Chapter 10

Testing

Redundancy is the essence of testing.

10.1 Introduction

At this point, we encourage the reader to review Section 2.4.

10.1.1 Systematic testing

It is common practice to ignore testing until after implementation, and to discard the tests shortly after acceptance of the software. This ad hoc approach to testing is ineffective: because the testing is developed too late to influence design decisions, the resulting software is often hard to test. This approach to testing is also expensive, because the testing is not reused during maintenance.

A systematic approach to testing requires that the testing be

- *planned*: to permit design for testability.
- *documented*: so that the test cases can easily be understood and the adequacy of the test cases can be evaluated, and
- *maintained*: so that the test cases can be executed after every change to the software.

To perform systematic testing, we maintain two work products for each module in SHAM: the *Test Plan* (TP) describes the strategy for selecting and executing the tests, and the *Test Implementation* (TI) implements the TP.

When designing test cases, it is important to keep in mind Dijkstra's law of testing (see Section 2.4): "program testing can be used to show the presence of bugs, but never their absence" [31]. This means that the focus of testing should be to detect program errors; *not* to show that the program is free of errors. Although this difference is a subtle and mostly psychological one, its consequences are important. It means we should design test cases so that they are most likely to expose errors.

Despite the limitations of testing, it is important as an independent check on code that has been carefully designed and inspected.

10.1.2 Testing tasks

We distinguish six testing tasks.

1. Build the test harness.
2. Generate the test inputs.
3. Determine the expected outputs for each of the test inputs.
4. Execute the test cases, monitoring the behavior of the program.
5. Compare the actual outputs to the expected outputs.
6. Evaluate the test results, and decide whether the program is ready to be put into production.

Note that the last three steps incur costs every test run. As a result, to reduce the overall cost of testing, it is most advantageous to automate these steps as much as possible.

10.1.3 Overview

In Section 10.2, we discuss the purpose and contents of a TP and a TI. This is followed by a discussion of the distinction between system and module testing, and an explanation of why module testing is important. In the next section, we describe our method for selecting test inputs, based on functional testing. Section 10.5 describes the PGMGEN testing tool and illustrates it with the TIs of *stack* and *symtbl*. The design of a system has a major influence on the ease with which the modules in the system can be tested, and design for testability is the subject of Section 10.6. Sections 10.7 and 10.8 discuss the testing of the BSHAM and the ISHAM modules. Although module testing is important, we cannot ignore system testing; the SHAM system testing is discussed in Section 10.9. In Section 10.10, we review the work product criteria and the verification procedures for a TP and a TI. While we discuss the testing of every SHAM module, we do not present every TP and TI in full detail. The full work products may be found in Appendix F.

10.2 Work product definition

10.2.1 Test Plan

The TP for module M is intended for those considering running or modifying the testing of M. It serves as a specification for the TI for M: it describes the strategy used for selecting test cases and for executing these test cases.

A TP contains four required sections.

- **assumptions**: defines any assumptions, not contained in the MIS, on which the testing depends.
- **test environment**: describes the environment, such as test scaffolding, in which the testing is performed.
- **test case selection strategy**: describes how test cases are selected.
- **test implementation strategy**: describes the key aspects of the TI.

In addition, the **considerations** section is sometimes used for information that does not fit in any of the required sections.

10.2.2 Test Implementation

The TI for module M implements the TP for M as simply and inexpensively as possible. The TI includes the test scaffolding, such as drivers and stubs, the test data files, and the procedures, both manual and automated, required to execute the tests. The SHAM modules with a call-based interface contain at least an interactive or a batch driver, and often both. Typically, the interactive driver prompts the user for an access routine name and parameter values, invokes the access routine and, for a get call, displays the return value. The batch driver, on the other hand, typically contains a large number of test cases and it is automated so that it can be easily run after every change to the module.

Part of the interactive test driver for the *stack* module is shown in Figure 10.1. A constant is defined for each access routine, and `main` contains a loop that repeatedly prompts the user to select an access routine using the function `nextcall`. The case statement inside the loop prompts the user for the parameters, if any, of the access routine, and invokes the access routine. For example, the function `readint` prompts the user for an integer parameter. For a get call, the value returned by the call is printed. The interactive driver also provides access to the routine `g_dump`, which is implemented for debugging purposes and displays the concrete module state (see Section 9.2).

The interactive test drivers are convenient for executing small numbers of test cases and for debugging, where the behavior of one test case determines what other test cases are interesting. However, thorough and systematic module testing requires that large numbers of test cases are executed after every change to the module. Executing these with the interactive drivers would be a tedious

Figure 10.1: *stack* interactive driver—`main`

```
#define QUIT 0
#define S_INIT 1
#define S_PUSH 2
#define S_POP 3
#define G_TOP 4
#define G_DEPTH 5
#define G_DUMP 6

...

main()
{
     int reply,i;

     while ((reply=nextcall()) != QUIT) {
          switch(reply) {
          case S_INIT:
               ps_s_init();
               break;
          case S_PUSH:
               i = readint("Enter element:");
               ps_s_push(i);
               break;
          case S_POP:
               ps_s_pop();
               break;
          case G_TOP:
               i = ps_g_top();
               printf("returns %d\n",i);
               break;
          case G_DEPTH:
               i = ps_g_depth();
               printf("returns %d\n",i);
               break;
          case G_DUMP:
               ps_g_dump();
               break;
          }
     }
     return(0);
}
```

and error-prone task, and automation is desirable. We automate those aspects of testing that are most tedious and repetitive: the tedious aspects because they are typically easiest to automate, and the repetitive tasks because the payoff of automation will be highest for those tasks. We do not attempt to automate those steps where manual approaches are cost-effective, or steps where it is unclear how we can automate. In particular, we do not automate the selection of test inputs, but we do automate the execution of the test cases, and the comparison of actual with expected behavior.

To automate the testing, most of the modules in SHAM contain a batch driver. For many of the modules, we use the PGMGEN test-driver generation tool to generate the batch driver. For the other modules, we use customized batch drivers.

10.3 Module and system testing

Due to a lack of controllability and observability, it is hard to test a module M when it is installed in a production system. M's access routines are often not directly accessible. If M is a general-purpose module, some of its access routines may not be called at all in a particular production system. For example, if SHAM works as intended, none of the exception handlers will ever be called. Thus it is impossible to test the exception handlers using the production code of SHAM. To thoroughly test a module, we need to test it in isolation from its production environment.

While it is important to test each module in isolation, we also need to perform *integration* and *system testing*. With integration testing, we test combinations of modules that can be tested as single subsystems. Finally, we need to test the entire system. The extent of integration and system testing depends on the size of the system, the reliability requirements, and the amount of module testing that has been performed.

10.3.1 Top-down testing

Integration testing can be performed *top-down* or *bottom-up*. In *top-down* testing, we start by testing a top-level module using stubs, and gradually replace the stubs by production code. Although we can use stubs for some access routines of a module and production code for others, we typically use stubs for all or none of the access routines of a module. Therefore, in the following, when we refer to stubs for a module M, we mean stubs for all the access routines of M.

An example of top-down testing of BSHAM is shown in Table 10.1, where the t_is represent various stages in the progression of testing. We start by testing the *sham* coordinator using stubs for the other modules. After this we can proceed in several ways. In this case, we replaced the stubs for *load* and *token* by their MIs. We could have replaced only *load* by its MI, but it is hard to thoroughly

Table 10.1: Top-down testing of BSHAM

module	t_1	t_2	t_3	t_4
sham	MI	MI	MI	MI
load	stubs	MI	MI	MI
exec	stubs	stubs	stubs	MI
token	stubs	MI	MI	MI
absmach	stubs	stubs	MI	MI

Table 10.2: Bottom-up testing of BSHAM

module	t_1	t_2	t_3
sham	not used	not used	MI
load	not used	MI	MI
exec	not used	MI	MI
token	MI	MI	MI
absmach	MI	MI	MI

test *load* using stubs for *token*. At the next stage, we use the MI for *absmach*, and finally we replace the stubs for *exec* by the MI.

10.3.2 Bottom-up testing

In *bottom-up* testing, we first test low-level modules using test drivers, and gradually replace the drivers by higher-level modules. An example of how we could test BSHAM with bottom-up testing is shown in Table 10.2. First we test *token* and *absmach* using drivers. Note that we can test these in either order or even in parallel. We then test *load* and *exec* using drivers and the MIs for *token* and *absmach*. Again, *load* and *exec* can be tested in either order or in parallel. Finally, we test the *sham* Coordinator module using the MIs for all other modules.

