Model-Based Formal Specification Directed Testing of Abstract Data Types

Xiaoping Jia
Institute for Software Engineering
Department of Computer Science and Information Systems
DePaul University
Chicago, IL 60604
E-mail: jiax@jia.cs.depaul.edu

Abstract

A model-based formal specification directed software testing approach is presented. It provides a test oracle and introduces a new coverage criterion for the functional (black-box) testing based on formal specifications. Given the source code of an implementation, its formal specification, and the retrieve functions, a test driver can be generated to conduct, evaluate, and measure the functional test. Although the complete specification conformance of an implementation can not be established by testing alone, our approach provides a strong necessary condition of complete specification conformance.

1 Introduction

This paper discusses the application of model-based formal specifications in software testing. A formal specification-directed testing approach is presented. The proposed testing approach provides a test oracle for checking whether the input-output pairs of test cases are acceptable, and introduces a new coverage criterion for the functional (black-box) testing based on formal specifications. This paper focuses the testing of *abstract data types* (ADT's). However, the proposed approach can be generalized and applied to the testing of other types of software systems. Without loss of generality, the Z notation [1] is adopted as the specification language, and C++ is adopted as the implementation language.

The model-based formal specifications [2, 3] model the functionality of software systems using well-defined mathematical objects such as sets and functions. Figure 1 shows a formal specification of an ADT *DataDictionary* using the Z notation, in which

$[NAME, INFO]$

DataDictionary

$Dict : NAME \pfun INFO$

Initialize

$\Delta DataDictionary$

$\mathrm{dom}\ Dict' =$

Find

$\Xi DataDictionary$
$name? : NAME$
$info! : INFO$

$info! = Dict(name?)$

Insert

$\Delta DataDictionary$
$name? : NAME$
$info? : INFO$

$name? \notin \mathrm{dom}\ Dict \Rightarrow$
$\quad Dict' = Dict \cup (name? \mapsto info?)$
$name? \in \mathrm{dom}\ Dict \Rightarrow$
$\quad Dict' = Dict \oplus (name? \mapsto info?)$

Delete

$\Delta DataDictionary$
$name? : NAME$

$Dict' = \{name?\} \rhd Dict$

Figure 1: The Z specification of *DataDictionary*.

```
class _Entry {
public:
  String _name;
  String _info;
};

class _DataDictionary {
private:
  _Entry _Dict[MAXENTRIES];
  int    _NumOfEntries;
public:
  _DataDictionary(void);
  void _Find(const String& _name, String& _info);
  void _Insert(const String& _name,
               const String& _info);
  void _Delete(const String& _name);
};
```

Figure 2: An array implementation of the *DataDictionary*: class definition.

the *DataDictionary* is modeled as a partial function from *NAME* to *INFO*. The mathematical object *partial function* in the specification is well-defined and easy to manipulate, however, it is not suitable for efficient implementation of *DataDictionary*. To efficiently implement *DataDictionary*, the partial function in the specification must be implemented using concrete data structures that are supported by the implementation language such as arrays, linked lists, or binary trees, etc. The class definition of an array implementation of *DataDictionary* in C++ is shown in Figure 2. The operation implementation will be shown later in Figure 5.[1]

The goal of software testing is to discover whether an implementation conforms to its specification. A formal definition of the conformance of an implementation to its specification is established in section 2. The formal specification directed testing approach is presented in section 3. The formal specification coverage criterion and heuristics are discussed in section 4 along with a case study. Although the complete conformance of an implementation to its specification cannot be established by testing alone, the proposed testing approach can provide a strong necessary condition of complete specification conformance.

