PROBE: A FORMAL SPECIFICATION-BASED TESTING SYSTEM

Ahmed Al-Amayreh
Department of Mathematics and Computers
Salalah College of Education
Sultanate of Oman

Abdullah Mohd Zin
Department of Computer Science
National University of Malaysia
Malaysia

Abstract

The aim of software verification is determining how well the software conforms to its specification. Testing is the most popular method of software verification. For software testing to be done effectively, there is a need to select proper test cases such that all aspects of the software can be tested. This paper describes a new approach for specification-based testing. The formal specification of the software is used as the basis for generating test cases. Instead of generating test cases directly from the formal specification, they are generated from an executable PROLOG specification, which is translated from the formal specifications of the software under test. An automatic testing system called PROBE has been developed based on the new approach.

Keywords: Specification-based testing, software testing, logic programming, Z formal notation, test case generation

1. INTRODUCTION

Testing plays a vital role in software development. It is a practical means of detecting program errors that can be highly effective if performed rigorously. Despite the major limitation of testing, that it can only show the presence of errors and never their absence, it will always be a necessary verification technique (Myers 1979).

Formal methods are useful for specifying and designing software. However, formal specifications can play an important role in software testing. It is impossible to test software without specifications of some kind. Goodenough and Gerhart (1975) note that testing based only on program implementation is fundamentally flawed. In the testing literature, little research has dealt specification-based testing.

Research on testing using model-based specifications such as VDM (Joans 1986) and Z (Spivey 1989) has followed increased interest. Hall (1988), an early proponent of specification-based testing, shows how to derive test data from Z specification.

In his approach, Hall generates the test cases from Z specification. His technique can be shown to generate test cases that are reliable and valid. However, the main problem with this technique is that it is not capable of being used for generating test cases automatically. The approach for generating test cases from formal specification developed by Al-Amayreh, Zin and Daraman...
PROBE: A Formal Specification-Based Testing System

(1997) is an improvement over the one described by Hall. Instead of generating test cases directly from the specification, the specification is translated first into PROLOG, then the PROLOG specification is used to generate the test cases. A system called PROBE has been built based on the new approach (Al-Amayreh 1997; Zin, Al-Amayreh and Foxley 1997). The overall structure of PROBE is shown in Figure 1.

2. THE Z INTO PROLOG TRANSLATOR (ZP)

The main function of this tool is to translate the program’s Z specification, which is produced from the requirement analysis phase of the software development process, into a PROLOG specification. This tool is required since the Z specification is not executable. PROLOG was chosen as the object language of ZP for the following reasons (Denney 1991; Zin 1993):

1. PROLOG is more widely available. It is available for most mainframes and even personal computers from several vendors, so it is practical to use in real-world engineering environments.

2. Both Z and PROLOG are based on first order predicate logic, so it is possible for automatic translation to be done from Z schemes into PROLOG clauses.

3. Z specification notations include abstract data types, algebraic specifications and formal grammars, which can be implemented directly in PROLOG.

4. Nondeterminism, which presents problems for functional specification languages, can be expressed naturally in PROLOG.

5. Specifications stated in PROLOG are executable.

It has been shown that the process of generating PROLOG specification from Z specification can be done almost automatically. A system for generating PROLOG specification from Z specification is described by Zin. With the availability of an automatic Z into the PROLOG translator, the problem of generating test cases from Z formal specification can now be reduced into the problem of generating test cases from PROLOG specification.

3. THE TEST CASE GENERATOR

The Test Case Generator (TCG) is the core of the PROBE system. The main function of this tool is to generate almost adequate test cases from the PROLOG specification produced by ZP. It has been argued that PROLOG is well-suited tool for generating test cases from formal specification (Denney 1991; Zin 1993). Because of the suitability of PROLOG and equally because the specification is in PROLOG, PROLOG is used as a tool to generate the test cases from the PROLOG specification. One reason that makes PROLOG suitable to generate test cases is its ability to provide partitions of the variables’ domains automatically.
However, generating test cases from PROLOG specifications is not without problems. Backtracking, term ordering, instantiation, and cut predicate are obstacles that confront those who want to generate test cases from PROLOG specifications.

3.1 Backtracking

Backtracking is one of the powerful features in PROLOG. But in the test case generation process, uncontrolled backtracking will generate test cases from specific clauses infinitely and never get around to generating other cases from the rest of the clauses. In other words, due to its depth-first strategy, the standard PROLOG interpreter does not go beyond the first infinite clause. To generate adequate test cases, all clauses would have to be explored.

