Constructing an Automated Testing Oracle: An Effort to Produce Reliable Software

Luqi; Yang, Hongji; Zhang, Xiaodong

Constructing an Automated Testing Oracle: An Effort to Produce Reliable Software, with H. Yang, X. Zhang, Proc. of COMPSAC 94, Taipei, Taiwan, pp. 228-233.
http://hdl.handle.net/10945/42335

Constructing an Automated Testing Oracle: An Effort to Produce Reliable Software

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Hongji Yang and Xiaodong Zhang
De Montfort University
Leicester
England

Abstract

Achieving reliability in practice is becoming a dominant issue in software engineering. This paper describes part of a systematic approach to producing reliable software that is based on automated support for software testing. Our approach constructs an automated testing oracle based on software specifications written in the Z specification language. The contextual part of the specification describes the set of legal inputs to the program and the semantics part describes the meaning of the given input data. The potential roles of this approach in improving software reliability are discussed and some future opportunities are indicated.

1 Introduction

Reliability of software is starting to replace cost and schedule overruns as the dominant problem in software development. This issue is particularly prominent for systems whose malfunction may result in lost lives, injuries, or financial losses. For practical impact, we need sound automatable methods for software analysis, synthesis and certification that fit together to cover the entire software development and evolution process. It is practically impossible to produce error-free software systems that solve real (complex) problems by purely manual development methods because human error rates are too high.

Given that complete automation of software development and evolution is not feasible in the near future, some realistic research goals include the development of:

- A consistent set of accurate mathematical models covering all tasks in the software development process. This is needed to enable integration of the methods and tools for solving problems related to different aspects of software development.
- Automated synthesis methods for tractable subproblems that can be certified to always produce correct results. In cases where this is possible, this approach provides both reliability and productivity gains.
- Interactive computer-assisted synthesis methods that guarantee absence of errors for less tractable subproblems. This approach combines the benefits of human creativity with the accuracy provided by computer application of sound formal methods.
- Analysis and certification methods capable of detecting and diagnosing errors for subproblems that cannot be covered by error prevention techniques. If parts of the process must remain manual, then automated assistance for locating and removing errors and for certifying that no errors are left are needed for those aspects.

The work reported in this paper makes a modest contribution to the last of these research goals. We take the conventional approach of checking the reliability of programs via testing, and link it to formal specifications and automated decision support for determining whether or not observed test outputs conform to the specification.

One of the benefits of this approach is that it enables quality assurance efforts based on testing to be combined with correctness proofs for selected aspects of system behaviour that are determined to be particularly important or safety-critical. This lets the project management allocate concentrated skill and effort to the parts of the system where the consequences of failure are the most severe, and enables a common requirements document to support a mixture of both approaches for increasing software reliability.

2 Previous Work

Previous work on improving software reliability has mostly been focused on the code level, and practical applications have mostly relied on testing techniques. Some of the solid results in the area are surveyed here.

Successful execution of test sets constructed by random sampling from a probability distribution can provide statistical degrees of confidence in lower bounds on the mean number of executions between failures if actual input values correspond to the given probability distribution [6]. This kind of statistical reliability assurance is sufficient in cases where input distributions are predictable and non-zero failure rates can be tolerated. Statistical assurances are not sufficient for critical applications where even one failure is unacceptable. Statistical reliability measures can also be misleading if real input distributions are unstable or unknown, because there exist input distributions with high failure rates for any deterministic program that is not completely error-free.

For some specialised classes of programs, there exist methods for constructing a finite set of test cases whose successful execution can establish correctness of the program for all possible inputs [5,9]. This is not possible in the general case: testing can show the presence of software errors but it cannot certify their absence for unconstrained programs.

3 Future Opportunities Related to Software Testing

Advances in software analysis, synthesis and certification are essential for realising trusted software systems. Work in this area should be expanded beyond the traditional domains of testing code in a programming language and proving that programs satisfy formal specifications, to include software products at all stages of development from requirements analysis to system evolution.

Error prevention is possible both in cases where a software development task can be completely automated, and in cases where an automated tool realises all of the designer's decision in constrained ways that do not allow the designer to make a mistake, or that eliminate some kinds of mistakes. Some examples are meaning-preserving software transformations, which prevent divergences between specifications and the code [3], and syntax-directed editors, which prevent the creation of programs that do not conform to the syntax of the programming languages.

