Generating a Test Oracle from Program Documentation

work in progress

Dennis Peters
peters@mcmaster.ca

David L. Parnas
Software Engineering Research Group
CRL, McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

A fundamental assumption of software 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 [17]. 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 paper discusses ongoing work to produce a tool that will generate a test oracle from formal program documentation.

In [9],[10] and [11] 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 the test oracle -- it simply must be evaluated for each test execution (input & output) to assign pass or fail. In contrast to [1], this paper discusses the testing of individual programs, not objects as used in [1]. Consequently, the method works with program documentation, written in terms of the concrete variables, and no representation function need be supplied. Documentation in this form, and the corresponding oracle, are illustrated by an example.

Finally, some of the implications of generating test oracles from relational specifications are discussed.

1.0 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 strict application of 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, and 2) complete and precise enough to allow thorough analysis, both manually and mechanically. The use of tabular expressions to represent relations [12] is one of the cornerstones of these techniques.

All software testing research and practice assumes that there is some mechanism, an *oracle*, for determining whether or not the output from a program is correct [17]. In many cases this oracle consists of a manual observation of the test input and output, which can be time consuming, tedious and error prone. If, however, a program is mathematically documented, it should be possible to derive an oracle from that documentation.

This paper describes on-going work aimed at developing an automated Test Oracle Generator (TOG) tool that, given a relational program specification [9] using tabular expressions [12], will produce a program that will act as an oracle. This oracle procedure will take as input a test execution (i.e. an input, output pair) from the program under test and will return ***true***[1] if the pair satisfies the relation described by the specification, or ***false*** if it does not. A brief introduction to the documentation techniques is given, followed by an illustration of how this documentation can be used to produce an oracle. Finally some of the implications of this technique are discussed.

1. ***true*** and ***false*** are used to represent predicate values, whereas the value of boolean valued program variables are represented by TRUE and FALSE.

1.1 RELATED WORK

Several authors have described tools which can be used to compare the results of a test execution with some pre-defined ‘correct’ data. In [8], 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 [6], 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 (as opposed to functional) specifications cannot be used. For example, the program specified in Table 2 is required to indicate an occurrence of the value of x in the array B, if one exists. If that value occurs in B in more than onc 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 to be invalid.

The latter limitation is partially overcome by the “program testing assistant” described by Chapman in [3]. 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 -- it requires that the user once had a version of the program considered correct.

Other systems, such as described by Luckham et al. in [7], allow program code to be annotated with assertions which are evaluated as the code is executed. While these systems are capable of checking that the value returned by a program has the desired properties (expressed in terms of the values of other program variables in that scope), they are limited in that they can only make assertions about variable values at a particular point in the program and thus cannot check that variables have been changed in the correct manner (i.e. new values expressed in terms of their previous values).

If a program is formally documented, then it should be possible to use the specification as an oracle, so the expected output need not be given by the user. In [16], Stocks and Carrington discuss deriving oracle specifications from model based specifications using the Z notation and in [15], 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) specified using algebraic specifications, e.g. [1],[2] & [4]. These specification techniques address a different problem from those used in this work in that they are required to document the intended properties of the 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.

2.0 PROGRAM DOCUMENTATION METHOD

The documentation which is the input to the TOG is internal design documentation for a single procedure, i.e. it describes the intended behaviour of a procedure in terms of its effect on the actual concrete data structure (variables). This is distinct from module interface documentation, which describes the externally observable behaviour of a group of programs, which together implement an abstract data type, without reference to the concrete data structure used in their implementation (see [10]). This section describes the relational program documentation method used in this work which is based on that described in [11] 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.

2.1 TERMINOLOGY

2.1.1 RELATIONAL SPECIFICATION

In [9], the use of Limited Domain Relations (LD-relations) to specify programs is described. 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, and C_L be the set of start states for which the program must 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 <x, y> is an element of R_L, and
- for all starting states, x, in C_L, P will always terminate.

Note that if $x \notin$ domain(R_L) then P cannot terminate such that P satisfies L.

If P satisfies L for all possible starting states then P can be said to be *correct* with respect to L.

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

In this paper we say that a program is specified by an LD-relation referred to as the *program relation*. If the competence set is not given it is assumed to be the domain of R_L.

