Proc. Estonian Acad. Sci. Engng., 1997, 3, 2, 127–142
https://doi.org/10.3176/eng.1997.2.06

KNOWLEDGE-BASED TEST GENERATION FROM SPECIFICATIONS

Jaak TEPANDI

Department of Informatics, Tallinn Technical University, Ehitajate tee 5, EE-0026 Tallinn, Estonia; e-mail tepandi@cc.ttu.ee

Received 18 July 1996, revised 20 August 1996, accepted 17 March 1997

Abstract. A method for specification-oriented generation of test data is proposed. The method is based on a systematic application of heuristic knowledge of typical error situations, planning of the testing process, minimization of test cases, and evaluation of the results of testing. The test generation algorithm relies on the testing problem specification and on the knowledge base. The specification describes the program to be tested and the external characteristics of the problem. The program is specified by its input/output Jackson structure diagrams augmented with the links representing additional knowledge of the problem. The knowledge base includes link patterns and expert system rules. The approach has been evaluated by prototyping it as an expert assistant and by performing four experiments. The experiments involved testing of a total number of 37 programs and interviewing human testers. The method can be extended to various types of software specification methods.

Key words: AI-supported information systems engineering, software process modelling and support, specification-based testing, test data generators, minimization of test data.

1. INTRODUCTION

Program specification and program text both provide valuable information for program testing [1, 2]. Although specification-based testing is one of the most important testing methods, research in the area of specification-oriented test data generation methods has lagged behind other research in testing automation [3]. The majority of automated test tools use the program-based approach [2–5]. This is understandable because the text of a program is certain to exist in a formal notation and can therefore be analyzed. However, this is not always the case with the specification: it is often missing, or is much less formal than the text.

The main disadvantages of the current specification-based testing are: (1) treatment of requirement combinations needs a large number of test runs, (2) it is difficult to find a set of test cases which maximizes the probability of finding errors [6]. Little attention is paid to three important topics: the test data generation based on the current specification notations [3], the use of specific knowledge of typical error situations, and the minimization of test data. Below we will discuss these in more detail.

Among the specification languages used for test data generation are finite-state automata, predicate calculus, abstract data types and high-level procedural languages [3, 7–9]. Typically, practical system analysis and design notations are not used for this purpose. Some reasons were mentioned above: these notations are often more or less informal, and the specification developed is not always sufficient to generate an output from a given input. Furthermore, the main aim of the current design techniques is to improve the quality of software products, thus diminishing the need for testing. However, this need will never vanish entirely. Specifications for complicated programs become obscure and clumsy [10]; they must also be verified and validated [11].

The range of valuable test data selection criteria is wide. These are primarily of general (for example, of syntactical or logical) nature: such as, all statements in the program must be executed at least once, the specification must be consistent. Such criteria provide a good basis for test generation. However, a rich body of empirical knowledge of testing [1] is rarely used in testing automation systems. Examples of such knowledge include suggestions like "If the program has to search for a given value in a file, then test a file with this value as the last one", "If the program has to deal with dates, then try February the 29th". Advances in artificial intelligence and, particularly, in expert systems have simplified application of this knowledge. However, although essential, research in the area of knowledge-based testing has received less attention [12].

So far, test minimization has not been a centre of interest. For that purpose, orthogonal Latin Squares have been proposed in [13]. Here we will apply this method to specification-based testing.

The next section discusses the testing scenario underlying our approach and proposes a principle of effective test data selection relying on heuristics and minimization. The third section introduces a test data generation method based on this principle. The final sections report the results obtained from the evaluation of the approach and state the conclusions.

2. THE TESTING SCENARIO

In real situations, the problem of "How to test a program?" is not pure. Rather, for instance, one has to test a program developed by a certain group of people, to be used in certain applications, to be tested within limited resources. A

minimization of testing resources is always essential. Thus we have a testing problem – a request to test a program using all the information available about it and minimizing the expenses.

To solve this testing problem, we will apply the testing process. Typically, it consists of establishing the goals of testing, passing through different testing phases (such as unit testing, integration testing), and evaluating testing results. The choice of tests for each phase cannot be entirely determined before the phase begins. For example, the modules, in which more errors have been found, should be tested more thoroughly. Therefore, a testing phase includes establishing the goals of the phase and iteration of testing steps. Each step consists of test planning, test data generation, test case design, test execution, evaluation of test results, and evaluation of the program.

