SPECIFICATION DIRECTED MODULE TESTING

by

Ian Hayes

Technical Monograph PRG-49

July 1985

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Author's address from September 1985:
Department of Computing Science
Queensland University
St. Lucia
Queensland 4067
Australia

SPECIFICATION DIRECTED MODULE TESTING

Abstract

If a program is developed from a specification in a mathematically rigorous manner, work done in the development can also be utilized in the testing of the program. We can apply the better understanding afforded by these methods to provide a more thorough check on the correct operation of the program under test. This should lead to earlier detection of faults (making it easier to determine their causes), more useful debugging information, and a greater confidence in the correctness of the final product. Overall, a more systematic approach should expedite the task of the program tester, and improve software reliability.

The testing techniques described in this paper apply to the testing of abstract data types (modules, packages). The techniques utilize information generated during refinement of a data type, such as the data type invariant and the relationship between the specification and implementation states. The techniques are illustrated by application to the implementation of a symbol table as an ordered list and as a height balanced tree.

Index Terms

Module testing; software reliability; specification language - Z; abstract data types, modules, packages; data type invariant, retrieval function, pre- and post-conditions.

Introduction

Rigorous program development, such as that advocated in Jones' excellent book [5], can do much to increase our confidence in software we produce. The development of a program starts from a high-level specification, which is then refined through one or more stages to produce the final program. Rigorous methods rely heavily on mathematics to specify the software to be developed, and to formalise the relationship between the specification and an implementation. The work done in formalising these relationships can be of great benefit to program testers in developing a thorough testing strategy that will trap errors as early as possible and thus be a aid to debugging.

Given a rigorously developed program it is possible to prove that it meets its specification. If such a proof is performed mechanically (and we trust the verifier) then testing should not be required; given the current state of the art, however, complete mechanical verification is a rarity and is expensive in resources. If the proof is done by hand then there is still room for error and hence room for testing. Rigorous methods can help greatly to increase our understanding of the program that we are developing and hence reduce the number of errors in the initial version of the program. However, we are still prone to make mistakes through oversights and typographical errors and without mechanical verification we will still require testing, especially on larger, more complex programs where errors could more easily slip in unnoticed. By making use of rigorous methods in testing we can increase our confidence in the correctness of the final product in a relatively straightforward manner that requires more moderate resources than complete mechanical verification.

The testing techniques described in this paper apply to the testing of abstract data types (modules, classes, packages, clusters). An abstract data type consists of some data, which we will refer to as its state, and a set of operations on that state. It is a good unit for testing purposes because it represents a coherent whole and, because the operations are all working on the same state, parts of the testing code are common to all the operations; in many cases it would be difficult to test an operation without having the other operations on the data type available. Testing of abstract data types can make use of the data type invariant for checking the consistency of the state between operations, the pre-condition for distinguishing errors in the module under test from those in the test program, and the relation between the specification and implementation states along with the individual operation input-output relations for testing the correctness of the operations.

We will illustrate the testing technique by following through the development and testing of a symbol table module. The notation used in this paper will be based on the specification language Z [1, 6]; programs will be given in a Pascal-like notation.

Symbol Table Specification

This example specifies a symbol table with an operation to update an entry. We will describe the table by a partial function from symbols (SYM) to values (VAL).

```
ST______________________________
| st : SYM ⇸ VAL
|_______________________________
```

The arrow $\nrightarrow$ indicates a function from SYM to VAL that is not necessarily defined for all elements of SYM (hence "partial"). The subset of SYM for which it is defined is its domain of definition

$$\mathrm{dom}(st)$$

If a symbol s is in the domain of definition of st ($s \in \mathrm{dom}(st)$) then $st(s)$ is the unique value associated with s ($st(s) \in VAL$). The notation $\{ s \mapsto v \}$ describes a function which is only defined for that particular s:

$$\mathrm{dom}(\{ s \mapsto v \}) = \{ s \}$$

and maps that s onto v:

$$\{ s \mapsto v \}(s) = v$$

More generally we can use the notation

$$\{ x_I \mapsto y_1, x_2 \mapsto y_2, \ldots , x_n \mapsto y_n \}$$

where all the x_k's are distinct to define a function whose domain is

$$\{ x_1, x_2, \ldots , x_n \}$$

and whose value for each x_k is the corresponding y_k. For example, if we have the following mapping

$$st = \{ \text{“John”} \mapsto v_1, \text{“Mary”} \mapsto v_2 \}$$

which maps “John” onto v_1 and “Mary” onto v_2, then the domain of st is the set

$$dom(st) = \{ \text{“John”}, \text{“Mary”} \}$$

and

$$st(\text{“John”}) = v_1$$
$$st(\text{“Mary”}) = v_2$$

The notation

$$\{\}$$

is used to denote the empty function whose domain of definition is the empty set.

We are describing a symbol table by modelling it as a partial function. This use of a function is quite different to the normal use of functions in computing where an algorithm is given to compute the value of the function for a given argument. Here we use it to describe a data structure. There may be many possible models that we can use to describe the same object. Other models of a symbol table could be a list of pairs of symbol and value, or a binary tree containing a symbol and value in each node. These other models are not as abstract because many different lists (or trees) can represent the same function. The list and tree models of a symbol table tend to bias an implementor working from the specification towards a particular implementation. In fact, both lists and trees could be used to implement such a symbol table. However, any reasoning we wish to perform involving symbol tables is far easier using the partial function model than either the list or tree model.

