PROCEEDINGS
The Eighth International Symposium on
Software Reliability Engineering
November 2-5, 1997

This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Dreyer, J., Program Segmentation for Controlling Test Coverage, Proc. 8th IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 72-83

Program Segmentation for Controlling Test Coverage

F. Belli J. Dreyer
University of Paderborn, Dept. of Electrical and Electronics Engineering,
33095 Paderborn, Germany
Email: belli@uni-paderborn.de dre@omega.uni-paderborn.de

Abstract

In this paper we present a new control-flow based approach to dynamic testing of sequential software. A practicable number of test cases is generated by using the boundary-interior path testing strategy [11] and by dividing the test units into test segments (program fragments composed of one statement or a sequence of statements). The size of the test segments can be adjusted by means of a parameter, i.e. the thoroughness of the test coverage can be adapted to the needs of the tester. The selection of test cases is performed by constructing path-classes for each test segment. The coverage criteria constructed by means of our approach (test segment coverage criteria) are fulfilled if at least one path from each path-class is covered. A validation of our approach is given by comparing the fault detection capabilities of test segment coverage criteria with the fault detection capabilities of branch testing, multiple-condition testing, LCSAJ testing and all-uses testing using n test cases for each item (e.g. branch) to be covered. The comparison demonstrates that, compared with the other testing criteria, greater fault detection probabilities can be achieved if a test segment coverage criterion is used.

1 Introduction

The quality of software can be validated by means of dynamic testing. Dynamic testing involves the execution of the program using previously selected *test cases*. A test case is an ordered pair of an input and the output, expected as the outcome of the program execution. The generated output is compared with the expected output. Note that the expected output can specify more than one acceptable result. The test succeeds if the execution result is in conformity with the expected output.

Dynamic testing techniques have been divided into two classes: Techniques where the selection of test cases is based on the program specification and techniques where the selection of test cases is based on the implementation. The goal of implementation-based testing is to *cover* the program code according to a specified execution thoroughness degree. *Statement coverage* is a weak coverage criterion. Dynamic testing reaches 100% statement coverage, if all statements in the program are executed at least once. *Branch testing* implies traversing each branch of the control-flow graph at least once. The coverage of all paths through the control-flow graph is called *path testing*. Among the presented implementation-based testing criteria, path testing is the strongest criterion, because it strictly subsumes all other presented strategies. (Comparison criteria and a comparison of software testing approaches are given in [6].) But path testing is generally not practicable, since even small size programs can have large (potentially infinite) number of paths. A solution to this problem can be given by the reduction of the number of test cases by a coverage criterion that can be constructed efficiently. There exist several testing techniques to bridge the gap between branch testing and path testing. They include techniques based on *data flow analysis* [13, 15, 16, 9], *LCSAJ testing* [18], *boundary-interior path testing* [11], *structured testing* [12] and *Ct coverage* [3].

Techniques based on data flow analysis concentrate on the *definitions* and *references* of program variables. A program variable is said to be *defined* in a segment, if a value is assigned to the variable inside the segment. A program variable is said to be *referenced* in a segment, if the first action containing the variable inside the segment uses the actual value of the variable. A well-known data-flow oriented approach is *all-uses* testing [16]. The all-uses testing strategy requires the coverage of at least one path segment from each definition to all references reached by that definition.

LCSAJ (linear code sequence and jump) testing is a control-flow oriented testing strategy, but it is based on the program text rather than the control-flow graph. A linear code sequence and jump is a sequential piece of code. It is characterized by a start line, an end line and a line where it jumps to (target line). A start line is a target line of a previous jump or the first line of the program. An end line is

any line that can be reached from a start line by an unbroken sequence of code.

Boundary-interior path testing [11] reduces the number of test cases by classifying paths that differ only in the number of times they iterate loops. Thus, instead of using all paths of a group, only few representative paths are selected. Boundary-interior path testing distinguishes two classes of paths corresponding to each loop. Paths belonging to the first class enter the loop but do not iterate it, whereas paths belonging to the second class iterate the loop at least once. Structured testing [12] and Ct coverage [3] are similar in the sense that both approaches specify that paths through the control-flow graph that are equivalent up to a iteration count k belong to the same path class.

In this paper we introduce a new control-flow based approach for controlling test coverage of sequential software. Our approach for parallel systems is general and less strict [2]. The approach for sequential software is based on boundary-interior path testing [11] and regulates the thoroughness of test coverage by dividing the program to be tested according to a parameter C into test segments (program fragments composed of one statement or a sequence of statements). The parameter C determines the size of the test segments. It can be used to adjust the thoroughness of test coverage and consequently the number of test cases to the needs of the tester.

Section 2 describes the coverage criterion which is based on boundary-interior path testing. In Section 3, our program segmentation approach for controlling test coverage is introduced. The following section illustrates the approach by means of an application. In Section 5 our approach is validated by comparing the fault detection capabilities of coverage criteria constructed by means of our approach (test segment coverage criteria) with the fault detection capabilities of branch testing, multiple-condition testing, LCSAJ testing and all-uses testing. Finally, the conclusion is presented.

2 The bi-coverage criterion

The approach presented in this paper is based on the boundary-interior path testing strategy [11]. The selection of test cases is performed by constructing path-classes and by covering from each class one path. The generated path-classes are given in form of regular expressions. We decided to deploy regular expressions because of their conciseness and mathematical rigor and because this textual language can be processed directly without internal transformations. A short introduction to regular expressions is given in the appendix. In the following the generation of all words corresponding to a language described by a regular expression R is called *expansion of R*.

Definition 2.1 *$\mathcal{R}(I)$ denotes the regular expression generated from program I according to Fig. 12 in the appendix.*

$\mathcal{R}(I)$ describes the control-flow structure of the program, i.e. an atom of the regular expression corresponds to a branch. Let $L(\mathcal{R}(I))$ be the language denoted by $\mathcal{R}(I)$. If $w \in L(\mathcal{R}(I))$, then w can be considered as a path through the program I. This means that all the paths through the program I can be generated by expanding $\mathcal{R}(I)$. By *boundary-interior expansion* (bi-expansion) of $\mathcal{R}(I)$ we denote a restricted expansion based on the boundary-interior path testing strategy. The bi-expansion of $\mathcal{R}(I)$ generates the path-classes.

Definition 2.2 *The bi-expansion of regular expressions is composed of two steps:*
Step 1 corresponds to the bi-expansion of the loops and Step 2 bi-expands the union expressions. Let Q be a regular expression.

- *bi-expansion of loops*
  Let Q^* denote a loop generated during the bi-expansion and $L(Q^*) = \{\lambda, Q, Q\,Q, Q\,Q\,Q, \ldots\}$.
  - *A regular expression of the form Q^* is bi-expanded into the equivalent regular expression $\lambda + Q + (Q\,Q\,Q^*)$.*
  - *A regular expression of the form Q^+ is bi-expanded into the equivalent regular expression $Q + (Q\,Q\,Q^*)$.*

  Loops of the form Q^* are not bi-expanded.
- *bi-expansion of unions*
  A regular expression of the form $(Q + R)$ is divided into the following two regular expressions: Q and R. The bodies of loops are not bi-expanded.