The expert methods for developing the tests depend on the program, its environment, the expert background, and on other factors. One possible approach could be as follows. The goal of testing is to find bugs in the program with a minimal waste of resources. Thus, when an expert starts testing, s/he applies his/her experience to elaborate tests for the most error-prone parts of the program (compare with testing scenarios from [1]). Besides, s/he tries to minimize the amount of test cases. Similar ideas are used in technology and business. For example, fault tree analysis provides a systematic approach to the identification of high risk areas of technical systems [14]; business strategies are analyzed by asking specific questions concerning possible areas of failure [15]. We have made use of the same principles. They can be summed up as follows.

The testing scenario is based on a systematic application of heuristic knowledge of typical error situations, planning of the testing process, minimization of test cases, evaluation of the testing results and acquisition of new heuristics.

The testing scenario includes the following steps.

1. Plan: evaluate the program-independent (external) characteristics of the testing problem, establish the goals of testing, plan the level of details of the specification, plan the amount of test cases, and plan the testing process.

2. Specify: develop and edit the program specification (internal characteristics of the testing problem). This step may be omitted if the specification is inherited from the earlier stages of program development.

3. Assembly: generate test data with instructions on the elaboration of most effective tests.

4. Modify: elaborate test data, select correct output for the input data and vice versa.

5. Monitor: execute tests and evaluate test results.

6. Diagnose & Predict: evaluate the program to be tested, forecast the program behaviour, and decide whether to stop or continue testing.

This scenario illustrates once more the complexity of the testing task – it involves several generic categories of knowledge engineering applications [16].

3. THE TEST DATA GENERATION

The testing problem specification for the testing assistant includes two components: program specification and environment specification. The first defines internal, the second – external characteristics of the testing problem. Program specification is based on Jackson structure diagrams [17, 18] for its inputs and outputs. As these diagrams are not sufficient for effective generation of test data, we will augment structure diagrams with the links providing complementary information about the problem. The knowledge base includes the link patterns representing general knowledge of typical error situations in programs, and the expert system rules representing knowledge of the influence of the external characteristics on the testing process.

3.1. Structure diagrams: a basis for systematic generation of test data

The principles underlying our testing scenario do not fix the program specification formalism. We have chosen structure diagrams for the program input and output data to be the basis of this formalism, for they have proved to be a valuable tool for specifying the program and explaining it to the user. These diagrams specify the problem completely for only a special class of problems [19]. Thus, in general, they are not sufficient for deriving a program text.

A structure diagram is a labelled tree. The labels denote the names of the data structures and elements for program input or output, plus the node types. Each node belongs to one of the following three types: SEQUENCE, ITERATION, or SELECTION. A node of type SEQUENCE does not carry a type label. A node of type ITERATION is marked with an asterisk "*", a node of type SELECTION – with a symbol "o". The root must be of type SEQUENCE. All sons of one parent must be of the same type. Figure 1 shows an example of a structure diagram. It describes an input for a program processing rainfall data. According to this diagram, the input denoted RAINFALL consists of the iteration DAY-RFS of the daily rainfalls DAY-RF, each of which may be positive, zero, or negative. The iteration is followed by a sentinel EOD, indicating the end of input. The iteration may be repeated for zero or more times, thus the shortest input described by the diagram in Fig. 1 comprises only the EOD element. Denoting POSITIVE as P, ZERO as Z, NEGATIVE as N and EOD as E, this sequence may be described as <E>. Another possible example input described by this diagram is <PPE>, a sequence of two positive elements followed by EOD. Some more examples include <PPPNZPZPZNE>, <ZZZZE>, <ZPNNPPZZE>, and so on.

To apply specification-based testing, one has to elaborate tests for the equivalence classes of input/output data, for boundary (degenerate) input/output values and for erroneous data [1]. On the basis of structure diagrams, it is possible to generate test data for each case [20, 21].

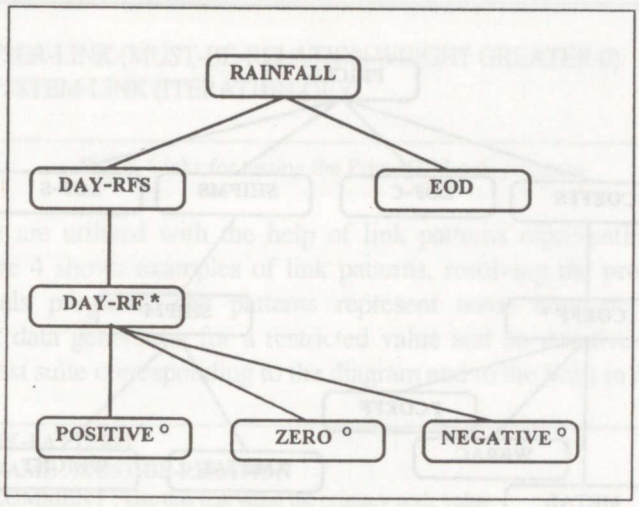

Fig. 1. A structure diagram.