10.3.3 Top-down versus bottom-up testing

One advantage of top-down testing is the early availability of an executable program for the end-user, so that he can give feedback as soon as testing is started. Another advantage is that integration testing occurs early in the testing, so that flaws in the interface design can be detected as early as possible. The major disadvantages of top-down testing are that observability and controllability are typically poor when testing the lower-level modules, and that the cost of developing and maintaining the stubs is high. Although simple stubs are easy to generate, they provide very little support for thorough testing. More sophisticated stubs are expensive to develop.

The advantage of bottom-up testing is that it provides better controllability and observability than top-down testing. A disadvantage of bottom-up testing is the cost of developing and maintaining the test drivers. Another disadvantage is that we cannot demonstrate an executable to the end-user until we have implemented all the lower-level modules.

In testing SHAM, we use a mixture of top-down and bottom-up testing. To improve controllability and observability we test the standalone modules using test drivers. For the other modules, we use a mixture of test scaffolding and production code, where the critical tradeoff is between the benefits realized through isolation, and the cost of developing and maintaining the test scaffolding.

10.4 Test case selection

We describe two methods for selecting test cases: *functional testing* and *structural testing*. With functional testing we base our tests primarily on the specification of the module. With structural testing we base our tests on the internal structure of the code implementing the module. Finally, we describe our approach for selecting test cases, which uses functional testing to select test cases and structural analysis as a cross-check on their adequacy.

10.4.1 Functional testing

Functional testing provides us with a systematic approach for choosing special values for test cases. Consider an access routine $f(p_1, ..., p_n)$. For each p_i, we choose a set S_i of special values, and test f on every tuple in $S_1 \times ... \times S_n$. The choice of special values is determined by the parameter types and, in some cases, the implementation of the access routine. The special values include both normal-case and exceptional values for each parameter. In some cases, there are dependencies between the sets. For example, a value for p_1 may be special only for certain p_2 values.

In addition to choosing special values for each parameter of each access routine, we also choose special values for the internal module state. However, we need to consider only normal-case values for the module state, because the exception-detection code should prevent exceptional values.

To illustrate the choice of special values, we present two heuristics. The *interval rule* applies to an integer parameter restricted to an interval $[L, U]$. For normal-case testing, at least three special values are chosen: the boundary points L and U, and at least one value interior to $[L, U]$. For exception testing, special values are chosen on the boundary and interior points of $(-\infty, L-1]$ and $[U+1, \infty)$. Thus, for an integer parameter restricted to $[1, 100]$, we might choose the following special values: $\{-1000, 0, 1, 50, 100, 101, 1000\}$. Sometimes we can apply the interval rule indirectly. For example, for the *stack* module, the stack size is restricted to the interval $[0, \texttt{PS\_MAXSIZ}]$, and thus we test it for an empty stack, a stack of size `PS_MAXSIZ`, and a stack with some, but less than `PS_MAXSIZ`, integers.

The second heuristic applies to a parameter with an enumerated type. If the number of elements in the enumerated type is small, we test the parameter for every value in the enumerated type. For example, for the *token* module, since there are only three types of tokens, we include test cases for all of these. If exhaustive testing is too costly, we divide the elements up into classes of "similar" ones and select at least one value from each class. For example, for an ASCII-character parameter we might include one alphabetic character, one digit, one punctuation mark, and one non-printable character. Similar heuristics can be applied to choose special values for other parameter types.

Now that we know how to choose special values for individual parameters, let us see how to combine these to come up with a set of test cases for access routines. Consider an access routine $f(p_1, p_2)$ with two parameters. Let us assume that p_1 is an integer restricted to the interval $[1, 100]$, and that p_2 belongs to the enumerated type $\{red, green, blue\}$. A suitable set of normal-case values for p_1 is $\{1, 50, 100\}$, and a suitable set of exceptional values is $\{-1000, 0, 101, 1000\}$. Since the enumerated type for p_2 contains only three values, we include all of these. There are no exceptional values for p_2, since we can use the compiler to ensure that f is always called with a value belonging to the enumerated type. To test f, we use all combinations of special values for both p_1 and p_2. Since p_1 has $3 + 4 = 7$ special values and p_2 has 3 special values, we should test f for all 21 combinations of these.

Thus, ignoring special values for the module state, a simple access routine such as the one above requires 21 test cases. This is a characteristic of our approach to functional testing: it leads to a large number of test cases. It thus appears that automated support is essential to perform functional testing. Fortunately, automation is feasible because the large number of test cases result from simple combinations.

10.4.2 Structural testing

With structural testing we select test cases based on the internal structure of the program. The motivation for structural testing is that we want to exercise or "cover" as many parts of the program as possible. We select our test cases so that a certain aspect of the source code is covered. We consider three types of coverage for structural testing: *statement*, *branch*, and *path coverage*.

The simplest form of coverage is *statement coverage*, where we select test cases so that every statement in the program is executed at least once. Consider the C function `tst` shown in Figure 10.2. To achieve statement coverage for `tst` a single test case suffices; for example, `tst`(2) will do. In the following, we abbreviate a set of test cases for `tst` by the set of parameter values. For example, the above test set is represented by the set $\{2\}$.

With *branch coverage*, we require that the set of test cases execute every branch in the program at least once. That is, every decision in the program has to evaluate to *true* and *false* at least once. For `tst`, we need at least two test

Figure 10.2: Implementation of `tst`

```
void tst(x)
int x;
{
     if (x > 0)
          pos = pos+1;
     if (x % 2 == 0)
          even = even+1;
     return;
}
```

Figure 10.3: Faulty implementation of `tst`

```
void tst(x)
int x;
{
     if (x >= 0)
          pos = pos+1;
     return;
}
```

cases to achieve branch coverage, for example, $\{-1, 2\}$. For -1 both decisions in `tst` evaluate to *false*, and for 2 both evaluate to *true*.

A path of control flow through a program is *feasible* if there exist values for the parameters of the program that exercise that path. With *path coverage*, we require that the set of test cases execute every feasible path through a program at least once. Since there are two decisions in `tst`, there are four paths of control flow through it, all of which are feasible. The test set $\{-2, -1, 1, 2\}$ achieves path coverage.

Unfortunately, path coverage is rarely practical. In programs with loops, the number of feasible paths is often infinite, and even when it is not, the number is typically so large that path coverage is impractical. Moreover, it is in general undecidable whether or not a path through a program is feasible. In practice, most programs contain many infeasible paths. To address these shortcomings, variations on path coverage have been proposed. However, all forms of structural coverage, including these proposals, suffer from the following weaknesses.

- *Structural coverage is not sufficient.* Many simple faults are not detected even by path testing, the most demanding coverage measure. For example, consider the faulty implementation of `tst` in Figure 10.3. There are two faults in this program: the first condition should be `x > 0`, and the second if-statement is omitted. Yet, the test set $\{-1, 1\}$ achieves path coverage, and the program behaves correctly for this test set. This indicates two reasons why structural coverage alone is not sufficient: it is not suited for detecting missing functionality such as the second if-statement, and it does not select special values that should be tested, such as 0 in the case of `tst`.

- *Insufficient automated support.* Although tools exist for measuring structural coverage, they can typically measure only statement and branch coverage. For example, the Unix utility *tcov* measures statement coverage. There are considerable problems with building a tool for measuring path coverage. First, since the number of paths is typically infinite or very large, there is the problem of presenting large volumes of data to the tester. Second, since there is no general way of detecting which paths are feasible, many of the paths will never be executed. Discovering which ones are feasible would have to be left to the tester. Third, there is the problem of displaying, in an understandable manner, an arbitrary path through a program.

10.4.3 Our approach

In the preceding sections we explained that

1. functional testing provides a systematic approach to test case selection that can be partially automated, and
2. structural testing provides little or no guidance with the selection of test cases, but statement coverage can easily be measured for a given set of test cases.

Therefore, our approach to testing uses functional testing for the selection of test cases and statement coverage analysis as a cross-check on their adequacy.

In particular, to select test cases for module M, we first consider M's MIS. Based on the MIS, we use functional testing to select special values for the access routine parameters and the module state. For the access routine parameters, we select normal-case and exceptional values; for the module state we select only normal-case values. We then consider M's MID and MI to see if there are any other special values, not suggested by the MIS, that we should test for. Finally, when executing the test cases, we use the Unix utility *tcov* to measure the statement coverage achieved by our test cases. The details of *tcov* are discussed in Section 10.10. We require that our test cases achieve 100% statement coverage. We view this 100% statement coverage as a necessary, but not a sufficient, condition for a test set. It is used as a simple check on the test case selection strategy, not as a goal in itself.

