicate or function.

4.7 Compilation and Execution

The oracle consists of three groups of code: that generated by the TOG, in *outFile*.cc (where *outFile* is the output file name specified by the user, as described in Section 5.1.2), and the two sets of object classes, in indPred.cc and Table.cc, which are not generated by the TOG but are used by the TOG generated code. As is the norm for C++ programs, each source file has a corresponding ‘include’ file with the file name extension “.h”.

To use the oracle, a test harness program, which calls the oracle procedures and the PUT and reports the results, must be written. It should include the oracle header file (*outFile*.h) which declares the oracle procedure prototypes. The test harness and the oracle code must be compiled and linked to produce an executable program. Figure 1 is a flowchart of a possible test harness design and Section 7.1 discusses some other possible designs in further detail.

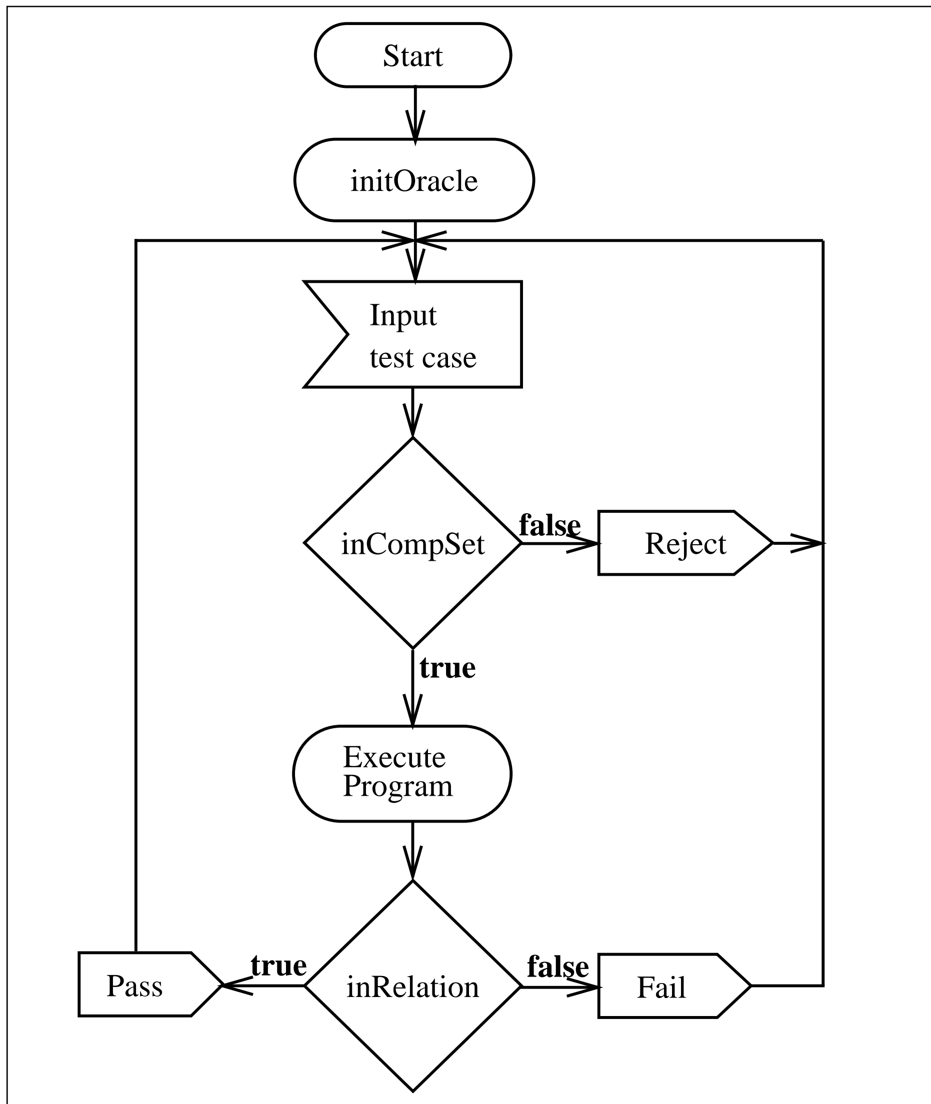

FIGURE 1 - Sample Test Harness Flowchart

5 Test Oracle Generator Design

This chapter briefly describes the requirements and design of the prototype TOG.

5.1 Requirements

To reiterate, the requirements of the TOG are that it accept a program specification in the form described in Chapter 3 and produce the code for an executable test oracle such as that described in Chapter 4.

5.1.1 Assumptions

It is assumed that the mathematical expressions used in the specifications have been input and saved using the table holder module (see Section 5.2.2.6). This assumption affects the Specification File and Expression modules, which are described in Section 5.2.2.1 and Section 5.2.3.2, respectively.

The oracle code is constructed using two sets of object classes: Tabular expressions (`normTable`, `invTable` and `vecTable`) and IDPs (`IndPred_<type>`) implemented in Table.cc and indPred.cc, respectively. These are assumed to be present and correct. This assumption affects the Code module which is described in Section 5.2.3.3.

5.1.2 User Interface

The user interface to the TOG is a 'command line interface' similar to that of many compilers. This has the advantage that it can be invoked by standard tools such as 'make'. The command line syntax is as follows:

tog [**-l***errlevel*] [**-h**] [**-o** *outFile*] [*specFile*]

Options:

-l*errlevel* Set the message logging level to errlevel where *errlevel* is one of **D**, **I**, **W** or **S** (Debug, Info, Warning, Serious). Only messages with seriousness equal to or greater than *errlevel* will be written to the log file (TOG_logfile). The default *errlevel* is **W**.

-h Output a help message (and do nothing).

-o *outFile* Use *outFile* as the base name for the oracle output. The files *outFile*.cc and *outFile*.h are produced. The default name is **oracle**.

specFile Generate the oracle from the specification in *specFile*. If no file name is given then input is read from standard input.

5.1.3 Input Format

The input to the TOG is in the form of a specification file which contains the information as described in Chapter 3. The file consists of a sequence of items, each of which define either a constant, variable, auxiliary predicate, auxiliary function, inductively defined predicate or the program relation. The last item in the file is the user definitions text. The format of the specification file is described in detail in Appendix B.

5.1.4 Anticipated Changes

It is expected that the following items are likely to change within the useful life of the TOG.

- Specification file format
- Oracle programming (output) language
- Oracle design

- User interface

5.2 Module Decomposition

The TOG is implemented by a set of *modules*, each of which encapsulates a set of design decisions. Several of the modules can be further sub-divided into sub-modules which encapsulate more specific design decisions. The benefits of this encapsulation are twofold: the design is easier to understand because of this separation of concerns, and it is easier to change the TOG since the decisions affected by the change are likely to be isolated

For the purpose of illustrating the system design, the *Module Uses Relation* is used. Module A is said to *use* Module B if some programs in Module A rely on the correct behaviour of some programs in Module B to accomplish their task.[1] Figure 2 illustrates the Module Uses Relation for the TOG for the first level module decomposition. Table 4 gives the Module Uses relation for the sub-modules.

5.2.1 User Interface (TOG_main.c)

The User Interface module acts as the main controlling module for the TOG. It encapsulates the interpretation of command line arguments and the sequence of invocation of other modules. It uses the Specification interface module to read the specification from the file, the Output module to initialize the output files and Oracle Generation module to produce the oracle code.

1. Note that the module uses relation is derived from the program uses relation discussed in [24].

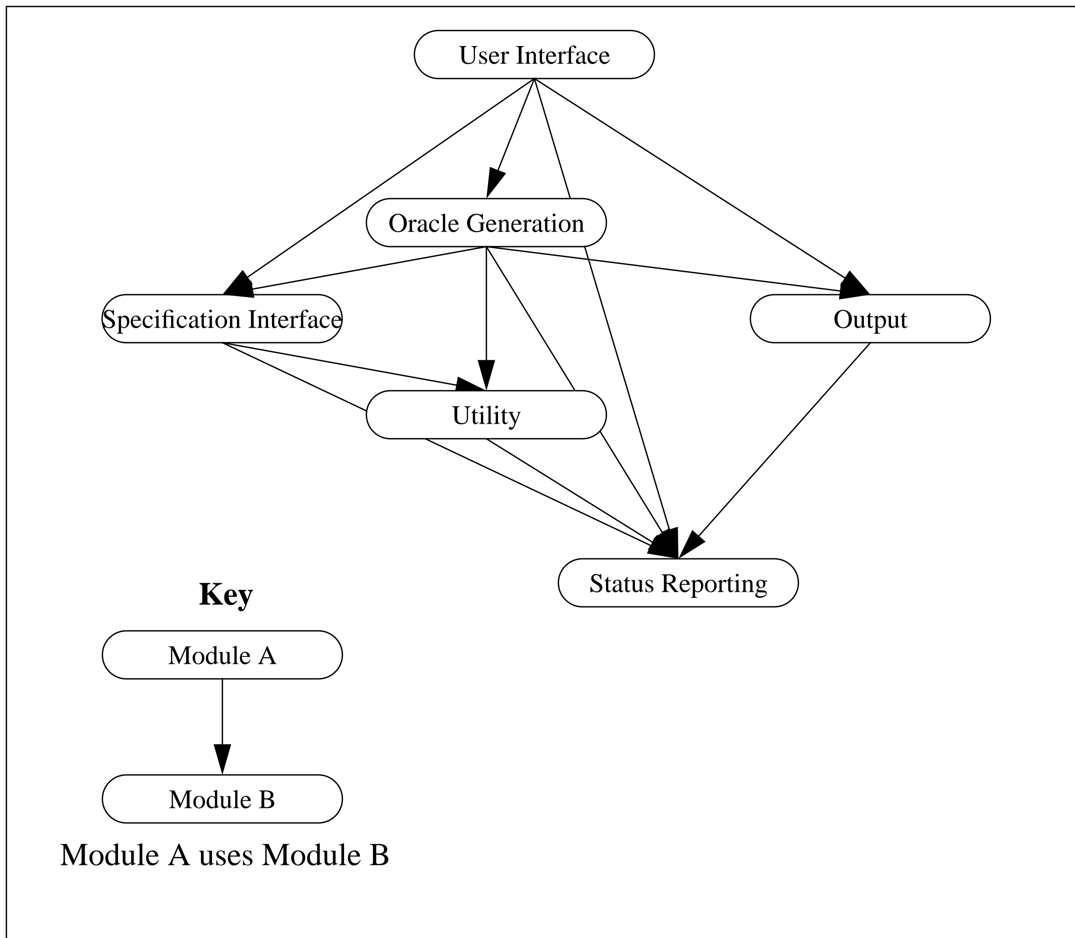

FIGURE 2 - First Level Decomposition Module Uses Relation

5.2.2 Specification Interface

The Specification Interface module is responsible for providing access to the PUT specification information. It is sub-divided into the following sub-modules.