At the current state of the art, the entire software development process cannot be covered by tools and techniques that prevent all errors. This is a sign of immaturity rather than intrinsic difficulty. It is commonly believed that error prevention is more difficult than error detection, but this is not always the case. For example, checking whether an equational specification for an abstract data type is consistent and complete is an undecidable problem. Nevertheless, there exists an error prevention technique that guarantees that every specification that can be generated according to the rules is complete and consistent. These rules are simple enough to be applied and checked by a text editor, and they are sufficiently loose to accommodate the styles of specification that normally occur in practice [2].

Until we can effectively cover the entire process by error-preventing tools and techniques, we must also consider support for detecting, locating, and correcting errors. One key area for future research is in program testing. We need to expand the domains in which firm conclusions about satisfying specifications can be drawn from finite sets of test cases constructed by definite and effective methods, and to systematically check assumptions about the operating environment on which the design of a software system depends.

Software development deals with information of many different kinds, at different levels of abstraction. We summarise some types of software analysis, synthesis and certification problems (relating to software testing) that should be investigated.

Requirements Requirements consist of models of the problem, the expected environment of the proposed system, and specific goals for the system. An important aspect of requirements analysis is achieving and maintaining consistency as the analysts discover and record the requirements. A promising approach to this problem is providing automated support for calculating and maintaining derived properties and consequences of the requirements, and for tracing dependencies to determine the causes of conflicts and inconsistencies. Better algorithms for this process and primitives suitable for expressing and effectively maintaining dependencies in software requirements should be investigated [11]. This part is particularly difficult to automate because it is concerned with the transition from informal human needs to formal software documents.

Specifications Specifications define the responsibilities and interface of a proposed system. The primary measure of the adequacy of a specified system is whether it will meet the needs of the user. This question is best addressed by experimental rather than analytical techniques because it addresses the problem of checking the correspondence between a formalised specification and the actual and informal needs of the users. One way of approaching this problem is via prototyping and operational scenarios. Operational scenarios are common tasks in the customer's problem domain, expressed in the user's terms. Such scenarios serve as test cases for the specifications, whose purpose is to determine whether a proposed interface is adequate for carrying out all of the tasks the users will have to perform. Such a test passes if the facilities provided by the proposed system interface can be combined to carry out the tasks in the operational scenario, and provide a systematic means for exercising a prototype in a demonstration to the users.

Design Designs decompose a proposed system into a hierarchy of subsystems. The primary reliability property of a decomposition is whether it will correctly realise the specification at the next higher level. Testing can reveal some faults of this kind early in the process if we can test relative to the specifications of the subsystem before detailed implementations are available. More definite conclusions about this type of problem can be provided by mathematical proof techniques. The problem is easier to solve than the general proof of correctness problem at the code level because the module interconnection language can be considerably simpler than a programming language. Most of the analysis can be carried out at the specification level, since the problem is to check whether a given combination of specified components will satisfy the required properties of the composite.

Code More work is needed on the construction of finite complete test sets, and on characterising the set of faults whose absence is guaranteed by successful execution of the test set. A complete test set is a set of test cases which is guaranteed to detect any error in a particular well-defined class of errors. Automated techniques for constructing the required test oracles from the formal specifications of the code to be tested are an important component of this work. A weakness of statistical approaches to testing is the size of the test set required for certifying that systems have low failure frequencies, which makes manual examination of test results impractical. To apply these techniques in practice, we need automated methods for deciding whether or not the outputs produced by a test case conform to a specification.

Automatable methods for synthesis of efficient code from formal specifications via meaning-preserving transformations should also be investigated. Of particular interest are systems that can choose transformations without explicit human guidance, or with guidance from general declarative advice that can be formulated without explicit reference to the details of the current state of the derivation and does not require explicit human interaction during the derivation process.

Evolution Software modifications are notorious for introducing errors. Symbolic representations for the parts of the input space and the output space of a program affected by a given change to the code are useful for testing and evaluating a modification for conformance with the expected results. Software slicing is one of the relevant technologies for addressing this aspect of the problem.

4 Constructing an Automated Testing Oracle

From the above analysis, we conclude that formal methods and automation are two essential factors in improving the current situation of software testing. An effort should be made along this direction to produce reliable software. Therefore we consider how to construct an automated testing oracle based on software specification written in a formal specification language.