To illustrate our approach to test case selection, consider the test case selection strategy for the *stack* module, shown in Figure 10.4. Recall the *stack* MIS semantics from Figure 7.1. The contents of the stack are maintained as a sequence of integers. Since there is no reason to believe any value will be treated

Figure 10.4: *stack* Test Plan—**test case selection strategy**

test case selection strategy
  special values
    module state
      interval rule on size of stack: $[0, \texttt{PS\_MAXSIZ}]$
    access routine parameters
      none
  test cases
    For each of the special module state values,
      call `ps_s_push`, `ps_s_pop`, `ps_g_top`, `ps_g_depth`
      check exception behavior
      after set calls, check get call values

differently from any other value, there are no special values for the stack elements. We do apply the interval rule to the size of the stack; we test the module for an empty, a partially full, and a full stack. The only access routine parameter is to `s_push`, which is an integer. Again, it is reasonable to believe that all values will be treated the same, and so there are no special values for this parameter. For each special module state, we check both the exception behavior and, using get calls, the normal-case behavior.

10.5 Test driver generation

Although implementing test drivers manually is straightforward, it is also time-consuming, repetitive, and error-prone, and it produces code that is costly to maintain. As a result, test driver generation is a good candidate for automated support. For most modules with a call-based interface, we use the testing tool PGMGEN to generate batch test drivers from test scripts.

10.5.1 Test script language

A test case is described by providing a trace on a module and associating it with some aspect of the required behavior of the module in response to that trace. We represent a test case as a five-tuple

$$\langle trace, expexc, actval, expval, type \rangle$$

with the following interpretation.

trace: a trace used to exercise the module.

expexc: the name of the exception that *trace* is expected to generate (or `noexc` if no exception is expected).

Figure 10.5: *stack*—small test script

```
module
     ps_

accprogs
     <s_init,s_push,s_pop,g_top,g_depth>

exceptions
     <empty,full>

globcod
{%
#include "system.h"
#include "stack.h"
%}

cases
<s_init().g_top(), empty, dc, dc, dc>
<s_init().s_push(10), noexc, g_top(), 10, int>
```

actval: an expression (typically a get call) to be evaluated after *trace*, and whose value is taken to be the "actual value" of the trace.

expval: the value that *actval* is expected to have.

type: the data type of *actval* and *expval*.

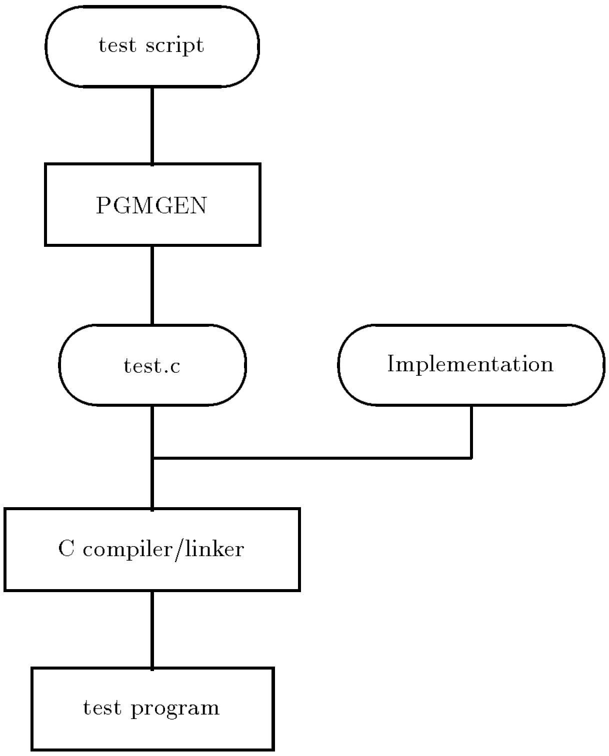
Below are two test cases for the *stack* module.

```
<s_init().g_top(), empty, dc, dc, dc>
<s_init().s_push(10), noexc, g_top(), 10, int>
```

In test cases developed solely to do exception checking, the *actval*, *expval*, and *type* fields contain `dc`, for don't care. The first trace initializes the module and calls `g_top`, which should signal the exception `empty`. The second trace pushes 10 onto the stack, and checks that `g_top` returns the correct value.

A complete test script for *stack* containing the above two test cases is shown in Figure 10.5. The `module` section defines the module prefix, which PGMGEN places in front of every access routine and exception name. The `accprogs` and `exceptions` sections define the list of access routines and the exceptions of the module. The `globcod` section contains global C code, delimited by the symbols `{%` and `%}`. PGMGEN places this global code at the top of the generated test driver. The test programmer can use the global code to define include files, stubs, and other functions that are called from the test cases. Finally, the `cases` section contains the test cases.

Figure 10.6: PGMGEN system flowchart



A test script may be viewed as a partial specification for a module, expressing its required behavior under specific circumstances. The purpose of PGMGEN is to generate a driver that will determine whether an MI satisfies this partial specification.

10.5.2 Test program generation

The system flow for PGMGEN is shown in Figure 10.6— ovals indicate human readable files and boxes indicate executable programs. The test programmer prepares the script using a text editor. PGMGEN reads that script and generates the C driver `test.c` which is compiled and linked with the MI. For example, for *stack*, the test script is stored in the file `stack.script`, the MI in `stack.c`, and the executable test program that is generated is called `stack_b`. When `stack_b` is executed, it runs the test cases from `stack.script` and reports any errors.

To generate the test driver, PGMGEN first generates code to record exception occurrences. Then, for each test case of the form

Figure 10.7: Steps performed for a PGMGEN test case

```
invoke c_1, ···, c_n, monitoring exception occurrences
compare the actual occurrences to expexc
if there are any differences
      print a message
else
      if actval ≠ expval
            print a message
      if any exceptions have occurred since c_n was invoked
            print a message
update summary statistics
```

$$\langle c_1 . \cdots . c_n, expexc, actval, expval, type \rangle$$

PGMGEN generates code that performs the steps outlined in Figure 10.7. Following the last case, code is generated to print summary statistics.

10.5.3 Example—*stack*

The production TP for the *stack* module is shown in Figure 10.8. The test environment section describes the test scaffolding used for testing *stack*. The test case selection strategy is explained in Section 10.4. The test implementation strategy section defines the key aspects of the TI: the C function `load`(n) is used to load the stack with the values $10, 20, \ldots, 10 \times n$, and *tcov* (discussed in Section 10.10) is used to measure statement coverage. Note that for the function `load` the values themselves are not important, but they should be unique and easy to generate.

The `globcod` and `cases` sections of the test script are shown in Figure 10.9; the `module`, `accprogs`, and `exceptions` sections are the same as in Figure 10.5. The `globcod` section defines the function `load`. Note that the `globcod` section is copied unchanged to the test driver by PGMGEN. This means that explicit module prefixes are required for the access routine calls in this section, such as `s_init` and `s_push` in Figure 10.9. In the `cases` section, we separate the cases for the three special module states. For each special module state, we check the return values of `g_top` and `g_depth` in that state, and after calls to `s_push` and `s_pop`. However, `s_pop` signals an exception for the empty stack, and `s_push` signals an exception for the full stack, and thus we do not check the return values of the get calls for these cases. For the partially full stack, there are test cases with a stack size of 3, and the expected behavior for these cases indicates that no exception should be signaled. This happens only if `PS_MAXSIZ` > 2, and hence the assumption in the TP.

The entire script for stack is 48 lines long and contains 16 test cases. The test driver generated by PGMGEN from this script is 453 lines: almost 10 times

Figure 10.8: *stack* Test Plan

assumptions
  `PS_MAXSIZ` > 2

test environment
  PGMGEN driver
  no stubs

test case selection strategy
  special values
    module state
      interval rule on size of stack: $[0,$ `PS_MAXSIZ`$]$
    access routine parameters
      none
  test cases
    for each of the special module state values,
      call `ps_s_push`, `ps_s_pop`, `ps_g_top`, `ps_g_depth`
      check exception behavior
      after set calls, check get call values

test implementation strategy
  `load`(n)
    loads stack with $10, 20, \ldots, 10 \times n$
  statement coverage measured using the Unix utility *tcov*

Figure 10.9: *stack* Test Implementation

```
globcod
{%
#include "system.h"
#include "stack.h"

static void load(n)
int n;
{
     int i;

     ps_s_init();
     for (i = 0; i < n; i++)
          ps_s_push((i+1)*10);
}
%}

cases

/*empty stack*/
<load(0).s_push(10), noexc, g_top(), 10, int>
<load(0).s_push(10), noexc, g_depth(), 1, int>
<load(0).s_pop(), empty, dc, dc, dc>
<load(0).g_top(), empty, dc, dc, dc>
<load(0), noexc, g_depth(), 0, int>

/*partially full stack*/
<load(2).s_push(30), noexc, g_top(), 30, int>
<load(2).s_push(30), noexc, g_depth(), 3, int>
<load(2).s_pop(), noexc, g_top(), 10, int>
<load(2).s_pop(), noexc, g_depth(), 1, int>
<load(2), noexc, g_top(), 20, int>
<load(2), noexc, g_depth(), 2, int>

/*full stack*/
<load(PS_MAXSIZ).s_push(0), full, dc, dc, dc>
<load(PS_MAXSIZ).s_pop(), noexc, g_top(), (PS_MAXSIZ-1)*10, int>
<load(PS_MAXSIZ).s_pop(), noexc, g_depth(), PS_MAXSIZ-1, int>
<load(PS_MAXSIZ), noexc, g_top(), PS_MAXSIZ*10, int>
<load(PS_MAXSIZ), noexc, g_depth(), PS_MAXSIZ, int>
```

Figure 10.10: *stack*—output produced by test script

```
Statistics:
        Number of test cases: 16
        Number correct: 16
        Percentage correct: 100.00
        Number of exception errors: 0
        Number of value errors: 0
```

the size of the script. When the driver is compiled and linked with a correct MI, it produces the output shown in Figure 10.10.