2 Specification Conformance

The Z specification of an ADT consists of two types of schemas: the *data schemas* that specify the composition of data items, such as schema *DataDictionary*, and the *operation schemas* that specify the operations of the ADT, such as schemas *Initialize* and *Insert*. The data schemas define the *abstract data domain* of the ADT, which is composed of mathematical objects such as sets and functions. To implement the ADT, the abstract data domain must be represented by a *concrete data domain*, which is composed of data structures supported by the implementation language such as arrays and lists.[2] We use $\mathbf{abs}[A]$ and $\mathbf{con}[A]$ to denote the abstract and concrete data domains of ADT A, respectively. Elements in the abstract and concrete data domains are known as *abstract* and *concrete states*, respectively. The predicates of the data schemas specify the *invariant property* of A. We use $inv(s)$, where $s \in \mathbf{abs}[A]$, to denote that state s satisfies the invariant property. In general, the correspondence between the entities in the abstract and concrete data domains can be represented by a binary relation between $\mathbf{abs}[A]$ and $\mathbf{con}[A]$, called the *data retrieve relation*. In a true implementation of A, each concrete state should represent a unique abstract state. Thus, the data retrieve relation is a total function from $\mathbf{con}[A]$ to $\mathbf{abs}[A]$, called the *data retrieve function*, and denoted $retr : \mathbf{con}[A] \rightarrow \mathbf{abs}[A]$.

Example

In the specification of *DataDictionary*, its abstract data domain is specified as a partial function, *Dict*, from *NAME* to *INFO*. In the implementation of *DataDictionary*, its concrete data domain is specified as an array, `_Dict`, of pointers to a structure `_Entry`. The data retrieve function of *DataDictionary* can be expressed as the following equation:

$$Dict = \{ i : 1 \,..\, \texttt{\_NumOfEntries} \bullet \texttt{\_Dict}[i].\texttt{\_name} \mapsto \texttt{\_Dict}[i].\texttt{\_info} \} \quad (1)$$

Note that the left-hand side of the equation is an abstract entity, and the right-hand side is an expression involving concrete entities only. The data retrieve functions indicate how abstract states can be constructed from concrete states. The data retrieve function (1) indicates that each element of array `_Dict`, which is a structure `_Entry` with fields `_name` and `_info`, corresponds to a maplet belongs to the abstract function *Dict*. ■

The signature of an operation schema *op* specifies the abstract input and output domains of the operation, denoted $\mathbf{absI}[op]$ and $\mathbf{absO}[op]$ respectively. The predicates of the operation schema specifies the input-output relation:

$$\mathbf{absI}[op] \times \mathbf{abs}[A] \leftrightarrow \mathbf{absO}[op] \times \mathbf{abs}[A].$$

[1] We adopt the following naming convention: for a specification entity named *Name*, its counterpart in the implementation will be named as `_Name`.

[2] We shall use the term *abstract* to modify the entities associated with the specification and use the term *concrete* to modify the entities associated with the implementation.

Note that an operation may have side-effects on the state. We use $op(i, s, o, s')$, where $i \in \mathbf{absI}[op]$, $o \in \mathbf{absO}[op]$, and $s, s' \in \mathbf{abs}[A]$, to denote that with an initial state s, and input i, after invoking operation op, the resulting state s', and output o is an acceptable outcome. States s and s' are called the *pre-state* and *post-state*, respectively.

Let $\mathbf{conI}[op]$ and $\mathbf{conO}[op]$ denote the concrete input and output domains of operation op, respectively. Similar to the data retrieve relation, the relations between the entities in the abstract and concrete input/output domains are total functions called the *input* and *output retrieve functions*, and denoted *retrI* and *retrO*, respectively. Suppose that $\overline{op}$ is an implementation of operation op. An invocation of $\overline{op}$ is in the following form:

$(\overline{o}, \overline{s'}) = \overline{op}(\overline{i}, \overline{s})$

where $\overline{i} \in \mathbf{conI}[op]$, $\overline{o} \in \mathbf{conO}[op]$, and $\overline{s}, \overline{s'} \in \mathbf{con}[A]$. $\overline{s}$ and $\overline{s'}$ denote the concrete states before and after the invocation, respectively. Now we can formally define the conformance of an implementation of an operation to its specification.

Definition 1 $\overline{op}$ *conforms* to the specification of op, if one of the following conditions is satisfied:

a) If op is the initialization operation,
$(\varepsilon, \overline{s'}) = \overline{op}(\varepsilon, \perp) \Rightarrow inv(retr(\overline{s'}))$
where ε denotes void input/output, and $\perp$ denotes an undefined state.

b) Otherwise,
$inv(retr(\overline{s})) \wedge (\overline{o}, \overline{s'}) = \overline{op}(\overline{i}, \overline{s}) \Rightarrow$
$op(retrI(\overline{i}), retr(\overline{s}), retrO(\overline{o}), retr(\overline{s'}))$
$\wedge\ inv(retr(\overline{s'}))$.

