Using Test Oracles Generated from Program Documentation

Dennis K. Peters, *Student Member, IEEE* David Lorge Parnas, *Senior Member, IEEE*

Abstract—This paper illustrates how software can be described precisely using LD-relations, how these descriptions can be presented in a readable manner using tabular notations, and one way such descriptions can be used to test programs. We describe an algorithm that can be used to generate a test oracle from program documentation, and present the results of using a tool based on it to help test part of a commercial network management application. The results demonstrate that these methods can be effective at detecting errors and greatly increase the speed and accuracy of test evaluation when compared with manual evaluation. Such oracles can be used for unit testing, –in situ˝ testing, constructing self-checking software and ensuring consistency between code and documentation.

Index Terms—Program testing, test oracle, formal specification, finite state machine.

1 Introduction

The Software Engineering Research Group at McMaster University is studying ways to improve the quality and maintainability of software systems by using precise design documentation. Many authors have found that relational documentation, written using tabular expressions, is precise and readable [11], [7], [15], [12], [16], [25], [27]. Such documentation clearly communicates the intended behavior of the software without the reader needing to read the code. This contributes to the quality of the system by ensuring that programmers, reviewers, maintainers and designers of other components can understand the designer's intentions. In addition it can be used in testing and thus we can ensure that the documentation is consistent with the code, so it can be trusted.

Functional testing involves executing a program under test (PUT) and examining the output [13]. Functional testing requires an *oracle* to determine whether or not the output from a program is correct [37]. Often the oracle is a human, but the process can be time-consuming, tedious and error-prone. If the program documentation is mathematical, it is possible to derive a software oracle from it. The software oracle makes evaluation of test results inexpensive and reliable.

We have developed a Test Oracle Generator (TOG) that will produce a software oracle from design documentation that is:

- precise and relatively readable,
- a minimal statement of requirements,
- written in terms of the data structure, and
- written using a relatively expressive notation.

The generated oracle meets the following criteria.

- For any test result (input, output pair) the oracle can be used to determine whether or not the PUT satisfied the specification.
- It can be used to determine whether or not the specification allows termination for a particular test case, and, if it does, if the program is required to terminate for that case.
- It does not require precalculated "expected results" for the test cases.
- It does not require that there be a unique correct answer.
- It does not assume the existence of a previous version of the PUT that can be assumed to be correct.

Other authors have addressed the problem of generating an oracle (e.g., [2], [18], [32], [35]), but our approach has four characteristics that make it unique.

1. We use documentation that is separate from the code rather than embedded in it. This documentation summarizes the program for people who will not read the code.
2. We can use other tools to manipulate or check properties of the specifications.
3. We use tabular notations, which make our specifications much more understandable and easily reviewed.
4. Since we use LD-relations for our specifications, rather than the more popular precondition and postcondition pairs, we are able to express, and detect, those cases for which a specification either requires or allows a program to be nonterminating.

2 Program Documentation Method

As described in [23], design documentation for a computer system should comprise the following documents.

- **System requirements document**. Treating the complete computer system (hardware and software) as a "black-box", it describes the relationships between the values of environmental quantities that should be maintained by the system.
- **System design document**. Identifies the computers within the computer system and describes how they

• Department of Electrical and Computer Engineering, Communications Research Laboratory, McMaster University, Hamilton, Ontario Canada L8S 4K1 E-mail: peters@mcmaster.ca
• Department of Computing and Software, Faculty of Engineering, Communications Research Laboratory, McMaster University, Hamilton, Ontario Canada L8S 4K1
IEEE Log Number 105204.

communicate with each other and with peripheral devices.

- **Software behavior specification**. Describes the required behavior of the software.
- **Software module guide**. Describes the division of the software into modules by stating the responsibilities of each module.
- **Module interface specifications**. Each treats a module as a "black-box", identifying those programs that can be invoked from outside the module (the *access-programs*), and describing the externally-visible effects of using them.
- **Module internal design documents**. Each describes a module's data structure, the intended interpretation of that data structure, and the effect of each access-program on the module's data structure.

In this work we are interested in generating oracles from the part of a module's internal design document that describes the behavior of a single access-program. Other authors have discussed producing oracles from module interface specifications for abstract data types (ADTs) specified using algebraic, or "trace" specifications, e.g., [3], [5], [8], [36].

The remainder of this section describes the relational program documentation method used in this work (which is based on that described in [24]) and discusses the applicability of this method.

2.1 Program Variables and State Descriptions

We view a digital computer as a state machine consisting of a finite set of memory locations and input and output registers, each of which is itself a finite state machine. The state of the computer is a function of the states of all of its components.

The following terminology is adopted from [24], [27].

execution. A (possibly infinite) sequence of states of the machine.

starting state. The first state in an execution.

terminating execution. A finite execution.

stopping state. The last state in a terminating execution.

executions of a program. The set of possible executions described by the program text.

Frequently, intermediate states of an execution are not of interest—we only require that the stopping state be correct for each starting state. For such cases we define:

execution summary. A pair containing only the initial and final states of a terminating execution.

We define the following terminology for describing program behavior:

program variable. A state machine that is a component of the computer.

program variable value. The state of the program variable.

data structure. The set of program variables whose values affect, or are affected by a program or set of programs. In this paper the term *state* is assumed to refer to the state of a program's data structure.

state description. A tuple giving the value of each program variable in the data structure. Note that for some programs the length of this tuple may vary during execution.

program variable name (identifier) . A string used to identify a program variable in a program text (i.e., code). Note that in some languages a program variable may have more than one name and several variables may be referred to by the same name.

We adopt the following convention from [10], [24], among others.

Let P be a program and $x_0, \ldots, x_k$ be the names of program variables used in P, then

- "x_i'" (to be read "x_i after") denotes the value of the program variable x_i in the stopping state.
- "'x_i" (to be read "x_i before") denotes the value of the program variable x_i in the starting state.

For describing program testing we define the following.

test case. A description of a starting state for a program.

test execution summary. (TES) An execution summary in which the first state is a test case and the second state is the state in which the program terminated.

2.2 Relational Specification

The set of acceptable execution summaries for a program is a binary relation. An *LD-relation*, L, is a pair $\langle R_L, C_L \rangle$ where R_L is a binary relation and C_L is a subset of the domain of R_L, which we call the *competence set.* [21], [24], [27] The domain, D_L, 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 complete set of acceptable execution summaries, and C_L be the set of starting states for which all executions of the program terminate. Thus, 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 $\langle x, y \rangle$ is an element of R_L, and
- for all starting states in C_L, P will always terminate.