10.5.4 Embedded C code

In developing PGMGEN, our goal was to provide a test language powerful enough to describe the test cases we encountered in practice, but which was as cost-effective as possible. In particular, we wanted to minimize the training time for the test programmer, the cost of implementing and maintaining PGMGEN, and the cost of changing its target language and operating system. Therefore, we have allowed code written in C to be embedded freely in test scripts. As a result, there is no need in the script language for functions, macros, or iteration constructs—these are available in C and are presumably understood by the test programmer.

Besides the global C code in the `globcod` section, C code delimited by {% and %} may also be inserted in the following places in the `cases` section: between test cases, as a call in the trace of a test case, as the *actval* or *expval* field of a test case, and as a parameter of any call. As for the global C code, this code is copied unchanged to the test driver by PGMGEN.

Consider the *stack* test case

```
<s_init().s_push(10), noexc, g_top(), 10, int>
```

Suppose we want to test `s_push` and `g_top` not only for 10, but for all values in $\{10, 20, \ldots, 100\}$. We can write 10 test cases, but we can also embed the test case inside a for-loop, as in

```
{% for (i = 1; i <= 10; i++) %}
     <s_init().s_push(10*i), noexc, g_top(), 10*i, int>
```

The variable `i` is used both in the parameter to `s_push` and in the *expval* field of the test case. For this, `i` needs to be declared somewhere in the test script, e.g., in the `globcod` section. The code generated by a test case is a single C statement. Thus, to include two test cases inside a for-loop, we need to enclose them by { and }, as in

```
{% for (i = 1; i <= 10; i++) { %}
     <s_init().s_push(10*i), noexc, g_top(), 10*i, int>
     <s_init().s_push(10*i), noexc, g_depth(), 1, int>
{% } %}
```

10.5.5 Example—*symtbl*

To illustrate the use of embedded code, we consider the testing for *symtbl*. Part of the *symtbl* TP is shown in Figure 10.11. The MIS semantics for *symtbl* is shown in Figure 7.3. The abstract state is a set of symbol/location pairs. There are two characteristics of this state for which we choose special values: the size of the table, and the length of the symbols in the table. For the table size, we apply the interval rule to $[0, \texttt{ST\_MAXSYMS}]$. In this case, we select two interior points, 1 and $\texttt{ST\_MAXSYMS}/2$, because we feel that a table size of 1 is different enough from the other table sizes to include it as a special case. Note that this does not violate the interval rule, which states that *at least* one point from the interior of an interval should be selected. For the length of the symbols in the table, we apply the interval rule to $[0, \texttt{ST\_MAXSYMLEN}]$. Since there is only one symbol of length 0, we test it separately and for the special module states we choose tables with short symbols and symbols of length `ST_MAXSYMLEN`. The special values for access routine parameters are overlength symbols for `s_add`, and the empty string for the access routines that take a symbol as parameter. The test cases that are executed for each module state have been divided up into those test cases that should signal an exception and the normal-case test cases. For simplicity, we have shown only the normal-case test cases in Figure 10.11.

To loop over the special module states, we define C functions that iterate over the sequence of special module states using one of the sequence idioms discussed in Section 7.3. Although we are not designing a module interface, the idioms are still helpful. In this case, the function `t_init` initializes the sequence, `t_next` advances to the next special module state, and `t_end` indicates if the end of the sequence has been reached. Each special module state is characterized by n, the table size, and l, the length of the symbols in the table. There are also several functions that return information about the current state in the sequence: `t_siz` returns n, $\texttt{t\_sym}(i)$ the i-th symbol in the table, and $\texttt{t\_loc}(i)$ the location value of the i-th symbol.

Symbol values and their locations are unimportant, as long as they are unique and easy to generate. To define the symbol values, we use the function $\texttt{t\_mksym}(i, l)$, whose value is i in string form padded right with `'*'` characters to length l (or zero `'*'` characters if i has l or more digits). For a given n and l, the special module state is a table with the symbols $\texttt{t\_mksym}(i, l)$, for $i \in [0, n-1]$. For the location, we use a simple arithmetic formula that determines a unique location for each symbol.

We can now explain the assumption:

$$\texttt{ST\_MAXSYMLEN} \geq \text{length of } \texttt{ST\_MAXSYMS} - 1 \text{ in string form}$$

Figure 10.11: *symtbl* Test Plan—normal case test cases

assumptions

    ST_MAXSYMLEN $\geq$ length of ST_MAXSYMS $-$ 1 in string form
    ST_MAXSYMS > 0

...

test case selection strategy

    special values
        module state
            number of symbols in table: $\{0, 1,$ ST_MAXSYMS$/2,$ ST_MAXSYMS$\}$
            symbol length: short, ST_MAXSYMLEN
        access routine parameters
            st_s_add: strings of length $\{0,$ ST_MAXSYMLEN $+ 1, 2 *$ ST_MAXSYMLEN$\}$
            st_s_add, st_s_loc, st_g_loc, st_g_exsym: empty string
    test cases
        exceptions
            ...
        normal
            check st_g_exsym for empty string in empty table
            add the empty string, check and change its location
            for each special module state
                check table length
                check that a very long symbol is not in table
                for each i in [0, ST_MAXSYMS $-$ 1]
                    if i in [0, t_siz $-$ 1]
                        check t_sym(i) in table with correct location
                        check st_s_loc resets location
                    else
                        check t_sym(i) not in table

test implementation strategy

    C functions to support iterating over the special module states,
    viewed as a sequence:
        void t_init: initialize to the first state
        void t_next: load next state
        int t_end: return *true* if no states remain
    C functions to generate and check symbols in current state:
        int t_siz: number of symbols in current state
        char *t_sym(i): i-th symbol in current state
        int t_loc(i): location of i-th symbol in current state
        char *t_mksym(i, l): string consisting of i converted to ASCII,
            padded right with *'s to length l
    statement coverage measured using the Unix utility *tcov*

The maximum number of elements stored in the table is `ST_MAXSYMS`, and therefore the longest string stored in the table will be `ST_MAXSYMS` $-$ 1 in string form, possibly padded with `'*'` characters. Thus, if the assumption was violated, one of the special module states would contain a string with more than `ST_MAXSYMLEN` characters. However, this string could not be added to the module state, because `s_add` would signal the exception `maxlen` for it.

The normal-case test cases from the *symtbl* test script are shown in Figure 10.12. We have used indenting and comments taken from the TP to make the test script more readable. The embedded code reduces the size of the test script considerably: although the entire test script contains only 18 test cases, it produces a driver that executes 868 test cases.

10.5.6 Comparing actual and expected value

Consider the test case

$$\langle trace, expexc, actval, expval, type \rangle$$

As Figure 10.7 shows, after the calls in *trace* are executed, *actval* and *expval* are compared. To compare them, PGMGEN generates a call to the boolean C function whose name is `cmp_` followed by *type*. The arguments to this function are *actval* and *expval*. For example, for the *stack* test case

```
<s_init().s_push(10), noexc, g_top(), 10, int>
```

PGMGEN generates the call `cmp_int(ps_g_top(), 10)` (note that PGMGEN places the prefix `ps_` in front of the call to `g_top`). This C function should return *true* when $actval = expval$, and *false* otherwise.

When the above function returns *false*, PGMGEN generates a call to the C function whose name is `prt_` followed by *type*, again with *actval* and *expval* as its arguments. The purpose of this function is to print a message indicating that $actval \neq expval$ and displaying both values.

PGMGEN provides the `cmp_` and `prt_` functions for the data types `bool` (*boolean*), `char`, `float`, `int`, and `string`. For example, the implementation of `cmp_bool` and `prt_bool` are shown in Figure 10.13.