Specifications are of great importance in testing, for they describe what the software ought to do and must necessarily form the basis for the verification testing of the functionality of the software [7,8,10,12,16]. The use of a formal specification allows the development phase and test preparation to be performed concurrently. A formal specification must have a mathematical basis (usually formal logic) and employ a formal notation to model a system.

We can consider making use of formal specifications to construct an automatic oracle for checking or predicting the expected output of a software system.

A formal specification language is needed to describe both the syntactic (format and contextual parts) and semantic aspects of the software. The contextual part of a specification describes all the legal input to the program; the semantics part describes the meaning (expected output) of each given input data [6]. In terms of this idea, an oracle can be constructed from a context-free grammar, together with the related "meaning". Figure 1 shows a scheme of the technique.

A *parser* is a program which performs syntax analysis. A parser determines whether the stream of tokens from the input forms a valid sentence in the source language grammar. If so, a parse tree can be unambiguously derived. A parser generator is a complete programming language that can generate automatically a parse table. Some parser generators are widely used now. YACC [4] is an outstanding one that is widely available under Unix and some other operating systems. YACC takes a specification of a programming language grammar and semantic actions and produces an LALR(1) parsing table and a shift-reduce parser. The source program is read as a stream of tokens. A lexical analyser must be provided separately, typically using the Lex [4] lexical analyser generator.

A *test data generator* is a program that can generate syntactically and contextually correct test data (input data) through a right-most derivation of the attribute syntax given in the specification. Given the syntax and a representation for the test-domains, the algorithm for generating the test domain partition is produced for a particular specification. Typical test data are selected in terms of the given test domain partition. The work of this latter part does not come within the scope of this paper and hence is not described in detail here.

An *interpreter* is a program that simulates the behaviour of the software under validation, by 'execution' of the semantics, and produces the expected results relating to the test domain. It needs to give special semantics to a formal specification language and then interpret the formal specification language using this semantics. YACC can associate semantic actions with the parsers it generates. The user may specify actions that are executed whenever a rule (production) or part of rules, is recognised. These actions can return values and access values returned by previous actions. These features of YACC can also be used to write an interpreter in C.

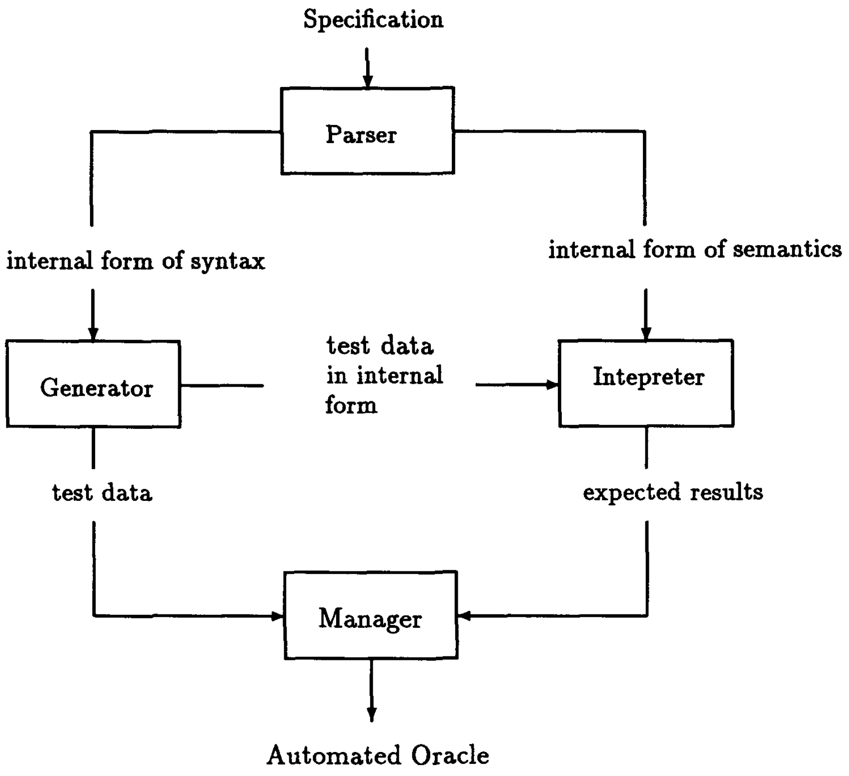

Figure 1: A Scheme of Automatic Oracle

A *manager* is a program that aggregates the generated test data and expected outputs, and prints a testing oracle in the form of a case table.