Note that if a starting state $x \notin \text{domain}(R_L)$ then P cannot terminate in a way that satisfies L. For deterministic programs, R_L is a function. By convention, when C_L is exactly the domain of R_L (which is always true for deterministic programs), C_L need not be given.

In this paper a program is assumed to be specified by an LD-relation referred to as the *specification relation.*

In addition the documentation may include some text in the syntax of the programming language defining any basic symbols (e.g., constant names) and operators (e.g., structure element access) that are used in the specification.

2.3 Inductively Defined Predicates

To enable the oracle to enumerate the elements of a set over which an expression is quantified, we use a special form of inductive definition for the characteristic predicate of the set. We define an *inductively defined predicate*, P, as the characteristic predicate of a set, **S**, formed in the following way. Given a triple, $\langle \mathbf{I}, \mathrm{G}, \mathrm{Q} \rangle$, where:

- **I** (initial) is a finite set of elements,
- G (generator) is a function,
- Q (don't quit) is a predicate, and

$$(\forall x \in \mathbf{I}, (\exists m, (\forall j, (0 < j < m \Rightarrow \mathrm{G}^j(x) \in \mathrm{dom}(\mathrm{G}))) \wedge \neg \mathrm{Q}(\mathrm{G}^m(x)))) \quad (1)$$

where $\mathrm{G}^i(x)$ represents $\mathrm{G}(\mathrm{G}^{i-1}(x))$ for $i > 1$, and $\mathrm{G}(x)$ for $i = 1$.

S is the least set formed by the following rules:

1. $\mathbf{I} \subseteq \mathbf{S}$
2. $(\forall x, (x \in \mathbf{S} \wedge \mathrm{Q}(x)) \Rightarrow \mathrm{G}(x) \in \mathbf{S})$

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

1. $\mathbf{S}_0 = \mathbf{I}$
2. $\mathbf{S}_{n+1} = \mathbf{S}_n \cup \{\mathrm{G}(x) \mid x \in \mathbf{S}_n \wedge \mathrm{Q}(x)\}$

It can be proven that $\exists N, \mathbf{S}_{N+1} = \mathbf{S}_N$ and thus $\mathbf{S} = \mathbf{S}_N$ and **S** is finite.

An inductive definition for the predicate, P, is given 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:

$$\mathrm{P}(\mathbf{int}\ x) \doteq \begin{cases} \mathbf{I} = \{MIN\} \\ \mathrm{G}(x) = x + 1 \\ \mathrm{Q}(x) = x < MAX \end{cases}$$

2.4 Predicate Logic

We describe an LD-relation by giving the characteristic predicate of the relation, domain[1] and competence set. To ensure that the meaning of the specification is clear, we require that such expressions be total (i.e., they always have a clearly defined value, either *true* or *false*, regardless of the values of their arguments). In writing program specifications, however, it is often necessary to use functions which are not total. To overcome this problem we use the logic described in [26], which allows partial functions while ensuring that predicates are total.

This logic differs from traditional logic only in that primitive relations—those that are not defined using the logic—are *false* if one or more of their argument terms is a function application with argument values outside the function's domain. For example, if F and G are functions, ">" and "=" are primitive relations, and x is not in the domain of F then "$\mathrm{F}(x) > \mathrm{G}(x)$" and even "$\mathrm{F}(x) = \mathrm{F}(x)$" are *false*. Note that in many other logics the expression "$\mathrm{F}(x) = \mathrm{F}(x)$" would be a tautology by the "axiom of reflexivity". This definition of primitive relations is useful since it allows expressions using partial functions to be written as if the functions were total and to have the usual meaning in most cases, and to have a clearly defined meaning in all cases. A more thorough treatment of this logic is found in [26], [28].

The standard logical operators are used ($\wedge, \vee, \neg, \Rightarrow$) and they have their usual interpretation.

1. The domain information is redundant but we don't have a tool that derives the domain definition from the relation description. We require the domain information to determine termination requirements.

Quantification is permitted but, in order to ensure that oracles terminate, it must be restricted to a finite set. For our oracles, only the following forms are permitted, where $\mathrm{P}(x)$ must be an inductively defined predicate and $\mathrm{Q}(x)$ is any predicate expression:

Universal. $(\forall x, \mathrm{P}(x) \Rightarrow \mathrm{Q}(x))$

Existential. $(\exists x, \mathrm{P}(x) \wedge \mathrm{Q}(x))$

This restriction does not significantly limit the expressiveness of the logic for practical specifications since computer systems are always restricted to finite sets.

Complicated or frequently used expressions can be extracted from the program specification and used to define auxiliary predicates or functions, which can be applied in other expressions. The arguments to an application of an auxiliary predicate or function are treated as terms in the usual way and must be evaluated before the auxiliary predicate or function is evaluated. Auxiliary functions may be partial, in which case the characteristic predicate of the domain must be supplied.

2.5 Tabular Expressions [2]

We extend the notation for representing mathematical functions and relations to include the multidimensional *tabular expression* forms described in [16], [17], [1], [25]. These expressions are equivalent to expressions written in a more traditional manner, but many people have found them to be often easier to read and understand. Tabular expressions are particularly well suited to describing conditional relations of the forms that frequently occur in program specifications. In this paper, expressions written in a conventional (i.e., non-tabular) manner will be called *scalar expressions*.

There are several different types of tabular expressions, which are interpreted as either predicate expressions or terms. The TOG tool can handle any tabular expression described using the model presented in [1] as long as the expressions in the cells conform to the restrictions described in Section 2.4. The examples in this paper are of two forms, which [25] refers to as *mixed vector* and *normal predicate* tables. The interpretation of these types of tables is described by way of examples.

A tabular expression is constructed from scalar expressions and *grids*—indexed sets of cells that contain terms or predicate expressions, which may themselves be tabular. The table in Fig. 1 is an example of a two-dimensional mixed vector table. Cells in the column header contain predicates that are evaluated to determine which column is applicable—the one for which the column header is *true*. Cells in the row header contain a variable name followed by either "|" (read "such that") or "=". Rows that have "|" in the corresponding row header cell contain predicate expressions in their main grid cells, while those that have "=" contain terms.