Initially the symbol table is empty

$$st = \{\}$$

The update operation can change the symbol table. We represent the effect of such an operation by the relationship between the symbol table before the operation and the symbol table after the operation. We use

$$
\begin{array}{|l}
\Delta ST \\
\hline
\quad ST_0 \\
\quad ST \\
\hline
\end{array}
$$

to represent the state before (ST_0) and the state after (ST). The above defintition of ΔST is equivalent to the following one in which ST_0 and ST have been expanded

$$
\begin{array}{|l}
\Delta ST \\
\hline
\quad st_0 \ : \ SYM \nrightarrow VAL \\
\quad st \ \ \ : \ SYM \nrightarrow VAL \\
\hline
\end{array}
$$

We use the convention that zero subscripted symbol table (st_0) represents the state before an operation and the undecorated (st) the state after. (This convention is slightly different to the convention used in the references [5, 6] both of which use undecorated variables for the state before (st) and primed variables for the state after (st'); the convention used in this paper allows some simplification of the assertions used in programs.)

The operation to update an entry in the table is described by the following schema

$$
\begin{array}{|l}
Update \\
\hline
\quad \Delta ST \\
\quad s? : SYM \\
\quad v? : VAL \\
\hline
\quad st = st_0 \oplus \{ \ s? \mapsto v? \ \} \\
\hline
\end{array}
$$

A schema consists of two parts: the declarations (above the centre line) in which variables to be used in the schema are declared, and a predicate (below the centre line) containing predicates giving properties of and relating those variables. In the schema Update the second line declares a variable with name "s?" which is the symbol to be updated. The third line declares a variable with name "v?" to be the value to be associated with s? in the symbol table. By convention names in the declarations ending in "?" are inputs and names ending in "!" will be outputs; the "?" and "!" are otherwise just part of the name.

The predicate part of the schema states that it updates the symbol table (st_0) to give a new symbol table (st) in which the symbol $s?$ is associated with the value $v?$. Any previous value associated with $s?$ (if there was one) is lost.

The operator $\oplus$ (function overriding) combines two functions of the same type to give a new function. The new function $f \oplus g$ is defined at x if either f or g are defined, and will have value $g(x)$ if g is defined at x, otherwise it will have value $f(x)$

$$\mathrm{dom}(f \oplus g) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$$

$$x \in \mathrm{dom}(g) \qquad\qquad\qquad \Rightarrow (f \oplus g)(x) = g(x)$$

$$x \notin \mathrm{dom}(g) \wedge x \in \mathrm{dom}(f) \Rightarrow (f \oplus g)(x) = f(x)$$

For example

$$\begin{aligned} &\{\text{ "Mary"} \mapsto v_1,\ \text{"John"} \mapsto v_2 \} \oplus \{ \text{"John"} \mapsto v_3,\ \text{"George"} \mapsto v_1 \} \\ &\qquad = \{ \text{"Mary"} \mapsto v_1,\ \text{"John"} \mapsto v_3,\ \text{"George"} \mapsto v_1 \} \end{aligned}$$

For the operation Update above the value of $st(x)$ is $v?$ if $x = s?$, otherwise it is $st_0(x)$ provided x is in the domain of st_0. In Update we are only using $\oplus$ to override one value in our symbol table function, however, the operator $\oplus$ is more general: its arguments may both be any functions of the same type.

For a symbol table module we would normally define further operations to lookup and delete entries in the table. For the purposes of illustrating testing, however, we will only consider the Update operation.

If we were not allowed to know the internal structure of the implementation of the symbol table, this specification would give us all the information we needed to test that implementation. At one level this provides a reasonable testing strategy but, as will be demonstrated, if we are allowed knowledge of the implementation we can construct a more rigorous test of that implementation.

Implementation as an Ordered Sequence

We will first consider implementing a symbol table as an ordered sequence and later as a height balanced binary tree. The testing techniques do not have as much to offer for the simpler ordered sequence implementation, but it will serve to illustrate the ideas involved before moving on to the more complicated balanced tree implementation.

Each item in the ordered sequence will consist of a pair of symbol and corresponding value.

$$\mathtt{Item} \mathrel{\widehat{=}} \mathtt{SYM} \times \mathtt{VAL}$$

We also define selector functions sym and val to select the symbol and value, respectively, from an item.

$$\mathtt{sym} : \mathtt{Item} \rightarrow \mathtt{SYM}$$
$$\mathtt{val} : \mathtt{Item} \rightarrow \mathtt{VAL}$$

such that for $\mathtt{it} : \mathtt{Item}$ we have

$$\mathtt{it} = (\mathtt{it.sym}, \mathtt{it.val})$$

The state is given by

```
SST______________________________________
| sst : seq Item
|--------------
| ordered(sst)
|________________________________________
```

where $\mathtt{ordered}(s : \mathbb{N} \mathrel{⤀} \mathtt{Item}) \mathrel{\widehat{=}}$
$$(\forall i, j : \mathrm{dom}(s) \bullet i < j \Rightarrow s(i).\mathtt{sym} <_S s(j).\mathtt{sym})$$

where we are assuming there is some total order ($<_S$) on symbols. The state is modelled by a sequence of items, sst. The domain of the sequence, dom(sst), is the set of integers that are valid indexes into the sequence. The invariant states that sst is in strictly ascending order on symbols. Initially the sequence would be empty.

