MODULE TEST CASE GENERATION

Daniel Hoffman and Christopher Brealey
University of Victoria
Department of Computer Science
P.O. Box 1700
Victoria, B.C., Canada V8W 2Y2

Abstract

While considerable attention has been given to techniques for developing complex systems as collections of reliable and reusable modules, little is known about testing these modules. In the literature, the special problems of module testing have been largely ignored and few tools or techniques are available to the practicing tester. Without effective testing methods, the development and maintenance of reliable and reusable modules is difficult indeed.

We describe an approach for systematic module regression testing. Test cases are defined formally using a language based on module traces, and a software tool is used to automatically generate test programs that apply the cases. Techniques for test case generation in C and in Prolog are presented and illustrated in detail.

1 INTRODUCTION

The fundamental goal of our research is to improve system quality and reduce maintenance costs through systematic module regression testing. While considerable attention is given to testing during software development, this is not the only time testing is required. As Brooks points out:

> As a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire test bank of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice such *regression testing* must indeed approximate this theoretical ideal, and it is very costly [1, pg. 122].

While system testing is usually emphasized, module testing is also important. It is difficult to thoroughly test a module in its production environment, just as it is difficult to effectively test a chip on its production board. IEEE testing standards [2] emphasize the benefits of testing software components, not just complete systems.

Our research focuses on reducing the cost of module regression testing. Since regression tests are developed once and run many times, our efforts are directed towards reducing the costs of test case execution and evaluation. We propose tests which are developed manually, with automated support, and which then run fully automatically. We rely on the test programmer's knowledge of the implementation programming language, and on his ability to effectively select test cases.

2 MODULES AND INTERFACES

Following Parnas [3], we define a *module* as a programming work assignment, and a *module interface* (hereafter just *interface*) as the set of assumptions that programmers using the module are permitted to make about its behavior. An *interface specification* is a statement, in some form, of these assumptions. We view a module as a black box, accessible only through a fixed set of *access programs*. The *syntax* of the specification states the names of the access programs, their parameter and return value types, and the names of the *exceptions* that each access program may generate. Any constants or types provided by the module are also described. The *semantics* of the specification state, for each access program call, the situations in which the call is legal, and the effect that invoking the call has on the legality and return values of other calls. By convention, in access program names we use the prefix **s_** (set) to indicate calls which set internal module values and **g_** (get) to indicate calls which retrieve those values.

Access Program	Inputs	Outputs	Exceptions
s_init			
g_space		integer	
s_addsym	string		maxlen legsym tblfull
s_delsym	integer		notlegid
g_legsym	string	boolean	
g_legid	integer	boolean	
g_sym	integer	string	notlegid
g_id	string	integer	notlegsym

Figure 1. Symbol Table (symtbl) Interface Syntax

These ideas are illustrated on a simple table module which is used as an example throughout this paper. The Symbol Table (*symtbl*) module maintains a set of symbol/identifier pairs. The syntax is shown in Figure 1; the semantics are described informally below and formally elsewhere [4]. Up to S unique symbols may be stored, each up to N characters in length. Each symbol has a unique integer identifier, assigned by *symtbl* from the range $[0, S-1]$. The access programs are divided into three groups.

1. **s_init** initializes the module and must be called before any other call. **g_space** returns the amount of available space in the table.

2. `s_addsym(s)` adds the symbol s to the table, signaling `maxlen` if s is longer than N, `legsym` if s is already in *symtbl*, and `tblfull` if `g_space` = 0. `s_delsym(i)` deletes the symbol with identifier i and signals `notlegid` if no symbol in the table has that identifier.
3. `g_legsym` and `g_legid` are the characteristic predicates of the set of legal symbols and the set of legal identifiers, respectively. `g_sym(i)` returns the symbol with identifier i, signaling `notlegid` if `g_legid(i)` is false; `g_id(s)` returns the identifier for symbol s, signaling `notlegsym` if `g_legsym(s)` is false.

3 TEST PROGRAM GENERATION

Below we describe a *test script language*, used to describe test cases, and the test program generator PGMGEN, which generates test drivers in the C language from scripts.