A mixed vector table is interpreted by selecting the appropriate column (i.e., the one with a *true* header cell expression) and conjoining the predicate expressions formed

2. Readers familiar with [16], [17], [25] may want to skip this section.

	$x < 0$	$x = 0$	$x > 0$
$y \mid$	$y > 6$	*true*	$y = 0$
$z =$	$x - y$	10	x

$$((x < 0) \wedge (y > 6) \wedge (z = x - y)) \vee$$
$$((x = 0) \wedge (z = 10)) \vee$$
$$((x > 0) \wedge (y = 0) \wedge (z = x))$$

Fig. 1. Mixed vector table and equivalent scalar expression

by that column in the following way: If, for a cell C, the corresponding row header cell, H, contains "|" then the predicate expression is simply the predicate expression in C. If, on the other hand, H contains a string of the form "$x =$" (where "x" is any variable name) then the predicate expression is "$x = C$".

The table in Fig. 2 is an example of a two-dimensional normal predicate table. It is interpreted by selecting the row and column for which the respective header cell expressions are *true* and evaluating the predicate expression in that row and column of the main grid.

	$x > 0$	$x \leq 0$
$z > 10$	$y < 5$	$y > x + 2$
$z \leq 10$	$x + y < z$	$y > 5$

$$((x > 0) \wedge (z > 10) \wedge (y < 5)) \vee$$
$$((x > 0) \wedge (z \leq 10) \wedge (x + y < z)) \vee$$
$$((x \leq 0) \wedge (z > 10) \wedge (y > x + 2)) \vee$$
$$((x \leq 0) \wedge (z \leq 10) \wedge (y > 5))$$

Fig. 2. Normal predicate table and equivalent scalar expression

2.6 Applicability of the Method

These documentation techniques are useful for specifying any imperative program that is required to give output through program variables upon termination. 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 through peripheral devices, 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 have been investigated by other members of our research group.[4]

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* during execution then call procedure x" or "don't call x more than n times" (as for the Dutch National Flag example discussed in [24]), are also difficult to document using these methods. This is because relational specifications do not allow any restrictions on the intermediate states of an execution. One 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 procedure "x" was called); however, even with this solution another form of specification (e.g., an algebraic or trace specification) must be given for the data structure to state that the added data structure elements actually represent the intended information.

3 Oracle Generation

Fig. 3 gives an example of how the program documentation might be presented. It should be clear that a program that evaluates the characteristic predicate of the specification relation can be used as an oracle. Our prototype TOG tool, which is part of the Table Tool System (TTS) described in [34], generates an oracle in the form of C language procedures that may use some C++ objects. The initial version of the tool is described in detail in [30]; extensions to handle new types of tabular expressions are described in [1].

Our design allows the TOG to use optimization techniques to reduce the time required for oracle execution. Since it is likely that a program will be tested for many test cases, and the oracle will often be much slower than the PUT, minimizing oracle execution time can be important.

Note that the choice of C and C++ as oracle implementation language was based on the implementation language of the examples selected for illustration. If the intended application were different, the oracle design could be translated with no significant change.

3.1 Oracle Interface

The interface to the oracle consists of the four access programs described in Table 1.

TABLE 1
ORACLE ACCESS PROGRAMS

Program Name	Return Type	Description
`initOracle`	`void`	Initialization program. To be called before the first oracle program is called.
`inRelation`	`bool`	Evaluates the characteristic predicate of the relational component of the specification relation. Takes the value of the data structure in the initial state and final state (i.e., the TES) as arguments.
`inCompSet`	`bool`	Evaluates the characteristic predicate of the competence set of the program relation. Its argument is the value of the data structure in the initial state.
`inDomain`	`bool`	Evaluates the characteristic predicate of the domain of the program relation. Its argument is the value of the data structure in the initial state.

The programs `inCompSet` and `inDomain` can be used to avoid executing the PUT using test cases for which either there is no acceptable result (i.e., the test case is not in the domain) or the PUT may be nonterminating (i.e., the test case is not in the competence set). Note that for test cases that are in the domain but are not in the competence set, it is acceptable for the PUT to be nonterminating, but if it does terminate, the results can be checked using `inRelation`. The problem of deciding when an execution

Program Specification

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

external varibles:

D_{find} = *true*

C_{find} = *true*

R_{find} =

	$(\exists i, \text{bRange}(i) \wedge$ `'B[`i`]` $=$ `'x`$)$	$(\forall i, \text{bRange}(i) \Rightarrow \neg($`'B[`$i$`]` $=$ `'x`$))$
j' |	`'B[`j'`]` $=$ `'x`	*true*
`present'` $=$	TRUE	FALSE

$\wedge$ NC(`'B`, `'x`, `B'`, `x'`)

Auxiliary Predicate Definitions

NC(**int** 'a[], **int** 'b, **int** a'[], **int** b')
$\doteq (\forall i, \text{bRange}(i) \Rightarrow$ 'a[i] = a'[i]$) \wedge$ ('b = b')

Inductively Defined Predicates

bRange(**int** i)

$$\doteq \begin{cases} \mathbf{I} = \{0\} \\ \text{G(i)} = \text{i} + 1 \\ \text{Q(i)} = \text{i} < (N-1) \end{cases}$$

User Definitions

```
#include "defs.h"
#define N 10            /* size of array to search */
```

Fig. 3. Example Specification

is to be considered nonterminating has no good solution, and is beyond the scope of this work.

Ideally an oracle should indicate why a test execution fails, so that program (or specification) faults can be easily isolated. Unfortunately, since we use program specifications that may allow several correct stopping states for a particular test case, it is not always possible for an oracle to determine why a TES has failed.

The following C function prototypes give the syntax of the oracle access programs for the specification illustrated in Fig. 3. The convention of translating the variable names so that "'x" becomes "p_x" and "x'" becomes "x_p" is adopted to conform to C syntax rules.

```
void initOracle(void);
bool inRelation(int p_B[], int p_x,
                int B_p[], int x_p, int j_p,
                bool present_p);
bool inDomain(void);
bool inCompSet(void);
```

3.2 Internal Design

Each of the oracle access programs (with the exception of `initOracle`) evaluates a predicate expression. The TOG generates the appropriate code by traversing the syntax tree of the expressions in a depth-first order (i.e., innermost subexpressions are processed first) writing the code to evaluate each subexpression into a buffer, which is then used to construct the code for the "parent" expression. This process continues until the root expression has been translated. The procedure for implementing each type of subexpression is described in the following sections. It should be clear to the reader how the algorithms described in these code fragments could be easily translated into another imperative programming language.