Example: bi-expansion of $(a^*\ b)^*$

- bi-expansion of the inner loop generates $(\,(\lambda + a + a\ a\ a^*)\ b)^*$. Let c denote $(\lambda + a + a\ a\ a^*)$.
- bi-expansion of the outer loop generates $\lambda + c\ b + c\ b\ c\ b\ (c\ b)^*$
- bi-expansion of unions generates
  λ, b, $a\ b$, $a\ a\ a^*\ b$, $b\ b\ (c\ b)^*$, $b\ a\ b\ (c\ b)^*$, $b\ a\ a\ a^*\ b\ (c\ b)^*$, $a\ b\ b\ (c\ b)^*$, $a\ b\ a\ b\ (c\ b)^*$, $a\ b\ a\ a\ a^*\ b\ (c\ b)^*$, $a\ a\ a^*\ b\ b\ (c\ b)^*$, $a\ a\ a^*\ b\ a\ b\ (c\ b)^*$, $a\ a\ a^*\ b\ a\ a\ a^*\ b\ (c\ b)^*$.

Definition 2.3 *Let $\mathcal{R}(I)$ and S be regular expressions and $B(\mathcal{R}(I))$ denote the set of regular expressions generated by bi-expanding $\mathcal{R}(I)$. If and only if $S \in B(\mathcal{R}(I))$ then S is called* bi-class of $\mathcal{R}(I)$.

Whereas an expansion generates words, i.e. regular expressions without any operator except the concatenation, a bi-expansion generates regular expressions that may contain all operator-types. The following example shows that a language denoted by a bi-class may comprise an infinite number of words (paths). Let $S \in B(\mathcal{R}(I))$ and $S = b\ a\ a\ a^*$. Then $L(S) = \{b\ a\ a,\ b\ a\ a\ a,\ b\ a\ a\ a\ a,\ \ldots\}$.

Definition 2.4 *Let $\mathcal{R}(I)$ be the regular expression generated from program I according to Definition 2.1. A program I is called* bi-covered *if and only if from each $S \in B(\mathcal{R}(I))$ at least one path has been tested.*

Note that a regular expression of the form

$$(Q_{11} + Q_{12})\ (Q_{21} + Q_{22})\ \cdots\ (Q_{n1} + Q_{n2})$$

contains 2^n bi-classes. Although the bi-coverage criterion reduces the number of required test cases considerably, complete bi-coverage is often still intractable. The complete bi-coverage of a test unit containing 20 non nested if statements requires the coverage of 1 048 576 (2^{20}) bi-classes.

3 Test segment coverage

In this chapter our approach for controlling test coverage will be introduced. The approach is based on the bi-coverage criterion. In order to reduce the number of classes to be covered, the implementation is divided automatically into test segments, i.e. program fragments composed of one statement or a sequence of statements. A statement can be either a simple statement (e.g. an assignment) or a compound statement (e.g. an iteration of statements, a sequence of statements, a selection containing statements). I_k is used in the following to denote test segment k. The size of the test segments and consequently the number of generated test segments are regulated by means of a parameter C that specifies the maximum number of bi-classes belonging to a test segment. For each test segment bi-classes are generated. The operation of the segmentation routine is outlined in the following by means of short examples. Let S be a regular expression and $bi^{\#}(S)$ denote the number of bi-classes generated by bi-expanding S.

- Example 1: The program to be tested is composed of a sequence of three compound statements $s1, s2$ and $s3$ with $bi^{\#}(R(s1)) = 4$, $bi^{\#}(R(s2)) = 6$ and $bi^{\#}(R(s3)) = 8$. Let $C = 30$. As $bi^{\#}(R(s1)) \leq 30$, statement $s1$ is assigned to test segment I_0. As $bi^{\#}(R(s1)) * bi^{\#}(R(s2)) \leq 30$, $s2$ is also assigned to I_0. $bi^{\#}(R(s1)) * bi^{\#}(R(s2)) * bi^{\#}(R(s3)) > 30$, therefore test segment I_1 is generated inside I_0 and $s3$ is assigned to I_1. Test segments are considered at the next higher level as a primitive statement, i.e. $bi^{\#}(R(I_0)) = 4 * 6 * 1 = 24$ and $bi^{\#}(R(I_1)) = 8$.

- Example 2: The program is composed only of an if-statement $s1$. Let $s2$ denote the compound statement in the *then*-part of $s1$, $s3$ denote the compound statement in the *else*-part of $s1$, $C = 30$, $bi^{\#}(R(s1)) = 50$, $bi^{\#}(R(s2)) = 22$ and $bi^{\#}(R(s3)) = 28$. As $bi^{\#}(R(s1)) > 30$, either $s2$ or $s3$ has to be assigned to I_1. The segmentation routine always selects the alternative with the greatest number of bi-classes, i.e. the program is segmented as follows:

```
/* beginning of test segment 0 */
if (x)
   s2;
else /* beginning of test seg. 1*/
        s3;
     /* end of test segment 1 */
/* end of test segment 0 */
```

  Consequently $bi^{\#}(R(I_0)) = 22 + 1 = 23$ and $bi^{\#}(R(I_1)) = 28$.

Definition 3.1 *The set of all the paths through the control-flow graph of the program that bi-cover the bi-class i of $\mathcal{R}(I_k)$ is called* test segment class (s-class) $sc(I, k, i)$.

Paths through the program control-flow graph belonging to the same s-class differ inside the corresponding test segment only in the number of times they iterate a loop.

Definition 3.2 *Let I^n denote the set of test segments generated from program I by means of the segmentation routine using $C = n$. A test segment coverage criterion s_n is fulfilled if and only if from all $sc(I, k, i)$ of all $I_k \in I^n$ at least one path has been tested.*

Note that a test segment coverage criterion s_n (s_n coverage) considers only branch-combinations inside test segments. If branches are located in different test segments, it does not require that the branches have to be traversed during one program execution.

The reduction of test cases caused by dividing a program into test segments is shown in Fig. 1 by means of examples. For simplicity it is assumed that the programs contain only non-nested if-statements and all test segments contain the same number of if-statements. It is also assumed that the test segments are tested separately. In case the test segments are not tested separately, the number of paths required to reach s_n coverage of a program with more than one test segment is smaller, since one path covers several s-classes corresponding to different test segments.