Test Script Language

Our test scripts are written in terms of module *traces*: sequences of access program calls on the module. Elsewhere traces have been used in connection with the formal specification method of the same name [5, 6] in which logic assertions are used to characterize module behavior on *all* possible traces. Although the trace specification method is powerful, considerable skill is required to devise these assertions. It is straightforward, however, to describe module behavior for any *particular* trace. For example, consider the following traces on the symtbl module. (When writing traces, we separate adjacent calls with a period.)

```
s_init().s_addsym("cat").g_legsym("cat")
s_init().s_addsym("cat").g_id("dog")
```

The first trace initializes *symtbl*, adds the symbol `"cat"` and checks to see if `"cat"` is a legal symbol. In the second trace, the `g_id` call generates the exception `notlegsym` because the symbol `"dog"` is not in *symtbl*.

We describe test cases by providing a trace and associating it with some aspect of the required behavior of the module following that trace. We represent a test case as a five-tuple

$$< trace, expexc, actval, expval, type >$$

with the following meanings:

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).

actval
: an expression (typically a get call) to be evaluated after *trace*, 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, based on the traces described above. In test cases developed solely to do exception checking, the *actval*, *expval*, and *type* fields contain `empty`.

```
<s_init().s_addsym("cat"),
    noexc, g_legsym("cat"), 1, boolean>
<s_init().s_addsym("cat").g_id("dog"),
    notlegsym, empty, empty, empty>
```

A test script consists of a list of access program and exception declarations, a list of test cases, and optional global C code. C code, delimited by the symbols "{%" and "%}", may also be embedded in test cases and provides the test programmer with expressions and control structures not supported by PGMGEN. 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 which will determine whether an implementation satisfies this partial specification.

Test Program Generation

Although implementing test drivers manually is straightforward, it is also tedious and error-prone, and produces code that is costly to maintain. As a result, test driver generation is a good candidate for automated support. In this section we briefly describe how PGMGEN accomplishes this task; a more detailed description is available [7].

Initially, code is generated to record exception occurrences. Then, for each test case of the form:

$$< c_1.c_2.....c_N, expexc, actval, expval, type >$$

code is generated to:

```
invoke c1, c2, ..., cN
compare the actual occurrences of exceptions
    against expexc
if there are any differences
    print a message
else
    if actval ≠ expval
        print a message
    if any exceptions have occurred since cN
        was invoked
        print a message
update summary statistics
```

Following the last case, code is generated to print summary statistics.

In order to automate driver generation, we have made the following assumptions about access program invocation and exception signaling. Each access program is implemented as a C function. For each exception, there is a C function of that name, serving as exception handler. When an exception occurs, the module implementation must call the appropriate function. The module user is expected to implement the exception handlers to take whatever action he deems suitable. In a *symtbl* implementation, each set and get call is implemented as a C function and the functions `legsym`, `maxlen`, `notlegid`, `notlegsym`, and `tblfull` are invoked when the corresponding exception is detected. However, the *symtbl* user implements these exception functions. PGMGEN, for example, implements the exception handlers to set flags for monitoring exception occurrences.

We have developed PGMGEN scripts for over 20 modules [7], including many of the modules in the PGMGEN implementation itself. Our test case selection has been based primarily on *functional testing* [8]. We have found the scripts significantly easier to develop and maintain than the manually generated drivers used previously. In particular, the scripts are roughly an order of magnitude shorter than the generated drivers.

4 TEST CASE GENERATION IN C

While PGMGEN allows a test programmer to write and maintain scripts which are significantly simpler than their corresponding C drivers, test scripts themselves often become lengthy. The effort to test multiple combinations of calls and parameter values produces scripts that are long and repetitive — 100 or more cases is not unusual even for implementations only several hundred lines in length. To reduce script size, we have developed a scheme for replacing long lists of test cases with templates driven by "embedded code" loops written in C.

General Approach

- Choose the test basis T

  Choose a subset T of the set of all traces on the module. Each of these traces may be viewed as an abstract state of the module corresponding to the actual state of the implementation after the calls in the trace have been invoked. Typically T is chosen from the *normal form* [6], a set of traces chosen to abstractly represent the entire state space of the module. For testing purposes, the traces in T must be easy to generate and it must be easy to compute, for each $t \in T$, the expected behavior of access programs immediately following t.
- Provide a generator for T

  Choose a representation for the elements of T, a means for requesting these elements one at a time, and a method for accessing the characteristics of the current element. Analogous to the *iterator* construct of programming languages such as CLU [9], the generator simplifies test scripts by separating the means for generating elements from the processing done on the current element.
- Apply the test cases

  Use the generator described above to write loops, each executing a set of test cases once for each $t \in T$. Invoke access programs to determine if the implementation behavior is correct with respect to t. Check that set and get calls operate correctly and exercise both normal and exception behavior. Use the generator to provide the expected behavior for these test cases.