According to this idea, an expected output related test-domain can be obtained from a formal specification. The pre-requisite for realising this idea is a description of syntax of the specification language. The Z specification language has concrete syntax and semantics, and is almost fully defined [13,14].

5 Experimental Results

A prototype based on the above approach has been implemented on a SUN workstation and experiments have been carried out with the prototype system. An example of a banking system [13] for transferring money from a source account to a destination account is illustrated here. The input file of the specification (in LaTeX-like format which can be read by a preprocessing tool FUZZ [15]) can be written as:

```
\begin{schema}{Transfer}
  \Delta Bank \\
  amount?: \nat \\
  src?, dst?: ACCT\\
  report!: MESSAGE
\where
  src? \neq dst?  \\
  bal(src?) \geq amount? \\
  bal' = bal \oplus
    \{src? \mapsto bal (src?) - amount?, \\
    dst? \mapsto bal(dst?) + amount? \}  \\
  report! = "OK" \\
 \lor\\
  src? = dst?  \\
  report! ="Same \ account
                 for \ src \ and \ dst" \\
 \lor\\
  src? \neq dst?  \\
  bal(src?) \leq amount? \\
  report! = "Not \ enough \ money
                            in \ src" \\
\end{schema}
```

When this file is input to the system, if there is any syntax error in the file, the system will report the syntax error, otherwise, a specification and a test case table are printed as follows respectively:

Transfer

$\Delta Bank$
$amount? : \mathbb{N}$
$src?, dst? : ACCT$
$report! : MESSAGE$

$src? \neq dst?$
$bal(src?) \geq amount?$
$bal' = bal \oplus \{src? \mapsto bal(src?) - amount?,$
$dst? \mapsto bal(dst?) + amount?\}$
$report! = \text{"}OK\text{"}$
$\vee$
$src? = dst?$
$report! = \text{"Same account for src and dst"}$
$\vee$
$src? \neq dst?$
$bal(src?) \leq amount?$
$report! = \text{"Not enough money in src"}$

bal	bal $\geq$ Amount	bal $\geq$ Amount	bal < Amount
Amount?	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$
src?	src? $\neq$ dst?	src? = dst?	src? $\neq$ dst?
dst?	src? $\neq$ dst?	src? = dst?	src? $\neq$ dst?
report!	OK	same account ...	not enough money ...
bal'	bal' = bal $\oplus$ sth	bal' = bal	bal' =bal

The bal' = bal $\oplus$ sth in the above table is:
bal' = bal $\oplus$ {src? $\mapsto$ bal (src? - amount?, dst? $\mapsto$ bal(dst?) + amount?}

Having established the test-domains table, the next step is to select typical test cases from the set. The process is:

1. Assign a unique number to each equivalence class,
2. Until all valid equivalence classes have been covered by test cases, cover as many of the uncovered value equivalence classes as possible,
3. Until all invalid equivalence classes have been covered by test cases, write a test case that cover one, and only one of the uncovered invalid equivalence classes.

For a Z specification, selections will be based on the chosen state. In this banking system example, the state *bal* of the system consists of the balance of each account:

Bank

$bal : ACCT \mapsto N$

The expected outputs for the example are therefore *report!* and *bal'*.

Testing such a system must require choosing an initial state as well as input data. We might select the initial state to be

bal:
4256 $\rightarrow$ 200
8957 $\rightarrow$ 320

For this state, the first set of data can be chosen as below:

Test data 1:
src? = 4256 $\rightarrow$ 200
dst? = 8957 $\rightarrow$ 320
Amount := 100

Test data 2:
src? = 4256 $\rightarrow$ 200
dst? = 4256 $\rightarrow$ 320
Amount := 100

Test data 3:
src? = 4256 $\rightarrow$ 200
dst? = 8957 $\rightarrow$ 320
Amount := 500

A test case table is thus obtained:

	TD$_1$	TD$_2$	TD$_3$
bal	as above	as above	as above
amount	100	100	500
src?	2546	2546	2546
dst?	8957	2546	8957
report!	ok	same account	not enough money
bal'	4256 $\rightarrow$ 100 8957 $\rightarrow$ 420	not changed	not changed

The above table shows that the expected outputs should be:

Test data 1: report! =ok
bal':
src' = 4256 $\rightarrow$ 100
dst' = 8957 $\rightarrow$ 420
Amount := 100

Test data 2:
report! = same account
bal' = bal not changed

Test data 3:
report! = not enough money
bal' = bal not changed

6 Conclusion

The above example shows that the expected output can be automatically given in the oracle table when test data are selected from the input conditions.