3.2.1 Scalar Expressions

Scalar expressions can easily be represented using the looping constructs and logical operators included in most programming languages. This is illustrated below using the C operators.

Logical Operators. Except when they are the root node of a quantified expression, logical operators are directly translated to their equivalents as given in Table 2, in which P and Q represent arbitrary predicate expressions.

TABLE 2
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`

Primitive Relations. The standard programming language relational operators are combined with information about the domain of any partial functions. For example, the predicate expression "F(j') = x", where F is a partial function, is translated into the following code fragment.

```
(F_domain(j_p) && (F(j_p) == x))
```

Inductively Defined Predicates. An IDP is implemented as a C++ object class that provides access methods to enumerate the elements of the set it characterizes. This interface is convenient since IDPs are primarily used to characterize sets for quantification purposes. Our approach encapsulates (hides) the algorithm for determining the elements of the set and allows independent copies of the IDP to be created as needed during execution of the oracle code.

An array is used to represent the initial set component of the IDP definition, and two procedures implement the generating expression ("G") and continuation expression ("Q"). For example, the definition for bRange (see Fig. 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 denoted by the parentheses "()", and two named methods: `first` and `next`. The operator method "(e)" returns `TRUE` if `e` is in the set characterized by the IDP, and `FALSE` otherwise. The method `first` initializes the object's internal variables and returns the first element of the array representing **I**. The method `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), where e is the most recently returned element, is *false* and there are elements of **I** that have not been returned, then the next element of **I** is returned.
3. Otherwise, there are no further elements of the set so an element not in the set is returned; (e) returns `FALSE`.

When an IDP is used in an expression, the oracle code creates an object from an appropriate class (determined by the type of the argument). The array and procedures corresponding to the predicate definition are passed as arguments to the constructor function, which is responsible for initializing a new copy of an object. This is illustrated by the object `bRange` of type `IndPred_int`, which is used in the quantification example, below.

Quantification. Quantifier expressions are implemented using loops that invoke the methods to enumerate the elements of the set characterized by the IDP. The root node of the quantification expression (i.e., the "$\wedge$" for existential or "$\Rightarrow$" for universal) evaluates 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 inductively defined predicate). To ensure that evaluation is fast, the loops are designed to terminate as soon as the result of the quantification is known (i.e., first positive instance for existential quantification, first negative instance for universal quantification).

For example, the quantification $(\exists i, \text{bRange}(i) \wedge \text{`B}[i] = \text{`x})$, which is in the first cell of the column header of the table in Fig. 3, is translated to the code below.

```
static bool
table_H2_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);
}
```

3.2.2 Tabular Expressions

Two possible implementations for tabular expressions were considered:

1. translate the tabular expressions into equivalent scalar expressions, and then translate them into equivalent C statements as described above; or
2. use a set of procedures that evaluate a tabular expression using procedures for the cell expressions.

The second option was chosen for the following reasons.

- It allows the semantics of tabular expressions to be hidden in the tabular expression evaluation procedures, so both the TOG and the oracle are less complicated.
- Since the algorithm for interpreting a table doesn't change for different specifications, the code can be designed to reduce the number of cell expressions that need to be evaluated and hence improve performance.

To implement tabular expressions we have four classes of C++ table objects each of which implements one of the classes of tabular expressions described in [1] (normal, inverted, vector and decision). An object of one of these classes is instantiated to implement each nonscalar expression.

Procedures that implement the expressions contained in each cell of the table are generated and pointers to these are used by the table object.

Table objects have two methods that are used to evaluate the expression: `findCell` determines if the values of the arguments are in the domain of the expression represented by the table, and `value` evaluates the table.

3.2.3 Auxiliary Predicates and Functions

As mentioned in Section 2.4, auxiliary predicates and functions are expressions that are either complicated or used repeatedly. A procedure is created for each auxiliary predicate or function, with the expression implementation forming the body of the procedure. If a domain expression for an auxiliary function is given, a procedure is produced to implement that expression as well. For example, consider an auxiliary function defined as follows:

$$\textbf{int}\ \text{guarded\_B}(\textbf{int}\ \text{b}[],\ \textbf{int}\ \text{i}) \doteq \begin{cases} \text{b[i]} \\ \textbf{domain} = 0 \leq \text{i} < N \end{cases}$$

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 Trial Application

We have applied these techniques to the testing of three programs from a commercial product. We hope that this will help us to demonstrate the practicality and effectiveness of these methods, and to gain an appreciation of their strengths and weaknesses. This section describes the programs that were tested, the procedure we used and the results.

4.1 Program Description

The programs used in the trial application are part of a network management application produced by Newbridge Networks Corporation of Kanata, Ontario, Canada. Together the programs 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. Newbridge provided us with the code (about 350 lines) and an informal description of its intended behavior. We developed the formal specifications for three of the module's access programs:

- `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.

Figs. 4, 5, and 6 give the specification for `HashAdd`. The specifications of `HashFind` and `HashRemove` reuse many of the same auxiliary predicates and functions that are used for `HashAdd`, so the total documentation for the module is about five pages (cf. about seven pages for the code).

4.2 Test Procedure

A *test harness* calls the PUT for a set of test cases and checks the results using the oracle. It may also perform such tasks as collecting statistics on the number of failed tests, etc. Usually test harnesses will not need to be changed to accommodate modifications to the specification or PUT.

We used a single test harness together with the oracle programs generated for the three hash module access programs. The input to the test harness was 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 executed only 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.

TABLE 3
TEST SUITE DESCRIPTIONS

Suite	Number of commands			
	Add	Remove	Find	Total
A	3303	3363	3334	10000
B	4984	2516	2500	10000
C	498	256	246	1000

Test suites were generated randomly using the C language uniform distribution random number generator. The tests were approximately uniformly distributed between the two hash tables. Table 3 summarizes the test suites.

To verify that the testing procedures does, in fact, detect errors when they occur, some errors were introduced into the hash module. This was done by making small changes to the code so it no longer satisfied its specification. While we did not follow any formal procedure for selecting the changes to be made, we feel they do represent typical programming errors (see Table 4). Each modified version of the hash module (presumably containing only one code fault) was tested separately using the same test suites.

4.3 Test Results

Table 4 summarizes the results of the testing.

Since the hash module is part of a commercial software system, and had previously been carefully inspected and tested at Newbridge, it is not surprising that no errors were detected for the unmodified cases. As can be seen from tests No. 3 through No. 8, the testing procedures were successful in detecting all of the inserted errors. The large number of rejected test cases in tests No. 5 and No. 7 was caused by the fact that the code modifications introduced for these tests destroyed the integrity of the data structure, resulting in many tests cases that were not in the competence set of the specification relation.