2.1.2 PREDICATE LOGIC

In writing program specifications it is often necessary to use functions which are *partial*, i.e. their domain is a subset of the possible values of their arguments, but it is a desirable property that predicates always be *total*, i.e. they always have a clearly defined value (***true*** or ***false***) regardless of the values of their arguments. The predicate logic described in [13] has this property and so a subset of it is used in this work.

This logic differs from traditional logic in that a primitive relation is defined as being ***false*** 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 x is not in the domain of F then "F(x) > G(x)" is always ***false***, as is "F(x) = F(x)" ('>' and '=' are primitive relations). Note that in many other logics the latter expression is taken to be equivalent to ***true*** by the "axiom of reflexivity".

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

Quantification is permitted but must be restricted to a finite set, which is described by an inductively defined predicate (see below). The following forms are permitted, 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

We define an *inductively defined predicate*, P, on <type> 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> $\rightarrow$ <type>,

Q is a predicate on <type>, and

$\exists m, \forall x \in I, \neg Q(G^m(x))$. [EQ 1]

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

It can be proven that $\exists N$, $S_{N+1} = S_N$. (In fact, we can take N = m from EQ 1, above.) Thus $S = S_N$ and S is finite.

Thus we can give an inductive definition for the predicate, P(x), 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

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

2.1.4 TABULAR EXPRESSIONS

Mathematical functions and relations are represented using the multi-dimensional *tabular expression* notation described in [12]. These expressions 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 the kinds of conditional relations that frequently occur in program specifications.

There are several different types of tabular expressions described in [12] which are interpreted as either predicate expressions or terms. In this paper only one form of tabular expression, the *mixed vector* table, will be used and tables will be at most 2-dimensional. A tabular expression is constructed from conventional (scalar) expressions and *grids* -- indexed sets of cells that contain terms or predicate expressions, which may themselves be tabular. The interpretation of a mixed vector table is described by way of an example. A more thorough treatment is given in [12].

TABLE 1. Mixed Vector Table

		$x < 0$	$x = 0$	$x > 0$	H_2
	y |	$y > 6$	***true***	$y = 0$	
H_1	z =	$x - y$	10	x	G

Table 1 is an example of a 2-dimensional mixed vector table. Cells in the column header, H_2, contain predicates which are evaluated to determine which column is applicable. Cells in the row header, H_1, contain a variable name followed by either '|' (read 'such that') or '='. Rows that have

2. 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.

'|' 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 column for which the column header cell expression evaluates to ***true*** and conjoining the predicate expressions formed by that column in the following way: If, for a cell C, the corresponding row header cell, H_1<i>, contains a '|' then the predicate expression is simply the predicate expression in C. If, on the other hand, H_1<i> contains an string of the form "x =" (where 'x' is any variable name) then the predicate expression is formed by appending the contents of C to the contents of H_1<i>. Thus, Table 1 is a representation of the following predicate:

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

2.1.5 BEFORE AND AFTER VALUE

The following convention for denoting the value of program variables before and after a program is executed is used [11].

Let P be a program and $x_i, \ldots, x_k$ be the program variables used in P. Then

- "x_i' " (to be read "x_i after") denotes the value of the programming variable x_i after execution of P.
- " $'x_i$" (to be read "x_i before") denotes the value of the programming variable x_i before execution of P.

2.2 DOCUMENTATION COMPONENTS

For the purposes of this work, program documentation is said to consist of a program specification together with the definitions of auxiliary predicates, functions and non-primitive data types, constants and functions used in the program specification. Each of these components is described below.

2.2.1 PROGRAM SPECIFICATION

A *program specification*, as illustrated in Table 2, consists of three components: (1)The *program invocation* gives the name and type of the program and lists the name and type of its actual arguments. (2)The *external variable list* lists the name and type of all external variables used in the program relation expression. (3)The *program relation* defines the LD-relation that specifies the behaviour of the program. It may include an expression that gives the characteristic predicate of the competence set and must include an expression that gives the characteristic predicate of the relation.

2.2.2 AUXILIARY PREDICATES & FUNCTIONS

An *auxiliary predicate* is a named predicate expression. It has a name, a list of arguments, and a definition which is either an ordinary predicate expression, written in terms of the arguments, or a triple which defines an inductively defined predicate (see section 2.1.3). Its name, together with a list of actual arguments, can be used in any expression where a predicate expression is permitted. It is evaluated by substituting the actual argument values for their corresponding arguments in the definition and evaluating the resulting predicate expression.