5.2.2.1 Specification File (TOG_spec.c)

This module extracts the specification from a file and stores it in the appropriate information storage modules, described below, for retrieval by the Oracle Generation module. It encapsulates the specification file format and the algorithm for reading it. The information about the auxiliary predicates, and functions and inductively defined predi-

TABLE 4 - Module Uses Relation

Module	Level	Uses
User Interface	7	Oracle Structure Expression Specification File Output Status Token Message Logging
Oracle Structure	6	Expression Code Context Specification File Output Line Buffer Status Token Message Logging
Expression	5	Code Applications Table Holder Line Buffer Status Token Message Logging
Code	4	Context Procedures Constants Variables Applications Inductively Defined Predicates Procedures Line Buffer Status Token Message Logging
Specification File	3	Constants Variables Applications Inductively Defined Predicates Table Holder Status Token Message Logging

TABLE 4 - Module Uses Relation

Module	Level	Uses
Context	3	Procedures Name Table Status Token Message Logging
Constants	2	Status Token Message Logging
Variables	2	Id Table Status Token Message Logging
Applications	2	Id Table Status Token Message Logging
Inductively Defined Predicates	2	Id Table Status Token Message Logging
Procedures	2	File Output Status Token Line Buffer Message Logging
Id Table	1	Message Logging
Name Table	1	Message Logging
Output	1	Message Logging Line Buffer

cates used in the specification are stored in lists which can be enumerated using the module access programs.

TABLE 5 - Specification File Module Access Programs

Name	Type	Arguments	Description
TOG_specOpen	void	FILE *	Read the specification from the file.
TOG_specGetCompSet	Expn		Return the competence set expression for the specification relation.
TOG_specGetCSArgs	int	Id **ids	Return the argument Ids for the competence set expression.
TOG_specGetRelation	Expn		Return the relation expression for the specification relation.
TOG_specGetRelArgs	int	Id **ids	Return the argument Ids for the relation expression.

TABLE 5 - Specification File Module Access Programs

Name	Type	Arguments	Description
TOG_specGetDomain	Expn		Return the domain expression for the specification relation.
TOG_specGetDomArgs	int	Id **ids	Return the argument Ids for the domain expression.
TOG_specNextAuxPred	bool		Make the ‘next’ auxiliary predicate current. If there are no more return BOOL_FALSE
TOG_specGetAuxPredId	Id		Return the Id of the current auxiliary predicate.
TOG_specGetAuxPredDef	Expn		Return the definition of the current auxiliary predicate.
TOG_specGetAuxPredArgs	int	Id **ids	Return the argument Ids for the current auxiliary predicate.
TOG_specNextAuxFunc	bool		Make the ‘next’ auxiliary function current. If there are no more return BOOL_FALSE
TOG_specGetAuxFuncId	Id		Return the Id of the current auxiliary function.
TOG_specGetAuxFuncDef	Expn		Return the definition of the current auxiliary function.
TOG_specGetAuxFuncDomain	Id		Return the Id of the domain predicate of the current auxiliary function.
TOG_specGetAuxFuncType	char *		Return the type of the current auxiliary function.
TOG_specGetAuxFuncArgs	int	Id **ids	Return the argument Ids for the current auxiliary predicate.
TOG_specNextIndPred	bool		Make the ‘next’ inductively defined predicate current. If there are no more return BOOL_FALSE
TOG_specGetIndPredId	Id		Return the Id of the current inductively defined predicate.
TOG_specGetIndPredI	int	char **i	Return the ‘I’ component of the current inductively defined predicate.
TOG_specGetIndPredG	Id		Return the Id of the ‘G’ component of the current inductively defined predicate.
TOG_specGetIndPredQ	Id		Return the Id of the ‘Q’ component of the current inductively defined predicate.
TOG_specGetUserDef	char*		Return the user definitions text.

5.2.2.2 Constants (TOG_const.c)

The representation (name) of every constant used in the specification is stored using this module. It encapsulates the data structure used for storing this information.

TABLE 6 - Constants Module Access Programs

Name	Type	Arguments	Description
TOG_constInit			Initialize the module internal data structure.
TOG_constLoad	bool	FILE *	Load a table of constant names from the file.
TOG_constDelete		Id	Delete a constant from the table.
TOG_constGetName	char *	Id char *buf	Write the name of the constant with the given Id into buf.

5.2.2.3 Variables (TOG_vars.c)

The name and type of each variable used in the specification is stored using this module. It encapsulates the data structure used for storing this information. The idTable module is used to implement efficient storage with fast retrieval of the information.

TABLE 7 - Variables Module Access Programs

Name	Type	Arguments	Description
TOG_varsInit			Initialize the module internal data structure.
TOG_varsLoad	bool	FILE *	Load a table of variable names and types from the file.
TOG_varsDelete		Id	Delete a variable from the table.
TOG_varsGetName	char *	Id char *buf	Write the name of the variable with the given Id into buf.
TOG_varsGetType	char *	Id char *buf	Write the type of the variable with the given Id into buf.

5.2.2.4 Applications (TOG_applic.c)

An application is any function or predicate used in the specification including both those defined in the specification (i.e. auxiliary functions and predicates) and those which are defined in the oracle programming language. The *form* of an application is an array of arity + 1 strings which are used with the argument expressions to construct the code which evaluates the application invocation. The name, arity, form and domain predicate Id of an application used in the specification is stored using the Applications module. It encapsulates the data structure used for storing this information. The idTable module is used to store the information.

TABLE 8 - Applications Module Access Programs

Name	Type	Arguments	Description
TOG_appInit			Initialize the module internal data structure.
TOG_appAdd		Id char *name int arity	Add an application to the module.
TOG_appDelete		Id	Delete a application from the table.
TOG_appSetForm		Id int num char *form	Set the 'num'th form string for the application with the given Id.
TOG_appGetForm	char *	Id int num	Return a pointer to the 'num'th form string for an application.
TOG_appSetDomain		Id Id domain	Set the Id of the domain predicate for the application.
TOG_appGetDomain	Id	Id	Return the Id of the domain predicate for the application.
TOG_appGetArity	int	Id	Return the arity of the application.
TOG_appGetName	char *	Id	Return the name of the application.

5.2.2.5 Inductively Defined Predicates (TOG_indPred.c)

The *instantiation* of an IDP is a string which is used in the oracle code to declare an instance of the IDP. The name, type and instantiation information for an IDP used in the specification is stored using the Inductively Defined Predicate module. It encapsulates the data structure used for storing this information. The idTable module is used to store the information.

TABLE 9 - Inductively Defined Predicates Module Access Programs

Name	Type	Arguments	Description
TOG_indPredInit			Initialize the module internal data structure.
TOG_indPredAdd		Id char *name char *type	Add an IDP to the module.
TOG_indPredDelete		Id	Remove an IDP from the module.
TOG_indPredSetInstantiation		Id char *inst	Set the instantiation of the IDP.
TOG_indPredGetInstantiation	char *	Id	Return the instantiation of the IDP.

TABLE 9 - Inductively Defined Predicates Module Access Programs

Name	Type	Arguments	Description
TOG_indPredGetType	char *	Id	Return the type of the IDP.
TOG_indPredGetName	char *	Id	Return the name of the IDP.

5.2.2.6 Table Holder (libtblhold.a)

The components of a specification that are mathematical expressions are stored using the Table Holder module, which is not a TOG module, but is the central module of the TTS. This module is described in further detail in [17].

5.2.3 Oracle Generation

The Oracle Generation module is responsible for converting the specification into the oracle implementation. It is sub-divided into the following sub-modules.

5.2.3.1 Oracle Structure (TOG_oracle.c)

The structure of the oracle (i.e. the procedures and their names) and the algorithm for constructing this structure is encapsulated by this module. It retrieves the components of the specification from the Specification Interface module and uses the other Oracle Generation sub-modules to construct the oracle.

TABLE 10 - Oracle Structure Module Access Program

Name	Type	Arguments	Description
TOG_oracle			Construct the oracle.

5.2.3.2 Expression (TOG_expn.c)

Mathematical expressions are decomposed into their component sub-expressions by this module. It encapsulates the interface to the Table Holder module and the algorithm for constructing procedures to implement expressions. The Code module is used to convert expressions into code.

TABLE 11 - Expression Module Access Program

Name	Type	Arguments	Description
TOG_expn	TOG_Line	Expn Path TOG_Line buf TOG_Cntxt parent	Translate the sub-expression at 'Path' and return the code to evaluate it in buf.

5.2.3.3 Code (TOG_code.c)

The knowledge of the syntax of the oracle programming language (C for this prototype) is encapsulated by this module. It converts the components of an expression into C code and combines them to form the oracle procedures, objects and statements.

TABLE 12 - Code Module Access Programs

Name	Type	Arguments	Description
TOG_codeConst	char *	Id id char * buf TOG_Cntxt cntxt	Write the code for the constant into buf.
TOG_codeVar	char *	Id id char * buf TOG_Cntxt cntxt	Write the code for the variable into buf.
TOG_codeQLE		Id quant_id Id var_id Id ind_pred_id TOG_Line exp TOG_Cntxt quant_proc	Write the code to evaluate the quantification on the expression given in exp. The code is the body of quant_proc.
TOG_codeApplic	TOG_Line	Id id int arity TOG_Line args[] TOG_Line line TOG_Cntxt cntxt	Write, into line, the code to evaluate the given application with the given arguments.
TOG_codeNewTable	TOG_Cntxt	int num_dims int shape[] TOG_TableType	Write the code to instantiate a table object.
TOG_codeTableCell		TOG_Cntxt table TOG_Cntxt cell TOG_TableValueType vT TOG_Line expn	Construct a function for a table cell.