4.3.1 Performance

To see how quickly our oracles perform, we measured the execution time for running 10,000 tests of the unmodified hash module (tests No. 1 and No. 2) on our DEC Alpha running OSF/1 V2.0. The results are summarized in Table 5. Our measurements are total elapsed time for the whole test suite, so they do not distinguish between execution of the test harness, the hash module programs or the oracle programs. These times are much less than the time it would take to manually verify the results of 10,000 test executions.

Program Specification

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

external varibles: `struct hashStruct *AHashArray[A_HASH_SIZE]`
`struct hashStruct *BHashArray[B_HASH_SIZE]`

$D_{HashAdd} = \underline{true}$

$C_{HashAdd} = (\text{`theTable} = \texttt{HASH\_A} \vee \text{`theTable} = \texttt{HASH\_B}) \wedge (\forall i, \text{ALists}(i) \Rightarrow \text{sorted}(\text{`AHashArray}[i])) \wedge (\forall i, \text{BLists}(i) \Rightarrow \text{sorted}(\text{`BHashArray}[i]))$

$R_{HashAdd} =$

$$\left(\begin{array}{l} (\text{`theTable} = \texttt{HASH\_A} \vee \text{`theTable} = \texttt{HASH\_B}) \wedge \\ (\forall i, \text{ALists}(i) \Rightarrow \text{sorted}(\text{`AHashArray}[i])) \wedge \\ (\forall i, \text{BLists}(i) \Rightarrow \text{sorted}(\text{`BHashArray}[i])) \end{array} \right) \Rightarrow$$

	`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)

Fig. 4. `HashAdd` specification, Part I

Inductively Defined Predicates

ALists(**unsigned int** i)
$$\doteq \left\{ \begin{array}{l} \mathbf{I} = \{0\} \\ \text{G}(i) = \text{i} + 1 \\ \text{Q}(i) = \text{i} < (\texttt{A\_HASH\_SIZE} - 1) \end{array} \right.$$

BLists(**unsigned int** i)
$$\doteq \left\{ \begin{array}{l} \mathbf{I} = \{0\} \\ \text{G}(i) = \text{i} + 1 \\ \text{Q}(i) = \text{i} < (\texttt{B\_HASH\_SIZE} - 1) \end{array} \right.$$

Auxiliary Predicate Definitions

Aequal(**struct HashStruct** * before[], **struct HashStruct** * after[])
$\doteq (\forall i, \text{ALists}(i) \Rightarrow \text{listsEqual}(\text{before}[i], \text{after}[i]))$

Bequal(**struct HashStruct** * before[], **struct HashStruct** * after[])
$\doteq (\forall i, \text{BLists}(i) \Rightarrow \text{listsEqual}(\text{before}[i], \text{after}[i]))$

inList(**struct HashStruct** * list, **unsigned int** Id)
$\doteq \neg(\text{list} = \texttt{NULL}) \wedge (\text{list}\texttt{->identifier} = \text{Id} \vee \text{inList}(\text{list}\texttt{->hashNext}, \text{Id}))$

insertedA(**struct HashStruct** * before[], **struct HashStruct** * after[], **unsigned int** Id, **struct hashStruct** * ptr)
$$\doteq \left(\forall i, \text{ALists}(i) \Rightarrow \left(\begin{array}{l} (\neg(\texttt{hash}(\text{Id}, \texttt{A\_HASH\_SIZE}) = i) \wedge \text{listsEqual}(\text{before}[i], \text{after}[i])) \vee \\ \text{insertedList}(\text{before}[i], \text{after}[i], \text{Id}, \text{ptr}) \end{array} \right) \right)$$

insertedB(**struct HashStruct** * before[], **struct HashStruct** * after[], **unsigned int** Id, **struct hashStruct** * ptr)
$$\doteq \left(\forall i, \text{BLists}(i) \Rightarrow \left(\begin{array}{l} (\neg(\texttt{hash}(\text{Id}, \texttt{B\_HASH\_SIZE}) = i) \wedge \text{listsEqual}(\text{before}[i], \text{after}[i])) \vee \\ \text{insertedList}(\text{before}[i], \text{after}[i], \text{Id}, \text{ptr}) \end{array} \right) \right)$$

insertedList(**struct HashStruct** * before[], **struct HashStruct** * after[], **unsigned int** Id, **struct hashStruct** * ptr)
$\doteq \neg(\text{after} = \texttt{NULL}) \wedge$

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)
$\doteq \text{inList}(\text{table}[\texttt{hash}(\text{Id}, \text{size})], \text{Id})$

Fig. 5. `HashAdd` specification, Part II

Auxiliary Predicate Definitions (continued)

listEqual(**struct HashStruct** * left, **struct HashStruct** * right)

$\doteq$

	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)

$\doteq$ (left = NULL ∧ right = NULL) ∨
(¬(left = NULL ∨ right = NULL) ∧ (left−>identifier = right−>identifier) ∧ (left−>data = right−>data))

sane(**struct HashStruct** * list)

$\doteq$ list = NULL ∨ (list = list−>sanityCheck ∧ ¬(list−>hashNext = NULL) ⇒
((list−>identifier < list−>hashNext−>identifier) ∧ sane(list−>hashNext)))

sorted(**struct HashStruct** * list)

$\doteq$ list = NULL ∨ (¬(list−>hashNext = NULL) ⇒
((list−>identifier < list−>hashNext−>identifier) ∧ sorted(list−>hashNext)))

User Definitions

```
#include "test.h"
#include "hash.h"

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

#define SUCCESS         1
#define FAIL            0
```

Fig. 6. `HashAdd` specification, Part III

TABLE 4
SUMMARY OF HASH MODULE TEST RESULTS

No.	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`	881	119	0
4	C	Neglect to append added item to existing list in `HashAdd`	511	489	0
5	C	Neglect to set "identifier" in `HashAdd`	3	6	991
6	C	Always return NULL from `HashRemove`	737	263	0
7	C	Neglect to rejoin list when element removed in `HashRemove`	29	14	957
8	C	Use wrong size to calculate hash index for table B	746	254	0

TABLE 5
ORACLE PERFORMANCE

No.	Test Suite	Elapsed Time (min:sec)	Time per command (msec)
1	A	2:30	15
2	B	5:30	34

The performance is different for the two different test suites because the hash module uses dynamic memory, so the size of the data structure changes depending on the number of elements stored in the tables, and hence there is more processing required for larger tables. Test suite B has about twice as many "add" commands as "remove", so the tables will grow during execution of the test suite (the commands are randomly ordered in the test suite), whereas with test suite A the tables will not increase in size. This affects the speed of all of the programs, but is most significant for the oracle and test harness components—the hash module programs are specifically designed to give good performance for large tables. In particular the test harness must copy the entire data structure before each command is executed so it will be much slower for large tables.