The equivalence classes of input/output data correspond to the normal mode of executing the program. In this case, every iteration is repeated for a fixed number of times with some or all the possible selection combinations considered. For example, if an iteration is repeated twice, a possible sequence for the diagram in Fig. 1 will be <ZZE>, i.e. it will consist of two zero elements, followed by EOD. All the possible inputs generated for a twofold iteration include nine sequences: <PPE>, <PZE>, <PNE>, <ZPE>, <ZZE>, <ZNE>, <NPE>, <NZE>, <NNE>. The boundary values occur for the zero or one iteration, resulting in four sequences <E>, <PE>, <ZE>, <NE> for the diagram in Fig. 1. The erroneous data consist of all the normal mode and boundary tests with some data elements replaced by erroneous ones. Denoting an erroneous element (a character input, for example) by R, one has the following error situations: <RPE>, <PRE>, <PPR>, etc.

A situation where this method is not applicable may be illustrated by a specification for a PricesOfMetals program computing prices of metals for various types and weights (Fig. 2; the program is given in [22] and discussed in [23]). An error in the program (when an error is detected in the input data and an error message is printed, the wrong metal type is displayed) will never be noticed as long as the first shipment is correct and the metal with the incorrect weight is the same as the previous metal examined. It is difficult to generate test data that guarantee revealing this bug with the aid of structure diagrams only. We will augment structure diagrams with links between the nodes and apply knowledge of errors occurring together with certain types of links.

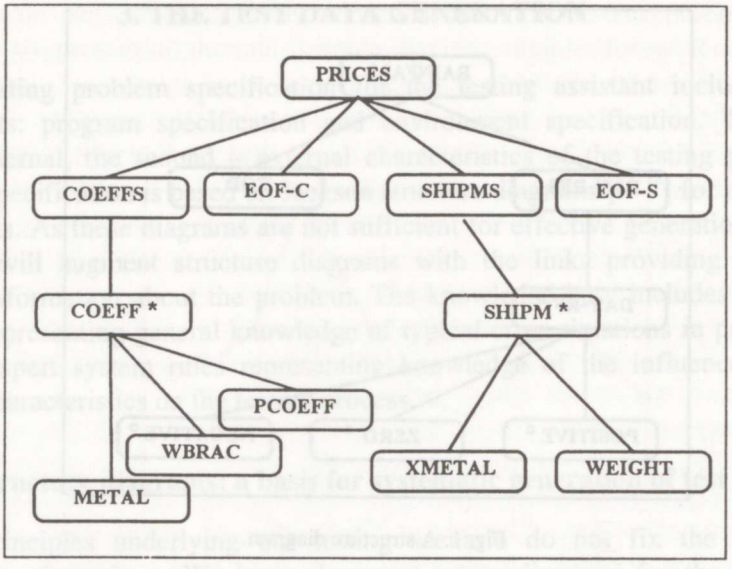

Fig. 2. A structure diagram for the PricesOfMetals program.

3.2. Links and link patterns: utilizing testing heuristics

There is a large evolving body of knowledge of typical error situations; some examples were given in the introduction. Most existing testing automation methods do not make use of it, reinvent it anew each time they have to test a program, or have this knowledge only in a built-in form. This may result in inefficient testing, in too large amounts of test data or in a testing system that is not able to adapt to changing programming environments.

To use this knowledge, we will augment structure diagrams with the links giving additional information about the problem. A link is determined by the link name, primary node name and (optionally) parameters. The primary node name expresses the user's view on the most important relationship between the link and the diagram. For example, the WEIGHT must be greater than zero, the EMPLOYEE names must be sorted in alphabetical order. Parameters may include, for instance, secondary node names, numbers, texts, expressions.

The links may be defined by the user or by the system. The (partially) augmented diagram for the PricesOfMetals program includes a user link MUST-BE-RELATION and activates a system link ITERATION-OF (Fig. 3). The MUST-BE-RELATION link specifies the primary node (WEIGHT) and two parameters (the relation name and a numerical value). The ITERATION-OF link has no parameters. Its primary node is a variable, thus any node in the diagram may be substituted for it.

```
USER-LINK (MUST-BE-RELATION WEIGHT GREATER 0)
SYSTEM-LINK (ITERATION-OF x)
```

Fig. 3. Links for testing the PricesOfMetals program.