TABLE 12 - Code Module Access Programs

Name	Type	Arguments	Description
TOG_codeTableSetHeader		TOG_Cntxt table TOG_Cntxt cell int header int cell_num	Write the code in 'initOracle' to initialize the header functions of a table.
TOG_codeTableSetMain		TOG_Cntxt table TOG_Cntxt cell char *index int num_dims int cell_num[]	Write the code in 'initOracle' to initialize the main grid functions of a table
TOG_codeNewPrivateProc	TOG_Cntxt	TOG_Cntxt parent char *name char *type	Create a new function internal to the oracle.
TOG_codeNewPublicProc	TOG_Cntxt	TOG_Cntxt parent char *name char *type	Create a new function which is an access program to the oracle.
TOG_codeNewIndPred	TOG_Cntxt	Id char *i int num char * g_name char * q_name	Construct the context for an inductively defined predicate object.
TOG_codeAddIndPred		Id id TOG_Cntxt cntxt	Write the code to instantiate an inductively defined predicate object.
TOG_codeAddQuantVar		Id id TOG_Cntxt cntxt	Add a variable to the context for use in quantification.
TOG_codeMakeHeading		TOG_Cntxt cntxt	Construct the code which forms the function heading.
TOG_codeMakePrototype	TOG_Line	TOG_Cntxt cntxt TOG_Line buf	Write a prototype for the context in buf.
TOG_codeConstructCall	TOG_Line	TOG_Cntxt caller TOG_Cntxt callee TOG_Line buf	Construct the code for 'caller' to call 'callee'.
TOG_codeTableValue	TOG_Line	TOG_Cntxt caller TOG_Cntxt table TOG_TableValueType type TOG_Line buf	Construct an invocation of the table 'value' method.
TOG_codeComment	TOG_Line	char *text TOG_Line line	Write the text as a comment.

TABLE 12 - Code Module Access Programs

Name	Type	Arguments	Description
TOG_codeProcValue		TOG_Cntxt TOG_Line val	Write the code so that the cntxt returns ‘val’.
TOG_codeAddIndexDecl	char *	int num_dim	Declare a variable ‘index’ in ‘initOracle’.
TOG_codeSetInitCntxt		TOG_Cntxt init	Set the ‘initOracle’ context.
TOG_codeCellElement	char *	TOG_TableValueType type	Translate the table value type into its name.

5.2.3.4 Context (TOG_context.c)

A *context* (`TOG_Cntxt`) is an ADT which represents a procedure in the oracle code and the collection of variables known within that procedure. It contains a list of arguments and internal variables, the name and type of the procedure and a TOG_Proc which contains the statements which make up the procedure. The Context module is used to manage knowledge of contexts and to avoid conflicts between variable names.

TABLE 13 - Context Module Access Programs

Name	Type	Arguments	Description
TOG_cntxtInit			Initialize the module internal data structure.
TOG_cntxtCleanUp			Destroy all contexts.
TOG_cntxtCreate	TOG_Cntxt	TOG_Cntxt char *name	Create a new context.
TOG_cntxtDestroy		TOG_Cntxt	Destroy a context.
TOG_cntxtGetName	char *	TOG_Cntxt char *buf	Return the name of the context.
TOG_cntxtSetType		TOG_Cntxt char *type	Set the type of the context.
TOG_cntxtGetType	char *	TOG_Cntxt	Return the type of the context.
TOG_cntxtGetVar	char *	TOG_Cntxt Id id char *buf	Return the name of a variable within a context.
TOG_cntxtAddVar		TOG_Cntxt Id id char *name bool isArg	Add a variable to the context.

TABLE 13 - Context Module Access Programs

Name	Type	Arguments	Description
TOG_cntxtRemoveVar		TOG_Cntxt Id id	Remove a variable from the context.
TOG_cntxtGetCode	TOG_Proc	TOG_Cntxt	Return the procedure for the context.
TOG_cntxtGetFirstArg	Id	TOG_Cntxt	Return the first argument to the context.
TOG_cntxtGetNextArg	Id	TOG_Cntxt	Return the next argutment to the context.
TOG_cntxtOutput		TOG_Cntxt	Output the procedure for the context using the output module.
TOG_cntxtAddTmpVar	char *	TOG_Cntxt char *name	Add a temporary variable to the context.

5.2.3.5 Procedures (TOG_procedures.c)

A *procedure* (`TOG_Proc`) is an ADT which represents the lines of text that form the source code for a function. The Procedures module encapsulates the data structure for storing these lines of text. The output module is used to write the text to the disk.

TABLE 14 - Procedures Module Access Programs

Name	Type	Arguments	Description
TOG_procInit			Initialize the module data structure.
TOG_procCreate	TOG_Proc		Create a new procedure.
TOG_procDestroy		TOG_Proc	Destroy a procedure.
TOG_procOutput		TOG_Proc	Write the procedure to the output file.
TOG_procAddHeading		TOG_Proc TOG_Line text	Add text to the procedure heading.
TOG_procAddDeclaration		TOG_Proc TOG_Line text	Add text to the procedure declarations.
TOG_procAddBody		TOG_Proc TOG_Line text	Add text to the body of the procedure.
TOG_procAddClosing		TOG_Proc TOG_Line text	Add text to the procedure closing.

5.2.4 Output

The Output module encapsulates text manipulation algorithms and data structures and the file system. It is sub-divided into the following sub-modules.

5.2.4.1 File Output (TOG_output.c)

The interface to the file system for writing the oracle source code files is encapsulated in the File Output module.

TABLE 15 - File Output Module Access Programs

Name	Type	Arguments	Description
TOG_outInit		char *outname	Initialize the module data structure. Open the appropriate output files.
TOG_outInclude		TOG_Line	Write text to the header file.
TOG_outTop		TOG_Line	Write text to the top of the code file.
TOG_outBody		TOG_Line	Write text to the body of the code file.
TOG_outClose			Close all files.

5.2.4.2 Line Buffer (TOG_line.c)

A *line* (`TOG_Line`) is an ADT which represents a sequence of arbitrary length strings of text. The Line Buffer module encapsulates the data structure for storing lines and provides access programs for their manipulation.

TABLE 16 - Line Buffer Module Access Programs

Name	Type	Arguments	Description
TOG_lineInit			Initialize the module data structure.
TOG_lineInsert	TOG_Line	TOG_Line char *text	Insert text at the begining of the line.
TOG_lineAppend	TOG_Line	TOG_Line char *text	Append the text to the end of the line.
TOG_linePack	TOG_Line	TOG_Line int width	Pack the line into strings of less than width characters.
TOG_lineGetText	char *	TOG_Line char *	Return a buffer containing all of the text in the line.
TOG_lineDestroy		TOG_Line	Destroy a line.
TOG_lineJoin	TOG_Line	TOG_Line ln1 TOG_Line ln2	Append ln2 to the end of ln1.

5.2.5 Utility Module

The TOG uses the following general purpose modules to store and manipulate data that can be referenced by integer or text keys.

5.2.5.1 Id Table (idTable.c)

The Id Table module facilitates the efficient storage and fast retrieval of a set of arbitrary data structures keyed by an integer identifier.

TABLE 17 - Id Table Module Access Programs

Name	Type	Arguments	Description
IDTab_create	IDTab	int size int dsSize	Create a table to store up to size items each of dsSize bytes
IDTab_destroy		IDTab	Destroy a table.
IDTab_insert	bool	IDTab int id void *dat	Add an item to a table.
IDTab_remove	bool	IDTab int id	Remove an item from a table.
IDTab_find	void *	IDTab int id	Retrieve an item from a table.

5.2.5.2 Name Table (nameTable.c)

The Name Table module facilitates the efficient storage and fast retrieval of a set of names (character strings) and their integer identifiers. Elements in the set can be retrieved quickly using either a name or an identifier as a key.

TABLE 18 - Name Table Module Access Programs

Name	Type	Arguments	Description
nameTableCreate	NameTable	int size	Create a name table for up to size elements.
nameTableDestroy		NameTable	Destroy a name table.
nameTableAdd	bool	NameTable char *name int id	Add an element to the name table.
nameTableDelete	bool	NameTable char *name	Delete the named element from the table.

TABLE 18 - Name Table Module Access Programs

Name	Type	Arguments	Description
nameTableDeleteIndex	bool	NameTable int id	Delete the identified element from the table.
nameTableGetIndex	int	NameTable char *name	Return the id of the named element.
nameTableGetName	char *	NameTable int id	Return the name of the identified element.
nameTableGetFirstIndex	int	NameTable	Return the id of the first element in the table.
nameTableGetNextIndex	int	NameTable	Return the id of the next element in the table, or NAME_NOT_FOUND if there are no more.

5.2.6 Status Reporting

The Status Reporting module is used by all modules of the TOG for the purpose of monitoring and reporting the status of other modules.

5.2.6.1 Error Token (TOG_error.c)

A status token is used to communicate success or failure status information between modules and between programs within the same module. This token is accessed through the Error Token module.

TABLE 19 - Error Token Module Access Programs

Name	Type	Arguments	Description
TOG_errSet		TOG_Token	Set the current error status.
TOG_errGet	TOG_Token		Return the current error status.
TOG_errGetStr	char *	TOG_Token	Return a descriptive string corresponding to the given error status.

5.2.6.2 Message Logging (sw_error.c)

To communicate information to the user for the purpose of debugging either the TOG or a specification, a log file is used. The Message Logging module provides an interface to this log file.

TABLE 20 - Message Logging Module Access Programs

Name	Type	Arguments	Description
SW_error		int level char *format <arguments>	Log a message to the logfile at the given error level. 'format' is a printf style format string and <arguments> are the arguments for it.
SW_errorInit		int level char *fileName	Open the named file as the logfile. Messages of level below 'level' are ignored.
SW_errorClose			Close the logfile.

5.3 Algorithm Overview

The algorithm for generating an oracle used by the TOG consists of the following steps:

1. Initialization: open files, initialize data structures.
2. Read Specification from file.
3. Create oracle program contexts.
4. Code Auxiliary Definitions: Create a C function for each, code the expression.
5. Code oracle programs: inDomain, inCompSet, inRelation.
6. Write and close files.
7. Free data structures.

5.3.1 Expression Coding

The mathematical expressions used in auxiliary definitions or in the specification relation are translated into code in the following manner: The expression syntax tree is traversed using a depth-first (i.e. innermost sub-expressions first) traversal and each sub-expression is implemented in turn as described in Chapter 4. The code that gives the value of each sub-expression is written into a buffer (`TOG_Line`) which is used to construct the code for the 'parent' expression. This process continues until the root expression has been implemented and the resulting code is used as the body of the procedure in the oracle.

6 Trial Application

In order to evaluate the practicality and effectiveness of the methods described in this thesis, and to gain an appreciation of their strengths and weaknesses, the methods were used to test some programs that are used in a commercial network management application.[1] This chapter gives a brief description of the programs to be tested and discusses the testing procedures, results of the testing and the lessons learned from this trial.

6.1 Program Overview

The programs to be tested together implement a module (hereafter known as the *hash module*) used to store elements (data structures) for quick retrieval using an integer key. This is achieved using two hash tables, referred to as table A and table B. Three of the module's access programs are tested: `HashAdd`, which adds an element to one of the tables, `HashFind`, which retrieves an element from one of the tables without changing the table, and `HashRemove`, which removes an element from one of the tables. Complete specifications and source code of these programs are given in Appendix A.