The values of the column labeled "1 test segment" are equal to the number of paths (and bi-classes) of the programs. They show that path testing is only practicable for small programs even if no loops are contained in the program.

if-statements	Number of paths to be tested			
	1 segment	2 segments	4 segments	8 segments
16	2^{16}	512	64	32
32	2^{32}	131072	1024	128
64	2^{64}	2^{33}	262144	2048

Figure 1. Reduction of test cases caused by segmentation

program	loops	if-statements	number of paths to be tested		
			$C = 4$	$C = 400$	$C = 40000$
LUDCMP	20	4	66	705	46118
COVSRT	17	3	57	474	33345
KENDL2	9	3	33	299	7694
REALFT	8	6	36	457	2460

Figure 2. Cost of test segment coverage criteria

3.1 Controlling test coverage

By changing the value of parameter C, the thoroughness of test coverage and the number of required test cases can be adjusted to the needs of the tester or to the projected duration of the test-phase. A small value for C leads to a great number of small test segments and therefore to a small number of test cases. In Fig. 2 the cost associated with test segment coverage criteria is shown using several programs. The programs were selected from the suite of programs called *Numerical Recipes* [17] because these programs were written by professional programmers, contain a substantial number of loops (see column two of Fig. 2) and are available in machine readable form. Fig. 2 demonstrates that, even for programs which contain a substantial number of loops, practicable test segment coverage criteria can be constructed by means of selecting an appropriate value for C. A simple strategy for adjusting the value for C is to start with a segmentation using a small value for C (e.g. 10). After segmenting the program, the number of s-classes is computed automatically. Then, the thoroughness of test coverage can be increased step by step by increasing C and initiating a new segmentation. This will lead to a small number of test segments and therefore to a great number of s-classes. C is increased until an s_n coverage criterion is reached which cannot be fulfilled during the projected testing duration. Finally the s_n coverage criterion of the previous step is selected for the test project. An adequate metric in order to specify the thoroughness of s_n coverage criteria is the number of generated segments.

3.2 Generation of s-classes

After an s_n coverage criterion has been selected, the class-generator has to be executed. It first generates for each test segment I_k an entry containing regular expression $R(I_k)$. The entry has following form: `segment(Segment_number, Regular_expression)`. Next all regular expressions are bi-expanded and for each bi-class the corresponding s-class is generated. The s-classes are generated using following steps:

1. FOR $k := 0$ TO *last_segment* DO
   bi-expand the regular expression $\mathcal{R}(I_k)$ and store the generated bi-classes. Let $bi(k,i)$ denote the i-th bi-class of $\mathcal{R}(I_k)$.
2. FOR $k := 0$ TO *last_segment* DO
   - Assign $\mathcal{R}(I)$ to variable $p[k]$.
   - Remove all paths from $p[k]$ that do not cover any bi-class of $\mathcal{R}(I_k)$. (This step is necessary since $R(I)$ can contain paths that do not traverse $\mathcal{R}(I_k)$.)
3. FOR $k := 0$ TO *last_segment* DO

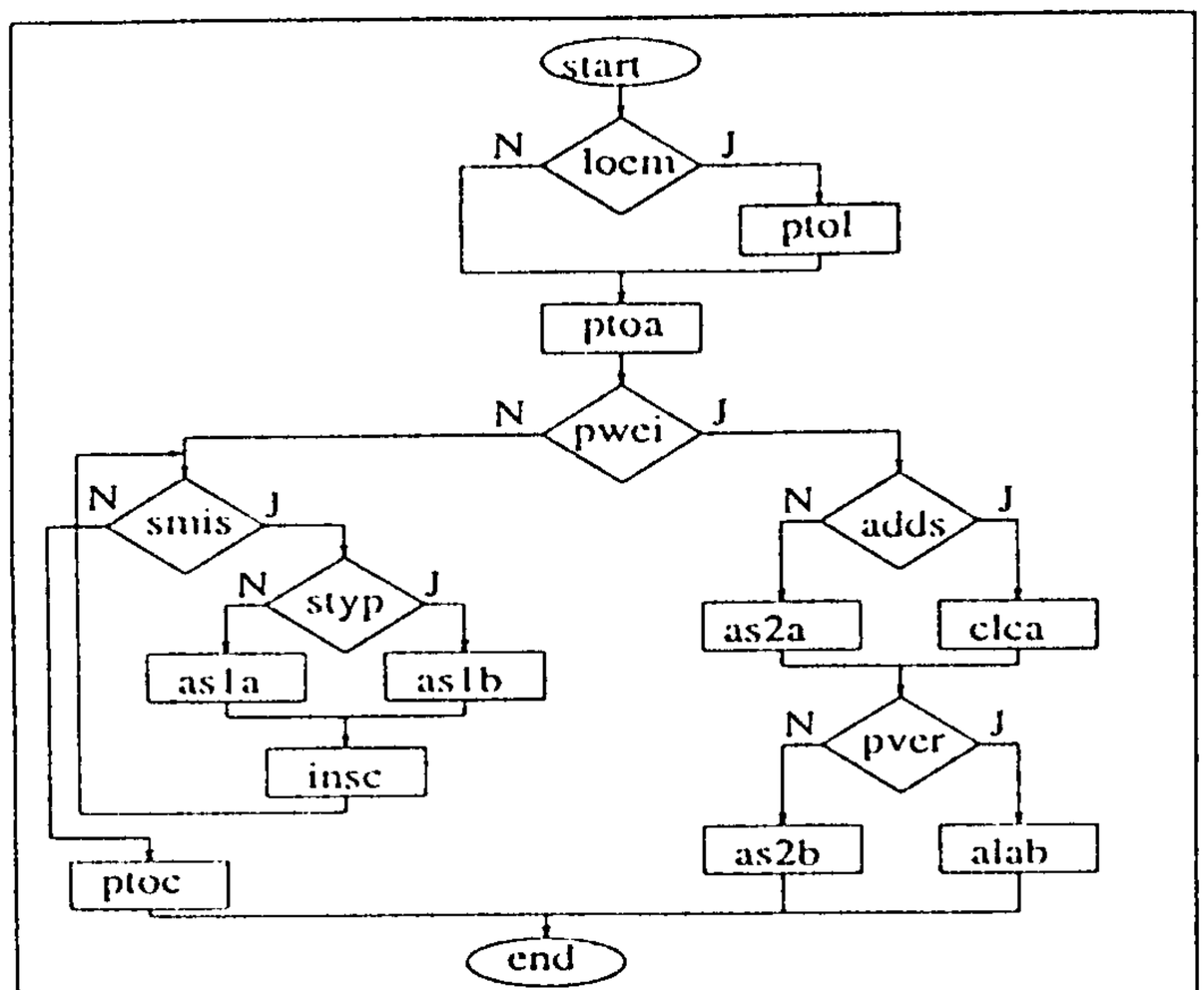
   FOR $i := 1$ TO last_bi_class DO
   - Assign $p[k]$ to variable *reg_expr*.
   - Replace inside *reg_expr* $\mathcal{R}(I_k)$ by $bi(k,i)$.
   - Create in the output file an entry of the form *s(k, reg_expr)*.

A comprehensive example that illustrates the steps is given in the following section.