Symtbl Example

We apply the above approach to testing a straightforward *symtbl* implementation. It is sufficient to know that symbols are stored in a fixed-size rectangular array with one row per symbol, identifers are zero-relative row indexes, linear search is used for symbol lookup, and the lowest identifier available is given to a newly added symbol.

- Choose the test basis T

  For *symtbl*, each table value corresponds to a trace consisting of an `s_init` call followed by zero or more `s_addsym` calls. We would like our tests to be based on tables for which the number of symbols and the length of the symbols vary. Symbol values are unimportant as long as they are unique and easy to generate. We define the function $mksym(i, n)$, whose value is i in string form padded right with '*' characters to length n (or zero '*' characters if i has n or more digits). We define $t_{s,n}$ as a table with s symbols where, for $i \in [0, s-1]$, the symbol with identifier i is $mksym(i, n)$. More precisely, $t_{s,n}$ is

  `s_init().s_addsym(`$mksym(0,n)$`).....`
  `s_addsym(`$mksym(s-1,n)$`)`

  For example, $t_{2,5}$ is

  ```
  s_init().s_addsym("0****").s_addsym("1****")
  ```

  Finally, we define T as $\{t_{s,n} \mid s \in \{0, S/2, S\} \wedge n \in \{0, N\}\}$ to focus testing on critical table sizes and symbol lengths.
- Provide a generator for T

  To represent the elements of T, we implement the C function `t_mksym` to compute *mksym* and declare `t_tbl` as an array of table size/symbol length pairs. Then, iterating over T is accomplished by indexing `t_tbl`: `t_init` initializes the index, `t_next` increments the index and loads the next table into *symtbl*, and `t_end` returns true when the index exceeds the number of elements in `t_tbl`. For the current table, `t_siz` returns the expected table size and `t_sym(i)` returns the expected symbol with identifier i, calculated using `t_mksym` and the current `t_tbl` element.
- Apply the test cases

  With the above functions in place, the test cases may be coded independently of the representation of T, as shown in Figure 2. In the outer loop, executed once for each table, are cases to check that `g_space` returns the correct value and that `g_legsym` works properly on a symbol much longer than N. The inner loop is executed once for each identifier in $[0, S-1]$. For an identifier i in the current table are cases to check that `t_sym(i)` is legal and has the correct identifier, and, similarly, that i is legal and is associated with the correct symbol. Also included are cases to check that `s_del(i)` is legal and that it deletes both i and `t_sym(i)`. Finally, there are cases to check that, when i is not in the current table, both `t_sym(i)` and i are illegal.

The full script, including the C functions described above and exception checks, is shown in the Appendix. For $S = 50$, the script produces a C driver which is 600 lines in length and executes 1842 test cases.

5 TEST CASE GENERATION IN PROLOG

In the previous section, we made use of C code to iterate over a list of relatively fixed test cases. To generate cases with significant variations, we have found Prolog more effective than C. (Some familiarity with Prolog is necessary to understand this chapter.)

Prolog Code

We base test case generation on three Prolog predicates. The first two are rewritten for each set of test cases; the third is used without change.

- `cases(`*trace*`,`*expbeh*`,`*gen*`,`*parmlist*`,`*maxcases*`)` describes the test case format. *trace* is a list of access program calls and *expbeh* is the expected behavior for *trace*. *gen* is the name of a Prolog predicate and *parmlist* is a list of parameters to *gen*. *maxcases* is the maximum number of cases to be generated.
- `gen(`*incase*`,`*outcase*`,`*parmlist*`)` instantiates *outcase*, based on *incase* and *parmlist*.

```
{% t_init();
t_next();
while (!t_end()) { %}
  < , noexc, g_space(), {%S-t_siz()%}, int>
  < , noexc, g_legsym({%t_mksym(0,T_MKSYMMAX)%}),
    0, bool>
{%for (i = 0; i < S; i++) {
    if (i < t_siz()) { %}
      /*t_sym(i) is legal, has correct id*/
      < , noexc, g_legsym({%t_sym(i)%}), 1, bool>
      < , noexc, g_id({%t_sym(i)%}), i, int>
      /*i is legal and has correct symbol*/
      < , noexc, g_legid(i), 1, bool>
      < , noexc, g_sym(i), {%t_sym(i)%}, string>
      /*s_del deletes i and symbol*/
      <s_del(i), noexc, empty, empty, empty>
      < , noexc, g_legsym({%t_sym(i)%}), 0, bool>
      < , noexc, g_legid(i), 0, bool>
{%  } else { %}
      /*t_sym(i) and i are not legal*/
      < , noexc, g_legsym({%t_sym(i)%}), 0, bool>
      < , noexc, g_legid(i), 0, bool>
{%  }
  }
  t_next();
} %}
```