6.2 Test Procedure

As discussed in Chapter 3, the methods described in this thesis are intended for testing individual programs, not modules. The methods described in [38] and [40] would be useful for testing the externally observable behaviour of this module. To effectively test the hash module using the methods described in this thesis, three oracle programs, one for each access program, are used with a single test harness.

1. Provided by Newbridge Networks Corp., Kanata, Ontario.

The input to the test harness is a series of commands, each of which instruct it to either add, remove, or find an element in one of the two tables. Each command, together with the state of the hash module before the command is executed, forms a test case for one of the programs. The test case is only executed if it is in the competence set of the program, as determined by the appropriate `inCompSet` program. The TES made up of the test case together with the description of the state of the hash module following execution and the values returned by the program, is passed to the appropriate `inRelation` program to determine pass or failure of the test.

Since the hash module is part of a commercial software system, it has previously been carefully inspected and tested, so it is expected that no errors will be detected by further testing. To verify that the testing procedures does, in fact, detect errors when they occur, it is necessary to introduce some errors into the hash module. This is done by making small changes to the hash module code so as to slightly alter its behaviour such that it no longer satisfies its specification. Each modified version of the hash module (presumably containing only one coding fault) is tested separately using the same set of test harness commands.

6.3 Testing Results

Test suites for testing the hash module were generated randomly using a simple program based on the C language uniform distribution random number generator (`rand`). The tests were approximately uniformly distributed between the two tables. Table 21 summarizes the test suites and Table 22 summarizes the results of the testing.

As can be seen from the information in Table 22 for tests 3 through 8, the testing procedures were successful in detecting all of the errors inserted in the code. The large number of rejected test cases in tests 5 and 7 are due to the fact that the code modifications

TABLE 21 - Test Suite Descriptions

Suite	Number of commands			
	Add	Remove	Find	Total
A	3303	3363	3334	10000
B	4984	2516	2500	10000
C	489	256	246	1000

TABLE 22 - Summary of Hash Module Test Results

#	Test Suite	Code Modifications	Passed	Failed	Rejected
1	A	Unmodified	10000	0	0
2	B	Unmodified	10000	0	0
3	C	Neglect to append existing list to added item in `HashAdd` (line 182)	881	119	0
4	C	Neglect to append added item to existing list in `HashAdd` (line 183)	511	489	0
5	C	Neglect to set 'identifier' in `HashAdd` (line 181)	3	6	991
6	C	Always return NULL from `HashRemove`	737	263	0
7	C	Neglect to re-join list when element removed in `HashRemove` (line 233)	29	14	957
8	C	Use wrong size to calculate hash index for table B (line 123)	746	254	0

introduced for these tests were such that the integrity of the data structure was not maintained correctly, and thus the tests cases were not in the competence set of the programs.

Since the test harness copies the entire data structure before each command is executed, and the time required to do this depends on the number of elements in the hash tables, the time required to execute a suite of tests is dependent on the size to which the tables grow and hence the ratio of 'add' to 'remove' commands in the suite. (Since 'find' commands do not change the size of the table, their frequency is not relevant to this analysis.) Test #1, which used test suite A in which the frequency of 'add' commands is approximately equal to that of 'remove' commands, took an elapsed time of about two and a half minutes[1] (approx. 15 ms per command). On the other hand, test #2, which used test suite

B in which the frequency of ‘add’ commands was about twice that of ‘remove’ commands, required an elapsed time of about five and a half minutes (approx. 34 ms per command). Both of these times are considered to be quite acceptable, especially considering how long it would take to manually verify the results of 10 000 test executions.

6.4 Discussion

As was the intention, the process of using these methods to test commercial software brought to light some of the difficulties of using these methods in a realistic software development situation. These difficulties are discussed in this section.

6.4.1 Specification Faults

One of the recognized dangers of using a formal specification to derive an oracle for testing software is that the oracle is only as good as the specification from which it was derived. On several occasions during preliminary testing of this module, it was thought that a fault had been discovered. On closer examination, however, it was realized that the fault was actually a fault in the specification, not the implementation of the hash module itself. Of course, if there is no specification, then there is no hope of generating an oracle.

Careful inspection is an obvious method of removing faults from the specification. However, if an oracle is generated from the specification, then other fault detection methods are possible. Experience has shown that inspection of the oracle code itself is a useful way of detecting faults in the specification—the structure of the code mirrors that of the specification. Also, it is possible to test the specification by executing the oracle with a TES for which the results are known (e.g. from a previous ‘correct’ version of the PUT or a TES that has been manually produced or checked).

---

1. All times are elapsed time running under OSF/1 V2.0 on a lightly loaded DEC Alpha.

6.4.2 Test Harness Construction

Since the `inRelation` programs in general take both starting and stopping state descriptions as arguments, the value of the data structure in the starting state is copied to other variables by the test harness before executing the PUT. Such a test harness is based on the design of the data structure, and the data structure must be exported from the PUT so that it can be accessed by the test harness.

In the case of the hash module, the data structure is complex enough that copying it is itself a potentially error prone activity and, in fact, in preliminary testing some errors were found in that portion of the test harness code. In addition, the program `hash_getTables` was required (an addition to the hash module) so that the data structure could be exported.

An alternative design is to have the hash module access programs directly invoke the test harness, which could copy the data structure and call the oracle programs as necessary. This implementation would likely involve more changes to the hash module code than that described above, and hence it is more likely to adversely affect the behaviour of the module. (The module that is tested should be as close as possible to the module that will be used in the real system.)

6.4.3 Non-Testable Properties

There is a class of properties which are impossible or impractical to test using an oracle generated using these methods. Two such properties are illustrated by the hash module: the requirement that a program should call another program some number of times, such as for `HashOperateOnNext`; and properties of the data structure that are not retained when the data structure is copied, such as the value of the '`sanityCheck`' field.

One of the arguments to `HashOperateOnNext` is a pointer to a program which is to be called by `HashOperateOnNext` for some subset of the elements in the hash table. It is not possible to give a relational specification of `HashOperateOnNext`, since its effect on the data structure and even the set of elements upon which it will act are not known without knowledge of the program which is its argument. It is, therefore, only possible to specify the program in conjunction with the program which is to be passed as its argument. Also, since the TES does not normally include information about which programs were called during the execution of the PUT, the oracle can only check the effect on the data structure of calling the given program, not that it was actually called the correct number of times. A potential solution to this problem is discussed in Section 7.2.

The `sanityCheck` field in the `hashStruct` data structure is used as a fault detection mechanism by this module, and presumably others in the system. The value of `sanityCheck` is set to be equal to the location in memory of the instance of the data structure (i.e. it is a pointer to itself). Unfortunately, if this data structure is copied to a new location in memory (i.e. so that it can be used as part of the start state in the TES) the value of `sanityCheck` will no longer have the desired property since its location in memory has changed, so it is not possible to ascertain the correctness of this value from a copy. For this reason the integrity of this field cannot be checked by `inRelation` for 'before' values of the data structure (this is overcome by the use of the '*sane*' auxiliary predicate only when referring to 'after' values).

6.4.4 Oracle Generation

When using these methods to test multiple programs with a single test harness, some of the assumptions made in the design of the oracle are not valid. Two such invalid assumptions are illustrated through the testing of the access programs of the hash module:

1) that it is sufficient to choose fixed names for the oracle access programs, and 2) that the auxiliary predicates and functions should be implemented by procedures internal to the oracle.

All oracles produced by the TOG have access programs with the same names (`initOracle`, `inRelation`, `inCompSet` and `inDomain`), so a name conflict results if more than one oracle is accessed by the same test harness. In the case of the hash module this was avoided by prefacing the name of each oracle program with the name of the hash module program for which it is the oracle. For example, the `inRelation` program for `HashAdd` is called `hashAdd_inRelation`. For this prototype, the renaming is done using a script controlled stream editor, but clearly support for different oracle program names could be added as an option for future versions of the TOG.

A large percentage of the auxiliary predicates defined in the hash module specification are used by more than one of the program specifications, so there is a fairly large amount of code duplication between the oracle code files. This duplication could be avoided by producing an additional code file which contains the auxiliary function and predicate implementations, and having each oracle use those programs as necessary.

7 Discussion and Conclusion

This chapter discusses potential applications for the TOG and some of the limitations of the methods.

7.1 Applications for This Work

This work is applicable to the same problems as is any methodology that produces an executable test oracle. The most obvious of these is in the ‘unit testing’ phase of software development during which small, relatively independent, components of the software, or ‘units’, are tested independently. A test harness, such as that illustrated in Figure 1 on page 36, can be used to invoke a PUT for some set of test cases. Calls to the oracle functions determine if each test case passed or failed, and these results are reported so that the PUT can be corrected.

Another possible application for an executable oracle is *in-situ* testing: The code for a software system can be modified by adding calls to the oracle programs for certain critical components, so that failures of these components during system operation (e.g. during system testing or beta trials) are reliably detected and reported. The behaviour of the resulting program is similar to that of those developed using the methods described in [18] or [35]. For *in-situ* testing, no test harness need be constructed since the PUT is called as usual by the system.

In [1], Antoy and Hamlet describe another way of using executable oracles to which this work can be adapted. Their specifications, and hence their oracles, are somewhat different from those used in this work in that they use algebraic specifications of ADTs and require that the user provide a ‘representation mapping’ to map from the con-

crete data structure to the abstract specification. Their oracles thus test that properties expressed in the specification hold in the abstract sense. Since relational program specifications, as used in this work, are in terms of the concrete data structure, no representation mapping is needed—the oracle tests that the concrete data structure is modified in the prescribed manner. To use oracles as generated by the TOG to create a self-checking ADT similar to Antoy and Hamlet's requires that a set of oracle programs be generated for each access program for the ADT (we assume that an ADT consists of a set of access programs that operate on a common data structure). Calls to these oracle programs can then be embedded in the ADT code in the manner described in [1].

A further application of this work—that of enforced documentation consistency—derives from the fact that the oracle is generated directly from the program documentation. It has been noted that one of the factors that reduces the value of program documentation is the fact that it cannot be relied upon to be accurate (i.e. consistent with the code) since programmers can easily modify the code without updating the documentation. If a TOG generated test oracle is always used to test a program before it is released, then we are assured that the documentation is consistent with the code. A correct program will only pass the tests if the documentation is accurate.

7.2 Limitations of the Method

As with any program documentation technique, relational program documentation is not ideally suited for all types of programs and hence it is difficult to apply this work in some cases. One class of programs that is difficult to document using this method are those that manipulate the data structure in a manner other than simply changing the value of variables. Examples of this include dynamic memory allocation (i.e. increasing the size of the data structure), input and output, and process control. It is difficult to

express the characteristics of the stop state for these programs since relational operators to represent such characteristics as “is a valid block of memory on the heap” do not exist, in general. Some of these problems are being addressed by Brian Bauer in his M.Eng. thesis [3].