3 Specification Directed Testing

To test an ADT A using the proposed formal specification directed testing approach, we assume that the following are given:

a) a model-based formal specification of A;

b) the source code of an implementation of A;

c) the retrieve functions between the implementation and the specification.

The formal specification directed testing process is outlined in Figure 3. It consists of the following stages:

1. **Specification compilation**
   The *specification compiler* reads the formal specification of A and the retrieve functions, and generates a *test driver*.

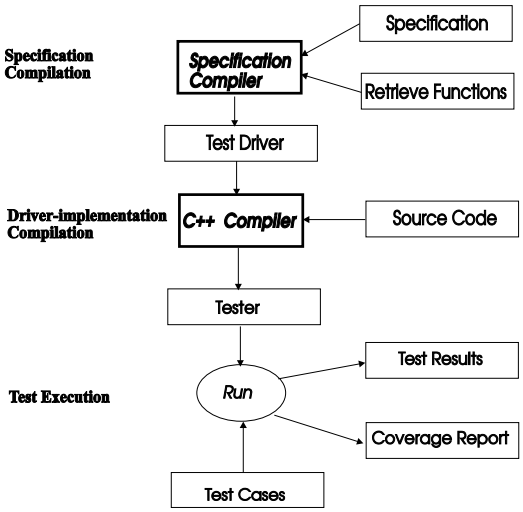

Figure 3: Formal Specification Directed Testing

2. **Driver-implementation compilation**
   The source code of the implementation and the generated test driver are compiled together to generate the executable *tester*.

3. **Test execution**
   The tester reads a sequence of test cases, executes the test cases and generates reports on:

   a) the execution results of the test cases;

   b) the specification coverage of the test cases.

3.1 The specification language

The specification compiler accepts the following two forms of Z specifications:

a) the LaTeX input with the `zed` style option defined by Mike Spivey [4].

b) a variation of the Z notation, called ZED/SL [5], that uses only the ASCII symbols.

Furthermore, in order for the test driver to check the validity of the input-output pairs of test cases by evaluating the truth values of the predicates in specifications, it is necessary to disallow infinite sets and quantification of variables bound by infinite sets. Infinite sets may only occur as the type specifiers.

3.2 The retrieve language

The retrieve language, called ZED/RL, is used to define the retrieve functions. A *retrieve scheme*, which

consists of all the retrieve functions, is in the following form:

`retrieve` ⟨*ADTName*⟩
  ⟨*RetrieveFunction*⟩ ...
  ⟨*RetrieveFunction*⟩
`end retrieve` ⟨*ADTName*⟩ .

Retrieve functions are defined for data types, data schemas, and operation schemas in the specification. A retrieve function is in one of the following forms:

a) data type:
   `type` ⟨*TypeName*⟩ `as` ⟨*DataType*⟩

b) data schema:
   `schema` ⟨*SchemaName*⟩ `as`
      ⟨*DataType*⟩
   `where`
      ⟨*RetrieveExp*⟩
   `end` ⟨*SchemaName*⟩ ;

c) operation schema:
   `schema` ⟨*SchemaName*⟩ `as`
      ⟨*FunctionPrototype*⟩
   `where`
      ⟨*RetrieveExp*⟩
   `end` ⟨*SchemaName*⟩ ;

TypeName and *SchemaName* are names in the specification. The *DataType* and *FunctionPrototype* in the retrieve functions are defined using the syntax of the implementation language, i.e., C++ syntax. A *RetrieveExp* is in the following form:

⟨*AbsName*⟩ : ⟨*RetrieveConstructor*⟩

where *AbsName* is a name in the specification, and *RetrieveConstructor* is an expression involving the names in the implementation only, and mathematical operations defined by Z such as set and function operations can be used. For example, assume that `u` and `v` are variables in the implementation, the following retrieve constructor denotes a maplet:

```
u -> v
```

where `->` is the ZED/SL symbol for $\mapsto$ in Z. Sets can be constructed using the following syntax:

```
{ i: 1..4 @ i -> i*i }
```

It constructs the following mapping:

$\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, 4 \mapsto 16\}$.