The links are utilized with the help of link patterns representing heuristics of testing. Figure 4 shows examples of link patterns, resolving the problems with the PricesOfMetals program. The patterns represent some heuristic knowledge of effective test data generation for a restricted value and an iterative input. Figure 5 illustrates a test suite corresponding to the diagram and to the links in Figs. 3 and 4.

```
LINK-PATTERN
  NAME : MUST-BE-RELATION
  COMMENT : Use this link when the primary node value
        must be compared with a numerical value
  PRIMARY-NODE : x
  ASSOCIATED-WITH : x
  PARAMETERS : y z
  CONSTRAINTS : IS-LEAF(x)
  HEURISTICS :
    IF TESTING-MODE(NORMAL) THEN SET-PARTITION (Select x y z)
    IF TESTING-MODE(BOUNDARY) AND
        BELONGS-TO(y (LESS-OR-EQUAL GREATER-OR-EQUAL) )
    THEN SET-PARTITION (Select x = z)
        .....
    IF TESTS-WITH-ERRORS(YES) AND
        BELONGS-TO(y (GREATER GREATER-OR-EQUAL EQUAL) )
    THEN SET-PARTITION (&ERROR : Select x LESS THAN z)
        .....
    IF AMOUNT-OF-TESTS(LARGE) AND
        TESTING-MODE(BOUNDARY) AND
        BELONGS-TO(y (GREATER-OR-EQUAL GREATER) )
    THEN SET-PARTITION (Select x A LITTLE BIT GREATER THAN z)

LINK-PATTERN
  NAME : ITERATION-OF
  COMMENT : Use this link when you have an iteration of a node
        and want to test its subordinates (leaves) more thoroughly
  PRIMARY-NODE : x
  ASSOCIATED-WITH : x
  CONSTRAINTS : IS-FATHER-OF-ITERATED-NODE(x)
  COMPUTE : COLLECT-LEAVES-OF-NODE-SUBTREE(x y)
  HEURISTICS :
    SET-PARTITION (Select different   y values)
    IF AMOUNT-OF-TESTS(LARGE)
    THEN SET-PARTITION (Select equal iterated y values)
        .....
```

Fig. 4. Two link patterns.

```
TEST:
  (Select different   (METAL WBRAC PCOEFF) values)
     METAL WBRAC PCOEFF
     METAL WBRAC PCOEFF
  EOF-C
  (Select different   (XMETAL WEIGHT) values)
     XMETAL
     (Select WEIGHT GREATER 0) WEIGHT
     XMETAL
     (&ERROR : Select WEIGHT LESS THAN 0) WEIGHT
  EOF-S
TEST:
  (Select different   (METAL WBRAC PCOEFF) values)
     METAL WBRAC PCOEFF
     METAL WBRAC PCOEFF
  EOF-C
  (Select different   (XMETAL WEIGHT) values)
     XMETAL
     (&ERROR : Select WEIGHT LESS THAN 0) WEIGHT
     XMETAL
     (Select WEIGHT GREATER 0) WEIGHT
  EOF-S
```

Fig. 5. A test suite.

Every link pattern includes the link name, the associated node name, the primary node name and heuristics. In addition, it may include comments, the link parameters, constraints, procedures for computing the values to be used in the test generation, and actions. The primary node name and link parameters are substituted from the link. As shown in Fig. 4, the associated node may coincide with the primary node. It may also be computed using the primary node and parameters.

3.3. The environment specification and the rule base: minimizing the test data

The amount of generated test data may be very large. To minimize it, one must take into account also the external characteristics of the testing problem, such as the responsibility of the task, the quality of the program, the reputation of the producer. To manage the amount and choice of test data, we have established several control parameters which depend on the external characteristics. Each of the parameters has a well-defined effect on the test data generation process, so these effects are built into the test data generation algorithm; moreover, the parameters may be used in link patterns and expert

system rules. On the other hand, the choice of the parameters' values for a given environment specification is heuristic. The current implementation of the method uses the following control parameters.