4.4 Difficulties

The process of using these methods to test commercial software highlighted some of the difficulties that might arise in a realistic software development situation. These difficulties are discussed in this section.

4.4.1 *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, we thought that a fault had been discovered, only to find that the fault was actually in our formal specification.

Careful inspection and the use of specification checking

tools, such as are being developed for the TTS [34], are obvious methods of removing faults from the specification; however, if an oracle is generated from the specification, then other fault detection methods are possible. 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). Documentation for software that is actively being used is important for maintenance, so finding an error in the specification can be almost as valuable as finding an error in the code.

4.4.2 Data Structure Accessibility

The oracle must be able to access the data structure to check its properties. In cases, such as the hash module, where the data structure is "hidden" in a module, this may require some modifications to the PUT. For the hash module we considered two solutions:

1. Add a program to the hash module to export the data structure so that the test harness could pass it to the oracle.
2. Modify the hash module programs to call part of the test harness, passing the data structure as arguments. The test harness could copy the data structure and call the oracle programs as necessary.

The first alternative was chosen since it involved no changes to the parts of the programs to be tested, whereas the other alternative would have required several changes. The programs that are tested should be as close as possible to those that will be used in the final system.

4.4.3 Complexity of Test Harness

The test harness must provide `inRelation` with the value of the data structure in both the starting and stopping states. To do this it must copy the data structure before executing the PUT. The complexity of such a test harness is dependent on the design of the data structure.

In the case of the hash module, the data structure consists of an array of sorted linked lists, which is a sufficiently complex data structure that the program to copy it is itself a potential source of errors. In fact, in preliminary testing, some errors were found in that portion of the test harness code.

4.4.4 Procedures that are not Programs

One of the procedures in the hash module, `HashOperateOnNext`, has an argument that is a pointer to a procedure that is to be called for some of the elements in the hash table. According to our definitions, `HashOperateOnNext` is not a program since its text does not determine a set of possible executions. (See [24] for a further discussion of the distinction between programs and procedures.) In order to specify the behavior of `HashOperateOnNext`, a specification of the program that is its actual argument is required so that its effect on the data structure can be determined. Also, as mentioned in Section 2.6, since the TES does not 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 actually was called the correct number of times. For these reasons `HashOperateOnNext` was not tested.

4.4.5 Location Sensitive Data Structures

A field in the hash module data structure, `sanityCheck`, is used as a fault detection mechanism by the hash module. 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 our test harness must copy the data structure to a new location in memory so that it can be used as part of the start state in the TES, which means that `sanityCheck` will no longer have the desired property. It is impossible to ascertain the correctness of this value from a copy. For this reason, the integrity of this field cannot be checked by `inRelation` for the "before" values of the data structure, so `inRelation` may report some false failures (i.e., TESs that pass are reported as failing). A solution to this problem would be to add to the start state description a variable representing the address of the data structure in the start state.

5 Discussion and Conclusions

5.1 Applications for This Work

In addition to the obvious application of this work to unit testing, it can be used in several other ways, as follows.

5.1.1 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 behavior of the resulting program is similar to those developed using the methods described in [18], [32]. For *in situ* testing, no test harness need be constructed. Of course, in such applications the performance of the oracle is a significant issue. The viability of such an application will depend on the amount of processing done by the oracle and the performance requirements of the system.

5.1.2 Self-checking ADTs

In [3], Antoy and Hamlet describe another application for executable oracles. They use algebraic specifications of ADTs and require that the user provide a "representation mapping" to map from the concrete data structure to the abstract specification. Their oracles are invoked by the ADT code to test that the axioms from the specification are valid following each change to the ADT value. 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. Oracles as generated by the TOG could be used to create a self-checking ADT similar to Antoy and Hamlet's by generating the oracle programs for each access program of the ADT

and embedding calls to the oracle programs in the ADT code.

5.1.3 Enforced Documentation Consistency

One factor that reduces the value of most 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 would only pass thorough testing if the documentation is correct.

5.2 Limitations of the Method

Clearly this work is limited to those programs that can be specified using relational techniques as discussed in Section 2.6. Some other limitations are as follows.

5.2.1 Oracle Termination

It is possible to write a specification for which the oracle will not terminate, or will only terminate after an unreasonable amount of time. Nontermination can be caused by a nonterminating recursion in an auxiliary definition, by errors in the definition of an inductively defined predicate or by a nonterminating "primitive" (i.e., defined in the programming language) function. Slow termination can be caused by quantification over large sets. For example, consider the well known "shortest path problem" for which a specification is given in [29]. 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. Nontermination can only be avoided by careful definition of auxiliary predicates and functions and by judicious use of well tested, or well verified primitive functions. For problems such as the shortest path problem, it is not practical to test the whole program against the specification for graphs with many paths. It may be practical in such situations, however, to test using small graphs or to test some of the subprograms used by the program and then to use other techniques to verify the top level code.

5.2.2 PUT Termination

As with any testing process, it is possible for the PUT to be a nonterminating program. 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 programs to detect that the PUT has not terminated, or has exceeded some reasonable time limit. Nontermination of the PUT must be detected by 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 might not terminate (i.e., those in the domain but not in the competence set) through `inDomain` and `inCompSet`, respectively.

5.2.3 Inaccessible Data Structure

An assumption in any testing method that checks properties of the data structure is that all elements of the data structure can be accessed by the oracle. In some cases this is not a valid assumption. For example, the data structure may include the computer display (i.e., the program is required to display some values). In these cases, some other form of oracle or additional test equipment (e.g., terminal simulator) are necessary.

5.3 Related Work

Several authors have described tools that can be used to compare the results of a test execution with some predefined "correct" data. In [20], Panzl describes three systems that verify the values of program variables against expected results described using a formal test language. Another system, described by Hamlet in [9], tests a program using a list of $\langle input, output \rangle$ 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, since they only compare expected and actual output, these methods are not appropriate when there are several acceptable answers.