The retrieve function (1) of *DataDictionary* is expressed in ZED/RL as follows:

```
Dict: { i: 1.._NumOfEntries @
       _Dict[i]._name -> _Dict[i]._info }
```
(2)

The retrieve scheme of *DataDictionary* are shown in Figure 4.

```
retrieve DataDictionary

type NAME as String;
type INFO as String;

schema DataDictionary as
   class _DataDictionary;
where
   Dict: { i: 1 .. _NumOfEntries @
            _Dict[i]._name -> _Dict[i]._info };
end DataDictionary;

schema Initialize as
   _DataDictionary::_DataDictionary( void );
end Initialize;

schema Find as
   void _DataDictionary::_Find(
            const String& _name, String& _info);
where
   name?: _name;
   info!: _info;
end Find;

schema Insert as
   void _DataDictionary::_Insert(
            const String& _name, const String& _info);
where
   name?: _name;
   info?: _info;
end Insert;

schema Delete as
   void _DataDictionary::_Delete(const String& _name);
where
   name?: _name;
end Delete;

end retrieve DataDictionary.
```

Figure 4: The retrieve scheme of *DataDictionary*.

3.3 The Specification Compiler

The specification compiler reads the specification and retrieve functions, and generates the source code for the *schema evaluators*, the *abstractors*; and the *test driver*.

3.3.1 The schema evaluators

For each schema in the specification, a schema evaluator will be generated to evaluate the truth value of the predicates of the schema. Since the predicates of the schemas involves abstract entities such as sets and functions, a C++ class library, called ZED/LIB, is developed to handle the construction, manipulation, and evaluation of expressions involving bags, sets, tuples, lists, mappings, functions, and predicates. Since infinite sets and quantification of variables bound by infinite sets are not allowed, the predicates of the schemas

can be translated to terminating boolean functions, called *schema evaluators*. For example, the predicates of schema *Insert* can be translated to the following boolean function:

```
Bool Evaluate_Insert(Function Dict0, Function Dict1,
                     NAME name, INFO info )
{  if ( Member(name, Domain(Dict0)) ) {
      return Equal(Dict1, Union(Dict0,
                           Maplet(name, info)));
   } else {
      return Equal(Dict1, Override(Dict0,
                           Maplet(name, info)));
   }
}
```

`Function` is a class defined in ZED/LIB. `NAME` and `INFO` are types introduced by *DataDictionary*. `Member`, `Domain`, `Equal`, `Union`, `Maplet`, and `Override` are all functions defined in ZED/LIB with their usual meanings.

3.3.2 The abstractors

The *abstractors* are generated from the retrieve functions to construct abstract entities from the concrete entities using ZED/LIB. For example the abstractor of *Dict* can be generated from retrieve function (2):

```
Dict = EmptySet;
for (int i=1; i<=_NumOfEntries; i++) {
  InsertSet(Dict, Maplet(_Dict[i]._name, _Dict[i]._info));
}
```

`EmptySet` and `InsertSet` are defined in ZED/LIB.

3.3.3 The test driver

The test driver is a test oracle. It repeatedly performs the following tasks:

a) read a test case, which consists of the name of the operation to be invoked and the input arguments;

b) invoke the operation, and save the concrete states before and after the invocation, as well as the input and output values;

c) construct the corresponding abstract states and the abstract input and output values using the abstractors;

d) invoke the appropriate schema evaluators on the abstract states and input-output values to check the validity of the operation invocation in step b), as well as the invariant properties.

4 Specification Coverage

When the formal specification of an ADT is provided, a new type of testing coverage criterion, called the *specification coverage*, can be used to complement various code coverage criteria.

4.1 The Criterion and Heuristics

Let S be a schema in the specification of ADT A, we use $\text{pred}\, S$ to denote the predicates of schema S, and $\text{sk}(S)$ to denote the skolemization of $\text{pred}\, S$.

$\text{sk}(S) \equiv c_1 \lor c_2 \lor \cdots \lor c_k$,

where c_i are conjunctions of atomic predicates.