Programs for which there is a requirement on the intermediate states that the computer may be in during execution, such as “if condition C is true then call procedure x” or “don’t call x more than n times” (as for the Dutch National Flag example discussed in [28]), are also difficult to document using these methods. This is because the specifications used in this work are relations which contain only the start state and stop state, and do not allow any restrictions on the intermediate states. Even if these can be formally specified, a TES does not include information that can be used to determine intermediate states, so the oracle can not determine if such a program meets the specification. One, somewhat artificial, solution to this problem is to add to the data structure information which represents the relevant information about the intermediate states (e.g. the number of times ‘x’ was called). Using that solution, it is difficult to state in a relational specification, and hence to test, that the added data structure elements actually represent the intended information.

It is also possible to write a specification for which the oracle will not terminate or will only terminate after an unreasonable amount of time. Non-termination can be caused by either a non-terminating recursion in an auxiliary definition, errors in the definition of an inductively defined predicate or a non-terminating ‘primitive’ (i.e. defined in the programming language) function. Slow termination can be caused, for example, by quantification over large sets. For example, consider the well known ‘shortest path problem’ for which a specification is given in [32]. An oracle based on this specification enumerates all possible paths through the directed graph to ensure that there is no valid path with a

smaller path weight—an O(n!) calculation. The responsibility for avoiding such non-terminating or slowly-terminating oracles rests with the user (i.e. specifier/verifier). Non-termination can only be avoided by careful definition of auxiliary predicates/functions and judicious use of well tested/verified primitive functions. For problems such as the shortest path problem, it is not practical to test the whole program against the specification for large graphs, but it may be practical to test some sub-programs called by it and then to use other techniques to verify the top level code.

It is also possible for the PUT to be a non-terminating program in some cases. Since the oracle programs can only be used either before the PUT is invoked or after it has terminated, there is no means for these functions to detect that the PUT has not terminated (or has exceeded some reasonable time limit). This responsibility must rest with the test harness. Note, however, that the oracle programs do provide a means of detecting test cases for which the PUT should not terminate (i.e. those not in the domain of the program relation) or may not terminate (i.e. those in the domain but not in the competence set) through `inDomain` and `inCompSet`, respectively.

Finally, in some cases it may be possible to document a program using the methods used in this thesis in such a manner as to make it impossible to generate an executable oracle from the documentation. For example, an oracle cannot be generated for programs with a data structure that includes items for which the state (value) cannot be determined by the oracle programs (e.g. the computer display). In these cases, some other form of oracle or additional test equipment (e.g. terminal simulator) are necessary.

7.3 Future Work

As described in Chapter 6, the TOG has been tested and evaluated using some small programs which has shown that the methods are viable in these cases. More experi-

ence with applying it to a wide variety of industrial software applications would allow us to draw more general conclusions about the viability and usefulness of the methods. Suggestions for improvements in the TOG and oracle designs would undoubtedly result from that work.

Experience has shown that there are some auxiliary predicates and functions, or forms of auxiliary predicates and functions, that frequently appear in specifications of the form used in this work. For example, it is often the case that a specification states that some set of variables are not changed—denoted by the auxiliary predicate 'NC' in [28]. In this work the specific form of this predicate must be specified in the documentation (see the definition for NC given in Section 3.3 on page 24). It would be convenient if this definition could be produced automatically from a shorthand notation as used in [28].

It has been suggested that, in cases where the specification relation is a function, (i.e. it contains only one stopping state for any given starting state), it would be possible for the oracle to output a description of the correct stopping state for each test case and allow the test harness to determine if the program is in the right state. It is not, in general, possible to automatically generate such an oracle from specifications of the form used in this work, even if they are functional. (e.g. Consider a specification for a program to solve a system of n linear equations in n unknowns—the specification is functional but an oracle that outputs the correct stopping state could not be generated automatically.) It may be possible to automatically generate such oracles for some limited set of specifications but that would require significant modifications to this work.

The TOG design was chosen carefully to allow the programming language used in the oracle to be changed easily, but only C has been used in this prototype. A more broadly applicable TOG would allow the user to choose among several popular program-

ming languages to facilitate interfacing with PUTs written in these languages. This could be accomplished by providing several Code sub-modules and having the Oracle Generation module select the appropriate one according to the user's request.

Finally, the TTS is envisioned as a set of interworking tools for documenting software. As the tools in this system mature, there will be opportunities to integrate related tools so that they appear seamless to the user. For example, the TOG could be invoked as a menu item from a tool for editing program documentation. This would require modifying or replacing the User Interface and possibly the Specification Interface modules of the TOG.

7.4 Conclusions

The development and application of the TOG prototype has shown that it is feasible to automatically generate executable test oracles from relational program documentation. The experience of applying these methods to industrial software has shown, however, that there are some limitations to these methods:

- The documentation used to generate the oracle is almost as complicated as the PUT and needs to be checked carefully.
- Certain classes of program behaviour cannot easily be documented and tested using these methods.
- A suitable test harness may be a non-trivial program, which must also be checked carefully.

Despite these limitations the availability of an executable oracle has the following benefits for the software testing process:

- faster test analysis, hence reduced cost, and
- reliable failure detection, hence increased value.

In addition, because the oracle is generated directly from the program documentation, which can be ensured to be consistent with the program as described in Section 7.1, the value and usefulness of this documentation is greatly increased.

Appendix A - Hash Module Documentation

A.1 Introduction

This appendix gives the complete specifications and code for a small module taken from a network management application developed by Newbridge Networks Corporation of Kanata, Ontario[1]. The specifications were developed by the author using the source code and an informal module description provided by the module designers as a guide. The specifications are used to generate an oracle and test the code as described in Chapter 6.

A.2 Internal Design Documentation

A.2.1 Informal Description

The module implements a collection of hash tables which are used for storing user data objects for quick retrieval. The data object contains a numeric key value which is used to locate it in a table. Each table is constructed using a fixed length array of pointers to the beginning of a linked list of data objects. Since each list is allowed to grow indefinitely, the number of elements in each table is not bounded. A simple hashing algorithm is used to calculate in which list an element is stored and each list is sorted in increasing order.

1. The code and design of this module are proprietary property of Newbridge Networks Corporation and are used here by permission. This information may not be copied or used in any manner without explicit written permission from Newbridge Networks Corp.

A.2.2 User Definitions

A.2.2.1 Constants

The following constants define the size of the hash table arrays and are parameters to the program function specifications, below. Both constants must be powers of 2 (i.e. 2^n).

```
#define A_HASH_SIZE (1 << 10)       /* 1024 for A hash table */
#define B_HASH_SIZE (1 << 4)        /* 16 for B hash table */
```

The following constants are used to indicate which table is to be used by an access program.

```
#define HASH_A 1 /* A structures */
#define HASH_B 2 /* B structures */
```

A.2.2.2 Data Structures

The following structure is used to contain a single element of the user's data.

```
struct hashStruct {
   struct hashStruct *sanityCheck;
                        /* can be used to help detect corruption */
   unsigned int identifier;         /* the external key */
   struct hashStruct *hashNext; /* ptr to next elem in list */
   int data;[1]
};
```

A.2.2.3 Hash Tables

There are two tables implemented by this module, they are statically allocated as follows.

```
static struct hashStruct *AHashArray[A_HASH_SIZE];
                                        /* A hash table */
static struct hashStruct *BHashArray[B_HASH_SIZE];
                                        /* B hash table */
```

---

1. Any user data could be defined here, a simple integer is used for testing.

A.2.3 Program Functions

A.2.3.1 HashAdd

TABLE 23 - HashAdd Program Description

unsigned int
HashAdd(unsigned int theTable, unsigned int theId, struct hashStruct *thePtr)

external variables:	struct hashStruct *AHashArray[] struct hashStruct *BHashArray[]

$\mathbf{C_{HashAdd}}$ = ('theTable = HASH_A ∨ 'theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))

$\mathbf{R_{HashAdd}}$ = (('theTable = HASH_A ∨ 'theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))) ⇒

	'theTable = HASH_A ∧	
	inTable('AHashArray, A_HASH_SIZE, 'theId)	¬*inTable*('AHashArray, A_HASH_SIZE, 'theId)
HashAdd =	FAIL	SUCCESS
AHashArray[]' |	*Aequal*('AHashArray[], AHashArray'[])	*sane*(AHashArray'[hash('theId, A_HASH_SIZE)]) ∧ *insertedA*('AHashArray[], AHashArray'[], 'theId, 'thePtr)
BHashArray[]' |	*Bequal*('BHashArray[], BHashArray'[])	*Bequal*('BHashArray[], BHashArray'[])

	'theTable = HASH_B ∧	
	inTable('BHashArray, B_HASH_SIZE, theId)	¬*inTable*('BHashArray, B_HASH_SIZE, theId)
HashAdd =	FAIL	SUCCESS
AHashArray[]' |	*Aequal*('AHashArray[], AHashArray'[])	*Aequal*('AHashArray[], AHashArray'[])
BHashArray[]' |	*Bequal*('BHashArray[], BHashArray'[])	*sane*(BHashArray'[hash(theId, B_HASH_SIZE)]) ∧ *insertedB*('BHashArray[], BHashArray'[], 'theId, 'thePtr)

A.2.3.2 HashRemove

TABLE 24 - HashRemove Program Description

struct hashStruct*
HashRemove(unsigned int theTable, unsigned int theId)

external variables: struct hashStruct *AHashArray[]
struct hashStruct *BHashArray[]

$\mathbf{C_{HashRemove}}$ = ('theTable = HASH_A ∨ 'theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))

$\mathbf{R_{HashRemove}}$ = (('theTable = HASH_A ∨ 'theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))) ⇒

	'theTable = HASH_A ∧	
	¬*inTable*('AHashArray[], A_HASH_SIZE, 'theId)	*inTable*('AHashArray[], A_HASH_SIZE, 'theId)
HashRemove |	HashRemove = NULL	*sameData*(HashRemove, *findElem*('AHashArray[hash('theId, A_HASH_SIZE)], 'theId))
AHashArray'[] |	*Aequal*('AHashArray[], AHashArray'[])	*sane*(AHashArray'[hash('theId, A_HASH_SIZE)]) ∧ *deletedA*('AHashArray[], AHashArray'[], 'theId)
BHashArray'[] |	*Bequal*('BHashArray[], BHashArray'[])	*Bequal*('BHashArray[], BHashArray'[])