$$\mathtt{sst} = [\,]$$

Before describing the Update operation on this state let us look at the relation between the ordered sequence model and the partial function model.

```
ST_SST
   ST
   SST
   ----------
   st = { it : rng(sst) • it.sym ↦ it.val }
```

where the range of the sequence, rng(sst), is the set containing all the items in the sequence.

ST_SST shows how, given a sequence representation, we can retrieve the partial function model of a symbol table by, for each item in the sequence, mapping its symbol to its value.

The update operation on the sequence model is given by

UpdateS

ΔSST

$s? : SYM$

$v? : VAL$

$$rng(sst) = rng(sst_0) \cup \{ (s?, v?) \}$$

where

ΔSST

SST_0

SST

The invariant on the states ensures that the final state sst is ordered; the predicate part of UpdateS ensures that the final sequence contains the correct values.

The following is a possible implementation written in a Pascal-like notation. It uses the simple scheme of appending the new pair to the sequence and then rippling it down the sequence into the correct place to maintain the ordering.

```
UpdateS(s? : SYM, v? : VAL):

    { sst = $sst_0$ ∧ ordered($sst_0$) }

    sst := sst $\frown$[ (s?,v?) ];
    i := |sst|;

    { Inv: rng(sst) = rng($sst_0$) ∪ {(s?,v?)} ∧
          1 ≤ i ≤ |sst| ∧
          ordered((1..i-1)◁sst) ∧ ordered((i..|sst|)◁sst) }

    while i ≠ 1 cand sst(i-1).sym $>_S$ sst(i).sym do
       begin
         swap(sst(i-1), sst(i));
         i := i - 1
       end

    { Inv ∧ (i = 1 ∨ sst(i-1).sym $\leq_S$ sst(i).sym) }
```

where

$\frown$ is concatenation of sequences,

[(s?,v?)] is a sequence containing a single item: that with symbol s? and value v?,

|s| gives the length of a sequence s,

(i..j) ◁ sst is the sequence sst with its domain restricted (◁) to values in the subrange i to j inclusive, and

cand is the conditional and operator: it only evaluates its second argument if its first argument is true.

Checking the Invariant

To test this implementation we will first write a procedure to check if the invariant holds. This will be used to check the invariant initially and then after every operation performed on the symbol table during testing. The invariant on the ordered sequence is

$$(\forall i, j : \mathrm{dom}(sst) \bullet i < j \Rightarrow sst(i).sym <_S sst(j).sym)$$

The following code should suffice to check this holds

```
k := 1;

{ Inv: ordered((1..k)◁sst) }

while k < |sst| cand sst(k).sym $<_S$ sst(k+1).sym do
     k := k + 1;

{ Inv ∧ (k ≥ |sst| ∨ sst(k).sym $\geq_S$ sst(k+1).sym) }

if k < |sst| then { sst(k).sym $\geq_S$ sst(k+1).sym }
     "report unordered sequence"
```

The above procedure is written solely for testing purposes. In this case the testing code is as complex as the update operation itself. For more sophisticated implementations the invariant check is generally (though not always) simpler and shorter than an operation. If the invariant check on a data structure is very simple and efficient then it is a good idea to leave the check on the invariant in the code when it is put into operation in order to aid earlier detection of faults that do occur in operational use.

The strategy of checking the invariant after every operation on the symbol table will catch a violation of the invariant immediately after the operation which caused it. To aid in debugging, diagnostic information such as the point at which the sequence is out of order and the corresponding items, should be displayed if the invariant check fails.

It is possible that the invariant check fails to detect an invalid state because there is an error in the invariant check that "cancels out" the error in the operation. In the majority of cases, however, we hope that the extra redundancy of the invariant check will not be of the cancelling out form. Perhaps using different people to code the testing and the module may help avoid this problem and make full use of the redundancy in detecting errors.

If we now run a series of tests on the "ordered sequence" implementation we should discover that it is incorrect: if the same symbol is inserted into the table more than once then the ordered sequence implementation will leave the first pair in the sequence when the second pair is inserted. This will cause our invariant check to fail because there will be two consecutive items with the same symbol whereas the invariant states that the sequence is in strictly ascending order (no duplicates). The invariant check will fail as soon as a symbol is inserted a second time. If we followed the advice given above and displayed the items which caused the invariant check to fail, it should be obvious that the problem is due to the duplicate entry.

If we did not perform the invariant check while testing, the error in the ordered sequence implementation would not be discovered immediately after the second insertion of the same symbol. The problem would probably be detected when we perform an operation that looks up the value associated with the duplicated symbol. This could happen at a point in the program far removed from the cause of the problem, and may not occur until a considerable time after the duplicate entry has been inserted; locating the cause of the problem could then be much more difficult.