Example

The skolemization of the schema *Insert* in *DataDictionary*, is the following:

$name? \notin \text{dom}\, Dict \land$

$\quad Dict' = Dict \cup (name? \mapsto info?) \lor$

$name? \in \text{dom}\, Dict \land$

$\quad Dict' = Dict \oplus (name? \mapsto info?)$ ■

Assume that the specification of ADT A consists of data schema D and operation schemas $OP_i, 1 \le i \le n$, and all schemas are skolemized: $\text{sk}(D) \equiv d_1 \lor d_2 \lor \cdots \lor d_{k_0}$ $\text{sk}(OP_i) \equiv c_{i,1} \lor c_{i,2} \lor \cdots \lor c_{i,k_i}$ where each $d_l, c_{i,j}$ is a conjunction of atomic predicates, k_0 is the number of conjunctions in $\text{sk}(D)$, and k_i is the number of conjunctions in $\text{sk}(OP_i)$.

We use $OP_i^{(j)}$ to denote that operation OP_i is invoked and makes $c_{i,j}$ true after the invocation. We use $D^{(l)}$ to denote that d_l is true for the current state. An *invocation sequence* is in the following form:

$$OP_{i_1}^{(j_1)}; D^{(l_1)}; \cdots; OP_{i_m}^{(j_m)}; D^{(l_m)}; \cdots$$

The specification coverage criterion can be simply stated as follows:

> *All possible invocation sequences must be tested.*

Since there is usually no limit on the number of operations to be invoked, there are infinite number of different invocation sequences. Thus complete specification coverage is infeasible. In order to conduct a reasonable test, heuristics must be used to limit the number of tests. We propose the following heuristics:

H1 For each operation schema OP_i, each $c_{i,j}$ must become true at least once. And for the data schema D, each d_l must also become true at least once.

H2 For a container ADT, i.e., an ADT manipulates a collection of elements, every element must participate in every operation of the ADT.

The heuristics are to be used as guidelines to derive test cases. The test driver can be augmented to track the specification coverage using the above heuristics:

- After each invocation of an operation, mark the condition of the operation schema that becomes true, and mark the condition of the data schema that becomes true.
- For a container ADT, every time an operation is invoked, mark all the elements that participated in the operation.

At the end of a test run, a report on the extent of the coverage according to the heuristics can be generated.

4.2 A Case Study

Figures 2 and 5 show the class definition and the operation implementation of an array implementation of *DataDictionary* in C++ . Note that there are a few errors in the implementation. The first error is an off-by-one error in `_Find` that the `"<"` sign should have been a `"<="` sign. This causes a premature termination of the search process before the last entry has been examined. The second error is another common mistake that the `break` statement in `_Insert` should have been a `return` statement. This causes a duplicate entry being inserted when the name is already defined.

Using the heuristics H1 and H2 as guidelines, the following set of test cases can be derived for *DataDictionary*:

```
1. Insert: "elephant", "a kind of animal"
2. Insert: "apple",    "a kind of fruit"
3. Insert: "cabbage",  "a kind of vegetable"
4. Find:   "apple"
5. Find:   "elephant"
6. Find:   "cabbage"
7. Insert: "elephant", "a kind of mammal"
8. Delete: "apple"
9. Delete: "cabbage"
10. Delete: "elephant"
```

This set of test cases satisfies the coverage heuristics, and it reveals inconsistency between the implementation and the specification:
a) Test case 6 returns `undefined`, while according to the specification it should return the definition of `cabbage`. (caused by error 1.)
b) Test case 10 results in a non-empty dictionary, while according to the specification the dictionary should be empty. (caused by error 2.)

```
_DataDictionary::_DataDictionary(void)
{ _NumOfEntries = 0; }

void _DataDictionary::_Find(
        const String& _name, String& _info)
{  for (i = 0; i < _NumOfEntries-1; i++) {
       // Error 1: '<' should be '<='.
      if (_name == _Dict[i]->_name) {
         _info = _Dict[i]->_info;
         return;
      }
   }
   _info = "Undefined!";
}

void _DataDictionary::_Insert(
        const String& _name, const String& _info)
{  _Entry* _entry = new _Entry;
   _entry->_name = _name;
   _entry->_info = _info;
   for (int i = 0; i <= _NumOfEntries-1; i++) {
      if (_name == _Dict[i]->_name) {
         delete _Dict[i];
         _Dict[i] = _entry;
         break;
         // Error 2: should be a return statement.
      }
   }
   _Dict[_NumOfEntries] = _entry;
   _NumOfEntries++;
}

void _DataDictionary::_Delete(const String& _name)
{  for (int i = 0; i <= _NumOfEntries-1; i++) {
      if (_name == _Dict[i]->_name) {
         delete _Dict[i];
         _NumOfEntries--;
         _Dict[i] = _Dict[_NumOfEntries];
         break;
      }
   }
}
```

Figure 5: An array implementation of the *DataDictionary*: operation implementation.