	'theTable = HASH_B ∧	
	¬*inTable*('BHashArray[], B_HASH_SIZE, theId)	*inTable*('BHashArray[], B_HASH_SIZE, theId)
HashRemove |	HashRemove = NULL	*sameData*(HashRemove, *findElem*('BHashArray[hash('theId, B_HASH_SIZE)], 'theId))
AHashArray'[] |	*Aequal*('AHashArray[], AHashArray'[])	*Aequal*('AHashArray[], AHashArray'[])
BHashArray'[] |	*Bequal*('BHashArray[], BHashArray'[])	*sane*(BHashArray'[hash('theId, B_HASH_SIZE)]) ∧ *deletedB*('BHashArray[], BHashArray'[], 'theId)

A.2.3.3 HashFind

TABLE 25 - HashFind Program Description

struct hashStruct*
HashFind(unsigned int theTable, unsigned int theId)

external variables: struct hashStruct *AHashArray[]
struct hashStruct *BHashArray[]

$\mathbf{C_{HashFind}}$ = (theTable = HASH_A ∨ theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))

$\mathbf{R_{HashFind}}$ = *Aequal*('AHashArray[], AHashArray'[]) ∧
Bequal('BHashArray[], BHashArray'[]) ∧
(('theTable = HASH_A ∨ 'theTable = HASH_B) ∧
(∀i, *ALists*(i) ⇒ *sorted*('AHashArray[i])) ∧
(∀i, *BLists*(i) ⇒ *sorted*('BHashArray[i]))) ⇒

	'theTable = HASH_A	'theTable = HASH_B
HashFind |	*sameData*(HashFind, *findElem*('AHashArray[hash('theId, A_HASH_SIZE)], 'theId)	*sameData*(HashFind, *findElem*('BHashArray[hash('theId, B_HASH_SIZE)], 'theId)

A.2.4 Auxiliary Predicate Definitions

ALists(unsigned int i)
$\stackrel{df}{=}$ inductiveDef[{0}, i+1, i < (A_HASH_SIZE-1)]

BLists(unsigned int i)
$\stackrel{df}{=}$ inductiveDef[{0}, i+1, i < (B_HASH_SIZE-1)]

Aequal(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id)
$\stackrel{df}{=}$ (∀i, *ALists*(i) ⇒ (*listEqual*(before[i], after[i]))

Bequal(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id)
$\stackrel{df}{=}$ (∀i, *BLists*(i) ⇒ (*listEqual*(before[i], after[i]))

deletedA(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id)
$\overset{df}{=}$ (∀i, *ALists*(i) ⇒ ((¬(hash(Id, A_HASH_SIZE) = i) ∧ *listEqual*(before[i], after[i])) ∨ *deletedList*(before[i], after[i], Id)))

deletedB(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id)
$\overset{df}{=}$ (∀i, *BLists*(i) ⇒ ((¬(hash(Id, B_HASH_SIZE) = i) ∧ *listEqual*(before[i], after[i])) ∨ *deletedList*(before[i], after[i], Id)))

deletedList(struct hashStruct *before, struct hashStruct *after, unsigned int Id)
$\overset{df}{=}$ ¬(before = NULL) ∧

before->identifier = Id	¬(before->identifier = Id)
listEqual(before->hashNext, after)	*sameData*(before, after) ∧ *deletedList*(before->hashNext, after->hashNext, Id)

inList(struct hashStruct *list, unsigned int Id)
$\overset{df}{=}$ ¬(list = NULL) ∧ (list->identifier = Id ∨ *inList*(list->hashNext, Id))

insertedA(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id, struct hashStruct *ptr)
$\overset{df}{=}$ (∀i, *ALists*(i) ⇒ ((¬(hash(Id, A_HASH_SIZE) = i) ∧ *listEqual*(before[i], after[i])) ∨ *insertedList*(before[i], after[i], Id, ptr)))

insertedB(struct hashStruct * before[], struct hashStruct * after[], unsigned int Id, struct hashStruct *ptr)
$\overset{df}{=}$ (∀i, *BLists*(i) ⇒ ((¬(hash(Id, B_HASH_SIZE) = i) ∧ *listEqual*(before[i], after[i])) ∨ *insertedList*(before[i], after[i], Id, ptr)

insertedList(struct hashStruct *before, struct hashStruct *after, unsigned int Id, struct hashStruct *ptr)
$\overset{df}{=}$ ¬(after = NULL) ∧

after->identifier = Id	¬(after->identifier = Id)
after = ptr ∧ after->data = ptr->data ∧ *listEqual*(before, after->hashNext)	*sameData*(before, after) ∧ *insertedList*(before->hashNext, after->hashNext, Id, ptr)

inTable(struct hashStruct *table[], unsigned int size, unsigned int Id)
$\overset{df}{=}$ *inList*(table[hash(Id, size)], Id)

listEqual(struct hashStruct *left, struct hashStruct *right)
$\stackrel{df}{=}$

	left = NULL	¬(left = NULL)
right = NULL	***true***	***false***
¬(right = NULL)	***false***	*sameData*(left, right) ∧ *listEqual*(left->hashNext, right->hashNext)

sameData(struct hashStruct * left, struct hashStruct *right)
$\stackrel{df}{=}$ (left = NULL ∧ right = NULL) ∨ (¬(left = NULL ∨ right = NULL) ∧
(left->identifier = right->identifier) ∧ (left->data = right->data))

sane(struct hashStruct * list)
$\stackrel{df}{=}$ (list = NULL) ∨ ((list = list->sanityCheck) ∧ ¬(list->hashNext = NULL) ⇒
((list->identifier < list->hashNext->identifier) ∧ *sane*(list->hashNext)))

sorted(struct hashStruct * list)
$\stackrel{df}{=}$ (list = NULL) ∨ (¬(list->hashNext = NULL) ⇒
((list->identifier < list->hashNext-> identifier) ∧ *sorted*(list->hashNext)))

A.2.5 Auxiliary Function Definitions

struct hashStruct * *findElem*(struct hashStruct *list, unsigned int Id)
$\stackrel{df}{=}$

list = NULL	¬(list = NULL) ∧ (list->identifier = Id)	¬(list = NULL) ∧ ¬(list->identifier = Id)
NULL	list	*findElem*(list->hashNext, Id)