Checking the Pre-Condition

The invariant check in the above example failed because the implementation was incorrect. In general, the invariant check can fail either because of an incorrect implementation or because the testing program incorrectly used the operations of the module. In the latter case, a failure can be caused if the pre-condition of an operation does not hold when the operation is invoked. In our example UpdateS has a pre-condition of true so the testing program can never use the operation incorrectly. At this stage let us not try to correct the implementation of UpdateS but rather change the original specification to include the following pre-condition stating that the symbol to be updated is not already in the symbol table

$$s? \notin dom(st_0)$$

Having now changed our specification (a tactic widely used in practice but not really recommended as the most appropriate solution in general) it is the test program that is now incorrect if it calls UpdateS with a symbol that is already in the table. In order to distinguish between a failure of the implementation and a failure of the test program we can insist (at least for testing purposes) that the operations should check that their pre-conditions hold and if not report an error. For our symbol table example, checking the pre-condition that the symbol to be inserted is not already in the table can be achieved by adding the following code at the end of the current implementation

```
{ $rng(sst) = rng(sst_0) \cup \{(s?, v?)\} \wedge$
     $1 \leq i \leq |sst| \wedge$
     $ordered((1..i-1) \lhd sst) \wedge ordered((i..|sst|) \lhd sst) \wedge$
     $(i = 1 \vee sst(i-1).sym \leq_S sst(i).sym)$ }
if i > 1 cand sst(i-1).sym = sst(i).sym then
     "report symbol already in table"
```

Note that the above check only discovers that the pre-condition does not hold after it has modified the data structure. This is reasonable if all we do on a pre-condition failure is to print a message and abort; we should not attempt to carry on testing any further.

If the pre-condition checks are inexpensive then it is prudent to leave them in the code permanently. If they are too expensive to leave in then we should at least have the ability to re-introduce them during the testing of any program that makes use of the module so that errors in its use of the module are detected as early as possible. A good rule is to design module interfaces in such a way that the pre-condition can always be checked efficiently. This is an essential requirement for public interfaces such as operating system calls or widely used packages; it can help sort out debates about which component is at fault.

Checking the Input-Output Relation

Checking invariants and pre-conditions is not a thorough test of an implementation; the implementation could be quite disastrously wrong and still maintain the invariant. To thoroughly check an algorithm we also need to check that it conforms to the input-output relation of the specification.

To perform such checking by testing we need to compare the results of two implementations of the same high-level specification. To illustrate the technique on our symbol table example let us assume that we have available a (very high-level) programming language with maps and operations on maps as primitives. (In practice, such programming languages are not generally available; when we consider the more involved example of testing balanced trees we will make use of a simpler implementation, namely the ordered list implementation described above, to provide a cross-check.) The operation to update a symbol table can be coded in our very high-level programming language as

```
Update(s? : SYM, v? : VAL):

    st := st ⊕ { s? ↦ v? }
```

where the state for this implementation is identical to that in the original specification.

We now have two implementations, Update and UpdateS, of the operation to update a symbol table. The states that the two implementations work on are quite different - in one case a mapping and in the other an ordered sequence - so the two are not directly comparable. In order to perform a cross-check between the "mapping" implementation and the "ordered sequence" implementation we need to implement a retrieval function that extracts a mapping from an ordered sequence. We can then compare the extracted mapping with that from the "mapping" implementation both initially and after every operation; each operation being performed on both implementations before the retrieval and comparison test.

The relation between the "mapping" and "ordered sequence" states is defined by the retrieval relation ST_SST given previously. The following code will retrieve the output mapping st! from the input sequence sst?

```
ST_SST(sst? : seq Item, st! : ST):

    i := 0;
    st! := {};

    { Inv: st! = { it:rng((1..i)◁sst?) • it.sym ↦ it.val } }
    while i ≠ |sst?| do
      begin
        i := i + 1;
        st! := st! ⊕ { sst?(i).sym ↦ sst?(i).val }
      end

    { st! = { it : rng(sst?) • it.sym ↦ it.val } }
```

The retrieved mapping can then be compared directly with that used in the mapping implementation

```
if st! ≠ st then
      "report input-output relation check failed"
```

Any error detected by the comparison may indicate an error in either

- the "ordered sequence" implementation,
- the "mapping" implementation,
- the ordered sequence to mapping retrieval function, or
- the comparison itself.

The last three should normally be less likely because they should be somewhat simpler. However, they cannot be ruled out as possible causes of errors and if an error is detected further investigation will be required in order to determine which of the above is the cause and to find the actual fault. In more complex cases the retrieval function may need to be refined by a series of steps and may itself need testing before it is put to use.

