Appeared in Proceedings of ISSRE'98, 5–7 November 1998, pp. 174–178 (IEEE Computer Society Press).

Model-Based Testing of a Highly Programmable System

S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott
Bellcore Applied Research, 445 South Street, Morristown NJ 07960, USA
{sid, jain, karun, jleaton, lott}@bellcore.com

Abstract

The paradigm of model-based testing shifts the focus of testing from writing individual test cases to developing a model from which a test suite can be generated automatically. We report on our experience with model-based testing of a highly programmable system that implements intelligent telephony services in the U.S. telephone network. Our approach used automatic test-case generation technology to develop sets of self-checking test cases based on a machine-readable specification of the messages in the protocol under test. The AETG™ software system selected a minimal number of test-data tuples that covered pairwise combinations of tuple elements. We found the combinatorial approach of covering pairwise interactions between input fields to be highly effective. Our tests revealed failures that would have been difficult to detect using traditional test designs. However, transferring this technology to the testing organization was difficult. Automatic generation of cases represents a significant departure from conventional testing practice due to the large number of tests and the amount of software development involved.

Keywords: model-based software testing, automatic test-case generation, ISCP, AETG software system.

1. Introduction

Telephone users expect extremely high reliability of telephone networks and services, and both equipment and service providers dedicate themselves to satisfying those expectations. However, especially in the current era of competition in telecommunications markets, resources must be applied judiciously. The goal of our work was generating comprehensive test sets for a telephone network element that would significantly enhance the reliability of the system yet cost little to generate. By enhancing the productivity of the test organization, we expect they will be able to respond quickly to an increasingly rapid cycle of product releases while maintaining high standards of system reliability.

We report on our experience from two projects that applied principles of model-based testing to generate black-box tests for Bellcore's Intelligent Services Control Point (ISCP). Model-based testing for the ISCP promised two significant advantages over manual test creation. The first and most compelling advantage was coping with change: in response to changes in the product or the test criteria, the test organization would simply regenerate a test suite automatically. Second, we hypothesized that the use of pairwise-complete tuples generated by Bellcore's AETG™ software system[1] would reveal more failures than tuples selected by the traditional, manual approach. Previous experience with model-based testing using the AETG software system was highly promising [4].

The problems in the work of automatically generating test cases for a reactive system using a test model include at least the following items:

1. Selecting valid and invalid values for individual fields
2. Combining individual field values into test-case input tuples such that constraints among fields are satisfied
3. Choosing number of tests (test termination criteria)
4. Developing scaffolding for running the tests
5. Calculating the expected results (i.e., an oracle)
6. Demonstrating the effectiveness of the effort.

A conventional approach was used to select individual values (item 1), much like the approach in [5]. Values were selected based on the data type, with emphasis on boundary and average values.

This work used a novel approach for items 2 and 3, namely the AETG software system that was developed in Bellcore's Applied Research division [4]. The AETG software system supports the generation of input test data (tuples) in which every distinct value possible for each tuple element appears at least once with every other distinct value

[1] AETG is a registered trademark of Bellcore.

of every other tuple element, a notion called pairwise coverage. The size of the test suite (item 3) is determined by the number of tuples required to attain pairwise coverage. The value of the AETG software system is that it achieves pairwise coverage with test suites that are a tiny fraction of the size that would be required for exhaustive testing.

The work required to develop test scaffolding dominated the projects (item 4). We spent considerable time automating the creation of that scaffolding.

Because individual test cases were relatively simple, expected outputs could be computed easily (item 5). Expected outputs and comparison logic were embedded in some test cases, making them self-checking.

Since this was a pilot of model-based testing, we wanted to demonstrate its effectiveness by measuring the difference between the old and new approaches (item 6). However, our work supplemented existing test suites (rather than replacing them), so calculating the return on this investment is difficult. Preliminary results are discussed later.

2. Background and Related Work

This section gives an overview of the system under test and discusses related work on model-based testing.

2.1. System under test

The system under test was Bellcore's Integrated Services Control Point (ISCP). The ISCP implements telephony services in the telephone networks deployed in the U.S. and elsewhere. This reactive system receives messages from a telephone end office (switch) concerning a telephone call, executes some logic to choose a suitable response, and sends a message to direct continued processing of the call. Mass-market services implemented by the ISCP include toll-free numbers, dialing by voice, and user authentication for wireless handsets.

The ISCP software is built from several million lines of code and is a highly programmable system. The ISCP project manages a rapid release cycle due to specific tailoring done for individual customers. This tailoring is necessary for the system to function properly in a customer's network, and affects the system sufficiently that each tailored system is tested separately. Because the releases are closely related, model-based testing promised significant advantages to the project.