1. The test data elaboration level (recommendations/diagram). Throughout this paper, we are discussing the structure diagram level generation of the test data if not explicitly stated otherwise. On the first level, only recommendations are given.
2. Mode of test data generation (normal/boundary).
3. Inclusion of errors into the data (yes/no).
4. Amount of test data (large/medium/small/an integer). The value may be an integer determining the maximum test suite size.
5. Number of iterations (an integer).
6. Mode of iteration (concatenation/combination).
7. Interaction between sequential subproblems (yes/no). The choice depends on whether sequential subparts of the specification interact or not.
8. Combining mode (all combinations/Latin Squares method/each leaf at least once).

For example, the test suite in Fig. 5 is generated for the following control parameters:
TEST-ELABORATION-LEVEL(DIAGRAM), TESTING-MODE(NORMAL), TESTS-WITH-ERRORS(YES), AMOUNT-OF-TESTS(SMALL), NUMBER-OF-ITERATIONS(2), MODE-OF-ITERATION(COMBINATION), INTERACTION-BETWEEN-SUBPROBLEMS(YES), COMBINING-MODE(ALL).

3.4. The test data generation algorithm

The test generation algorithm utilizes the components of the specification in the following way. The structure diagram is used for systematic generation of test data. The links partition the data into complementary heuristics-based equivalence classes. The control parameters determine the type of the test data and keep the amount of data within required limits.

Figure 6 shows an algorithm that implements the principle "Systematic application of heuristic knowledge with minimization" for the structure diagram level generation of the test data. Here, the ROOT function returns the name of the root of the structure diagram. The function LEFT-SON(N) gives the name of the left son of node N or NIL if N is a leaf. Some comments on the algorithm will follow.

The SET-NODE-ENVIRONMENT procedure uses link actions to set the node processing environment. In particular, the test generation control parameters are used by default. A list of active recommendations is composed for each link associated with NODE. This list gives complementary information concerning elaboration of the test data if there is only one element in the list. An example of such a heuristics is (Select different (WEIGHT XMETAL) values) in Fig. 5. If there are several elements in the list, its effect will be additional

partitioning and therefore increasing amount of the test data. For example, the heuristics (Select WEIGHT GREATER 0) and (&ERROR: Select WEIGHT LESS THAN 0) partition the values of WEIGHT node into two distinct classes, the last of them including erroneous input.

```
procedure GENERATE-TESTS:
   Evaluate the control parameters;
   NODE-TESTS (ROOT, TEST-SUITE);
   Print test cases from TEST-SUITE
end GENERATE-TESTS.

procedure NODE-TESTS (NODE, TEST-SUITE):
   comment   Elaborate actions of links associated with NODE to set the current
                environment of NODE ;
   SET-NODE-ENVIRONMENT (NODE, ENV);
   comment   Use the structure diagram to form the component test suites;
   if LEFT-SON (NODE)=NIL
   then LIST-OF-SUITES:=(((NODE)))
   else
      SON-LIST:=list of all sons of NODE;
      LIST-OF-SUITES:=list of all non-empty test suites for all nodes from
                SON-LIST
   endif;
   comment   Use links to establish complementary partitions of the test data ;
   LINK-LIST:=list of links associated with NODE;
   LIST-OF-PARTITIONS:=list including one element - a list of active
                recommendations - for each link from LINK-LIST;
   comment Use the control parameters to combine the complementary partitions
                and sons' suites into TEST-SUITE ;
   FORM-SEQUENTIAL-SUITES(LIST-OF-SUITES, ENV,
                LIST-OF-SEQ-SUITES);
   if LIST-OF-SEQ-SUITES = ()
   then TEST-SUITE:=()
   else
      APPEND(LIST-OF-PARTITIONS, LIST-OF-SEQ-SUITES,
                PARTITIONS-AND-SUITES);
      COMBINATIONS(PARTITIONS-AND-SUITES, ENV, CSUITE);
      ELABORATE(CSUITE, ENV, TEST-SUITE)
   endif
end NODE-TESTS.
```

Fig. 6. The test data generation algorithm.

The FORM-SEQUENTIAL-SUITES procedure uses the list of the sons' test suites and the current environment to elaborate selections and iterations. Some examples follow. If LIST-OF-SUITES is empty, then the result is also an empty list. If the sons are of type SELECTION, then their test suites are merged. If the

son is of type ITERATION, then the elaboration is determined by the iteration control parameters.

Each of the LIST-OF-PARTITIONS and LIST-OF-SEQ-SUITES lists may be thought of as a sequence of selections of elements – instructions or test cases. These two lists are appended. The COMBINATIONS procedure takes the result and replaces the sequence of selections of elements by the selection of element sequences according to the combining control parameters. The last sequences are concatenated, so the depth of the structure diagram is reduced by one. The combining procedures include, for example, the following. If the testing resources are large, it may be suitable to generate all the combinations of selection elements. For medium resources, it may be useful to include all pairwise combinations of elements into TEST-SUITE. This is a generalization of the Latin Squares method for the test data generation proposed in [13]. If a small amount of test data is desirable, then it may be sufficient to ensure that each element occurs in the node's test suite.