When we combine input-output relation checks with invariant and pre-condition checks we get a thorough test mechanism for operations on the "ordered sequence" symbol table implementation. It is almost certain that the redundancy incorporated into the above checks is sufficient to catch any fault manifested during testing. Furthermore, the fault will have been isolated to a particular operation and if appropriate diagnostics have been added to the checking code the cause should be easily found. However, we are only dealing with a testing strategy and like all testing it does not exclude the possibility of latent errors: errors that did not occur on the test cases used but could occur on other cases. Such latent errors show the inherent weakness of program testing when compared with program verification. To reduce the possibility of latent errors left after testing we should use our knowledge of the implementation to ensure that it is thoroughly exercised; all parts of the code should be tested. The selection of test cases is covered in other treatments of program testing [4] and will not be pursued further here.

Height Balanced Binary Trees

In the "ordered sequence" implementation the procedures to test the invariant and retrieve the symbol table are both as complicated as the operation to update an item. We will now consider a more involved example in which the invariant testing and retrieval function are somewhat simpler than the operations.

Height balanced binary trees were invented by Adel'son-Velskii and Landis [2] to provide a binary search tree with worst case insert and delete times of O(log N) where N is the number of nodes in the tree. A binary tree is height balanced if at every node in the tree the heights[1] of its left and right subtrees differ by at most one. The beauty of a height balanced tree is that its worst case height is at most 45% greater that that of an equivalent perfectly[2] balanced tree, and insertion and deletion of nodes can be performed by examining a path from the root to a node unlike perfectly balanced trees. Search, insert and delete operations can all be performed in O(log N) time in the worst case which should be compared with a worst case time of O(N) for these operations on an ordinary (unbalanced) tree.

The major disadvantage of balanced trees[3] is that the algorithms to manipulate them are considerably more complicated than those for an unbalanced tree. Fortunately, for the purposes of this paper we do not need to delve into the details of these operations in order to illustrate the approach to testing them. The interested reader is referred to one of the many books on algorithms that discuss operations on balanced trees in detail. One such book is Wirth's "Algorithms + Data structures = Programs" [7]. To give a crude idea of the complexity of the operations on balanced trees, the Pascal versions given by Wirth consist of 63 lines for insertion (p220-1) and 92 lines for deletion (p223-5). These figures should be compared with those for unbalanced trees: 19 lines for insertion (p205) and 18 lines for deletion (p211). Not only are the balanced tree operations considerably longer than their unbalanced tree counterparts, they are, in the opinion of the author, a good deal more subtle and more liable to erroneous implementation.

[1] the height of a binary tree is the maximum number of nodes on a path starting at its root and descending down the tree.

[2] a perfectly balanced tree is a binary tree in which at every node the number of nodes in its left and right subtrees differ by at most one.

[3] for the remainder of this paper we will abbreviate "height balanced binary tree" to "balanced tree".

As promised earlier we do not need to look in detail at the implementation of the operations on balanced trees. What we do need to look at closely, however, is the state invariant for a balanced tree. A tree is given by

```
Tree ≙ Node | nil
```

That is, a Tree is either a Node or it is the special value nil, where

```
Node______________________________________
│  sym : SYM
│  val : VAL
│  bal : -1..1
│  left,
│  right : Tree
├─────────────────
│  (∀ s : syms(left)  • s <_S sym) ∧
│  (∀ s : syms(right) • sym <_S s) ∧
│  bal = height(left) - height(right)
└─────────────────────────────────────────
```

where

$$\mathrm{syms} : \mathrm{Tree} \rightarrow \mathbb{P}\ \mathrm{SYM}$$

such that for $n : \mathrm{Node}$

$$\mathrm{syms}(\mathrm{nil}) = \{\}$$

$$\mathrm{syms}(n) = \mathrm{syms}(n.\mathrm{left}) \cup \{\ n.\mathrm{sym}\ \} \cup \mathrm{syms}(n.\mathrm{right})$$

and

$$\mathrm{height} : \mathrm{Tree} \rightarrow \mathbb{N}$$

such that for $n : \mathrm{Node}$

$$\mathrm{height}(\mathrm{nil}) = 0$$

$$\mathrm{height}(n) = \max(\mathrm{height}(n.\mathrm{left}),\ \mathrm{height}(n.\mathrm{right})) + 1$$

The trees are both ordered and balanced. A tree is ordered if at each node in the tree all the symbols in its left subtree are less than the symbol at the node which is less than all the symbols in its right subtree. A tree is balanced if at every node the difference in heights between the left and right subtrees is equal to the bal field of the node (which can only take on values in the range -1 . . 1).

The relation between a balanced tree and the high level specification of a symbol table is given by

```
ST_BT
  ST
  BT
  ─────────────
  st  = { node : nodes(t) • node.sym ↦ node.val }
```

where

$$nodes : Tree \rightarrow \mathbb{P}\ Node$$

such that for $n : Node$

$$nodes(nil) = \{\}$$

$$nodes(n) = nodes(n.left) \cup \{ n \} \cup nodes(n.right)$$

Checking the Invariant

As before we can write a procedure to check the state invariant: the tree is both balanced and ordered. A procedure to check that a tree is balanced follows. It performs a post order traversal of a tree checking that each subtree is balanced and returning the height of the tree so that the higher level checking that the tree is balanced can take place.

```
Balanced(t? : Tree, h! : integer):

        if t? = nil then
              h! := 0
        else
            begin
              var hl, hr : integer;
              Balanced(t?.left,  hl);
              Balanced(t?.right, hr);
              { hl = height(t?.left) ∧ hr = height(t?.right) }
              if hl - hr ≠ t?.bal then
                    "report unbalanced tree"
              h! := max(hl, hr) + 1
            end
```

We have assumed here that the implementation of our programming language will trap any assignment of a value outside the range $-1..1$ to the bal field of a node; if this were not the case then a check that the bal field of each node is in this range should be added to the above procedure. The procedure to check that a tree is ordered is straightforward and is omitted here.