2.2. Model-based testing

Much work has been done on automated generation of test data and test cases, with significant emphasis on testing compilers using automatically generated bits of code. (As used in this paper, test data means test inputs; a test case includes both inputs and expected outputs.) Ramamoorthy et al. discuss the use of symbolic execution to discover the test data that will achieve statement, branch, and path coverage [12]. Bird and Munoz discuss specific test-case generators with emphasis on generating self-checking code to support compiler testing [1]. Ince surveys previous work on selecting test data [8]. Ostrand and Balcer discuss work that is quite close to ours, in that they generate functional test suites automatically using formal test specifications [10]. Camuffo et al. report on an approach based on annotated grammars that seems especially suited to generating tests for compilers [3]. Maurer reviews the advantages and disadvantages of generating test data with context-free grammars [9]. Burgess reviews the main techniques in constructing systems to generate test data [2].

The use of techniques from experimental design to choose test-case tuples is relatively new. Cohen et al. report on previous experience with developing and using the AETG software system [4]. Heller discusses the use of design of experiment structures to choose test cases [7]. Dunietz et al. report on their experience with attaining code coverage based on 2-way, 3-way, and higher coverage of values within test tuples [6].

Category	Examples
Arithmetic	add, subtract, multiply
String	clrbit, setbit, concat, match
Logical	and, or, xor
Time and date	datestr, timestr, date+, time+
Table	addrow, delrow, selrow

Table 1. Manipulators tested in project 1

3. Project 1: Basic Manipulators

The first project addressed the automatic generation of test cases for basic manipulators provided by the ISCP software. These manipulators are basic infrastructure used in every release of the software. Approximately two staff years were devoted to this effort.

3.1. Scope of the tests to be generated

Table 1 summarizes the manipulators that were tested. Individual data values were chosen manually, with special attention to boundary values. Tuples (i.e., combinations of test data) were generated by the AETG software system to achieve pairwise coverage of all valid values. Testing of table manipulators was slightly different because both tables and table operations were generated.

All manipulators were tested using service logic on the ISCP that performed each operation, compared the result

Field	Values
Type of operand 1	int, float, hex
Value of operand 1	min, max, nominal
Operator 1	$+, *, /, -$
Type of operand 2	int, float, hex
Value of operand 2	min, max, nominal
Operator 2	$+, *, /, -$
Type of operand 3	int, float, hex
Value of operand 3	min, max, nominal

Table 2. AETG software system relation for an expression with 3 operators

Field	Values
Column 1 data type	hex, int, float, string, date
Used as key?	yes, no
Use in sel. criteria?	yes, no
Column 2 data type	hex, int, float, string, date
Used as key?	yes, no
Use in sel. criteria?	yes, no
Column 3 data type	hex, int, float, string, date
Used as key?	yes, no
Use in sel. criteria?	yes, no

Table 3. Relation for testing 3-column tables

to an embedded expected value, and reported success or failure. The effort to create the required service logic required more time than any other project element. The only way to create service logic on the ISCP is via a graphical service creation environment (known as SPACE). The GUI test-automation tool *QA Partner* was used to drive the GUI.

Each test was initiated by sending a message, and the result was indicated by the contents of a return message. Appropriate system inputs (messages) had to be created.

3.2. Testing arithmetic/string manipulators

Table 2 shows a model (an AETG software system relation) for generating test cases. In this example, each test case consists of an arithmetic expression with two operators and three operands. The table lists all possibilities for each. An example test case could be "int min + float max * hex nominal" which might be implemented as "0 + 9.9E9 * ab." The AETG software system creates 18 test cases for covering all the pairwise interactions as compared to 11,664 test cases required for exhaustive testing. We created expressions with 5 operators. Instead of exhaustively testing $3^{12} * 4^5$ combinations, the AETG software system generated just 24 test cases. Similar tables were used to create test cases for the other basic manipulators.

After the test cases were generated, expected output was computed manually, which was feasible due to the small number of test cases. Appropriate logic was appended to the test cases so they would check and report their own results.

3.3. Testing table manipulators

Two steps were required to test table manipulators, namely generation of tables with data and generation of queries to be run against the newly generated tables.

In the first step, the AETG software system was used to generate table and selection schemas. A table schema specifies the number of columns, the data type of each column, and for each column an indication whether that column is a key for the table. A selection schema states which columns will participate in a query. Table 3 gives a relation for creating table and selection schemas for three-column tables.

Except for the addrow operation, all the other operations have to specify a selection criteria. For the example given in Table 3, the AETG software system creates 24 table and selection schemas instead of approximately 8000 in the exhaustive case. Since ISCP imposed a limit of 15 columns on their tables, we decided to model tables with 15 columns only. Instead of exhaustively testing $5^{15} * 2^{30}$ test cases, the AETG software system created only 45 test cases.

Following the generation of table and selection schemas, instances were created for each. Exactly one instance was created for each table schema; a random data generator was used to populate the table instance. For each selection schema that was generated for a particular table, six selection instances were created. For example, if the selection schema for a table indicated that only columns 1 and 2 participate, one selection instance might look like "table1.column1 = 1 AND table1.column2 = ABC."

Of the six selection instances (six was chosen arbitrarily), three selections were for rows that existed in the table and three were for rows that did not exist. The target rows for the successful selections were randomly chosen from the newly generated table instance by a program; rows at the beginning, middle, and end of the table were favored. The three unsuccessful queries were generated by invalidating the three successful cases.