For other data types, the tester must define these functions, typically in the `globcod` section of the test script. These functions can also be used to define customized comparison and printing functions for standard data types. For example, the function `cmp_float` provided by PGMGEN uses an exact comparison to compare two floating-point numbers. For certain applications it is impossible to define the actual value of a floating-point number with such accuracy. For these cases the tester could define the functions `cmp_fuzz` and `prt_fuzz` as shown in Figure 10.14. `cmp_fuzz` considers two floating-point numbers equal if they are within `epsilon` of each other. Note that `prt_fuzz` also prints out the value of `epsilon`. To use these functions in a test case, the *type* field of the test case should be defined as `fuzz`.

Figure 10.12: *symtbl* Test Implementation—normal case test cases

```
/*check g_exsym for empty string in empty table*/
<s_init(), noexc, g_exsym(""), 0, bool>
/*add the empty string, check and change its location*/
<s_init().s_add("",10), noexc, g_exsym(""), 1, bool>
< , noexc, g_loc(""), 10, int>
<s_loc("",20), noexc, g_loc(""), 20, int>

{%
t_init();
t_next();
while (!t_end()) {
%}
     /*check table length*/
     < , noexc, g_siz(), t_siz(), int>
     /*check that a very long symbol is not in table*/
     < , noexc, g_exsym(t_mksym(0,2*ST_MAXSYMLEN)), 0, bool>
{%
     for (i = 0; i < ST_MAXSYMS; i++) {
          if (i < t_siz()) {
%}
               /*check t_sym(i) in table with correct location*/
               < , noexc, g_exsym(t_sym(i)), 1, bool>
               < , noexc, g_loc(t_sym(i)), t_loc(i), int>
               /*check s_loc resets location*/
               <s_loc(t_sym(i),t_loc(-i)), noexc,
                       g_loc(t_sym(i)), t_loc(-i), int>
{%
          } else {
%}
               /*check t_sym(i) not in table*/
               < , noexc, g_exsym(t_sym(i)), 0, bool>
{%
          }
     }
     t_next();
}
%}
```

Figure 10.13: `cmp_bool` and `prt_bool`

```
int cmp_bool(aval,eval)
int aval,eval;
{
     if ((eval == 0) && (aval == 0)) return(1);
     else if ((eval != 0) && (aval != 0)) return(1);
     else return(0);
}

int prt_bool(aval,eval)
int aval,eval;
{
     printf("\tExpected value:%d.  Actual value:%d\n",eval,aval);
}
```

Figure 10.14: `cmp_fuzz` and `prt_fuzz`

```
float epsilon = 0.00001;

int cmp_fuzz(aval,eval)
float aval,eval;
{
     float diff;

     diff = eval-aval;
     if (diff > epsilon || diff < -epsilon) return(0);
     else return(1);
}

int prt_fuzz(aval,eval)
float aval,eval;
{
     printf("\texpval:%f.  actval:%f.  epsilon:%f\n",eval,aval,epsilon);
}
```

10.6 Design for testability

Both the module decomposition and the module interface design influence the testability of a module. To make testing affordable and effective, we need to consider testability of a module at design time, before the implementation is started. In this section, we review the key principles in design for testability, *controllability* and *observability*. While the need for controllability and observability is well-known in hardware testing, it is often ignored in software testing. Yet it is critical when software interacts with the environment, as is the case with most software systems. Poor controllability and observability also make it difficult to automate the testing, which is necessary to make testing affordable.

A common example of poor controllability in software is the lack of explicit initialization. Consider the three test cases

```
<s_init().s_push(10).s_push(20), noexc, g_top(), 20, int>
<s_init().s_push(10).s_push(20), noexc, g_depth(), 2, int>
<s_init().s_push(10).s_pop, noexc, g_depth(), 0, int>
```

for the *stack* module. Without the access routine `s_init`, these would have to be changed to

```
<s_push(10).s_push(20), noexc, g_top(), 20, int>
< , noexc, g_depth(), 2, int>
<s_pop.s_pop, noexc, g_depth(), 0, int>
```

where we rely on the fact that the stack is empty when the first test case is reached. Without `s_init`, a large number of changes would have to be made to the *stack* test script shown in Figure 10.9. The need for explicit initialization is recognized in hardware testing, where integrated circuits frequently contain reset circuitry.

A common example of poor observability in software is the undisciplined use of print statements. Exceptions are often signaled by printing a message, which makes it hard to automate the testing of exceptions. For example, it is not clear how we could use PGMGEN to test the *stack* module if exceptions were signaled by a call to `printf` rather than a call to an exception handler.

Another cause for poor controllability or observability is external interaction. While the lack of explicit initialization and the uncontrolled use of print statements can be avoided, external interaction cannot be completely eliminated. Every system will have modules that interact with other modules. To improve the testability of such a module, we isolate it where possible and affordable, and we limit the amount of interaction by careful interface design. Most systems also interact with the environment, for example, by reading input from the keyboard or by printing output to the screen. To improve the testability of SHAM modules, we have isolated the interaction with the environment in a small number of modules, so that the other modules can be tested without controllability and observability problems. Examples are discussed in the next two sections.

Figure 10.15: *token* Test Implementation—`cmp_valtyp` and `prt_valtyp`

```
int cmp_valtyp(actvtp,expvtp)
tk_valtyp *actvtp,*expvtp;
{
     if (!strcmp(actvtp->val,expvtp->val))
          return(actvtp->typ == expvtp->typ);
     else
          return(0);
}

void prt_valtyp(actvtp,expvtp)
tk_valtyp *actvtp,*expvtp;
{
     printf("Expected value:<%s,%d>. Actual value:<%s,%d>\n",
             expvtp->val,expvtp->typ,actvtp->val,actvtp->typ);
}
```

10.7 BSHAM Test Plans and Implementations

10.7.1 The *token* TP and TI

Since *token* does not have any controllability and observability problems we test it with PGMGEN. Recall that the abstract state for *token* (Figure 7.5) is a sequence of tokens. In this case, the special module state values we select are determined by the number of tokens in the sequence and their types. There is a minimum but no maximum number of tokens defined, and we apply the interval rule to $[0, \infty)$: we test for states with 0, 1, and 3 tokens. Since there are only three token types, we include test cases for all of these. Every token type, except for `TK_BADTOK`, has a maximum length associated with it, and we apply the interval rule to the length of the tokens. For `TK_BADTOK`, we include test cases for tokens that are "almost" of some other type. For example, we include a token consisting of `TK_MAXINTLEN` $+ 1$ digits, which would be an integer, except that it is too long. The only access routine for which we include special parameter values is `s_str`, for which we apply the interval rule to the length of the string and vary the number of blanks before and after tokens in the string.

The access routine `sg_next` returns a value of type `tk_valtyp`, a structure containing the value of the token and its type. To compare expected and actual values of this type, we must define `cmp_` and `prt_` functions as discussed in Section 10.5.6. We define the functions `cmp_valtyp` and `prt_valtyp` shown in Figure 10.15 in the `globcod` section of the test script. Both functions take two pointers to `tk_valtyp` as their arguments. To compare the two values, `cmp_valtyp` compares both the value of the token and its type. Similarly, `prt_valtyp` prints out the values and types of both its parameters.

Figure 10.16: *load* Test Plan—**test environment**

test environment
sham Coordinator used as driver
stubs for *absmach* and *exec*, production code for *sham* and *token*
input stored in files
output saved in files, checked with delta testing
directory structure:
`load/`
`input/` - test cases stored one per file
`exp/` - expected results of test case (same file name)
`act/` - actual results of test case (same file name)

10.7.2 The *absmach* TP and TI

absmach is also tested using PGMGEN. The abstract state for *absmach* (Figure 7.8) consists of the accumulator, the program counter, and the memory. The special module state values that we select are determined by the effect of these three on the behavior of `sg_exec`. In particular, we include state values so that every SHAM run-time exception occurs at least once and every SHAM instruction is executed at least once. For each instruction, we check the effect it has on the accumulator and the program counter. For instructions that also alter memory contents, we check that the change is made correctly. For the access routine parameters, we apply the interval rule to the parameters of `s_acc`, `s_pc`, `s_mem`, and `g_mem`.

10.7.3 The *load* TP and TI

The *load* MIS semantics is shown in Figure 7.10. *load* interacts with both the environment and other modules. It reads input, produces output, and calls access routines from *token* and *absmach*. Before we select test cases, we must decide on the test environment we are going to use: how to provide input, how to check the output, and whether to use stubs or production code for *token* and *absmach*.