For balanced trees the invariant checking is far less complicated than the operations; it is more akin to the complexity of the operations on the simpler unbalanced trees, requiring only straightforward tree traversal algorithms. The great value of the invariant check is that if an operation otherwise works correctly but manages to corrupt the data type invariant the fault will be detected immediately after the operation rather than at some indeterminate time in the future when an operation tries to access the corrupted part of the data structure. Not only is the detection in this latter case well after the fault it may be on an operation other than the one that caused the corruption; other than detecting that there is an error one has been given little help in diagnosing the fault.

Given this invariant check procedure our testing can now check that the invariant holds initially and then after each operation during testing. The invariant checking above requires O(N) time versus the O(log N) time for the operations themselves. Hence it is not sensible to leave the invariant check in the program after testing. After all, the point of using balanced trees was to take advantage of their worst case O(log N) performance; if we were to leave the invariant check in the code the performance would always be O(N) and hence worse than the unbalanced tree which, while being O(N) worst case, is only O(log N) average case.

The invariant check given above is a far more stringent test that the state of a module is consistent than any that can be carried out purely from knowledge of the high-level specification even if one is given a retrieval function to extract the abstract state. It is possible that the implementation could be incorrect in a way that does not affect the high-level correctness. For example, the implementation may correctly maintain an ordered tree but it may be incorrectly balanced. In this case the operations would appear to work correctly but in some cases would not be as efficient. Such a fault could only be detected externally by timing operations and would require the testing to generate a badly balanced tree. With knowledge of the internal operation of the algorithm in the invariant check it is far less likely that an incorrect implementation would go undetected.

Checking the Pre-Condition

As with the "ordered sequence" implementation a pre-condition check can be incorporated into the implementation using balanced trees. This will detect any incorrect use of the operations by the testing program. For balanced trees a simple constant-time check (which should be left in the code permanently) can be incorporated into the update operation. As this is quite simple to do, but to explain requires detailed knowledge of the update operation on balanced trees, we will not elaborate the pre-condition check for balanced trees here.

Checking the Input-Ouput Relation

As with the "ordered sequence" implementation we need to check that the input-output relation is satisfied. For this example we will not assume that we have available a very high-level programming language with mappings as primitives. In order to cross-check the input-output relation we need a second (simpler) implementation of a symbol table. Fortunately we have just that in our "ordered sequence" implementation. To perform the cross-check we need a retrieval function that extracts an ordered sequence from a balanced (ordered) tree. The relation between ordered sequences and balanced trees is given by

```
SST_BT
  SST
  BT
  ─────────────
  { node : nodes(t) •  node.sym ↦ node.val }
  = { it : rng(sst) •  it.sym   ↦ it.val }
```

Extracting an ordered sequence from an ordered tree can be achieved by the following tree traversal algorithm

```
TreetoSequence(t? : Tree, sst! : seq[Item]):

    if t = nil then
            sst! := []
    else { t ≠ nil }
        begin
            var lsst, rsst : seq[Item];
            TreetoSequence(t?.left,  lsst);
            TreetoSequence(t?.right, rsst);
            sst! := lsst⁀[(t?.sym, t?.val)]⁀rsst
        end
```

The sequence retrieved by TreetoSequence is compared with the sequence maintained by the "ordered sequence" implementation after each operation is performed (on both implementations). The code for the comparison is straightforward and has been omitted here.

For the height balanced binary tree example the procedures required to use the testing techniques outlined in this paper require only a fraction of the time necessary for a programmer to develop the somewhat more sophisticated balanced tree operations. The extra time is well spent in terms of increasing one's confidence in the correct operation of the algorithms, but furthermore the techniques are likely to actually save time: if there are errors in the operations the testing will isolate the errors quickly and provide useful diagnostics to aid in debugging.

Discussion

When implementing abstract data types in a programming language with facilities to support them (for example, Modula modules, Ada packages, or Clu clusters) the invariant check and retrieval procedures will both have to be part of the module as they need access to the internal data structure which should not be accessible externally. This will probably imply that the person responsible for the module should write these when writing the module (although as mentioned earlier there are good reasons for having a separate person write them). In practice this probably represents a reasonable line of demarcation between the module writer and tester as these functions provide everything that the tester needs from the module internals to apply the testing techniques.

The author has used the techniques described above to test an implementation of B-trees [3]: balanced multi-way trees suitable for secondary storage data bases. B-trees are more complicated data structures than height balanced trees and the algorithms to manipulate them have a number of special cases that can easily lead to errors in implementation. In the testing of the B-tree implementation the techniques described above were able to isolate two errors (one omission and the other a swap of variable names) and give good hints as to the nature of the fault; in this respect the invariant check, which for the B-tree is involved but not difficult to implement, was particularly useful in detecting faults as soon as possible after their prime cause. The use of these techniques certainly increased the author's confidence in the correctness of the final implementation - especially that the algorithms actually implemented B-trees rather then some other (strange) variety of multi-way trees.