Figure 2. symtbl test script — normal case

- casegen(f) generates a test script in file f, based on the following pseudocode

  do
    invoke *gen*([*trace*,*expbeh*],*outcase*,*parmlist*)
    convert *outcase* to PGMGEN syntax and write to f
    write converted *outcase* to f
  until (*gen* fails or *maxcases* cases are generated)

Trace Equivalence

Trace T_1 is equivalent to T_2 ($T_1 \equiv T_2$) if T_1 and T_2 are indistinguishable with respect to future module behavior, i.e., T_1 and T_2 leave the module in the same abstract state. Trace equivalences provide a good basis for test case generation. Roughly speaking, in Section 4, we tested whether a given trace put the module in the *correct* state — here we test whether equivalent traces put the module in the *same* state.

Exhaustive testing of trace equivalence is rarely possible — both the set of equivalent traces and the set of traces constituting "future behavior" are typically infinite. Therefore, for testing purposes, a subset of the equivalent traces and a subset of the future behavior must be selected.

Testing Trace Equivalence

Assuming the same *symtbl* implementation as in Section 4, we generate cases to test the equivalence

$$T \equiv T.\texttt{s\_addsym}(x).\texttt{s\_del}(\texttt{g\_id}(x))$$

i.e., "add/delete pairs cancel." This equivalence holds for all traces T and strings x and, by the transitivity of $\equiv$, for any number of add/delete pairs following T. For testing purposes, we fix T to be

```
s_init.s_addsym(a).s_addsym(b).s_addsym(c)
```

We follow T with between 1 and 3 add/delete pairs where the symbol d or e is added and the identifier 3 is deleted. We test the equivalence of these traces by invoking get calls to check that that the symbol a with identifier 0 is in the table and that the symbol d and identifier 3 are not in the table.

```
cases(
  [s_init,s_addsym(a),s_addsym(b),s_addsym(c),
    AddDels],
  [actval(ActVal),expval(ExpVal)],
  gen,
  [3,AddDels,ActVal,ExpVal],
  10000).
gen(T,T,[N,AddDels,ActVal,ExpVal]) :-
  adddel(N,AddDels), getcall(ActVal,ExpVal).
adddel(N,L) :-
  N > 0, adddel_1(N,L).
adddel(N,L) :-
  N > 1, N1 is N-1, adddel(N1,L).
adddel_1(0,[]).
adddel_1(N,[s_addsym(S),s_del(3)|Tail]) :-
  N > 0, new_symbol(S), N1 is N-1,
  adddel_1(N1,Tail).
new_symbol(d).  new_symbol(e).
getcall(g_legsym(a),1).  getcall(g_legid(0),1).
getcall(g_sym(0),a).  getcall(g_id(a),0).
getcall(g_legsym(d),0).  getcall(g_legid(3),0).
```

Figure 3. *cases* and *gen* predicates

Figure 3 shows the cases and gen predicates used to accomplish this testing. In cases, the trace begins with T as described above. The tail of the trace and the expected behavior are variables, passed as parameters to gen. gen's first parameter is the maximum number of add/delete pairs. gen is defined in terms of adddel and getcall. adddel(N,L) is true if L is a list of N or fewer add/delete pairs, where the added symbol is d or e and the deleted identifier is 3. getcall(C,X) is true of the 7 getcall/expected value pairs shown.

When casegen is invoked, a script with 98 cases results, from which PGMGEN produces a C driver of 2484 lines.

6 RELATED WORK

Considerable work has been done in the area of test case selection. The two basic approaches are *black-box* and *white-box* testing. In black-box testing, the tests are constructed based on the requirements of the program. Both *functional* testing [8] and *random* testing [10] are examples of black-box testing. White-box testing uses the internal structure of the program to select appropriate test cases [11, 12]. Mutation testing [13] is a test input analysis technique based on constructing variants, called *mutants*, of the program under test. A set of test cases is evaluated according to its ability to distinguish between a program and its mutants. Our test methods are neither black-box nor white-box, but combine both of these methods. While we emphasize the module interface and have been influenced by Howden's proposals for functional testing, we also base test cases on the module implementation.