The implemented class-generator is unable to recognize *infeasible paths*. An infeasible path is a path for which no input exists that causes it to be traversed. The problem of recognizing infeasible paths is, in general, not solvable [11]. A path generation strategy aimed at reducing the effects of infeasible paths is given in [4].

4 Application

To depict test segment coverage, a manufacturing system composed of pallets, a components store, two industrial robots and a conveyor will be used as example. The manufacturing system moves semifinished products and subcomponents into a robot cell and assembles product variants. In the following the control program of the manufacturing system shown in Fig. 3 is described:




```
Abbreviations:
adds: add subcomponent?
alab: adhere label on casing
as1a: assemble subcomponent a with robot 1
as1b: assemble subcomponent b with robot 1
as2a: assemble subcomponent a with robot 2
as2b: assemble b in casing with robot 2
clca: close casing of product
insc: insert and tighten screws, actualize smis
lamb: lambda (empty word, left branch of loem)
loem: load-station empty
ptoa: load pallet, move pallet to assembly cell
ptoc: place product on conveyor
ptol: move pallet to load-station
pver: product version x?
pwei: product weight greater than 4 kg?
smis: subcomponent missing?
styp: subcomponent of type b?
```

Figure 3. Flowgraph of an assembly program

If the load station is empty, a pallet is moved to the load station (`ptol`). After loading the pallet and moving the pallet to the assembly cell (`ptoa`) the assembly order is assigned to the adequate robot. If the product weight is not greater than 4 kg, robot 1 is used, otherwise the order is passed to robot 2. Depending on the product variant to be assembled, robot 1 assembles zero or more subcomponents on the product and places the finished product on a conveyor. Robot 2 has to execute two tasks. The tasks depend on the product variant (see Fig. 3).

4.1 Generation of s-classes

As each branch contains at most one statement, for simplicity, the branches are denoted in this example by the corresponding statements. Let $C = 8$ and '#n' denote test segment I_n. The generated segments corresponding to the program in Fig. 3 are following:

```
segment(0,(ptol + lamb) ptoa #1).
segment(1,(((as1a + as1b)insc)* ptoc) +#2).
segment(2,(as2a + clca)(as2b + alab)).
```

Next the steps for the generation of the s-classes are applied for $\mathcal{R}(I_2)$.

- The first step generates following bi-classes:

  ```
  bi(2,1)= as2a as2b,
  bi(2,2)= as2a alab
  bi(2,3)= clca as2b,
  bi(2,4)= clca alab.
  ```

- At step 2, first $\mathcal{R}(I)$ is assigned to $p[2]$.
  p[2] = (ptol + lamb) ptoa ((((as1a + as1b)insc)* ptoc) + ((as2a + clca) (as2b + alab))).
  It has to be assured that if $w \in L(p[2])$, then w bi-covers one of the bi-classes of $\mathcal{R}(I_2)$. Let $w =$ *ptol ptoa as1a insc ptoc*. Then $w \in L(p[2])$, but w does not bi-cover any bi-class of $\mathcal{R}(I_2)$. In order to assure that all w with $w \in L(p[2])$ bi-cover a bi-class of $\mathcal{R}(I_2)$, the string

  `(((as1a + as1b)insc)* ptoc) +`

  is removed from $p[2]$, i.e. $p[2]$ = (ptol + lamb) ptoa (as2a + clca) (as2b + alab).

- At step 3, following four entries (s-classes) are created:

  ```
  s(2,(ptol + lamb) ptoa as2a as2b).
  s(2,(ptol + lamb) ptoa as2a alab).
  s(2,(ptol + lamb) ptoa clca as2b).
  s(2,(ptol + lamb) ptoa clca alab).
  ```

A graphical overview of the s-classes corresponding to the segmented program is given in Fig. 4. Each path from the beginning to the end of the graph corresponds to an s-class. Not complete expanded regular expressions are represented by thick arrows and complete expanded regular expressions (words) are represented by thin arrows. Whereas thin arrows denote exactly one subpath, thick arrows denote one or more subpaths. A thick arrow indicates that it is irrelevant for the s_8 coverage criterion which of the subpaths belonging to the arrow is traversed.

4.2 Test coverage

All s-classes of I_1 in Fig. 4 are covered, if for instance the following 7 paths are traversed during the test phase:

- ptol ptoa ptoc
- ptol ptoa as1a insc ptoc
- ptol ptoa as1b insc ptoc
- ptol ptoa as1a insc as1a insc ptoc
- ptol ptoa as1a insc as1b insc as1a insc ptoc
- ptol ptoa as1b insc as1a insc ptoc
- ptol ptoa as1b insc as1b insc as1b insc ptoc

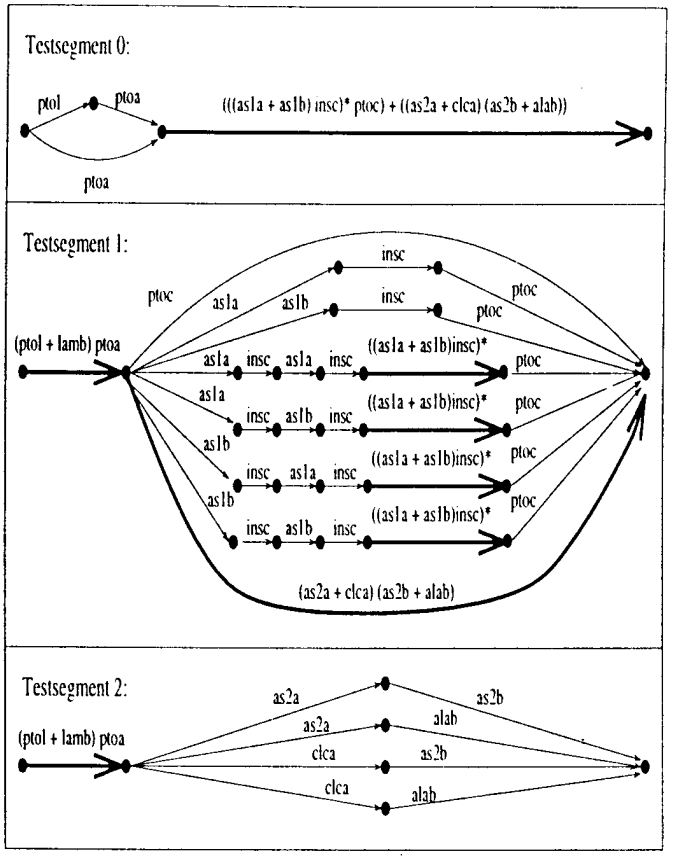


Figure 4. Test segment classes

The s-class `s(1,(ptol + lamb) ptoa ptoc)` can be covered by executing the instrumented program with following input values: `loem = TRUE, weight = 3 kg, smis = FALSE`.

The program in Fig. 3 contains a serious fault, since the traversal of path `ptol ptoa clca as2b` causes a collision. After closing the casing (`clca`) it is not possible to assemble a subcomponent inside the casing (`as2b`). As the fault involves the combination of two branches, the branch coverage criterion does not assure that this fault is discovered. The advantage of s_n coverage over branch coverage is that not only isolated branches, but also combinations of branches are considered.