In a similar manner, an *auxiliary function* is a named functional expression. It has a name, type, list of arguments and a definition which is an ordinary functional expression written in terms of its arguments.

2.2.3 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 specification 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.

3.0 EXAMPLE DOCUMENTATION

In this section the documentation is given for a small program, 'find', similar to that described in [11], which searches an integer array 'B' for a value given by 'x' and returns its index in 'j' and, using a boolean variable 'present', indicates if a match was found. The design of an oracle and the procedure for automatic generation are described in section 4.0 using this specification as a basis for examples.

3.1 PROGRAM SPECIFICATION

TABLE 2 - 'Find' specification

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

external variables:

$\mathbf{R_{find}}$(<'B[], 'x>, <B'[], x', j', present'>) =

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

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

3.2 AUXILIARY PREDICATES

bRange(int i)

$\stackrel{\mathrm{df}}{=}$ **inductiveDef**[{0}, i+1, i<(N-1)][3]

NC(int 'a[], int 'b, int a'[], int b')

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

3.3 User Definitions

```
#include “defs.h”

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

4.0 ORACLE DESIGN

Given program documentation as illustrated above, it should be clear that the characteristic predicate of the program relation can be used as an oracle -- if an input-output pair is in the program relation then the execution was successful. The challenge of this work then is to convert the documentation into a form that can be executed (evaluated) so that it can be used in an automated testing environment.

Since it is likely that a program will be tested for several input values using the same specification, it was decided that the TOG should convert the documentation into a form that could be compiled using a standard high-level language compiler. This has the advantages that the TOG output is in a form that can be understood by programmers and the resulting oracle executes relatively quickly. Since both the tabular documentation tools being developed at McMaster and the example programs selected for illustration of this tool are implemented using the C programming language, it was decided that the oracle should also be implemented in C or a derivative (e.g. C++).

4.1 INTERFACE

An oracle, in the context of this work, consists of a C source code file which contains four externally accessible programs, initOracle, inCompSet, inDomain, and inRelation. initOracle is used for initializing internal data structures etc. and is intended to be called only once for each execution of the oracle (an execution may involve evaluating any number of test executions). The latter three programs are boolean valued functions which evaluate the characteristic predicate of the competence set, domain and relation components, respectively, of the program relation. To simplify implementation, these all have the same arguments, which represent the ‘before’ and ‘after’ values of the arguments to the program under test. The ‘after’ values are not used by inDomain and inCompSet. Thus, for the example given above, the access program prototypes are[4]:

```
#define ARGS_PROTO int B[N], int x, \
            int p_B[N], int p_x, \
            int p_j, BOOL p_present

#define ARGS B, x, p_B, p_x, p_j, \
            p_present

void initOracle();
BOOL inCompSet(ARGS_PROTO);
BOOL inDomain(ARGS_PROTO);
BOOL inRelation(ARGS_PROTO);
```

The oracle procedures are constructed by traversing the syntax tree of the expressions in the program relation in a post-order manner (i.e. innermost sub-expressions are processed first) and constructing code to implement each sub-expression as described below.

4.2 SCALAR EXPRESSIONS

The scalar (i.e non-tabular) expressions used in this form of program documentation can easily be represented in C as described below. Each scalar expression is translated into equivalent C statements.

4.2.1 LOGICAL OPERATORS

Except when they are the root node of a quantified expression (see section 4.2.4), logical operators are directly translated to their equivalents as given in Table 3. (P and Q are arbitrary predicate expressions.)

TABLE 3. Logical Operator Conversions

Logical Operator	C Equivalent
¬P	`!P`
P ∨ Q	`P || Q`
P ∧ Q	`P && Q`
P ⇒ Q	`(!P) || Q`

Thus the top node of the program relation expression tree, which is the conjunction of the table expression with *NC*(B, x) is implemented in the following procedure:

```
BOOL
inRelation(ARGS_PROTO)
{
   return(find_tab.inRelation(ARGS)
```

3. This notation is used to denote an inductive definition.

4. The actual variable names used in the oracle are derived from those appearing in the specification. In this example 'B, 'x, B', x', j' and present' are represented by B, x, p_B, p_x, p_j and p_present, respectively.