Relatively little has been published on test case execution. The DAISTS system [14] focuses on module testing and

describes test cases using sequences of calls. Given a formal algebraic specification of the module under test, DAISTS automatically determines the correct behavior for a given test. Panzl [15] reports on regression testing of Fortran subroutines. He presents a test case description language and a program to automatically execute the cases, monitoring actual versus expected behavior. Choquet [16], Gerhart [17], Gorlick [18], and Wild [19] have all explored the use of Prolog for test case generation. Our work is most similar to the DAISTS work, which goes further than ours by providing a test oracle, but offers little for testing modules when an algebraic specification is unavailable.

7 CONCLUSIONS

We have argued the importance of systematic module regression testing and presented tools and techniques for accomplishing that task. We have made a conscious decision to base our testing on the module interface — both the test script language and the automated support provided by PGMGEN depend critically on this decision.

We have presented two test case generation techniques for situations where test scripts themselves become uncomfortably long. In the first technique, a set of base traces is chosen, a generator for that set is developed, and test cases focus on the behavior of calls executed just after the base traces. In the Prolog-based approach, sets of equivalent traces are chosen and cases are written to test if the implementation preserves the equivalence. The superiority of Prolog or C as a test generation language remains an open question.

With the ability to generate large numbers of test cases automatically, care must be taken when interpreting test case counts. Specifically, there is no simple connection between the number of test cases and either the quality of the test or the cost to develop the test. Consider the following script.

```
{% for (i = 0; i < 1000000; i++) do }%
    <s_init, noexc, g_legsym(i), 0, int>
```

This code will generate one million cases, yet is surely far less effective and far less expensive to develop than the script shown in Figure 2. We have found that test quality depends on careful selection of test cases and that test cost is dominated by the size and complexity of the test case generation code.

In the testing literature, a *test oracle* is typically assumed to exist and discussions focus on test input generation. In practice, the cost of examining outputs for correctness cannot be ignored — it is pointless to generate inputs if you cannot afford to check the outputs! We have found that it is often much easier to generate input/output pairs than to generate output given a random input. The latter requires a full implementation; the former is far easier. We have exploited this idea in our test case generation techniques.

REFERENCES

[1] F.P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975.

[2] *IEEE Standard for Software Unit Testing.* Soft. Eng. Tech. Comm. of the IEEE Computer Society, May 1987.

[3] D.L. Parnas and P.C. Clements. A rational design process: how and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.

[4] D.M. Hoffman. Practical interface specification. *Software - Practice and Experience,* 19(2):127–148, February 1989.

[5] Wolfram Bartussek and David L. Parnas. Using traces to write abstract specifications for software modules. In *Information Systems Methodology,* pages 211–236, Springer-Verlag, 1978. Proc. 2nd Conf. European Cooperation in Informatics, October 10-12, 1978.

[6] D.M. Hoffman and R. Snodgrass. Trace specifications: methodology and models. *IEEE Trans. Soft. Eng.*, 14(9), 1988.

[7] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance (accepted for publication),* IEEE Computer Society, October 1989.

[8] W.E. Howden. Functional program testing. *IEEE Trans. Soft. Eng.*, SE-6(2):162–169, March 1980.

[9] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[10] J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Trans. Soft. Eng.*, SE-10(4):438–444, July 1984.

[11] J.C. Huang. An approach to program testing. *ACM Computing Surveys,* 7(3):113–128, September 1975.

[12] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Soft. Eng.*, SE-2(3):208–215, September 1976.

[13] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. Principles Prog. Lang.*, pages 220–233, ACM, 1980.

[14] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[15] D.J. Panzl. A language for specifying software tests. In *Proc. AFIPS Natl. Comp. Conf.*, pages 609–619, AFIPS, 1978.

[16] N. Choquet. Test data generation using a prolog with constraints. In *Workshop on Software Testing,* pages 132–141, IEEE Computer Society, 1986.

[17] S. Gerhart. *A Test Data Generation Method Using Prolog.* Technical Report TR85-02, Wang Inst. of Grad. Studies, 1985.

[18] M.M. Gorlick, C.D. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a logical methodology for testing. *Journal of Logic Programming (to appear),* 1988.

[19] C. Wild. *The Use of Generic Constraint Logic Programming for Software Testing and Analysis.* Technical Report 88-02, Dept. of Computer Science, Old Dominion University, Norfolk, VA, 1988.