A.3 Hash Module Code

A.3.1 hash.c

```
/*****************************************************************
 * $RCSfile: hash.c,v $   $Revision: 1.5 $
 * $Date: 1994/12/19 19:24:33 $
 * $State: Exp $
 *
 * Example code.
 *
 *****************************************************************/

/*****************************************************************
 *                      REVISION HISTORY
 *
 * $Log: hash.c,v $
 * Revision 1.5  1994/12/19  19:24:33  peters
 * Added hash_getTable.
 *
 * Revision 1.4  1994/12/15  20:23:47  peters
 * Made ANSI compatible.
 *
 * Revision 1.3  1993/11/05  20:54:19  peters
 * Changed AAA to A and BBB to B.
 *
 * Revision 1.2  1993/09/03  19:02:12  peters
 * Reformated to improve readability.
 *
 * Revision 1.1  1993/09/03  18:10:43  peters
 * Initial revision
 *
 *
 *****************************************************************/

/*****************************************************************************

 File: hash.c

 Contents:
   This file provides routines that hash a 32-bit unsigned external ID in
   order to find the element within some data structures.  New structure types
   can be added by defining a new HASH_XXX constant and adding a new case
   statement to the routines.

   This file contains the following exported routines:
     HashAdd
     HashFind
     HashRemove
     HashOperateOnNext

   This file contains the following local routines:
     DoFind
```

```
  Copyright 1991,1992,1993 Newbridge Networks Corporation.
******************************************************************************/

#include <stdlib.h>
#include "sw_error.h"
#include "stuff.h"
#include "hash.h"

/* The following structure must be a valid overlay for the first part */
/* of the structure to be hashed. */

struct hashStruct {
  struct hashStruct *sanityCheck;    /* can be used to help detect corruption */
  unsigned int      identifier;      /* the external key */
  struct hashStruct *hashNext;       /* ptr to next structure in hash list */
};

/* The following arrays are used to hold the actual hash tables.  They are */
/* NOT sized to hold the maximum number of expected elements but instead are */
/* sized to hold the number of elements for the "average" use of the */
/* structures.*/
/* It must be ensured that in the largest expected use of the structures, */
/* the linear scanning required to handle overflows will not be greater */
/* than is desired.  To provide for a simple and cheap hash function, the */
/* size of the arrays MUST be a power of two.  Note that it is trivial to */
/* remove this restriction if ever required. */

#define A_HASH_SIZE (1 << 10)  /* 1024 for A hash table */
#define B_HASH_SIZE (1 << 4)   /* 16 for B hash table */

static struct hashStruct *AHashArray[A_HASH_SIZE]; /* A hash table */
static struct hashStruct *BHashArray[B_HASH_SIZE]; /* B hash table */

/* Now define the hash function itself.  We know that the hash module will */
/* be used for structures whose identifiers are assigned in increasing order */
/* starting from 1, i.e. 1, 2, 3, ...  Thus, a trival hash function is to */
/* AND the external ID with (XXX_HASH_SIZE - 1).  This will provide hash */
/* indexes of 1, 2, 3, ..., and then wrap around to 0, 1, 2, ... etc. */
/* Thus, the array will get filled up, and then wrap around and overflow */
/* will start.  Of course, by then some of the hash entries could have */
/* been freed up. */

#define HASH_FUNC(id,size) ((id) & ((size) - 1))

/*
 */
/******************************************************************************

  Routine: DoFind

  Description:
    Searches for the specified external id.  TRUE is returned if the
    id was found in the hash table, and FALSE otherwise.  In addition,
    a pointer to the hashNext field of the preceeding item is returned.
    Note that if there is no preceeding item, then the pointer points
```

```
    to the hash table array entry, i.e. to the head of the linked list.

*****************************************************************************/
static unsigned int
DoFind(register unsigned int theTable,     /* which hash table is being used */
   register unsigned int theId,         /* the external id to be hashed */
   register struct hashStruct ***trailPtr)/* trailing ptr for searching */
{
  register struct hashStruct **hashArrayPtr;    /* ptr into hash array */
  register struct hashStruct *curPtr;          /* ptr to current item */
  register int loopCount;                       /* count for infinite check */


  switch (theTable) {
    case HASH_A:
      hashArrayPtr = &AHashArray[HASH_FUNC(theId, A_HASH_SIZE)];
      break;
    case HASH_B:
      hashArrayPtr = &BHashArray[HASH_FUNC(theId, B_HASH_SIZE)];
      break;
  }

  *trailPtr = hashArrayPtr;      /* ptr to hashNext field of preceeding item */

  /* If the hash entry if free, then then item does not exist yet. */
  if ((curPtr = *hashArrayPtr) == NULL)
    return(FALSE);

  /* We have a collision for the hash index, so now we have to search */
  /* and try to find the item.  When the search terminates, trailPtr */
  /* points to the hashNext pointer of the previous item.  If the item */
  /* was not found, then trailPtr points to the hashNext pointer of what */
  /* should be the previous item. */

  /* We know that curPtr cannot be NULL to start, so use a do while loop. */
  /* Make sure that we do not loop infinitely (which could happen if the */
  /* linked list has been corrupted). */
  loopCount = MAX_HASH_LOOP;
  do {
    SanityCheck(curPtr);
    CheckInfiniteLoop(loopCount);
    if (theId <= curPtr->identifier)
      return(theId == curPtr->identifier);  /* return TRUE if item found */
    *trailPtr = &curPtr->hashNext;          /* point to hashNext of new prev
                    item */
    curPtr = curPtr->hashNext;              /* point to next item in list */
  } while (curPtr != NULL);

  return(FALSE);  /* if we are at the end of list then item not found */
} /*DoFind */

/*
 */
/******************************************************************************
```

```
  Routine: HashAdd

  Description:
    Adds the specified id and block of storage to the hash table.  It
    is an error to add an an item to the hash table if it already exists.

*******************************************************************************/
unsigned int                                   /* returns SUCCESS or FAILURE */
HashAdd(register unsigned int theTable,    /* which hash table is being used */
   register unsigned int theId,          /* the external id to be added */
   register unsigned int *thePtr)    /* pointer to the item to be added */
{
  register struct hashStruct *itemPtr;  /* ptr to item being added */
  struct hashStruct **trailPtr;         /* ptr to next ptr of previous item */

  if (DoFind(theTable, theId, &trailPtr)) {
    SW_error(ERR_SERIOUS, "Attempt to add existing external id %u", theId);
    return(FAIL);
  }

  /* The item was not found, so insert it into the list. */
  itemPtr = (struct hashStruct *)thePtr;
  itemPtr->identifier = theId;
  itemPtr->hashNext = *trailPtr;    /* item's next is the previous' next */
  *trailPtr = itemPtr;              /* previous' next is now the new item */

  return(SUCCESS);
}/* HashAdd */

/*
 */
/*******************************************************************************

  Routine: HashFind

  Description:
    Returns the address of the item with the specified id.  If no item
    currently exists with the id, then NULL is returned.

*******************************************************************************/
unsigned int *                    /* return address of item with external id */
HashFind(register unsigned int theTable,   /* which hash table is being used */
     register unsigned int theId)         /*the external id to be found */
{
  struct hashStruct **trailPtr;         /* ptr to next ptr of previous item */

  if (DoFind(theTable, theId, &trailPtr))
    return((unsigned int *)*trailPtr);
  else
    return(NULL);
}/* HashFind */

/*
 */
/*******************************************************************************
```

```
  Routine: HashRemove

  Description:
    Removes the item from the hash table with the specified id.  It is
    an error try to remove something which does not exist in the hash
    table.  The storage of the item itself is NOT freed.  This is the
    responsibility of the caller.  The address of the removed item
    is returned, or NULL if the item could not be found.

*****************************************************************************/
unsigned int *                    /* return address of item with external id */
HashRemove(register unsigned int theTable, /* which hash table is being used */
       register unsigned int theId)     /*the external id to be deleted */
{
  register struct hashStruct *itemPtr;  /* ptr to item being deleted */
  struct hashStruct **trailPtr;         /* ptr to next ptr of previous item */

  if (DoFind(theTable, theId, &trailPtr)) {
    /* Remove the item from the linked list and return address of item. */
    itemPtr = *trailPtr;
    *trailPtr = itemPtr->hashNext;  /* previous' next is now item's next */
    return((unsigned int *)itemPtr);
  } else {
    SW_error(ERR_SERIOUS, "Attempt to remove non-existant external id %u",
             theId);
    return(NULL);
  }
}/* HashRemove */

/*
 */
/*****************************************************************************

  Routine: HashOperateOnNext

  Description:
    Calls the passed in function with the pointers found in the  specified
    hash table. Will keep calling the specified function until that
    function returns something other than SUCCESS.
    Note that theId that is passed in is used as a place holder within the
    hash table, and does not mean that all ids > theId will be operated on.

*****************************************************************************/
unsigned int *                    /* return address of item with external id */
HashOperateOnNext(
    register unsigned int theTable,        /* which hash table is being used */
    register unsigned int theId,         /* seed value to find the next item */
    int      (*callActionFunc)(unsigned int *),/* the funnction to be called */
    int *rc)                        /* the return code of the called routine */
{
  struct          hashStruct   **trailPtr;  /* ptr to next ptr of previous
                       item */
  register struct hashStruct   *curPtr;     /* ptr to current item */
  register        int          loopCount;   /* count for infinite check */
```

```
  register        unsigned int hashIndex;   /* the index of the hash array */
  register        unsigned int startIndex;  /* starting hashIndex */
  register        unsigned int hashSize;    /* size of hash table being used */

  curPtr = NULL;

  switch (theTable) {
    case HASH_A:
      hashSize = A_HASH_SIZE;
      break;
    case HASH_B:
      hashSize = B_HASH_SIZE;
      break;
    default:
      return(NULL);
   }

  startIndex = HASH_FUNC(theId, hashSize);

  if (DoFind(theTable, theId, &trailPtr))
    curPtr = (*trailPtr)->hashNext;  /* point to next item in list */
  else
    curPtr = *trailPtr;

  /* Make sure that we do not loop infinitely (which could happen if the */
  /* linked list has been corrupted). */
  loopCount = MAX_HASH_LOOP;

  for (hashIndex  = startIndex; hashIndex < hashSize; hashIndex++){
    if (hashIndex != startIndex) { /* first pass may have curPtr already set */
      switch (theTable) {
        case HASH_A:
          curPtr = AHashArray[hashIndex];
          break;
        case HASH_B:
          curPtr = BHashArray[hashIndex];
          break;
      }
    }
    /* We know that curPtr could be NULL to start, so use a while loop. */
    while (curPtr != NULL) {
      SanityCheck(curPtr);
      CheckInfiniteLoop(loopCount);
      *rc = (*callActionFunc)((unsigned int *)curPtr);
      if (*rc != SUCCESS)
        return((unsigned int *)curPtr); /* address to pointer of current item */
      curPtr = curPtr->hashNext;        /* point to next item in list */
    }
  }
  return(NULL);
}/* HashOperateOnNext */

#ifdef TEST
/*****************************************************************************
   Routine: hash_getTable
```

```
   Description: Only included when the 'TEST' macro is defined.  This routine
   exports the addresses of the tables so that they can be copied for
   testing purposes.

****************************************************************************/
void
hash_getTables(void **tableA, int *sizeA, void **tableB, int *sizeB)
{
  *tableA = (void *)AHashArray;
  *sizeA = A_HASH_SIZE;
  *tableB = (void *)BHashArray;
  *sizeB = B_HASH_SIZE;
}

#endif
```

A.3.2 hash.h

```
/******************************************************************
 * $RCSfile: hash.h,v $   $Revision: 1.3 $
 * $Date: 1994/12/19 19:32:50 $
 * $State: Exp $
 *
 * Include file for example source hash.c
 *
 ******************************************************************/

/******************************************************************
 *                     REVISION HISTORY
 *
 * $Log: hash.h,v $
 * Revision 1.3  1994/12/19  19:32:50  peters
 * Use ANSI protyping.  Added hash_getTables().
 *
 * Revision 1.2  1993/11/05  20:55:37  peters
 * Changed AAA to A and BBB to B.
 *
 * Revision 1.1  1993/09/03  18:14:31  peters
 * Initial revision
 *
 *
 ******************************************************************/

/******************************************************************************

  File: hash.h

  Description:
    This file contains the declarations of extern variables and routines
    which are needed for users of the hash module.
```

```
  Copyright 1991,1992,1993 Newbridge Networks Corporation.

*****************************************************************************/

/* The following definitions indicate which hash table is being accessed */
#define HASH_A 1   /* A structures  */
#define HASH_B 2   /* B structures */

extern unsigned int HashAdd(unsigned int theTable, unsigned int theId,
   unsigned int *thePtr);
extern unsigned int * HashFind(unsigned int theTable, unsigned int theId);
extern unsigned int * HashRemove(unsigned int theTable, unsigned int theId);
extern unsigned int * HashOperateOnNext(unsigned int theTable,
   unsigned int theId, int (*callActionFunc)(unsigned int *), int *rc);
#ifdef TEST
extern void hash_getTables(void **tableA, int *sizeA, void **tableB, int
*sizeB);
#endif
```

A.3.3 stuff.h

```
/*****************************************************************
 * $RCSfile: stuff.h,v $   $Revision: 1.3 $
 * $Date: 1994/12/19 19:36:44 $
 * $State: Exp $
 *
 * Include file for util.c
 *
 *****************************************************************/

/*****************************************************************
 *                      REVISION HISTORY
 *
 * $Log: stuff.h,v $
 * Revision 1.3  1994/12/19  19:36:44  peters
 * Removed uneeded defines of NULL, TRUE & FALSE.
 *
 * Revision 1.2  1994/12/19  19:34:02  peters
 * Define ERR_SERIOUS & MemoryPanic()
 *
 * Revision 1.1  1993/09/03  18:16:37  peters
 * Initial revision
 *
 *
 *****************************************************************/


#define SUCCESS        1
#define FAIL           0

#define ERR_SERIOUS    SW_SERIOUS

#define MAX_HASH_LOOP  100000
```

```
#define CHECK_FAILURE  1
#define INFINITE_LOOP  2

#define SanityCheck(ptr) \
        if ((ptr) != (ptr)->sanityCheck) \
          { MemoryPanic(CHECK_FAILURE); }

#define CheckInfiniteLoop(theCount) \
        if (--(theCount) == 0) \
          { MemoryPanic(INFINITE_LOOP); }

#define MemoryPanic(code) printf(“Memory Panic: %d”, code); exit(code)
```

Appendix B - TOG Input File Format

B.1 Format Description

The TOG input file is a text file containing the complete specification for a program including all of the components described in Section 3.2. The components are identified by a single integer and can be arranged in any order in the file with the exception that the user definitions must be the last component. All expressions are in the format output by the `ExpnSave` access program of the Table Holder module, and can thus be input using the `ExpnLoad` access program. The following sections describe the format of each component type.

B.1.1 Constants

All constant names must be defined in a single table in the file. The beginning of the table is identified by the number 1 on a line by itself. The first line of the table is a number indicating the number of lines to follow. Each subsequent line is a number which is the symbol Id followed by a period (".") and the symbol name.