```
              && nc_B_x(ARGS));
}
```

4.2.2 PRIMITIVE RELATIONS

Since the logic used in this work differs from traditional logics in the definition of primitive relations, the standard programming language relational operators must be combined with information about the domain of partial functions. For example the predicate expression "'B[j'] = x", which appears in the first row of the first column of the table, is translated into the following procedure. (Arrays are treated as partial functions.)

```
static BOOL
find_g1_1(ARGS_PROTO)
{
   return(B_domain(p_j)
         && (B[p_j] == x));
}
```

4.2.3 INDUCTIVELY DEFINED PREDICATES

A set of classes of C++ objects has been defined to implement inductively defined predicates. Each of these classes has methods `first()` and `next()`, for enumerating the elements in the set characterized by the predicate, and an operator, `()`, which evaluates the predicate for an element. The definition of an inductively defined predicate (an auxiliary predicate definition) is converted into an array and two procedures for use by an object of one of these classes. For example, the definition for *bRange*(int i) (see section 3.2) results in 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));
}
```

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

4.2.4 QUANTIFICATION

Quantifier expressions are implemented using loops which call the procedures to enumerate the elements of the set which has the inductively defined predicate (see section 2.1.3) as its characteristic predicate. The root node of the quantification expression (i.e. the '$\wedge$' for existential or '$\Rightarrow$' for universal) is not implemented as described in section 4.2.1 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 inductively defined predicate). 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. first positive instance for existential quantification, first negative instance for universal quantification). Unfortunately, however, 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 is implemented as follows.

```
static BOOL
find_h2_1(ARGS_PROTO)
{
   BOOL result = TRUE;
   IndPred_int bRange(bRange_I, 1,
                      bRange_G,
                      bRange_Q);
   int i;

   i = bRange.first();
   while (result && bRange(i)) {
      result = !(B_domain(i)
                && (B[i] == x));
      i = bRange.next();
   }
   return(!result);
}
```

4.3 TABULAR EXPRESSIONS

Several classes of (C++) table objects which implement the various types of tabular expressions have been defined. An object of one of these classes is instantiated to implement each non-scalar expression. Although, as illustrated in section 2.1.4, tabular expressions could be translated into equivalent scalar expressions which could then be translated into equivalent C statements as for scalar expressions, it is felt that there are advantages to the table object approach. Firstly, the table objects encapsulate all knowledge of the semantics of tabular expressions so the TOG need not have this knowledge and is hence less complicated. Also, since the algorithm for interpreting a table is built in to the table object class (i.e. is not automatically generated), that code can be designed to minimize the number of cell expressions that need to be evaluated and hence improve performance.

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

Predicate expression table objects have two access programs (methods in C++ terminology) which may be used to evaluate the table: `findCell` determines if the current values of the arguments are in the domain of the table, and `inRelation` evaluates the table.

4.4 AUXILIARY PREDICATES & FUNCITONS

As mentioned in section 2.2.2, auxiliary predicates and functions are simply a shorthand notation for writing more complicated expressions. With the exception of inductively defined predicates, which are implemented as described in section 4.2.3, a procedure to implement each auxiliary predicate or function is generated, and appropriate calls to these are used in the code which implements other expressions.

4.5 USER DEFINITIONS

The user definition text is inserted at the beginning of the oracle code so that these basic definitions and symbols can be used by the oracle.

5.0 DISCUSSION

The documentation methods described in section 2.0 have been found to be useful for describing software behaviour. The mathematics (logic) used is neither new nor difficult so only a small amount of training should be required for specifiers, designers, programmers and users to become comfortable with it. The use of a single set of documentation by all people involved in the production and verification of a software system greatly reduces the risk of miscommunication or errors in translation. The addition of tools which allow the software to be tested directly against this documentation will enhance confidence in the implementation and speed up, and hence encourage, testing.

The requirement that the documentation be written so that it can be used to derive an effective test oracle does, however, impose some restrictions. For example, the use of primitive relational operators, most notably ‘=’, is only valid for the basic data types for which these operators are defined. A specifier might be tempted to write an expression such as “x' = *myFunc*('x)”, where x is a variable of non-primitive (i.e. abstract) type and *myFunc*() is an auxiliary function of that type, but it is impossible to evaluate this expression without the definition of ‘=’ for that type. The solution is to define an auxiliary predicate, say *absTypeEqual*(), that evaluates equality for the abstract data type, so the above expression could be written “*absTypeEqual*(x', *myFunc*('x))”.