The test environment section of the *load* TP is shown in Figure 10.16. Since *load* reads input from a file, it is hard to test it with PGMGEN, which has no facility to deal with access routines that read input. Therefore we must either build a customized driver to test *load*, or use the *sham* Coordinator. We use the *sham* Coordinator, because it is simple, it provides good controllability, and it saves us implementing a customized driver. Note that although we use the *sham* Coordinator, we are not interested in testing it at this point.

load uses *token* to retrieve the tokens from the input. To thoroughly test *load* with stubs for *token* would require stubs almost as complicated as the production code for *token*. In such a case, the production code is the clear choice, since it avoids the need to maintain the complicated stubs. On the other hand, *load* uses *absmach* only to load the memory, and the production code for *absmach* is non-trivial. We therefore use stubs for *absmach*. Finally, because we use the production *sham* Coordinator, we must decide whether or not to use stubs for *exec*. In this case the choice is easy: simple stubs suffice and the production code is complicated. In summary, we use stubs for *absmach* and *exec*, and production code for *sham* and *token*.

Since we use the *sham* Coordinator as a test driver, we can store the test inputs in several files. For each input file, we maintain a file that contains the expected output for that input. To organize the files, we use three directories:

- `input` contains the input files.
- `exp` contains the expected output files, one for each input file.
- `act` is used to store the actual output files for a test run, again one for each input file.

The output produced by *load* is non-trivial to check. We automate the checking using *delta testing*, where the expected output for a test run is the output from an earlier test run. If there are any differences between the actual and the expected output, the tester must verify which of the two is the correct output, and modify either the expected output or the program. A critical part of delta testing is the creation of the initial expected output. This can be done manually or using the initial version of the production code. In the latter case, the output should be checked carefully, because if it contains erroneous data, this may mask program errors in future test runs.

The test case selection strategy of the TP for *load* is shown in Figure 10.17. Note that *load* has no internal state and the only access routine parameter is the file pointer passed to `sg_load`. The special values for this parameter are determined by the contents of the file. We include test cases so that each load-time exception is exercised at least once, and so that every instruction is loaded at least once. For instructions that take an operand, we apply the interval rule to the operand (either an address or a SHAM integer). Finally, we include a test case that completely fills up the available memory.

Test cases are stored in several files, with names chosen to distinguish files that should generate load-time exceptions and those that should not. For example, Figure 10.18 contains the first three lines of the file `instr`, which should not generate a load-time exception.

Part of the TI consists of the implementation of the stubs for *absmach* and *exec*. We use simple stubs that store the values loaded into memory in an array and that print out the contents of this array after the load phase is completed successfully. These stubs (1) provide adequate observability, and (2) can easily combine the output checking for the stubs with the output checking for *load*. To implement this scheme, the stubs for *absmach* maintain an array `mem` of `AM_MEMSIZ` characters. The stub for `am_s_mem` stores values in `mem` and the stub

Figure 10.17: *load* Test Plan—**test case selection strategy**

test case selection strategy
- special values
  - module state
    - none
  - access routine parameters
    - input file for `ld_sg_load`:
      - every load-time exception for every instruction
      - every SHAM instruction at least once
        - interval rule for instructions with an operand
      - completely fill up memory
- test cases
  - load-time exceptions
    - `ldexc1`: all load-time exceptions except *NOMEMEXC*
    - `ldexc2`: *NOMEMEXC*
  - normal case
    - `instr`: every SHAM instruction
    - `fill`: completely fill up memory

Figure 10.18: *load* Test Implementation—part of test case

```
load 0
load 50
load 99
```

Figure 10.19: *load* Test Implementation—stubs for `am_s_mem` and `am_g_mem`

```
void am_s_mem(a,i)
int a,i;
{
     mem[a] = i;
}

int am_g_mem(a)
int a;
{
     return(mem[a]);
}
```

Figure 10.20: *load* Test Implementation—stub for `ex_s_exec`

```
void ex_s_exec()
{
     int i;

     for (i = 0; i < AM_MEMSIZ; i++) {
          printf("%4d",am_g_mem(i));
          if (i % 10 == 9)
               printf("\n");
     }
}
```

for `am_g_mem` retrieves values from `mem`. Both stubs are shown in Figure 10.19. Note that the stubs are much simpler than the production implementations because there is no need to perform exception checking. Figure 10.20 shows the stub for `ex_s_exec`, which makes calls to the stub for `am_g_mem` to print out the memory contents as a ten by ten array. For example, Figure 10.21 shows the output that is produced for the test file `instr`. Note that the first six memory locations correspond to the object code for the first three lines of `instr` shown in Figure 10.18.

The Unix commands shown in Figure 10.22 are used to perform a test run. For each file f in the directory `input`, BSHAM is run on f, output is redirected to `act`/f, and `act`/f is compared to `exp`/f with the Unix utility *diff*.

10.7.4 The *exec* TP and TI

Figure 7.12 shows the MIS semantics for *exec*. The BSHAM version of *exec* calls access routines from *absmach*. The ISHAM version produces output and calls

Figure 10.21: *load* Test Implementation—expected output

```
0   0   0  50   0  99   1   0   1  50
1  99   2   0   2  50   2  99   3   0
3  50   3  99   4   0   4  50   4  99
5   0   5  50   5  99   6   0   6  50
6  99   7   0   7 500   7 999   8   9
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0
```

Figure 10.22: *load* Test Implementation—shell commands

```
foreach f (input\*)
     echo input file: $f
     bsham $f >act/$f
     diff act/$f exp/$f
end
```

access routines from *absmach*, *scndr*, and *keybdin*. For both versions, since it is hard to write a stub for `am_s_exec` that provides good controllability, we use the production code of *absmach*. Similarly, for the ISHAM version, stubs for *keybdin* would provide poor controllability, and stubs for *scndr* would provide poor observability. We therefore use the production code for both these modules. To manually generate the sequence of calls that loads the memory in *absmach* would be quite tedious, and so we use the production code for *load* and *token* to load the memory from a file. Finally, since we use the production code for *load* and *token*, it becomes hard to use PGMGEN, and hence we use the *sham* Coordinator as a test driver.

In summary, to test *exec* we prefer to use the production code for all of SHAM. We therefore incorporate the testing of *exec* with the system testing; see Section 10.9. Note that this is not the only possibility, and that this decision influences not only the testing of *exec*, but also the system testing. During system testing, we now need to thoroughly exercise *exec*. For example, we should achieve 100% statement coverage for *exec*. As an alternative, we could provide separate testing for *exec*, in which case the system testing would be simpler.

10.7.5 The *sham* TP and TI

Although we can test *sham*, the SHAM Coordinator module, as a separate module, it is so simple that it is not worth it. Therefore, just as for *exec*, we incorporate the testing of *sham* with the system testing.

10.8 ISHAM Test Plans and Implementations

10.8.1 The *keybdin* TP and TI

Figure 7.15 shows the MIS semantics for *keybdin*. Since *keybdin* reads input from the keyboard, it has controllability problems and cannot be tested with PGMGEN. To automate the testing requires a means for storing and replaying keystroke sequences. Although tools exist that provide these services, we can adequately test *keybdin* with a simple interactive driver. The interactive driver we use repeatedly waits for input. It displays the characters that are entered on the screen, and terminates when the character `'q'` is entered.

Since there is no reason to believe *keybdin* treats any character differently from any other character, there are no special values to test for. We therefore test *keybdin* for just a few characters. Since we have to select a few characters, we use the fact that, in our case, the intended use for *keybdin* is to read the ISHAM commands from the keyboard. Therefore we include at least both ISHAM commands and one character that is not an ISHAM command in the test cases. Since we use an interactive driver to test *keybdin*, it it the tester's responsibility to run the test cases that are described in the TP. Similarly, it is his responsibility to check that the behavior of the implementation is correct for those inputs.

10.8.2 The *scngeom* TP and TI

scngeom provides run-time access to the layout of the screen, and it has no abstract state (Figures 7.11 and 7.17). The service provided by *scngeom* is straightforward. If there were any faults in the MI, these most likely would either cause obvious errors, such as a missing or cut off screen field, or cause errors with minor impact, such as a field that is not displayed in its correct position on the screen. Moreover, the normal-case behavior of *scngeom* is easily and thoroughly tested in the testing of the *scndr* module; see Section 10.8.4. The testing of *scngeom* therefore contains only test cases for exceptions, which cannot be tested from the *scndr* module. Since the exception checking for `g_row`, `g_col`, and `g_len` is the same, we further simplify the testing by testing only `g_row` thoroughly. To test `g_row`, we choose exceptional parameter values for all field names and for illegal row and column values determined by the interval rule. For `g_col` and `g_len`, we include one test case to ensure that the MI does at least some exception checking.