In order to force the backtracking and generate test cases from all clauses in the specification, a meta-interpreter (an interpreter for a language written in the language itself) is used instead of the standard PROLOG interpreter. Using a meta-interpreter, the number of iterations in each clause will be constrained to a specific number. This number is called *adequacy level*. Adequacy level is an integer number representing the maximum count of backtracking iterations in a clause during the generation of any test case. A proper adequacy level must be selected. If the selected adequacy level is too low, some of the test cases will be missed. If the selected adequacy level is too high, some of the test cases will be repeated. Some mechanism is still required to control and monitor the execution: stop a clause execution (if the adequacy level is reached) and start another clause, and so on. To handle this problem a deterministic automaton is used (Denney 1991). This automaton enables the meta-interpreter to monitor and control its progress and determines where it is and where it must go by recording the arcs of the automaton it crosses. The automaton can be generated automatically during the execution by translating the *goal reduction states* into automaton states. Goal reduction is the mechanism taken by PROLOG to solve goals.

3.2 Term Ordering

Bad term ordering in a clause executed by the standard PROLOG interpreter can return an erroneous result. This is caused by bad term ordering results during the execution of insufficiently instantiated clauses, or it could be caused by calling a logic program with a level of variable instantiation different from what is needed. This problem is solved by using a *constraint list*. If the clause is sufficiently instantiated it can be evaluated immediately. In this case there is no problem. If the clause is insufficiently instantiated, it will be referred to as a constraint and added to the constraint list until it is instantiated sufficiently. The meta-interpreter will backtrack repeatedly and evaluate any sufficiently instantiated clauses and then remove them from the list.

3.3 Instantiation

Instantiation occurs when a predicate is called with one of its input arguments insufficiently instantiated. The input argument should be a non-variable argument (bounded variable).

In order to overcome this problem, the meta-interpreter will generate a random number for any unbounded input argument. This random generator is intelligent enough to generate a proper random value.

3.4 Cut Predicate

The control predicate cut (!) in PROLOG needs special care when writing a meta-interpreter; it is notoriously tricky to deal with this in PROLOG interpreters. Actually, cut predicate has no place in PROLOG code intended for use as a software specification (Denney 1991). In the PROBE system, however, the PROLOG specification is not written directly. Instead, it is translated automatically from Z specification. So, it is assumed that the specifications may contain cuts. Therefore, the meta-interpreter must have the ability to handle this case.
The idea of handling this case basically starts from assigning a unique number to each clause in the specification path. This number is called clause level and it is generated and assigned automatically by the meta-interpreter. Once the meta-interpreter finds a cut predicate in the specification, the clause level will be saved (to return to it later) and then executes the cut predicate.

4. THE INPUT/OUTPUT ANALYZER

Test cases generated by TCG need some processing and modification in order for them handled by the tst diver so that they are suitable and compatible with the program under test. To perform this process and the modifications, an intelligent Input/Output Analyzer (IOA) was developed (Al-Amayreh 1997). As shown in Figure 1, the test cases generated by the TCG will be the input of the IOA. The functions of IOA can be divided into three categories:

- **Classification**: To classify each test case into two parts, the input part and the expected output parts, i.e., \(<I, E>\). The input part \(<I>\) will be used as input to the program under test, while the expected output part \(<E>\) will be compared to the actual output of the program. The comparison process will be performed by the Test Oracle (TO), which is a subtool of the test driver (TD).

- **Modification**: To modify the data types of the test cases so that they are compatible with the data types of the program’s input/output arguments.

- **Instantiation**: To substitute a random value for all uninstantiated variables in the test cases.

To perform its functions, IOA needs some information about the program under test such as the input arguments, the output arguments, and their data types and formats. This information must be defined in the input/output format file, which is created based on knowledge about the program or as a part of the specification.

The IOA will generate a table called Input/Output Table (IOT), which will contain the modified test cases. This table will be used later by the test driver to check the correctness of the program.

5. THE TEST DRIVER

The main function of the test driver is executing the program under test against the first part (the input part) of each test case in the IOT. The test driver then compares the program output with the second part (the expected output part) of each test case and reports the result of the whole testing process.

The program to be tested by the PROBE system must satisfy three conditions:

- Its Z specification is provided.
- Its executable version is available.
- It reads its input from the external file and writes its output into another external file.

6. CONCLUSION

PROBE is available and can be used to test and assess the quality of small-scale software such as programs used as assignments in a teaching programming environment. Extra work is needed to be done to improve the system so that it can handle more complicated software.

PROBE has been implemented on a PC running Microsoft Windows 95. It provides a Graphical User Interface, was implemented to be user-friendly, is graphically based, menu driven, and multi-windows interfaced.
7. REFERENCES