The latter limitation is partially overcome by the "program testing assistant" described by Chapman in [6]. 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, requires that the user once had a version of the program that was considered correct.

ANNA [18] and APP [32], allow program code to be annotated with assertions which are evaluated as the code is executed. If these assertions constitute a specification of the program, which is the intention of ANNA but not APP, then they can be used as an oracle. However, since the annotations used in theses 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 nonprogrammer "domain experts".

A *model-based* specification describes the intended behavior of a program in terms of operations on an abstract machine model, often a finite state automaton. Such specifications often do not meet our requirement of being a minimal statement of requirements because they contain information describing the model behavior, which is not part of the actual requirements.

In [35], Stocks and Carrington discuss using model-based specifications to derive "oracle templates", which describe, using the Z notation, a set of acceptable outputs for a given set of test cases. In [31], 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.

In [14], [19], Hörcher and Mikk discuss the generation of oracles (in the sense of this paper) from model-based specifications written using the Z notation. Their oracles

evaluate predicates on "specification variables" whose values are derived from the concrete data structure by way of an abstraction function, whereas our oracles evaluate predicates on the concrete data structure value, so no such abstraction function is required.

The Assertion Definition Language (ADL) project [2], [33] includes a system for generating oracles and test harnesses from procedure specifications. ADL is quite similar to our work, but doesn't support tabular notations, which we consider to be a significant factor improving the readability of our specifications. ADL also cannot be used to specify the nontermination cases described by the competence set and domain of the relation in an LD-relation. In addition, ADL uses "bounded" quantification (i.e., quantification of the form $(\forall x : S, P(x))$ or $(\exists x : S, P(x))$, where S is a set) as a primitive of the language, which, as discussed in [26], can complicate the expressions. Finally, in its current revision [2] the ADL primitive for defining the "domain" of quantification only supports ranges of integers, whereas our IDPs permit quantification over a much broader class of sets.

5.4 Future Work

As described in Section 4, the TOG has been tested and evaluated using some small programs which have shown that the methods presented in this paper are viable in these cases. More experience with applying them to a wide variety of industrial software applications would allow more general conclusions about the viability and usefulness of the methods to be drawn and would undoubtedly lead to suggestions for improvements in the TOG and oracle designs.

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 stated in a specification that some program variables are not changed—denoted using the shorthand "NC" in [24]. This work does not support this shorthand, so the expanded form of the predicate must be included in the documentation (see Fig. 3). It would be convenient if this definition could be produced automatically from such a shorthand. This would be straight-forward for the basic data types of a programming language, but is much more difficult for constructed types.

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. Consider a specification for a program to solve a system of n linear equations in n unknowns—the specification, which states that the final values of the unknowns satisfy the equations, is functional, but an oracle that outputs the correct stopping state could not be generated automatically by this tool. The oracle to check if the given solution is correct, however, can be generated by our tool and is quite efficient. It is possible for some limited set of specifications to automatically generate oracles that output expected results but that would require significant modifications to this work.

TABLE 6
Advantages and Disadvantages of Oracle Generation

Disadvantages	Advantages
The documentation used to generate the oracle can be almost as complicated as the PUT and needs to be checked carefully.	The value and usefulness of program documentation is greatly increased since it can be tested for consistency with the program.
A suitable test harness may be a nontrivial program, which must also be checked carefully.	Faster test analysis, hence reduced cost.
Certain classes of program behavior cannot easily be specified and checked using these methods.	Reliable failure detection, hence increased value.

Finally, 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 programming languages to facilitate interfacing with PUTs written in these languages. This could be accomplished by providing several additional submodules, one for each programming language, and having the TOG tool select the appropriate one according to the user's request.

5.5 Conclusions

The development and application of the TOG prototype has shown that it is feasible to automatically generate executable test oracles from tabular relational program documentation. Application of these methods to industrial software has demonstrated the limitations and strengths outlined in Table 6.

Acknowledgments

Jeff Zucker and Oliver Slupecki have offered helpful and insightful comments, which have helped to improve the clarity of this work. The anonymous referees were exceptionally helpful in providing thoughtful and detailed comments on the first submission of this paper.

This research was supported in part by the Government of Ontario through the Telecommunications Research Institute of Ontario (TRIO), the Government of Canada through the Natural Sciences and Engineering Research Council (NSERC) and Bell Canada. Brian Smith, Jonathan Bosloy, and Ed Ensing, all of Newbridge Networks Corporation, assisted by providing realistic examples.

References

[1] R.F. Abraham, "Evaluating Generalized Tabular Expressions in Software Documentation", MEng thesis, Dept. of Electrical and Computer Engineering, McMaster Univ., Hamilton, On-

tario, also printed as CRL346, Telecommunications Research Inst. of Ontario, Feb. 1997.

[2] *ADL Language Reference Manual for ANSI C programmers, Release 1.1*, Sun Microsystems Inc., Document Reference MITI/0002/D/R1.1, Dec. 1996.

[3] S. Antoy and R. Hamlet, "Objects that Check Themselves against Formal Specifications", TR 91-1, Dept. of Computer Science, Portland State Univ. School of Engineering and Applied Sciences, Portland, Oregon, 1991.

[4] B.J. Bauer, "Documenting Complicated Programs", MEng thesis, Dept. of Electrical and Computer Engineering, McMaster Univ., Hamilton, Ontario, also printed as CRL316, Telecommunications Research Inst. of Ontario, Dec. 1995.

[5] G. Bernot, M.C. Gaudel and B. Marre, "Software Testing Based on Formal Specifications: A Theory and a Tool", *Software Eng. J.*, vol. 6, pp. 387–405, June 1990.

[6] D. Chapman, "A Program Testing Assistant", *Comm. ACM*, vol. 25, no. 9, pp. 625–634, Sept. 1982.

[7] S.R. Faulk, J. Brackett, P. Ward and J. Kirby, jr., "The CORE Method for Real-Time Requirements", *IEEE Software*, vol. 9, no. 6, pp. 22–33, Sept. 1992.

[8] J. Gannon, P. McMullin and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing", *ACM Trans. Programming Languages and Systems*, vol. 3, no. 3, pp. 211–223, July 1981.

[9] R. Hamlet, "Testing Programs with the Aid of a Compiler", *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 279–290, July 1977.

[10] E.C.R. Hehner, "Predicative Programming Part 1", *Comm. ACM*, vol. 27, no. 2, pp. 134–143, Feb. 1984.