Another technique that can be used in testing programs is to check assertions such as loop invariants, at execution time. This could be useful if a fault is detected in an operation of an abstract data type but the cause is not obvious. Unfortunately expanding such assertions is non-trivial; in some cases the code to check a loop invariant can be more complicated than the original loop. The tactic of testing at the abstract data type level seems to provide the most benefits for the amount of effort involved; coding up assertions can be left to aid in debugging when a non-obvious error is detected, although it is probably better to go back to the original reasoning about the program and find the fault there.

The testing procedures should not be discarded once a module has been tested; they will be useful to anyone responsible for making changes to the module (where introduction of errors is more likely due to lack of understanding). The invariant check procedure is of more general use if data is kept on permanent storage devices. It can be used to check the consistency of the data after a hardware or software failure has occured. It cannot guarantee the correctness of the data but it can find inconsistencies which imply the data is incorrect and it can ensure that the data is in a state suitable for running the system.

References

1. Abrial, J.-R. The specification language Z: Basic library. *Oxford University Programming Research Group internal report*, 1980.

2. Adel'son-Velskii, G. M., and Landis, Y. M. An algorithm for the organization of information. *English translation in Soviet Math. Dokl.* 3, (1962), 1259-1262.

3. Bayer, R., and McCreight, E. M. Organization and maintenance of large ordered indices. *Acta Informatica 1*, 3 (1972), 173-189.

4. Beiser, B. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

5. Jones, C. B. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

6. Morgan, C. C., and Sufrin, B. A. Specification of the UNIX file system. *IEEE Transactions on Software Engineering 10*, 2 (March 1984), 128-142.

7. Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

Appendix: Notation

1. Definitions and declarations.

Let x, x_k be identifiers and T, T_k sets.

$LHS \mathrel{\widehat{=}} RHS$	Definition of LHS as syntactically equivalent to RHS.
$x : T$	Declaration of identifier x of type T.
$x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n$	List of declarations.
$x_1,\ x_2,\ \ldots\ ,\ x_n : T$	$\mathrel{\widehat{=}} x_1 : T;\ x_2 : T;\ \ldots\ ;\ x_n : T.$

2. Logical symbols.

Let P, Q be predicates and D declarations.

$\neg P$	Negation: “not P”.
$P \vee Q$	Disjunction: “P or Q”.
$P \wedge Q$	Conjunction: “P and Q”.
$P \Rightarrow Q$	Implication: “P implies Q” or “if P then Q”.
$\exists\, x : T \bullet P$	Existential quantification: “there exists an x of type T such that P”.
$\forall\, x : T \bullet P$	Universal quantification: “for all x of type T, P holds”.
$\exists\, x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \bullet P$	“There exist x_1 of type T_1, x_2 of type T_2, . . . , and x_n of type T_n, such that P holds.”
$\forall\, x_1 : T_1;\ x_2 : T_2;\ \ldots\ ;\ x_n : T_n \bullet P$	“For all x_1 of type T_1, x_2 of type T_2, . . . , and x_n of type T_n, P holds.”

3. Sets.

Let S and T be subsets of X; t, t_k terms; P a predicate and D declarations.

$t \in S$ Set membership: "t is an element of S".

$t \notin S$ $\widehat{=} \neg(t \in S)$.

$S \subseteq T$ Set inclusion: $S \subseteq T \ \widehat{=} \ (\forall x : S \bullet x \in T)$.

$S \subset T$ Strict set inclusion: $S \subset T \ \widehat{=} \ S \subseteq T \land S \neq T$.

$\{\}$ The empty set.

$\{ t_1, t_2, \ldots , t_n \}$

The set containing t_1, t_2, ... and t_n.

$\{ x : T \mid P \}$ The set containing exactly those x of type T for which P holds.

$(t_1, t_2, \ldots , t_n)$

Ordered n-tuple of t_1, t_2, ... and t_n.

$T_1 \times T_2 \times \ldots \times T_n$

Cartesian product: the set of all n-tuples such that the kth component is of type T_k.

$\{ x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \mid P \}$

The set of n-tuples $(x_1, x_2, \ldots , x_n)$ with each x_k of type T_k such that P holds.

$\{ x_1 : T_1; x_2 : T_2; \ldots ; x_n : T_n \mid P \bullet t \}$

The set of t's such that given all the x_k of type T_k, P holds.

$\{ D \bullet t \}$ $\widehat{=} \ \{ D \mid true \bullet t \}$

$\mathbb{P}\ S$ Powerset: $\mathbb{P}\ S$ is the set of all subsets of S.

$\mathbb{F}\ S$ Finite subsets of S.

$S \cup T$ Set union: given $S, T : \mathbb{P}\ X$,

$\widehat{=} \ \{ x : X \mid x \in S \lor x \in T \}$.