The ELABORATE procedure adjusts erroneous tests and the exact amount of test cases. If the node is a root, the test cases with exactly one erroneous data element are included into the test suite [1]. Otherwise, the test cases with at most one error are retained. If the test suite size has been given an integer value, then the corresponding number of test cases is selected from the test suite.

4. EVALUATION OF THE APPROACH

The approach has been evaluated by prototyping it in PROLOG as a testing assistant and by performing four experiments.

The specific testing knowledge is represented in the assistant's knowledge base – in link patterns and in the expert system rule base (the assistant's rule base is presented in [20]). Link patterns are given in the form similar to that illustrated in Fig. 4. The rules are given in the following format: RULE (15 (TESTING-WRT-JOB IS GOOD IF OVERALL-TESTING IS SATISFACTORY AND RESPONSIBILITY IS LOW) 4). They are used to control the testing process and to evaluate the control parameters. For example, a rule RULE (30 (NUMBER-OF-ITERATIONS IS 0 IF TESTING-MODE IS BOUNDARY AND TESTING-RESOURCES IS LIMITED) 5) says that if the son is of type ITERATION, the testing resources are limited, and the data are generated in the boundary mode, then the result of the combination is an empty list (with weight 5).

We have made four experiments to evaluate the efficiency of the approach and the correspondence of the testing model proposed above to the way humans try to validate their programs. Part of the experiments involved the testing assistant, part of them required interviewing and hand simulation.

To evaluate the efficiency of the method, we have examined 37 programs and program fragments from various sources. Most of them – 20 programs and fragments – were taken from the "Common Blunders" section and its Appendix ("Points to Ponder") of [22]. The specifications for these programs were mostly not available, and hence we wrote them based on the clarifying English text and on the implementation. There were 14 buggy programs and six programs that needed defensive programming with a total number of 35 errors. The method demonstrated 31 errors definitely, while detection of remaining four errors (for example, comparing an input against some fixed value to determine end of input data) was shown to be dependent on the current computing environment. Errors of this kind can be detected by rules like "If floating point numbers are used to terminate iteration, then..." and utilized to improve the specification-based evaluation of program reliability. Eight link patterns were sufficient to detect all the errors. Only one of the programs (Payments, p. 107–109) required generation of tests on the basis of the output structure diagram.

The second experiment involved evaluation of specification-based testing efficiency in terms of program-based adequacy criteria. A random sample of 12 programs was selected from textbooks [24–26]. A first nontrivial program, if it existed, was taken from each 100 consecutive pages of the books. All the programs had clear English descriptions of their intended function, so it was easy to draw up the specifications. Hand simulation of testing these programs showed that as tutorial programs, they contained relatively few "real" errors (there were boundary mode errors in four programs), but were mostly sensitive to erroneous input. Two additional link patterns were required. The main result of the experiment was that the test data, generated from specifications by the proposed method, satisfied the branch adequacy criterion for all the programs from the above sample and the previous experiment.

The objective of the third experiment was to evaluate the performance of the method when compared to that of a human testing expert. We compared the test data given in [1] for five programs (TRIANGLE, DIMENSION, MTEST, DISPLAY, BONUS) with the data generated by TESTER (for DIMENSION, MTEST, and TRIANGLE) or by hand simulation (for DISPLAY and BONUS). The first conclusion was that these programs required ten new link patterns – that is relatively more than those discussed above. The reason for this can be as follows. We suppose that the programs can be distributed into similar (in some respect) classes. Within each class, the number of necessary new link patterns grows fast at the beginning, but then starts diminishing rapidly. This process was clearly observed in the previous two experiments. The programs under consideration are of different kind and/or nearer to real applications. Therefore, initially a greater number of new link patterns was required. The second conclusion from this experiment states that the test data generated by TESTER on recommendation level were close to those suggested in [1], although these data did not coincide. The differences were mainly due to the ordering of the

recommendations and allocation of recommendations to the testing modes. For example, the tests for the equivalence classes of DIMENSION program in [1] included the first equivalence class – a test with one array descriptor; this case was considered a boundary test by TESTER system.