We have implemented our approach on a Sun SPARC-classic workstation under X-Windows. The graphical interface and the files generated after activating the menu-bar *Generate all* are shown in Fig. 5. The file displayed on the left side contains the segmented program of the introduced example. The comment `/* #1b */` denotes the beginning of I_1, `/* #1e */` denotes the end of I_1. The file displayed on the right side contains the generated s-classes.

5 Validation of the approach

In this section we compare the fault detection capabilities of test segment coverage criteria (s_4 testing and s_{150} testing) with the fault detection capabilities of branch testing, multiple-condition testing, LCSAJ testing and all-uses testing. Boundary-interior path testing was not considered, since it is due to the number of required test cases in general intractable. First we focus on errors that are always revealed by the considered testing criteria in order to compare their least fault detection capabilities. For many errors it cannot be assured that they are detected by any of the considered testing criteria. In the second part of this section we compare therefore the fault detection capabilities of the considered testing criteria by means of probabilistic measures.

5.1 Least fault detection capability

Two program types with substantially different characteristics were considered: numerical and string processing programs. The four numerical programs selected appeared in issues of the *Communications of the ACM* and were published in separate entity titled *Collected Algorithms from ACM* [1]. We decided to select this programs because they have several substantial features: they were implemented by professional programmers, they contain real faults and are given with a brief specification, a description of discovered bugs as well as proposed corrections. Considered errors:

- **Program 19:** The termination criterion of a loop is incorrect.
- **Program 49:** The brackets inside an expression are incorrect.
- **Program 50:** The index of the array S is incorrect.
- **Program 106:** An assignment and two decisions are incorrect.

Each considered testing criterion assures the detection of all errors. These results are not surprising, since all errors are "easy" to detect. For the given numerical programs, branch testing is more efficient than the other considered criteria because it requires less test cases.

Next a program for string processing is considered. The C-program given in Fig. 6 is based on an Algol program written by Naur [14]. Naur's program converts a text consisting of words separated by `BLANKS` or by `NL` (new line) characters into a line-by-line form that fulfills following rules: "1) line breaks must be made only where the given text has `BLANK` or `NL`; 2) each line is filled as far as possible as long as 3) no line will contain more than `MAXPOS` characters."

Goodenough and Gerhart [8] discovered in Naur's program seven errors. We corrected four errors, since none

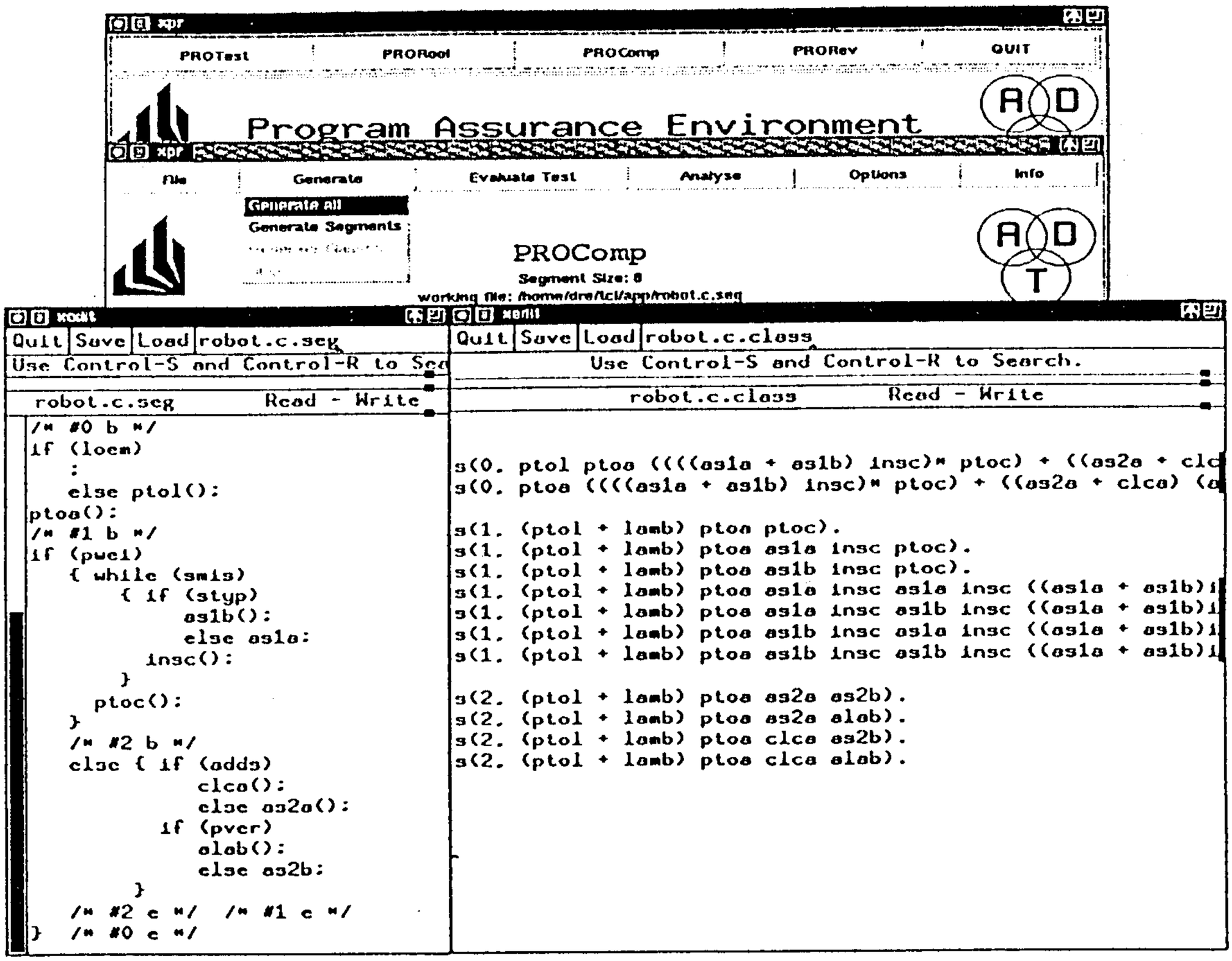


Figure 5. Graphical interface

of the considered testing criteria can assure their detection. Another error was discarded because it is "too easy" to detect. In other words, five errors were corrected because they are not suitable for comparing the fault detection capabilities of the considered testing criteria. Retained errors:

1. The first word of the output is preceded by a BLANK if the first character of the input is a BLANK or an NL.
2. Successive adjacent breaks (e.g. two blanks) are not condensed.