Also, the restriction of quantification to finite (presumably small) sets may be seen as complicating the documentation. As is pointed out in [13] “…practitioners do not want to use methods that require them to use many symbols to say simple things.” An example which is used to illustrate the benefits of the logic is the expression “($\exists$i, B[i] = x)”. To evaluate this expression, in the worst case, an oracle must evaluate “B[i] = x” for <u>every</u> integer i. Although we can assume that the set of integers is finite, this evaluation would certainly take longer than is practical. So, in the notation adopted in this work, the expression must be written as “($\exists$i, *bRange*(i) $\wedge$ B[i] = x)” which is slightly more complicated, but can be evaluated much more quickly (assuming that bRange characterizes a set much smaller than the set of integer representations on the computer). This, unfortunately, also introduces an extra step in the process of verification of a specification -- the (human or computer) verifier must check that the inductively defined predicate does not exclude any values of interest, which may be a non-trivial task where complicated expressions are involved.

It is quite possible to write a specification for which the oracle will not terminate or will only terminate after an unreasonable amount of time. Non-termination could be caused by either a non-terminating recursion in an auxiliary definition, errors in the definitions of an inductively defined predicate or a non-terminating ‘primitive’ (i.e. defined in the programming language) function. Slow termination could 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 [12]. An oracle based on this specification must enumerate 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 must rest 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.

6.0 CONCLUSIONS

The problems addressed by this work are seen as being of very practical importance. The use of formal documentation techniques has been shown to improve the quality of software (e.g. [5]) but the industrial community has been slow to accept them because they are seen as greatly increasing the up-front cost of system development without significant measurable benefit. The ability to test a program using its documentation as an oracle will greatly increase the value of such formal documentation by reducing the cost of testing and helping to ensure that errors that occur during testing are detected. The test oracle generator can also be used to ensure that documentation is kept up to date: if a program is always tested against the documentation then everyone is assured that the documentation is consistent with the program behaviour.

We believe that the generation of a useful test oracle from relational program documentation is both feasible and practical. Development of a Test Oracle Generator tool which uses the methods described in this paper is currently under way. Tools for allowing easy editing and manipulation of documentation using these techniques are also being developed.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the assistance of Jeff Zucker of McMaster University in the preparation of this paper, particularly in clarifying the sections on inductively defined predicates. Several helpful comments have also been received from the anonymous reviewers as well as from Arne Maus of the University of Oslo.

This research was supported in part by the Government of Ontario through the Telecommunications Research Institute of Ontario (TRIO) and the Government of Canada through the Natural Sciences and Engineering Research Council (NSERC).

REFERENCES

1. Antoy, S. & Hamlet, R., “Objects that Check Themselves against Formal Specifications”, *TR 91-1*, Dept. of Computer Science, Portland State University School of Engineering and Applied Sciences, Portland, OR. 18 pgs.
2. 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.
3. Chapman, D., “A Program Testing Assistant”, *Communications of the ACM*, Vol. 25, No. 9 (September 1982), pp. 625-634.
4. 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.
5. Gerhart, S., Craigen, D. & Ralston, T., “Experience with Formal Methods in Critical Systems”, *IEEE Software*, (January, 1994) pp. 21-28.
6. 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.
7. 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.
8. Panzl, D. J., “Automatic Software Test Drivers”, *Computer*, April 1978, pp. 44-50.
9. Parnas, D. L., “A Generalized Control Structure and Its Formal Definition”, *Communications of the ACM*, Vol. 26, No. 8 (August 1983), pp. 572-581.
10. 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.
11. Parnas, D. L., Madey, J. & Iglewski, M., “Formal Documentation of Well-Structured Programs”, *CRL Report No. 259*, Telecommunications Research Institute of Ontario (TRIO), September 1992, 37 pgs.
12. Parnas, D. L., “Tabular Representation of Relations”, *CRL Report No. 260*, Telecommunications Research Institute of Ontario (TRIO), November 1992, 17 pgs.
13. Parnas, D. L., “Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 9 (September 1993), pp. 856-862.
14. Peters, D. K., “Shortest Path Algorithm - Formal Program Documentation”, *CRL Report No.280*, Telecommunications Research Institute of Ontario (TRIO), February 1994, 11 pgs.
15. 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.
16. 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.
17. Weyuker, E. J., “On Testing Non-testable Programs”, *The Computer Journal*, Vol. 25, No. 4 (1982), pp. 465-470.