The previous experiments confirm the efficiency of the proposed method and, consequently, the test data generation principle underlying it. This conclusion is supported by the observation that human expert validation of programs, business procedures and technical applications follows in many cases in the same way (cf. section 2). It is interesting whether this principle can be considered a model of human non-expert program testing. Our fourth experiment was designed with several hypotheses in mind. They stated that the human non-expert test generation process (1) is not systematic, (2) is systematically following the specification text, (3) is systematically following any formal specification, (4) utilizes heuristics, (5) does not make use of heuristics, (6) uses minimization, and (7) does not use minimization. The experiments involved two eight-person groups of third-year computer science students who had an experience in programming and packages, but were not very strong in program testing. The groups were given two different versions of the BONUS program specification [1]. The versions differed in the order of descriptions (e.g. EMPTAB first versus DEPTTAB first), and in the hints indicating higher need for some kind of tests (e.g. an experienced programmer maintaining the database and a beginner implementing the BONUS program in one specification, versus an insecure payroll system and an expert implementing the program in another).

The experiment demonstrated that 15 students out of total 16 selected test data in a systematic way, although the grounds for the systematization were not necessarily the same as those implemented in the testing assistant. Ten students utilized more than one criterion of systematization. The most frequently used criteria were: the intrinsic ordering of concepts (for example, in both groups six students considered DEPTTAB first – this order might have been suggested by a vision of the database schema behind the BONUS program); exhaustive utilization of heuristics-based suspicious situations (e.g. considering together all the empty file tests, or all tests, dealing with equal/unequal data values – in eleven cases); mode of test generation (in eight cases). The order of presentation did not seem to be of much importance – at least, as far as more influential factors were involved. No conscious test data minimization effort was observed, neither in the form of combining different heuristics into one test, nor in the form of exploiting the hints mentioned above. This indicates that minimization may require higher skills from the testing personnel and therefore more sophisticated support from the testing assistant than other test generation activities. The experiment also points out the potential usefulness of exploiting different specification models for the test data selection.

The method can be applied to various types of specification formalisms. The structure diagrams correspond to suitably bracketed regular expressions [19]. The

last serve as a basis or as a sublanguage in many specification methods. For example, both the Data Dictionary definitions and process specifications in Structured Analysis [27] are given in an equivalent notation. Many notations for Structured Programming (pseudocode or Structured English, Nassi–Schneiderman diagrams [28], etc.) may be described with the help of regular expressions. The sequence, selection and iteration statements are sufficient to express any flowchartable program logic [29]; so here is a good chance to integrate the algorithm into the specification. Real-time systems are designed with the aid of structure diagrams [18], distributed processes – as sets of regular expressions [30], and so on. To make a better use of system specifications from the earlier development phases, it is desirable to build the test generation mechanism into a CASE tool. Last but not least, use of the test data generation principle proposed above is not restricted to any specification method.

5. CONCLUSION

We have proposed a hypothesis that effective test data selection process may be characterized as a systematic application of heuristic knowledge with minimization. This hypothesis has been applied to specification-based testing. The knowledge base structure, the testing problem specification, and the test data generation algorithm have been elaborated, a testing assistant prototype has been implemented, and experiments have been performed. The experiments have demonstrated that the proposed method can effectively be used for program testing; that the test data generated from specifications by the proposed approach, in many cases also satisfy the branch adequacy criterion; that in many aspects, the performance of the method is close to that of the human expert; and that the principle "systematic application of heuristics" can be taken as a model of a certain class of a non-expert program testing activity (although the grounds for systematization may differ from those implemented in our system).

REFERENCES

1. Myers, G. J. *The Art of Software Testing*. Wiley, New York, 1979.
2. Pressman, R. S. Adapted by D. Ince. *Software Engineering. A Practitioner's Approach*. McGraw-Hill, London, 1994.
3. Ince, D. C. The automatic generation of test data. *The Computer J.*, 1987, **30**, 1, 63–69.
4. Jensen, R. W. and Tonies, C. C. *Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1979.
5. Deason, W. H., Brown, D. B., Chang, K.-H., and Cross II, J. H. A rule-based software test data generator. *IEEE Trans. on Knowledge and Data Engineering*, 1991, **3**, 1, 108–117.