[11] K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Applications", *IEEE Trans. Software Eng.*, vol. 6, no. 1, pp. 2–13, Jan. 1980.

[12] D.N. Hoover and Z. Chen, "Tablewise, a Decision Table Tool", *Proc. Ninth Conf. Computer Assurance, COMPASS '95*, June 1995.

[13] W.E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, 1987.

[14] H.M. Hörcher, "Improving Software Tests using Z Specifications", *ZUM '95: The Z Formal Specification Notation, Lecture Notes in Computer Science 967*, J.P. Bowen and M.G. Hinchey, eds., pp. 152–166, Springer-Verlag, 1995.

[15] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl and B.E. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems", *IEEE Trans. Software Eng.*, vol. 17, no. 3, pp. 241–258, Mar. 1991.

[16] R. Janicki, "Towards a Formal Semantics of Parnas Tables", *Proc. 17th Int'l Conf. Software Eng., ICSE*, pp. 231–240, Apr. 1995.

[17] R. Janicki, D.L. Parnas, J. Zucker, "Tabular Representations in Relational Documents", *Relational Methods in Computer Science—Advances in Computing Science*, C. Brink, W. Kahl, and G. Schmidt, eds., pp. 184–196, New York: Springer Wien, 1997.

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

[19] E. Mikk, "Compilation of Z Specifications into C for Automatic Test Result Evaluation", *ZUM '95: The Z Formal Specification Notation, Lecture Notes in Computer Science 967*, J.P. Bowen and M.G. Hinchey, eds., pp. 167–180, Springer-Verlag, 1995.

[20] D.J. Panzl, "Automatic Software Test Drivers", *Computer*, pp. 44–50, Apr. 1978.

[21] D.L. Parnas, "A Generalized Control Structure and Its Formal Definition", *Comm. ACM*, vol. 26, no. 8, pp. 572–581, Aug. 1983.

[22] D.L. Parnas and Y. Wang, "The Trace Assertion Method of Module Interface Specification", TR89-261, Queen's Univ., Telecommunications Research Inst. of Ontario, Oct. 1989.

[23] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems", *Science of Computer Programming*, Elsevier-Science, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[24] D.L. Parnas, J. Madey and M. Iglewski, "Precise Documentation of Well-Structured Programs", *IEEE Trans. Software Eng.*, vol. 20, no. 12, pp. 948–976, Dec. 1994.

[25] D.L. Parnas, "Tabular Representation of Relations", CRL260, Telecommunications Research Inst. of Ontario, Nov. 1992.

[26] D.L. Parnas, "Predicate Logic for Software Engineering", *IEEE Trans. Software Eng.*, vol. 19, no. 9, pp. 856–862, Sept. 1993.

[27] D.L. Parnas, "Mathematical Description and Specification of Software", *Proc. IFIP Congress*, vol. I, pp. 354–359, North Holland, Aug. 1994.

[28] D.L. Parnas, "A Logic for Describing, not Verifying, Software", *Erkenntnis*, vol. 43, no. 3, pp. 321–338, Kluwer, Nov. 1995.

[29] D.K. Peters, "Shortest Path Algorithm: Formal Program Documentation", CRL280, Telecommunications Research Inst. of Ontario, Feb. 1996, to appear.

[30] D.K. Peters, "Generating a Test Oracle from Program Documentation", MEng thesis, Dept. of Electrical and Computer Engineering, McMaster Univ., Hamilton, Ontario, also printed as CRL302, Telecommunications Research Inst. of Ontario, Apr. 1995.

[31] D.J. Richardson, S.L. Aha and T.O. O'Malley, "Specification-based Test Oracles for Reactive Systems", *Proc. Int'l Conf. Software Eng., ICSE*, pp. 105–118, May 1992.

[32] D.S. Rosenblum, "A Practical Approach to Programming With Assertions", *IEEE Trans. Software Eng.*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

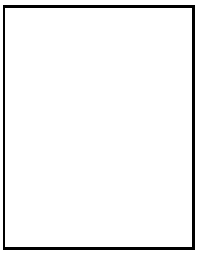
[33] S. Sankar and R. Hayes, "Specifying and Testing Software Components using ADL", SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, California, Apr. 1994.

[34] Software Engineering Research Group, "Table Tool System Developer's Guide", CRL339, Telecommunications Research Inst. of Ontario, Jan. 1997.

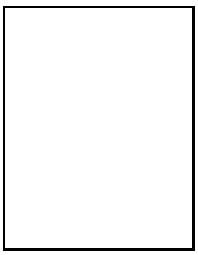
[35] P. Stocks and D. Carrington, "Test Template Framework: A Specification-Based Testing Case Study", *Proc. Int'l Symp. Software Testing and Analysis, ISSTA*, pp. 11–18, June 1993.

[36] Y. Wang, "Specifying and Simulating the Externally Observable Behavior of Modules", PhD thesis, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario, Canada, 1994.

[37] E.J. Weyuker, "On Testing Non-testable Programs", *The Computer J.*, vol. 25, no. 4, pp. 465–470, 1982.



Dennis Peters received his BEng (electrical) degree from Memorial University of Newfoundland in 1990 and the MEng (computer) degree from McMaster University, Hamilton, Ontario, in 1995. He is a PhD candidate in the Department of Electrical and Computer Engineering at McMaster University. He is currently researching techniques for documenting the requirements for computer systems and methods for using that documentation to automatically determine if a system satisfies the requirements. He also has a strong interest in software engineering education. He is a licensed Professional Engineer in the province of Ontario. He is a student member of the IEEE, a member of the IEEE Computer Society, ACM, and ACM SIGSOFT.



David Lorge Parnas received his PhD in electrical engineering from Carnegie Mellon University, and honorary doctorates from the ETH in Zurich, Switzerland, and the Catholic University of Louvain, Belgium. Dr. Parnas is a professor in the Faculty of Engineering Computing and Software Department at McMaster University, Ontario, where he is Director of the Software Engineering Program; he is also an associate member of the Department of Electrical and Computer Engineering. The author of more than 190 papers and reports, Dr. Parnas is interested in most aspects of computer system design. Dr. Parnas won an ACM Best Paper Award in 1979, and two Most Influential Paper awards from the International Conference on Software Engineering. He is the 1998 winner of ACM SIGSOFT's Outstanding Research award. He is licensed as a Professional Engineer in the province of Ontario. Dr. Parnas is a fellow of the Royal Society of Canada, a fellow of the ACM, a senior member of the IEEE, and a member of the IEEE Computer Society.