10.8.3 The *scnstr* TP and TI

The *scnstr* MIS semantics is shown in Figures 7.19 and 7.20. *scnstr* provides access routines that write a string to any screen position. Since the exception testing for *scnstr* does not pose any observability problems, we perform the exception tests with a PGMGEN script. The normal-case testing, on the other hand, poses observability problems because *scnstr* updates the terminal screen. Automating the testing of *scnstr* requires access to the screen contents. To provide such access with software is a complex task, and instead we test *scnstr* with a customized driver and check the output manually. To simplify the checking of the output, the customized driver displays patterns on the screen that are chosen to exercise special values and to be quickly recognizable as correct or not.

To select special values, note that only the position where strings are displayed are important, not the actual content of the string. We apply the interval rule to the row and column number for both the exception and normal-case testing. For the normal-case testing, we use a unique string for each special position on the screen.

The implementation for the exception testing is straightforward. The driver for the normal-case testing displays the unique strings in the special positions on the screen. To allow the tester time to verify the screen contents, the driver then waits for the tester to press a key. Once the tester has verified the screen contents, he can press any key, and the test driver terminates.

10.8.4 The *scndr* TP and TI

The *scndr* MIS semantics is shown in Figure 7.22. The purpose of *scndr* is to update the screen contents according to the information stored in *absmach*. Since it updates the screen contents, it has observability problems, and we test it with a customized driver.

scndr calls three other modules: *scngeom*, *scnstr*, and *absmach*. Since we decided to test the normal-case behavior of *scngeom* while testing *scndr* (see Section 10.8.2), we must use the production code for *scngeom*. This also means that it is not enough to check the values of the screen fields, the tester must also check their position on the screen. We could verify this by checking that the access routine `ss_s_str` of *scnstr* is called by *scndr* with the correct parameters. However, in this case, it is easy enough to verify this visually, and hence we use the production code for *scnstr*.

To verify that *scndr* correctly displays the information in *absmach*, we provide stubs for `am_g_mem`, `am_g_pc`, and `am_g_acc`, that return a unique value for each memory cell, the program counter, and the accumulator. For example, the stub for `am_g_mem` returns $10 \times a$ as the content of memory address a.

After initializing various modules, the test driver calls `sd_s_mem` to update the screen contents according to the values returned by the stubs for *absmach*. The driver then waits for the tester to press a key, so that he can verify the screen contents.

10.9 System testing

With system testing, we use production code for the entire system and, if possible, we test the system in its production environment. The most important goal in system testing is to test whether or not a system satisfies its RS. This includes testing performance and storage requirements, if these are part of the RS. Depending on the size and complexity of the system, system testing may also include some or all of the following testing techniques.

Volume and stress testing: to test whether the system can handle large volumes of data and heavy loads.

Reliability testing: to test how reliable a system is. For example, to test whether a system meets a certain mean-time-to-failure requirement.

Recovery testing: to test how well the system can recover from errors such as hardware failures or data errors.

Acceptance testing: to test whether the system meets the needs of the end user. It is typically performed by the end user.

In the following, we will only test whether BSHAM and ISHAM meet their respective RS.

10.9.1 BSHAM

In system testing, it is important to focus the testing on the parts of the system that were not tested during module testing. We have tested every SHAM module separately, except for *exec* and the *sham* Coordinator module. Since *sham* is so simple, the test case selection for system testing is mostly guided by the need to thoroughly exercise the *exec* module.

The **test environment** and **test case selection strategy** sections for the system TP of BSHAM are shown in Figure 10.23. In the testing of BSHAM, we distinguish test cases with command-line errors and test cases that read input from a file. Since BSHAM produces output for all the test cases, we store this output in a file so that we can use delta testing to automate the checking of the output. The expected output for the test cases with command-line errors is stored in the file `cmdlin.exp`, and the actual output is stored in the file `cmdlin.act`. For the other test cases, we use the same directory structure as for testing *load*: the directory `input` contains the input files, `exp` the expected output files, and `act` is used to store the actual output files for a test run.

Since the BSHAM system has no module state and no access routines, we cannot use these to select special values for test cases. In BSHAM, the special values that we want to test are determined by the command-line arguments and the contents of the input files. We test every command-line error once. For the other test cases, we distinguish three types: test cases that produce load-time exceptions, those that produce run-time exceptions, and those that do not

Figure 10.23: BSHAM Test Plan—**test environment** and **test case selection strategy**

test environment

entire BSHAM system
input stored in files
output saved in files, checked with delta testing
directory structure:
`sham/`
`input/` - test cases stored one per file
`exp/` - expected results of test case (same file name)
`act/` - actual results of test case (same file name)
`sham/cmdlin.exp` - expected results for command-line test cases
`sham/cmdlin.act` - actual results for command-line test cases

test case selection strategy

special values
command-line errors
each command-line error once
content of input file
one load-time exception
every run-time exception once
SHAM instructions
halt with pc $= \{0, \mathtt{SY\_MEMSIZ}/2, \mathtt{SY\_MEMSIZ} - 1\}$
print with interval rule on content accumulator
test cases
command-line errors
hardcoded in Makefile
load-time exceptions
`ldexc`: one load-time exception
run-time exceptions
`addrexc`: $mem[pc] = ADD.object$, $mem[pc+1] = 100$
`arithexc`: $acc = 500 + 500$
`noopexc`: $pc = 99$, $mem[pc] = LOADCON.object$
`objectexc`: $mem[pc] = 10$
normal-case
`halt[1-3]`: *HALT* instruction, check with *PRINT* instruction
`print`: print special values
`two+two,sum`: programs from Appendix of RS

Figure 10.24: BSHAM Test Plan—**test implementation strategy**

test implementation strategy
  target `runtestb` in `Makefile`
    test cases for command-line errors
    for each file f in `input/`
      `bsham` f `>act/`f
      `diff act/`f `exp/`f
  statement coverage for *sham* and *exec* measured using the Unix utility *tcov*
    100% coverage for statements not associated with ISHAM

produce any exceptions at all. For load-time exceptions, we note that we have already tested the *load* module. However, recall that in testing *load*, we used stubs for some of the modules in SHAM. We therefore include one test case with a load-time exception, to ensure that the replacement of stubs by production code has not introduced any errors.

Since we have not tested *exec* yet, we must test every run-time exception at least once. However, we have already tested that the access routine **`am_sg_exec`** from *absmach* signals the correct run-time exception, and so we only need to test whether or not *exec* prints the correct message for each exception. We therefore include no more than one test case for every run-time exception. Note that for run-time exceptions we can store only one test case per input file.

For the test cases that do not signal an exception, we also benefit from the testing of **`am_sg_exec`**. We assume that it returns the correct value and correctly changes the memory contents and the value of the accumulator and the program counter. We do need to test whether *exec* correctly deals with the return value of **`am_sg_exec`**. For return values other than **`AM_HALT`** and **`AM_PRINT`**, this is tested by the test cases for the run-time errors. For **`AM_HALT`**, we apply the interval rule to the value of the program counter in which the halt instruction is executed, and we include three test cases. For **`AM_PRINT`**, we apply the interval rule to the value of the accumulator. Finally, we include some non-trivial normal-case test cases as a "sanity check" of the whole system. For this purpose, we use the two SHAM programs from the appendix of the RS.

The **test implementation strategy** section for the TP of BSHAM is shown in Figure 10.24. To run the test cases with command-line errors, we invoke BSHAM with the incorrect command-line arguments. To run the other test cases, we use the same Unix commands as for *load* (see Section 10.7.3). After running the test cases, we use the Unix utility *tcov* to check the statement coverage of *exec* and *sham*. Since these modules contain **`#ifdef`**'s, the statements associated with ISHAM will not be executed. We do check that all statements associated with BSHAM have been executed at least once.

10.9.2 ISHAM

Although BSHAM has some controllability and observability problems, these are far worse in ISHAM. Besides reading the program from a file, ISHAM obtains input from the keyboard and prints information to the screen. However, in testing ISHAM, we use the fact that the command-line arguments and large parts of *exec* and *sham* are exercised in the BSHAM testing, Similarly, the keyboard input and screen output modules have also been tested separately. Therefore we only need to test that *exec* and *sham* correctly respond to the user commands entered at the keyboard. This requires only a few simple test cases, making it feasible to test ISHAM interactively: we manually provide the keyboard commands and manually verify the behavior.

The programs we use for testing ISHAM are a subset of the test cases for BSHAM. There are four programs we use to test ISHAM: one with a load-time exception, one with a run-time exception, and the two programs from the appendix of the RS.

For the program with the load-time exception, ISHAM should terminate after the load phase. For the program with the run-time exception, we step through the program until the exception occurs. We include only one load-time and one run-time exception because all we are testing is that ISHAM correctly responds to these. We have already tested that the exceptions are correctly signaled by `am_sg_exec` and that the correct exception message is produced by *exec*.