5 Conclusions

We have presented a formal specification directed testing approach for abstract data types. It provides a test oracle to validate the results of the invocations of test cases against the formal specification, and a testing coverage criterion and heuristics based on formal specifications.

The formal specification directed testing approach presented here differs from other formal specification based testing approaches and offers many advantages:

- The proposed testing approach is based on model-based formal specifications. Testing approaches based on other types of formal specifications have been proposed, such as Anna[6] based on

axiomatic specifications, and DAISTS[7] based on algebraic specifications. The recent development in model-based formal specifications, particularly Z and VDM, demonstrates that model-based specification techniques are more effective and practical for large-scale software systems. Thus, testing approaches based on model-based formal specifications are of far more practical importance.

- The proposed testing approach validates the testing results against specifications rather than assertions on the concrete state of programs.

  Anna provides a sophisticated mechanism, including predicates and quantifiers, to write assertions about various entities in programs such as variables, types, packages, and subprograms. Anna also provides a tool to check the validity of the assertions during run-time. All the entities involved in the assertions are concrete entities in the program. Hayes [8] presented a specification directed module testing approach using the Z notation. It focused on translating the model-based specifications to assertions on the concrete state of an implementation, so that these assertions can be checked at run-time. No tools assisting the translation was proposed.

  Checking the validity of the assertions on the concrete states of an implementation is not equivalent to checking the validity of the predicates in the specification, unless the assertions are consistent with the specification. It is not a trivial task to ensure the consistency between the assertions and the specification. The testing based on assertions means little when the assertions are not consistent with the specification.

  The proposed testing approach does not suffer from this problem, since it uses retrieve functions (abstractors) to construct the abstract states from the concrete states. Previously, retrieve functions are only discussed theoretically and applied to program transformation and verification [9], and have not been applied to automated testing of programs. Under the proposed testing approach, errors in either the implementation or the retrieve functions or both will result in failures during the test run.

- The proposed testing approach introduces a testing coverage criterion based on formal specifications. Deriving testing cases from model-based formal specification using skolemization was proposed by Scullard[10]. His criterion is similar to the heuristic H1 discussed here. The specification coverage proposed here is more extensive. Zweben *et al* [11] proposed a set of quite different coverage criteria based on model-based specifications. They considered specification-based analogues to control and data flow white-box coverage criteria. Their criteria focused on patterns of invocation, however ignored the structure of the predicates in the specification. These criteria can be used to complement and enhance the testing coverage criterion proposed here. Further study on formal specification based testing criteria and derivation of test cases based on formal specifications is required.

References

[1] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, London, 1989.

[2] I. Hayes (ed.), *Specification Case Studies*, Londen: Prentice-Hall, 1987.

[3] C.B. Jones, *Systematic Software Development Using VDM*, Londen: Prentice-Hall, 1986.

[4] M. Spivey, *A Guide to the* `zed` *Style Option*, 1989.

[5] X. Jia, *A Guide to ZED/SL*, Technical Report, No. TR-ISE-93-1, Institute for Software Engineering, DePaul University, 1993.

[6] D.C. Luckham, *Programming with Specifications — An Introduction to Anna, a Language for Specifying Ada Programs*, Springer-Verlag, 1990.

[7] J. Gannon, P. McMullin, and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing," *TOPLAS*, 3(**3**), July 1981, pp. 211-223.

[8] I. Hayes, "Specification Directed Module Testing," *IEEE Transactions on Software Engineering*, 12(**1**) January 1986, pp. 124-133.

[9] J.C.P. Woodcock, "A Tutorial on the Refinement Calculus," *VDM'91*, LNCS, Vol. 521, pp. 79-140, 1991.

[10] G.T. Scullard, "Test Case Selection Using VDM," *VDM'88*, LNCS, Vol. 328, pp. 178-186, 1988.

[11] S.H. Zweben *et al*, "Systematic Testing of Data Abstractions Based on Software Specifications," *Journal of Software Testing, Verification, and Reliability,* 1(**4**), pp. 39-45, 1992.