Furthermore we extended the specification by adding a fourth rule: the input text has to be followed by an end-of-text *(EOT)* character. All test cases were considered separately without checking whether a test case covers for instance more than one LCSAJ. The following table shows the number of test cases required to reach complete coverage of the criteria and indicates whether a criterion assures the detection (Y) of the first error or not (N).

	branch	mult.-condit.	LCSAJ	all-uses	s_4	s_{150}
test cases	13	21	16	36	12	133
error 1	N	N	N	N	N	Y

Only s_{150} testing assures the detection of the first error. s_{150} testing reveals the error because it requires the traversal of all subpaths through the control-flow graph that correspond to the first two executions of the body of the do-while loop (see Definition 2.2). Therefore it requires a test case with an input-text that starts with a BLANK or an NL. As a BLANK from the input-text is immediately written to the output-file, the error is also detected if the second character is an EOT or the first word of the input-text is larger than MAXPOS. s_{150} testing requires more test cases than the other criteria, but none of the other criteria can assure the detection of the first error even if all branches, LCSAJ's, etc. are covered several times.

In order to discard the first error, (for simplicity) we extend the specification with a fifth rule: the input-text never starts with a BLANK or NL. Consider now the second error. The following table indicates whether a coverage criterion assures its detection.

	branch	mult.-condit.	LCSAJ	all-uses	s_4	s_{150}
error 2	N	N	Y	N	Y	Y

```
bufpos = 0;
outcharacter(NL);
fill = 0;
Alarm = 0;
do
{incharacter(CW);
 if ((CW != BLANK) && (CW != NL) && (CW != EOT))
    if (bufpos == MAXPOS)
       Alarm = 1;
    else {bufpos = bufpos + 1;
          buffer[bufpos] = CW;
         }
 else {for (k = 1; k <= bufpos; k = k + 1)
           outcharacter(buffer[k]);
       if (CW != EOT)
          {if (fill + 1 + bufpos <= MAXPOS)
              {outcharacter(BLANK);
               fill = fill + 1;
              }
           else {outcharacter(NL);
                 fill = 0;
                }
           fill = fill + bufpos;
           bufpos = 0;
          }
      }
}
while ((CW != EOT) && !Alarm);
```

Figure 6. Program for string processing

In this case, three criteria assure the detection of the error: s_{150} testing, s_4 testing and LCSAJ testing. The remaining criteria cannot assure the detection of the second error, since they do not require a test case in which the body of the for-loop is not executed. Note that the body of the for-loop is not executed if and only if `(bufpos = 0) and ((cw = BLANK) or (cw = NL))`.

The experiments demonstrated that branch testing assures the detection of many errors without a great deal of effort. They showed also that branch testing is not suitable for the detection of errors which occur only if a combination of conditions is fulfilled. Only the s_{150} coverage criterion assures the detection of all considered errors. Although the all-uses criterion requires significantly more test cases than LSCAJ testing and s_4 testing, it cannot assure the detection of the second error inside the string processing program. The reason for this disadvantage is that it requires merely the coverage of *one* path from each definition to all references. The coverage of *all* paths from each definition to all references (all-du-paths testing [16]) is like boundary-interior testing often not practicable. LCSAJ testing and s_4 testing assure the detection of the second error inside the string processing program with only few test cases, but they cannot assure the detection of errors (like the first error inside the string processing program) which occur only if a subpath composed of more than two adjacent branches is traversed.

```
scanf("%d %d %d %d %d",&x1,&x2,&x3,&x4,&x5);
if ((x1 != 3) && (x2 != 6))
   r1 = x1 - x2;
   else r1 = 11;
if (x3 > 5)
   r2 = x3;
   else r2 = -x3;
if (x4 = 3)
  r3 = 6;
   else r3 = 4;
if (x5 > 6)
   r4 = 10;
   else r4 = - x5;
printf("%d %d %d %d",r1,r2,r3,r4);
```

Figure 7. Program fragment

5.2 Probabilistic measures

Frankl and Weyuker proposed comparing testing criteria by means of two probabilistic measures [5, 6]. One measure indicates the likelihood of detecting a fault, the other measure indicates the expected number of faults detected. The input domain, i.e. the set of possible inputs, is divided into subdomains according to the considered testing criteria. The test suite is generated by independent random selection of one test input from each subdomain. According to Frankl and Weyuker [5, 6] for each subdomain the test cases are generated separately without checking whether a test case covers other subdomains. Furthermore it is assumed that the program and the corresponding specification are given.

Used abbreviations:
I : implementation,
S : specification,
D_i : subdomain i,
d_i : size of D_i,
m_i : number of failure-causing inputs in D_i.

The probability M_a that a test suite chosen using test coverage criterion $\mathcal{T}$ will expose at least one fault is [5, 6]:

$$\mathrm{M}_a(\mathcal{T}, I, S) = 1 - \prod_{i=1}^{n}(1 - \frac{m_i}{d_i})$$

The expected number of failures detected (E) is defined as follows [5, 6]:

$$\mathrm{E}(\mathcal{T}, I, S) = \sum_{i=1}^{n} \frac{m_i}{d_i}$$

Consider the program fragment shown in Fig. 7.

condition	id	input values	number of inputs
(r1 = 11) ∧(r1 + r2 + r3 ≥ 27)	F_1	$x_1 \neq 3,\ x_2 = 6,\ x_3 = 10,\ x_4 = 3,\ 0 < x_5 \leq 10$	90
	F_2	$x_1 = 3,\ x_2 = 6,\ x_3 = 10,\ x_4 = 3,\ 0 < x_5 \leq 10$	10
	F_3	$x_1 = 3,\ x_2 \neq 6,\ x_3 = 10,\ x_4 = 3,\ 0 < x_5 \leq 10$	90
(r1 < 11) ∧ r1 + r3 + r4 ≥ 25	F_4	$x_1 = 10,\ x_2 = 1,\ 0 < x_3 \leq 10,\ x_4 = 3,\ x_5 > 6$	40

Figure 8. Failure-causing inputs

Let $((r1 = 11) \wedge (r1 + r2 + r3 < 27)) \quad \forall \quad ((r1 < 11) \wedge (r1 + r3 + r4 < 25))$ be the corresponding specification and let the variables x1, x2, x3, x4 and x5 be of the type `integer`. Suppose that the following precondition is fulfilled: $0 < xi \leq 10$ for all $i \in [1,5]$. The input values that cause failures are given in Fig. 8.

Example:
Let $x_1 = 3, x_2 = 5, x_3 = 10, x_4 = 3$ and $x_5 = 2$. In this case $r1 + r2 + r3 = 11 + 10 + 6 = 27$ (see row with id F_3 in Fig. 8), i.e. following expression of the specification is not fulfilled: $((r1 = 11) \wedge (r1 + r2 + r3 < 27))$.

All subdomains required to perform branch, multi-condition, LCSAJ, all-uses and test segment testing are given in Fig. 9. Note that only variables with additional restrictions are contained in column *subdomain*. Consider for instance subdomain D_4 of Fig. 9. D_4 comprises all input values of F_1, F_2 and F_3 (see Fig. 8) as well as 20 inputs of F_4. As the inputs corresponding to F_1, F_2, F_3 and F_4 are in this case disjoint, D_4 contains $90 + 10 + 90 + 20 = 210$ failure-causing inputs (m_4). In Fig. 10 for each testing criterion the corresponding subdomains, the required number of test cases, the value of M_a and the value of E are given.