For the two programs from the appendix, we step though the entire programs, checking that the screen contents is correctly updated. We also need to test the ISHAM command *EXIT*. For this, we use the simpler of the two programs of the appendix. We load the program three times, and execute the *EXIT* command (1) right after the program is loaded, (2) after several instructions have been executed but before the halt instruction is reached, and (3) when the halt instruction is reached.

After running the test cases, we use the Unix utility *tcov* to check the statement coverage of *exec* and *sham*. This time, we check that every statement inside the `#ifdef`'s for ISHAM have been executed at least once. All other statements in these modules are executed at least once in the testing of BSHAM.

10.10 Verification

In this section, we discuss the verification procedures for the TP and the TI.

10.10.1 Test Plan

Since a TP is not executable, we can verify it only by inspection. Figure 10.25 shows the criteria for a TP. Since the test case selection is primarily based on the MIS, a thorough understanding of the MIS is required to understand the TP.

Figure 10.25: Test Plan Criteria

For the module under test, M:

- **Audience**. Those considering running or modifying M's testing.
- **Prerequisites**. A thorough understanding of M's MIS and some understanding of M's MID.
- **Purpose**. Serve as a planning tool for development and evaluation of the test case selection strategy. Document the TI.
- **Additional criteria**.
  1. *Well-formed.* The TP follows the format described in Section 10.2.
  2. *Comprehensible.* The TP can be read and understood by the intended audience. It is sufficient to estimate the adequacy and the feasibility of the TI.
  3. *Adequate.* Test cases are planned to exercise the module as thoroughly as is practical. Tests are planned to invoke every access routine in normal and exceptional situations. 100% statement coverage is expected.
  4. *Feasible.* The TP can be implemented affordably.

While it is easy to verify that a TP is well-formed, it is a lot harder to verify the last three "additional criteria" for a TP. Dijkstra's law of testing tells us that we can never do enough testing to guarantee that there are no more bugs remaining. We thus have to find a balance between the adequacy of the test inputs and the cost of maintaining and running the test cases. This means that the TP must be comprehensible enough to be able to estimate both the adequacy and the feasibility of the proposed testing. The adequacy and feasibility criteria indicate where the balance should lie. On the one hand, test cases should at least exercise the normal and exceptional behavior of every access routine, and typically 100% statement coverage is expected. On the other hand, it should be possible to implement the testing affordably.

10.10.2 Test Implementation

Recall that the TI includes the test scaffolding, the test data files, and the procedures—both manual and automated—required to execute the tests. The verification criteria for a TI are shown in Figure 10.26. Since a TI is executable, we verify it both by inspection and by testing.

The only criteria that needs further explanation is the correctness criteria. Since the purpose of the TI is to implement the TP, we verify that it follows the TP as closely as possible. For example, in a PGMGEN test script, the order of the test cases should follow the order given in the TP. As shown in the *symtbl* test script (Figure 10.12), the correspondence between the TP and the TI can be clarified by using comments in the test script.

Figure 10.26: Test Implementation Criteria

For the module under test, M:

- **Audience**. Those considering running or modifying M's testing.
- **Prerequisites**. A thorough understanding of M's MIS and some understanding of M's MID.
- **Purpose**. Implement the TP as simply and inexpensively as possible.
- **Additional criteria**.

    1. *Well-formed.* The code in test scripts obeys the Code Format Rules shown in Appendix G. For each standalone module M, a simple interactive tester is provided, giving keyboard access to all access routines and displaying return values on the screen. In M.c, the testing access routine `g_dump` is implemented to display the module state on the screen.
    2. *Comprehensible.* The TI can be read and understood by the intended audience.
    3. *Correct.* The TI follows the TP. The tests run to completion, with correct results and 100% statement coverage.

In addition, we verify that the tests run to completion, with correct results and 100% statement coverage. We measure the statement coverage with the Unix utility *tcov*, which instruments a program with additional statements to keep track of how often every statement in the original program is executed. To measure statement coverage for an MI, it is compiled with a special option. This creates an instrumented object file, and a *tcov* data file. Each time the program is executed, coverage information is accumulated in the data file. Finally, the program *tcov* is run on the data file, which produces a version of the MI in which each statement is prefixed with the number of times it has been executed. To make them easy to find, statements that have never been executed are prefixed with the string "`#####`".

10.11 Summary

While system and integration testing are important, controllability and observability dictate that we test many modules individually. We have advocated the systematic testing of modules through two work products. The TP serves as specification, outlining the test case selection and execution strategies, and the TI implements the TP.

We have emphasized three ways to make the testing of modules affordable and effective.

1. *Select test cases most likely to expose errors.* Use functional testing for test case selection: base test cases on special values for access routine param-

eters, module state, and combinations of these. Use structural coverage, in particular statement coverage, as a check on the adequacy of the test cases.

2. *Automate repetitive and tedious tasks.* Where possible, automate the execution of test cases and the comparison of actual and expected outputs, because these steps are performed every test run. The execution of test cases is automated using the PGMGEN testing tool, customized test drivers, and Unix scripts. The comparison of outputs is automated using PGMGEN scripts and delta testing. Where automation is not possible, limit testing to a few cases for which the output is easy to check.

3. *Design for testability.* The key considerations in design for testability are controllability and observability. Typical problems with controllability and observability are caused by interaction with other modules and the environment. Although this interaction cannot be completely eliminated, we can limit the problems by isolating the module under test, and by restricting the interaction with the environment to a few modules.

10.12 Exercises

1. If we were to test BSHAM in a strictly top-down fashion, we would first test the *sham* Coordinator module using stubs for all the other modules. Define a TP for testing the BSHAM version of *sham* in this fashion.

2. Define the smallest possible number of PGMGEN test cases that still achieve statement coverage for the *stack* module. Do these test cases adequately test *stack*?

3. How many paths are there through the *token* access routine `s_str`? How many of those paths are feasible?

4. This question is based on the *stack* TI.

   (a) Modify the *stack* TI so that it does not make any calls to `s_init`, except for one call at the beginning of the script.

   (b) How would the TI of *stack* have to be changed if exceptions were signaled by printing a message rather than by calling an exception handler?

5. This question is based on the *symtbl* TP and TI. Modify the TP and TI so that it adequately tests an implementation that uses binary search (instead of linear search) to look up the symbols in the table.

6. This question is based on the *absmach* TP and TI. Currently, to test the behavior of `sg_exec` we test the return value, the value of the program

counter, and the value of the accumulator with three separate test cases. Define `cmp_` and `prt_` functions so that these three tests can be performed with a single test case. Modify the *absmach* TP and TI to make use of these two functions.

7. Consider the *sset* (simple set) module, which provides access to a set of at most N integers. The access routines for *sset* are shown in Table 3.24, and the **access routine semantics** are shown in Figure 3.4. Assume that the concrete state of *sset* is defined as

   state variables
   ```
   int set[N];
   int siz;
   ```

   where the array `set` is used to store the elements of the set, `siz` represents the size of the set, and the set elements are stored in `set`[0..`siz`−1]. Define a TP for the above module.

8. Implement the TP from the previous exercise.

9. Currently, the TP and TI of *scngeom* only test the exception behavior. The normal-case behavior is tested while testing *scndr*. Modify the *scngeom* TP and TI so that the normal-case behavior is also tested.

10. Define a TP and TI for the system test of BSHAM, assuming that no module testing is performed for any of the BSHAM modules.

10.13 Bibliographic notes

Two general textbooks on software testing that cover most of the topics discussed in this chapter are [39] and [80]. However, most of the testing research has focused on test case selection. Functional testing was first proposed in [81] and discussed in detail in [40]. Structural testing techniques such as path testing are discussed in [82]. Due to the infeasibility of path testing, other structural testing techniques such as data flow testing [83], mutation testing [84], and domain testing [85] have been proposed. Partition testing [86] mixes functional and structural testing in that it considers both the specification and the implementation for defining test cases. A recent text by Beizer [87] focuses on test case selection, and discusses many of the above, and other, test case selection techniques.

For test case execution, several tools similar to PGMGEN have been proposed in the literature. Panzl [88] reports on the regression testing of Fortran subroutines using an automated tool. The DAISTS system [89] performs module testing and describes test cases using traces. Given a formal algebraic specification of the module under test, DAISTS automatically determines the correct behavior for a given test and measures the coverage of the specification and the implementation. Frankl [90] has developed a tool for object-oriented testing using algebraic specifications. The Protest system [91] is similar to PGMGEN, except that test cases are defined by a Prolog program. Finally, the ACE tool [92] is an enhancement of PGMGEN that supports the testing of Eiffel and C++ classes.

Although very little has been published about test documentation, the IEEE Standard for Software Test Documentation is an excellent source of information on this topic [93]. The standard proposes a set of basic test documents, which is quite more elaborate than the TPs we use in SHAM. The standard also contains an example of how these documents can be used in practice.