6. EWICS TC7/ techniques for verification and validation of safety-related software. *Computers & Standards*, 1986, **4**, 2, 101–112.
7. White, L. J. Domain testing and several outstanding research problems in program testing. *INFIR*, 1985, **23**, 1, 53–68.
8. McMullin, P. R. and Gannon, J. D. Combining testing with formal specifications: a case study. *IEEE Trans. Software Eng.*, 1983, **SE-9**, 3, 328–335.
9. Richardson, D. J. and Clarke, L. A. Partition analysis: A method combining testing and verification. *IEEE Trans. Software Eng.*, 1985, **SE-11**, 12, 1477–1490.
10. Richter, C. A. An assessment of structured analysis and structured design. In *Proc. Int. Workshop on the Software Process and Software Environments.* New York, 1985, 75–83.
11. Boehm, B. W. Verifying and validating software requirements and design specifications. *IEEE Software*, 1984, **1**, 1, 75–88.
12. Brooks, F. P. No silver bullet – essence and accidents of software engineering. *Computer*, 1987, **20**, 4, 10–19.
13. Mandl, R. Orthogonal Latin Squares: an application of experiment design to compiler testing. *Commun. ACM*, 1985, **28**, 10, 1054–1058.
14. Uijta, H. Development of SUPKIT-II: computer aided fault tree analysis system. *J. Nuclear Sci. and Technology*, 1984, **21**, 8, 625–633.
15. Brickner, W. H. and Cope, D. M. *The Planning Process.* Winthrop, Cambridge, 1977.
16. Clancey, W. J. Heuristic classification. *Artif. Intell.*, 1985, **27**, 3, 289–350.
17. Jackson, M. A. *Principles of Program Design.* Academic, New York, 1975.
18. Jackson, M. *System Development.* Prentice-Hall, London, 1983.
19. Hughes, J. W. A formalization and explication of the Michael Jackson method of program design. *Software-Practice and Experience*, 1979, **9**, 3, 191–202.
20. Tepandi, J. J. A knowledge-based approach to the specification-based program testing. *Computers and Artif. Intell.*, 1988, **7**, 1, 39–48.
21. Roper, M. and Smith, P. A structural testing method for JSP designed programs. *Software-Practice and Experience*, 1987, **17**, 2, 135–157.
22. Kernighan, R. W. and Plauger, P. J. *The Elements of Programming Style.* McGraw-Hill, New York, 1974.
23. Budd, T. A. and Gopal, A. S. Program testing by specification mutation. *Comput. Lang.*, 1985, **10**, 1, 63–73.
24. Welsh, J. and Elder, J. *Introduction to Pascal.* Prentice-Hall, Englewood Cliffs, NJ, 1982.
25. Holt, R. C. and Hume, J. N. P. *Programming Standard Pascal.* Reston, Reston, Virginia, 1980.
26. Hill, L. A. Jr. *Structured Programming in Fortran.* Prentice-Hall, Englewood Cliffs, NJ, 1981.
27. DeMarco, T. *Structured Analysis and System Specification.* Prentice-Hall, Englewood Cliffs, NJ, 1979.
28. Nassi, I. and Schneiderman, B. Flowchart techniques for structured programming. *ACM SIGLPAN Notices*, 1973, **8**, 8, 12–26.
29. Boehm, C. and Jacopini, G. Flow diagrams, turing machines, and languages with only two formation rules. *Commun. ACM*, 1966, **9**, 5, 366–371.
30. Lauer, P. E., Shields, M. W., and Best, E. Design and analysis of highly parallel and distributed systems. In *Abstract Software Specifications.* LCNS 86, Springer, 1980, 451–503.

TEADMUSLIK TESTIDE GENEREERIMINE SPETSIFIKATSIOONI PÕHJAL

Jaak TEPANDI

On loodud testide genereerimise meetod, mis tugineb programmi spetsifikatsioonile. Meetodi aluseks on süstemaatiline tüüpiliste veaolukordade kohta käivate heuristiliste teadmiste rakendamine, testimisprotsessi planeerimine, testide minimiseerimine ja testimise tulemuste hindamine. Testide genereerimise algoritm põhineb probleemi spetsifikatsioonil ja teadmusbaasil. Spetsifikatsioon kirjeldab testitavat programmi ja selle väliseid omadusi. Programmi spetsifikatsiooniks on tema sisendi/väljundi struktuuridiagrammid, mida on täiendatud probleemi kohta lisainfot andvate seostega. Teadmusbaas sisaldab seoste mustreid ja ekspertsüsteemi reegleid. Vaadeldavat meetodit on hinnatud, luues testija assistendi prototüübi ja tehes neli eksperimenti. Eksperimendid hõlmasid kokku 37 programmi testimist ja testijate intervjueerimist. Meetodit võib kasutada erinevate tarkvara spetsifitseerimise formalismide puhul.