APPENDIX - C TEST SCRIPT

```
accprogs /* declare access programs and arity*/
  <s_init:0,g_space:0,s_addsym:1,s_del:1,
  g_legid:1,g_legsym:1,g_id:1,g_sym:1>
exceptions /*declare exceptions*/
  <legsym,maxlen,notlegid,notlegsym,tblfull>
```

```
globcod /*global C code*/
{%
#include "symtbl.h"
#define T_MKSYMMAX 1000
#define T_FILLCHAR '*'
#define T_TBLSIZ 5
static int i;
static struct {
  int siz;
  int symlen;
} t_tbl[] = {
  {0,0},
  {S/2,0},
  {S/2,N},
  {S,0},
  {S,N}
};
static int t_cur;
static char *t_mksym(i,len)
int i,len;
{
  static char buf[T_MKSYMMAX+1];
  int j;
  sprintf(buf,"%d",i); /*convert i to ASCII*/
  if (len > strlen(buf)) {
    for (j = strlen(buf); j < len; j++)
      buf[j] = T_FILLCHAR;
    buf[len] = '\0';
  }
  return(buf);
}
static void t_init()
{
  t_cur = -1;
}
static void t_next()
{
  int i;
  t_cur++;
  if (t_cur >= 0 && t_cur < T_TBLSIZ) {
    s_init();
    for (i = 0; i < t_tbl[t_cur].siz; i++)
      s_addsym(t_mksym(i,t_tbl[t_cur].symlen));
  }
}
static int t_end()
{
  return(t_cur >= 5);
}
static int t_siz()
{
  return(t_tbl[t_cur].siz);
}
static char *t_sym(i)
int i;
{
  return(t_mksym(i,t_tbl[t_cur].symlen));
}
%}
cases
/*****exceptions*****/
{% t_init();
t_next();
while (!t_end()) {
  for (i = 0; i < t_siz(); i++) { %}
    /*add an existing symbol*/
    <s_addsym({%t_sym(i)%}), legsym, empty, empty,
      empty>
    /*add overlength symbols*/
    <s_addsym({%t_mksym(0,N+1)%}), maxlen, empty,
      empty, empty>
    <s_addsym({%t_mksym(0,T_MKSYMMAX)%}),
      maxlen, empty, empty, empty>
{%/*add a symbol to a full table*/
  if (t_siz() == S) %}
    <s_addsym("x"), tblfull, empty, empty, empty>

{%} %}
  /*delete ids not in the table*/
  <s_del(-1), notlegid, empty, empty, empty>
  <s_del({%t_siz()%}), notlegid, empty, empty,
    empty>
  /*request symbols for ids not in the table*/
  <g_sym(-1), notlegid, empty, empty, empty>
  <g_sym({%t_siz()%}), notlegid, empty, empty,
    empty>
  /*request ids for symbols not in the table*/
  <g_id({%t_mksym(t_siz(),0)%}), notlegsym, empty,
    empty, empty>
  <g_id(""), notlegsym, empty, empty, empty>
{%t_next();
} %}
/*****normal case*****/
/*special - empty string should be a legal symbol*/
<s_init().s_addsym(""), noexc, g_legsym(""), 1,
  bool>
< , noexc, g_legid({%g_id("")%}), 1, bool>
{% t_init();
t_next();
while (!t_end()) { %}
  < , noexc, g_space(), {%S-t_siz()%}, int>
  < , noexc, g_legsym({%t_mksym(0,T_MKSYMMAX)%}),
    0, bool>
{%for (i = 0; i < S; i++) {
    if (i < t_siz()) { %}
      /*t_sym(i) is legal and has correct id*/
      < , noexc, g_legsym({%t_sym(i)%}), 1, bool>
      < , noexc, g_id({%t_sym(i)%}), i, int>
      /*i is legal and has correct symbol*/
      < , noexc, g_legid(i), 1, bool>
      < , noexc, g_sym(i), {%t_sym(i)%}, string>
      /*s_del deletes i and symbol*/
      <s_del(i), noexc, empty, empty, empty>
      < , noexc, g_legsym({%t_sym(i)%}), 0, bool>
      < , noexc, g_legid(i), 0, bool>
{%  } else { %}
      /*t_sym(i) and i are not legal*/
      < , noexc, g_legsym({%t_sym(i)%}), 0, bool>
      < , noexc, g_legid(i), 0, bool>
{%  }
  }
  t_next();
} %}
```