B.1.2 Variables

All variable names and types are defined in two adjacent tables in the file. The beginning of the first table is identified by the number 2 on a line by itself. The first table gives the Id and name for each variable in the same format as used for constants. The second table begins on the line immediately following the last line of the name table and gives the Id and type of each variable in the same format. The type of a variable is represented by the string of characters that would form its declaration in the C programming language, with the two character combination "`%s`" being a place-holder for the variable name. It is an error for the name and type tables to not contain the same set of Ids.

B.1.3 Program Specification

The beginning of the program specification is indicated by the number 6 on a line by itself. It is followed by the arity and formal argument Id list for the characteristic predicate of the competence set, and then by the expression which is that characteristic predicate. Following that is the arity, formal argument Id list and expression for the characteristic predicate of the domain of the specification relation and finally the arity, formal argument Id list and expression for the characteristic predicate of relational component of the specification relation. It is an error for a specification file to contain more than one program specification.

B.1.4 Auxiliary Predicate Definitions

The beginning of an auxiliary predicate definition is indicated by the number 3. It is followed by the Id, name, arity and formal argument Id list. The expression that is the definition of the predicate begins on the following line and is in a form that can be read by the Table Holder.

B.1.5 Auxiliary Function Definitions

The beginning of an auxiliary function definition is indicated by the number 4. The first line of the definition has the Id, name, and type of the auxiliary function. On the next line the first number is the arity of the function and it is followed by the Id of each of the formal arguments and finally the Id of the characteristic predicate of the function domain, or -1 for a total function. The auxiliary function definition expression begins on the next line.

B.1.6 Inductively Defined Predicate Definitions

The number 5 identifies the beginning of an IDP definition. It is followed by the Id and name of the IDP and the type of its argument. The next line contains the definition of the 'I' set which is a number (the cardinality of the set) and a string which is used to initialize an array to represent that set. The next line contains the Id of the G function (an auxiliary function) and that of the Q predicate (an auxiliary predicate).

B.1.7 Built-in Functions

The number 8 on a line by itself identifies the beginning of a built-in function declaration. This describes the method for generating code to invoke a function that is part of the programming language. The description consists of the Id, name and arity of the function followed by a list of arity+1 strings, each on a separate line, which, when written surrounding the actual argument representations, will invoke the function.

B.1.8 User Definitions

The number 7 identifies the beginning of the user definitions text. All of the text following it in the file is taken to be the user definitions text and is output to the beginning of the oracle code file (oracle.cc).

B.2 Formal Grammar

<spec> ::= <item_list> <user_def>

<item_list> ::= <item> | <item> <item_list>

<item> ::= <const_table> | <var_table> | <aux_pred> | <aux_func>
| <builtin_func> | <ind_pred> | <prog_relation>

<const_table> ::= **1** **CR**[1] <num> **CR** <const_list>

<const_list> ::= <const> | <const> <const_list>

<const> ::= <id>**.**<name> **CR**

<var_table> ::= **2** **CR** <num> **CR** <var_name_list> <num> **CR** <var_type_list>

<var_name_list> ::= <var_name> | <var_name> <var_name_list>

<var_name> ::= <id>**.**<name> **CR**

<var_type_list> ::= <var_type> | <var_type> <var_name_list>

<var_type> ::= <id>**.**<type> **CR**

<aux_pred> ::= **3** <id> <name> <arg_list> **CR** <expn>

<aux_func> ::= **4** <id> <name> <type> **CR** <arg_list> <id> **CR** <expn>

<builtin_func> ::= **8** **CR** <id> <name> <num> **CR** <form_list> **CR**

<form_list> ::= <form> | <form> **CR** <form_list>

<ind_pred> ::= **5** <id> <name> <type> **CR** <num> <I> **CR** <id> <id>

<prog_relation> ::= **6** **CR** <arg_list> **CR** <expn> <arg_list> **CR** <expn>
<arg_list> **CR** <expn>

<user_def> ::= **7** <text>

<arg_list> ::= <num> <args>

<args> := <id> | <id> <args>

---

1. **CR** represents the new-line character.

TABLE 26 - Formal Grammar Symbols

Symbol	Format	Interpretation
`<expn>`	As produced by the Table Holder	An expression.
`<form>`	Any characters.	The ith form is the text to precede the ith argument to the built in function.
`<I>`	A C array initialization.	The elements of the ‘I’ component of the IDP.
`<id>`	An integer.	A object identifier.
`<name>`	Alphanumeric characters.	The name of an object.
`<num>`	A natural number.	The number of elements to follow.
`<text>`	Any characters.	The user definitions text.
`<type>`	A C type.	The type of the variable or function.

References

1. Antoy, S. & Hamlet, R., "Self-Checking against Formal Specifications", *Proceedings of the Fourth International Conference on Computing and Information (ICCI)*, (May 1992), pp. 355-360.
2. ANSI/IEEE, "IEEE Standard Glossary of Software Engineering Terminology", *ANSI/IEEE Std. 729-1983*, American National Standards Institute, Institute of Electrical and Electronics Engineers, 1983.
3. Bauer, B. J., "Precise Documentation of 'Real' Programs, Masters Thesis Proposal", Communications Research Laboratory, McMaster University, September 1994. Private communication.
4. Bernot, G., Gaudel M. C. & Marre, B., "Software Testing Based on Formal Specifications: A Theory and a Tool", *Software Engineering Journal*, Vol. 6, pp. 387-405.
5. Chapman, D., "A Program Testing Assistant", *Communications of the ACM*, Vol. 25, No. 9 (September 1982), pp. 625-634.
6. Cheng, J. H. & Jones, C. B., "On the Usability of Logics Which Handle Partial Functions", *Proceedings of the Third Refinement Workshop*, Morgan, C. & Woodstock, J. (editors), Heidelberg, Germany: Springer-Verlag, 1990.
7. Dijkstra, E. W., "The Humble Programmer", *Communications of the ACM*, Vol. 15, No. 10 (October 1972), pp. 859-866.
8. Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Inc., 1976.
9. Farmer, W. F., "A Partial Functions Version of Church's Simple Theory of Types", *Journal of Symbolic Logic*, Vol. 55, No. 3, (September 1990), pp. 1269-1290.
10. Gannon, J., McMullin, P. & Hamlet, R., "Data-Abstraction Implementation, Specification, and Testing", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 3 (July 1981), pp. 211-223.
11. Goodenough, J. B. & Gerhart, S. L., "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2 (June 1975), pp. 156-173.
12. Gelperin, D. & Hetzel, B., "The Growth of Software Testing", *Communications of the ACM*, Vol. 31, No. 6 (June 1988), pp. 687-695.
13. Gries, D., *The Science of Programming*, Springer-Verlag, 1981.
14. Hamlet, R. G., "Testing Programs with the Aid of a Compiler", *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 4 (July 1977), pp. 279-290.
15. Hehner, E. C. R., "Predicative Programming Part 1", *Communications of the ACM*, Vol. 27, No. 2 (February 1984), pp. 134-143.
16. Howden, W. E., *Functional Program Testing & Analysis*, McGraw-Hill Book Compa-

ny, 1987.

17. Krasnor, C. & Parnas, D.L., "The Table Tool System: The Table Holder", *CRL Report No. 300*, Telecommunications Research Institute of Ontario (TRIO), May 1995.

18. Luckham, D.C., von Henke, F.W., Krieg-Brückner, B. & Owe, O., *ANNA A Language for Annotating Ada Programs Reference Manual, Lecture Notes in Computer Science 260*, Goos, G. & Hartmanis, J. (editors), Springer-Verlag, 1987.

19. Mills, H. D., "Function Semantics for Sequential Programs", *Proceedings of the IFIP Congress 1980*, North Holland 1980, pp. 241-250.

20. Myers, G. J., *The Art of Software Testing*, John Wiley & Sons, 1979.

21. Ostrand, T. J. & Balcer, M. J., "The Category-Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, Vol. 31, No. 6 (June 1988), pp. 676-686.

22. Panzl, D. J., "Automatic Software Test Drivers", *Computer*, April 1978, pp. 44-50.

23. Panzl, D. J., "A Language for Specifying Software Tests", *AFIPS National Computer Conference Proceedings*, Vol. 47 (1978), pp. 609-619.

24. Parnas, D. L., "On a 'Buzzword': Hierarchical Structure", *Proceedings of the IFIP Congress 1974*, North Holland 1974, pp. 336-339.

25. Parnas, D. L., "A Generalized Control Structure and Its Formal Definition", *Communications of the ACM*, Vol. 26, No. 8 (August 1983), pp. 572-581.

26. Parnas, D. L. & Wadge, W. W., "Less Restrictive Constructs for Structured Programs", *Technical Report 86-186*, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), October 1986, 16 pgs.

27. Parnas, D.L. & Madey, J., "Functional Documentation for Computer Systems Engineering (Version 2)", *CRL Report No. 237*, Telecommunications Research Institute of Ontario (TRIO), September 1991, 14 pgs.

28. Parnas, D. L., Madey, J. & Iglewski, M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, Vol. 20, No. 12 (December, 1994), pp. 948-976.

29. Parnas, D. L., "Tabular Representation of Relations", *CRL Report No. 260*, Telecommunications Research Institute of Ontario (TRIO), November 1992, 17 pgs.

30. Parnas, D. L., "Predicate Logic for Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 19, No. 9 (September 1993), pp. 856-862.

31. Parnas, D. L., "Mathematical Description and Specification of Software", *Proceedings of IFIP World Congress 1994,* Vol. I (August 1994), pp. 354-359.

32. Peters, D. K., "Shortest Path Algorithm - Formal Program Documentation", *CRL Report No. 280*, Telecommunications Research Institute of Ontario (TRIO), February 1994, 11 pgs.

33. Pressman, R. S., *Software Engineering A Practitioner's Approach*, Third edition, McGraw-Hill Book Company, 1992.

34. Richardson, D.J., Aha, S.L. & O'Malley, T.O., "Specification-based Test Oracles for Reactive Systems", *Proceedings of the 1992 International Conference on Software Engineering (ICSE)*, (May 1992), pp. 105-118.

35. Rosenblum, D. S., "A Practical Approach to Programming With Assertions", *IEEE Transactions on Software Engineering*, Vol. 21, No. 1 (January 1995), pp. 19-31.

36. Schach, S. R., *Software Engineering*, Aksen Associates Inc., 1990.

37. Stocks, P. & Carrington, D., "Test Template Framework: A specification-based testing case study", *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, (June 1993), pp. 11-18.

38. Wang, Y., *Specifying and Simulating the Externally Observable Behavior of Modules*, Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada (August 1994), 130 pgs.

39. Weyuker, E. J., "On Testing Non-testable Programs", *The Computer Journal*, Vol. 25, No. 4 (1982), pp. 465-470.

40. Woit, D. M., *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules*, Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada (February 1994), 77 pgs.