Branch coverage is achieved by covering the subdomains $D_1 - D_8$ (see Fig. 10), therefore

$$M_a(branch, I, S) = 1 - \left(1 - \frac{m_1}{d_1}\right) \cdots \left(1 - \frac{m_8}{d_8}\right)$$

and

$$E(branch, I, S) = \frac{m_1}{d_1} + \cdots + \frac{m_8}{d_8}.$$

The values of M_a and E for branch testing and s_4 testing can be compared directly, because both testing criteria require the same number of test cases. For the given example, s_4 testing has greater values for M_a and E than branch testing. However, examples can be constructed where branch testing has greater values for M_a and E than a test segment criterion with the same number of test cases. More significant is the fact that even for this small example with restricted input values ($0 < x_i \leq 10$) the values of M_a and E of the first five testing criteria are smaller than 0.1 (see Fig. 10). A greater value for M_a can be obtained by selecting by means of the variable C a test segment criterion with greater test segments. In order to compare the value M_a of s_{16} testing with the corresponding value of branch testing, branch testing is performed by selecting two test inputs from each subdomain. The probability M_b that a test suite chosen by selecting randomly k inputs from each subdomain of a test selection criterion $\mathcal{T}$ is [5]:

$$M_b(\mathcal{T}, I, S, k) = 1 - \prod_{i=1}^{n} \left[1 - \left(\frac{m_i}{d_i}\right)\right]^k$$

Consider Fig. 11 and Fig. 10. Whereas $M_b(branch, I, S, 2)$ is approximately two times greater than $M_a(branch, I, S)$, $M_a(s_{16}, I, S)$ is approximately six times greater than $M_a(s_4, I, S)$. The two approaches differ in that both branch testing strategies have the same subdomains and s_4 testing has other subdomains than s_{16} testing. The results of multiple-condition testing, LCSAJ and all-uses testing for the given example are similar to branch testing in the sense that the usage of two instead of one test case per subdomain approximately doubled the value of the corresponding M. Consider Fig. 9 and Fig. 10. s_4 testing has 8 subdomains ($D_{12} - D_{19}$) and s_{16} testing has 16 subdomains ($D_{20} - D_{35}$). Subdomains $D_{20} - D_{35}$ have more restrictions than subdomains $D_{12} - D_{19}$. By adding restrictions the subdomain size is decreased. Consider in Fig. 9 subdomain D_{19} and the values $d_{19} = 6000, m_{19} = 114$ and $\frac{m_{19}}{d_{19}} = 0.019$. Additionally to the restrictions of D_{19}, the subdomain D_{25} contains the restrictions $((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5)$. As d_{25} and d_{19} contain the same number of failure causing inputs, $\frac{m_{25}}{d_{25}}$ is considerably greater than $\frac{m_{19}}{d_{19}}$.

To sum up the validation by probabilistic measures, a significant advantage of test segment testing over the other four testing strategies using n test cases from each subdomain is that the subdomain sizes are reduced and therefore better values for M can be achieved, i.e. higher probabilities for detecting at least one fault can be achieved.

id	subdomain	d	m	$\frac{m}{d}$	$1-\frac{m}{d}$
D_1	$(x_1 = 3) \vee (x_2 = 6)$	19000	190	0.0100	0.9900
D_2	$(x_1 \neq 3) \wedge (x_2 \neq 6)$	81000	40	0.0005	0.9995
D_3	$x_3 \leq 5$	50000	20	0.0004	0.9996
D_4	$x_3 > 5$	50000	210	0.0042	0.9958
D_5	$x_4 \neq 3$	90000	0	0	1
D_6	$x_4 = 3$	10000	230	0.0230	0.9770
D_7	$x_5 \leq 6$	60000	114	0.0019	0.9981
D_8	$x_5 > 6$	40000	116	0.0029	0.9971
D_9	$(x_1 = 3) \wedge (x_2 \neq 6)$	9000	90	0.0100	0.9900
D_{10}	$(x_1 \neq 3) \wedge (x_2 = 6)$	9000	90	0.0100	0.9900
D_{11}	$(x_1 = 3) \wedge (x_2 = 6)$	1000	10	0.0100	0.9900
D_{12}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 \leq 5)$	7500	0	0	1
D_{13}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5)$	7500	95	0.0127	0.9873
D_{14}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 \leq 5)$	40500	20	0.0005	0.9995
D_{15}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 > 5)$	40500	20	0.0005	0.9995
D_{16}	$(x_4 \neq 3) \wedge (x_5 > 6)$	36000	0	0	1
D_{17}	$(x_4 \neq 3) \wedge (x_5 \leq 6)$	54000	0	0	1
D_{18}	$(x_4 = 3) \wedge (x_5 > 6)$	4000	116	0.0290	0.9710
D_{19}	$(x_4 = 3) \wedge (x_5 \leq 6)$	6000	114	0.0190	0.9810
D_{20}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 \leq 5) \wedge (x_4 = 3) \wedge (x_5 > 6)$	380	0	0	1
D_{21}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 \leq 5) \wedge (x_4 = 3) \wedge (x_5 \leq 6)$	570	0	0	1
D_{22}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 \leq 5) \wedge (x_4 \neq 3) \wedge (x_5 > 6)$	3420	0	0	1
D_{23}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 \leq 5) \wedge (x_4 \neq 3) \wedge (x_5 \leq 6)$	5130	0	0	1
D_{24}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5) \wedge (x_4 = 3) \wedge (x_5 > 6)$	380	76	0.2000	0.8000
D_{25}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5) \wedge (x_4 = 3) \wedge (x_5 \leq 6)$	570	114	0.2000	0.8000
D_{26}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5) \wedge (x_4 \neq 3) \wedge (x_5 > 6)$	3420	0	0	1
D_{27}	$((x_1 = 3) \vee (x_2 = 6)) \wedge (x_3 > 5) \wedge (x_4 \neq 3) \wedge (x_5 \leq 6)$	5130	0	0	1
D_{28}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 \leq 5) \wedge (x_4 = 3) \wedge (x_5 > 6)$	1620	20	0.0123	0.9877
D_{29}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 \leq 5) \wedge (x_4 = 3) \wedge (x_5 \leq 6)$	2430	0	0	1
D_{30}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 \leq 5) \wedge (x_4 \neq 3) \wedge (x_5 > 6)$	14580	0	0	1
D_{31}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 \leq 5) \wedge (x_4 \neq 3) \wedge (x_5 \leq 6)$	21870	0	0	1
D_{32}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 > 5) \wedge (x_4 = 3) \wedge (x_5 > 6)$	1620	20	0.0123	0.9877
D_{33}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 > 5) \wedge (x_4 = 3) \wedge (x_5 \leq 6)$	2430	0	0	1
D_{34}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 > 5) \wedge (x_4 \neq 3) \wedge (x_5 > 6)$	14580	0	0	1
D_{35}	$(x_1 \neq 3) \wedge (x_2 \neq 6) \wedge (x_3 > 5) \wedge (x_4 \neq 3) \wedge (x_5 \leq 6)$	21870	0	0	1
D_{36}	$(x_3 \leq 5) \wedge (x_4 \neq 3)$	45000	0	0	1
D_{37}	$(x_3 \leq 5) \wedge (x_4 = 3)$	5000	20	0.0040	0.9960
D_{38}	$(x_1 \neq 3)$	90000	130	0.0014	0.9986
D_{39}	no restriction	100000	230	0.0023	0.9977