The advantage of this approach is in the generation of an oracle which is functionally independent of any human decisions. This provides a strong foundation upon which a complete testing system can be built, i.e., by adding test case generation and gathering test coverage information. Additionally the system can provide motivation for the generation of a formal specification during the software development cycle. The system integrates Z formal specification techniques with the process of software testing.

The implementation is independent of other tools, in particular a compiler. An oracle is generated to model the particular specification expressed, instead of requiring compiler extensions to drive test cases through the program. The current implementation of the system can automatically generate an oracle only for small and comparatively simple Z specifications.

The system can also be used for computer-assisted requirements validation. In this context, the system is used to develop a set of test cases from the initially proposed specifications, before an implementation is available. The expected outputs that are automatically derived by the system are checked against the informal expectations of a group of prospective users of the system rather than against the actual outputs produced by the implementation. This social process is used to check the validity of the specification itself, so that requirements adjustments can be made before implementation. Thus the test oracle generator can also be viewed as a software prototyping and requirements validation tool that operates on symbolic Z specifications rather than on concrete implementations.

7 Future Work

It has been seen from the previous discussions that constructing an automatic oracle using formal specification is potentially of interest in software testing. To justify the arguments proposed in this paper, further implementation of the tool and experimenting with more examples (big examples, in particular) using the tool will be the main direction for future research. For example, a future research project may be to use the formal semantics of Z (such as denotational semantics and axiomatic semantics) to construct an automated testing tool. The formal semantics is its consequences for the practice of specification. It provides a foundation for a logical calculus for reasoning about a specification and deriving consequences from them. The successful application of formal methods in industry will be helped by software tools [1]. Using formal specifications means that all the early parts of the testing procedure are easy to carry out. Using formal specifications also means that valid results for each test case can be worked out with certainty. Thus the formal functional testing will become important in software testing.

References

[1] Abrial, J. R., Gardiner, P. H., Morgan, C. C. and Spivey, J. M., "A Formal Approach To Large Software Construction", Technical Report, Programming Research Group, Oxford, 1988.

[2] Antony, S., Forcheri, P. and Molfino, M., "Specification-based Code Generation", IEEE 23rd Hawaii International Conference on System Sciences, Jan. 1990.

[3] Bauer, F. L., Moller, B. B., Partsch, H. and Pepper, P., "Formal Program Construction by Transformation — Computer-Aided, Intuition-Guided Programming", *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, pp. 165–180 (February 1989).

[4] Bennett, J. P., *Introduction to Compiling Techniques: A First Course Using ANSI C, LEX and YACC*, McGraw-Hill, London, 1990.

[5] Bicevskis, J., Borozovs, J., Straujums, U., Zarins, A. and Miller(Jr.), E., "SMOTL - A System to Construct Samples for Data Processing Program Debugging", *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 1 , pp. 60–66 (Jan. 1979).

[6] Camuffo, M., Maiocchi, M. and Morselli, M., "Automatic Software Test Generation", *Information and Software Technology*, Vol. 32, No. 5 (June 1990).

[7] Hall, P. A. V., "Relationship between Specifications and Testing", *Information and Software Technology*, Vol. 33, No. 1 (1991).

[8] Horebeek, I. V. and Lewi, J., *Algebraic Specifications in Software Engineering*, Springer-Verlag, Berlin , 1989.

[9] Howden, W., *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.

[10] Howden, W. E., "Error, Testing Properties and Function Program Tests", in *Computer Porgram Testing*, North-Holland, 1981.

[11] Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer* , Vol. 22, No. 5 , pp. 13–25 (May 1989).

[12] Richardson, D. J., O'Mally, O. and Title, C., "Approaches to Specification-based Testing", *Software Engineering* , Vol. 14 , No. 8 (December, 1989).

[13] Spivey, J. M., *Understanding Z*, Cambridge University Press, 1988.

[14] Spivey, J. M., *The Z Notation*, Prentice-Hall International, London, 1989.

[15] Spivey, J. M., *The Fuzz Manual*, Oxford University, 1992.

[16] Tsai, W. T., Volovik, D. and Keefe, T. F., "Automated Test Case Generation for Programs Specified by Relational Algebra Queries", *IEEE Transactions on Software Engineering*, Vol. 16 , No. 3 (March 1990).