$S \cap T$ Set intersection: given $S, T : \mathbb{P}\ X$,

$\widehat{=} \{ x : X \mid x \in S \land x \in T \}$.

$S - T$ Set difference: given $S, T : \mathbb{P}\ X$,

$\widehat{=} \{ x : X \mid x \in S \land x \notin T \}$.

$|S|$ Size (number of elements) of a finite set.

4. Relations and functions.

A relation is modelled by a set of ordered pairs hence operators defined for sets can be used on relations. A function is a relation with the property that for each element in its domain there is a unique element in its range related to it. As functions are relations, operators defined for relations also apply to functions.

Let R be a relation; f be a function; A, B and S be sets; and x, x_k, y, y_k be terms.

$A \leftrightarrow B$ The set of relations from A to B: $A \leftrightarrow B \mathrel{\widehat{=}} \mathbb{P}\,(A \times B)$.

$x\ R\ y$ x is related by R to y: $x\ R\ y \mathrel{\widehat{=}} (x,\ y) \in R$.

$A \nrightarrow B$ The set of partial functions from A to B:
$\mathrel{\widehat{=}} \{\ f\colon A \leftrightarrow B \mid$
$(\forall\ a\colon A;\ b,\ b'\colon B \bullet a\ f\ b \wedge a\ f\ b' \Rightarrow b = b')\ \}$.

$x \mapsto y$ $\mathrel{\widehat{=}} (x,\ y)$

$\{\ x_1 \mapsto y_1,\ x_2 \mapsto y_2,\ \ldots\ ,\ x_n \mapsto y_n\ \}$
The relation $\{\ (x_1,\ y_1),\ (x_2,\ y_2),\ \ldots\ ,\ (x_n,\ y_n)\ \}$
relating x_1 and y_1, x_2 and y_2 . . . , x_n and y_n.

$f\ x$ The function f applied to x.

$\mathrm{dom}\ R$ The domain of a relation or function: for $R\colon A \leftrightarrow B$,
$\mathrm{dom}\ R \mathrel{\widehat{=}} \{\ a\colon A \mid (\exists\ b\colon B \bullet a\ R\ b)\ \}$.

$\mathrm{rng}\ R$ The range of a relation or function: for $R\colon A \leftrightarrow B$,
$\mathrm{rng}\ R \mathrel{\widehat{=}} \{\ b\colon B \mid (\exists\ a\colon A \bullet a\ R\ b)\ \}$.

$S \lhd R$ Domain restriction:
$\mathrel{\widehat{=}} \{\ x : X;\ y : Y \mid x\ R\ y \wedge x \in S\ \}$.

$S \ntriangleleft R$ Domain subtraction:
$\mathrel{\widehat{=}} \{\ x : X;\ y : Y \mid x\ R\ y \wedge x \notin S\ \}$.

$R_1 \oplus R_2$ Relational or functional overriding: for $R_1,\ R_2 : A \leftrightarrow B$,
$\mathrel{\widehat{=}} (\mathrm{dom}\ R_2 \ntriangleleft R_1) \cup R_2$.

5. Numbers.

$\mathbb{N}$ The set of natural numbers (non-negative integers).

$\mathbb{N}^+$ The set of strictly positive natural numbers.

$\mathbb{Z}$ The set of integers (positive, zero and negative).

$m..n$ The set of integers between m and n inclusive:
$m..n \mathrel{\widehat{=}} \{\ k\colon \mathbb{Z} \mid m \leq k \wedge k \leq n\ \}$.

6. Sequences.

Let X be a set; S be a sequence; and lower case variables be terms.

seq X The set of sequences whose elements are drawn from X:
$\mathrel{\widehat{=}} \{ S : \mathbb{N}^{+} \nrightarrow X \mid \mathrm{dom}\ S = 1..|S| \}$.

$|S|$ The length of sequence S.

$[\,]$ The empty sequence $\{\}$.

$S(i)$ The i th element in the sequence S.

$[x_1, x_2, \ldots, x_n]$ The sequence $\{ 1 \mapsto x_1, 2 \mapsto x_2, \ldots, n \mapsto x_n \}$.

$[s_1, s_2, \ldots, s_n] ^\frown [t_1, t_2, \ldots, t_m]$ Concatenation:
$\mathrel{\widehat{=}} [s_1, s_2, \ldots, s_n, t_1, t_2, \ldots, t_m]$

$\mathrm{rng}([s_1, s_2, \ldots, s_n])$ Range of a sequence: the set of items in the sequence:
$\mathrel{\widehat{=}} \{ s_1, s_2, \ldots, s_n \}$,
$\mathrm{rng}([\,]) = \{\}$.

7. Schema notation.

Schema definition:

```
SCH______________________________
| a: A
| b: B
|_______________
|
| predicate
|________________________________
```

A schema groups together some declarations of variables and a predicate relating these variables. The following conventions are used for variable names in those schemas which represent operations:

subscript "$_0$" state before the operation,
undecorated state after the operation,
ending in "?" inputs to the operation, and
ending in "!" outputs from the operation.

A schema S may be included within a schema T, in which case the declarations of T are merged with the other declarations of S (variables declared in both S and T must be the same type) and the predicates of S and T are conjoined.