Figure 9. Subdomains of the program fragment

coverage criterion	subdomains	test cases	M_a	E
branch	$D_1 - D_8$	8	0.0423	0.0429
multiple-condition	$D_2 - D_8, D_9 - D_{11}$	10	0.0614	0.0629
LCSAJ	$D_1 - D_8, D_{12}, D_{13}, D_{16}, D_{17}, D_{36}, D_{37}, D_{39}$	16	0.0622	0.0638
all-uses	$D_1, D_3, D_4, D_7, D_{38}, D_{39}$	18	0.0773	0.0800
s_4, 2 seg.	$D_{12} - D_{19}$	8	0.0605	0.0617
s_{16}, 1 seg.	$D_{20} - D_{35}$	16	0.3757	0.4246

Figure 10. M_a and E values of the coverage criteria

coverage criterion	test cases	$M_b(C, I, S, 2)$
branch	16	0.0829
multiple-condition	20	0.1190
LCSAJ	32	0.1188
all-uses	36	0.1461

Figure 11. M_b values of the coverage criteria

6 Conclusion

A control-flow based approach for controlling test coverage has been introduced. The approach is based on boundary-interior path testing and regulates the thoroughness of test coverage by means of program segmentation according to a parameter C which determines the size of the test segments. The thoroughness of test coverage can be adjusted therefore easily to the needs of the tester or the projected duration of the test-phase. Coverage criteria constructed by means of the segmentation approach are stronger than branch coverage, since (inside test segments) also combinations of branches are considered. The number of generated test segments has been proposed as metric in order to indicate the thoroughness of an s_n coverage criterion.

The fault detection capabilities of test segment coverage criteria have been compared with the fault detection capabilities of branch testing, multiple-condition testing, all-uses testing and LCSAJ testing using n test cases from each subdomain. The comparison has shown that, compared with the other four testing strategies, greater fault detection probabilities can be achieved if test segment coverage is used. The approach has been implemented for testing structured C-programs as well as not structured C-programs, i.e. C-programs containing *break* and *continue* statements. However, the approach can be used also for other imperative programming languages. It is planned to develop an approach for optimizing the segmentation parameter C according to application-specific test-requirements by means of an genetic algorithm and fuzzy-logic.

References

[1] Association for Computing Machinery. *Collected Algorithms from ACM*, New York, Vol. 1 1980.

[2] F. Belli and J. Dreyer. Systems specification, analysis, and validation by means of timed predicate/transition nets and logic programming. *Proceedings of the Sixth Symposium on Software Reliability Engineering*, pages 68–77, 1995.

[3] W. G. Bently and E. F. Miller. Ct coverage – initial results. *Software Quality Journal*, 2:29–47, 1993.

[4] A. Bertolino and Martina Marre. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, December 1994.

[5] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE*

Transactions on Software Engineering, 19(3):202–213, March 1993.

[6] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[7] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. *Proceedings of the Workshop on Software Testing*, pages 64–73, Banff, Canada, July 1986.

[8] J. B. Goodenough and S. L. Gerhard. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, pages 156–173, June 1975.

[9] P. M. Herman. A data flow analysis approach to program testing. *Australian Cumputing J.*, 8(3):92–96, November 1976.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[11] W. E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, 24:554–559, May 1975.

[12] W. E. Howden. Symbolic testing – design techniques, costs and effectiveness. National technical information service pb-268517, US National Bureau of Standards GCR77-89, Springfield, VA, 1977.

[13] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9:347–354, May 1983.

[14] P. Naur. Programming by action clusters. *BIT*, 9(3):250–258, 1969.

[15] S. C. Ntafos. On required element testing. *IEEE Transaction on Software Engineering*, 10:795–803, November 1984.

[16] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, April 1985.

[17] W. T. Vetterling, S. A. Teukolsky, W. H. Press, and B. P. Flannery. *Numerical Recipes — Example Book (C)*. Cambridge University Press, Cambridge, 1988.

[18] M. R. Woodward, D. Hedley, and M. A. Hennel. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6:278–286, May 1980.

Appendix

Let $\mathcal{A}$ be an alphabet, Q be a regular expression and $L(Q)$ the language denoted by Q. The regular expressions over $\mathcal{A}$ and the sets described by the regular expressions are defined as follows [10]:

$\emptyset$ is a regular expression and denotes the empty set. λ is a regular expression and denotes the empty word. $(\forall b \in \mathcal{A})$ b is a regular expression and $L(b) = \{b\}$. If Q and R are regular expressions in $\mathcal{A}$, so are

- Q^* (iteration, Kleene star operation),
- $Q\ R$ (concatenation, product),
- $Q + R$ (selection, union).

The abbreviation Q^+ is used in this work to denote $Q\ Q^*$.

Examples:

- $L((ab)^*) = \{\lambda,\ a\,b,\ a\,b\,a\,b,\ a\,b\,a\,b\,a\,b,\ \ldots\}$,
- $L((a+b)c) = \{a\,c,\ b\,c\}$.

Correspondences between C-programs and regular expressions are shown in Fig. 12. In order to use our approach also for testing not structured C-programs, *break* and *continue* statements are considered (see examples in Fig. 12). As the control-flow of programs which contain *goto*-statements can be represented by means of regular expressions, the introduced approach can be used also for programs with *goto*-statements.

program	regular expression
P; (statement)	S (symbol)
if (P_0) P_1; else P_2;	$S_0\ (S_1 + S_2)$
switch (P_0) case $x1$: $P1$; break; ... case xn : Pn; break;	$S_0\ (S_1 + S_2 + \cdots + S_n)$
while (P_0) do P;	$S_0\ (S\ S_0)^*$
do P; while (P_0);	$(S\ S_0)\ (S\ S_0)^*$
do $\{if\ (P_1)\ P_2$; else continue; P_3; $\}$ while (P_0);	$((S_1 S_2 S_3 S_0) + (S_1 S_0))^*$
do $\{if\ (P_1)\ P_2$; else break; P_3; $\}$ while (P_0);	$(S_1 S_2 S_3 S_0)^*\ (S_1 + \lambda)$

Figure 12. Statements and regular expressions