3.4. Results and Payoff

Table 4 summarizes the results. Approximately 15% of the generated test cases revealed system failures. The failures were analyzed to discover patterns, resulting in the identification of several problem classes. These problem classes included mishandled boundary values, unexpected actions taken on invalid inputs, and numerous inconsistencies between the implementation and the documentation.

Basic manipulators	
Total test cases	1601
Failed test cases	213
Failure classes	43

Table 4. Results from testing manipulators

Message parsing and building	
Total test cases	approx. 4,500
Failed test cases	approx. 5%
Failure classes	24

Table 5. Results from testing messages

Several of the failures were revealed only under certain combinations of values. For example, if a table had a compound key, and if only a subset of these key columns were specified in the selection criteria, then the system would ignore any non-key column in the criteria during selection.

After developing the test-generation system for one release of the ISCP software, test suites were generated for two subsequent releases with just one staff-week of effort each. In addition to increasing the reliability of the product, the testing organization gained a tool that can be used to generate a compact and potent test suite.

4. Project 2: Message Parsing and Building

The second project addressed the automatic generation of test cases for a particular message set supported by the ISCP. Test cases were required to exercise functionality for parsing and building the messages. Approximately four staff-years were dedicated to this effort.

4.1. The opportunity

The ISCP project created a model of the message set that would enable the development group to cope with changes. This machine-readable specification was an opportunity to leverage our experience with model-based testing.

4.2. Scope of the tests to be generated

Testing the message set's parsing and building functionality meant checking that all parameters from all messages could be read in from the network and sent out to the network. The message set under test consisted of 25 query messages and associated response messages (total 50 unique messages). A query message can come in to the ISCP from a network element or be sent out from the ISCP to other network elements. Similarly, a response message can come in to the ISCP or be sent out from the ISCP.

Each message had 0 to 25 parameters. Parameters included scalars (e.g., integers), fixed collections of scalars (structs), and variable-length collections of arbitrary types. Ultimately all parameters can be viewed as collections of scalars. Included in the generated tests were deliberate mismatches of values to rule out false positive matches.

4.3. The test-generation system

The first step in generating tests was extracting a model of the data (a test specification) from the message-set specification. Challenges that were overcome in developing the data model included null values for message parameters, complex parameters (e.g., lists and other variable-length types), and upper bounds on the total message size.

Message parameter values were chosen individually. The AETG software system was then used to construct messages (i.e., tuples of values) such that all pairwise combinations of parameter values were covered.

The strategy for testing an outgoing message was to build the message in the ISCP, send the message out, and compare the output with the expected result using a text-comparison tool. The strategy for testing an incoming message was to send in a message to the ISCP using a call-simulation tool, then to compare the message received with expected values embedded in the logic (making the case self-checking).

Following the selection of tuples, all required elements were generated. These elements included scripts to simulate calls, expected outputs, ISCP logic, and test specifications. Again the GUI test-automation *QA Partner* was used to create the logic on the ISCP.

Each test case was run by simulating a telephone call. Tests of incoming messages were initiated by sending a message with a full set of parameter values; success or failure was indicated by the contents of a return message. Tests of outgoing messages were initiated by sending a message with just enough information to cause the outgoing message to be sent; success or failure was determined by comparing the output with an expected output.

4.4. Results and Payoff

Table 5 summarizes the results. Failures were revealed while developing the test-generation system and running the generated tests. After analysis of all problems, 24 distinct failure classes were identified and submitted for repair.

Following the transfer of this technology to the testing organization, the ISCP project will be able to generate test suites for subsequent revisions of the message set at extremely low cost. Significantly, the test suite can be generated early in the release cycle, so the tests can be executed as soon as an executable version is available.

5. Conclusion

We offer lessons learned about systems that generate, document, execute, and evaluate thousands of test cases.

Model of the test data is fundamental. The model is comparable with an executable specification; like a specification, model development requires considerable domain expertise. For example, permissible data values and complex constraints among data values must be discovered and represented. Although a model-based test-generation system will require far more effort to develop than the model, development of the model should be allocated a significant portion of the up-front effort.

Model-based testing is a development project. The development, application, and ongoing maintenance of a test-automation system requires expertise from software developers and professional testers. This mix of skill sets is difficult to find in either a development or a testing organization.

Change must be managed to reduce human effort. Any change in the data model or generation tools generally means regenerating all down-stream data files. Because many support systems were involved in the generation of our test cases, a regeneration effort required considerable human attention. Future work will identify commonalities in successive versions of a generated test suite to avoid unnecessary regeneration.

Technology transfer requires careful planning. First, the generation system must respect local practices. For example, a test specification document may have to be generated. These sorts of issues can dramatically increase the effort required to develop the test-generation system. Second, automatic generation of cases is a significant departure from conventional testing practice due to the large number of tests and the considerable amount of development that is involved. Testers may not be comfortable with the approach and should therefore be involved throughout. Third, unlike a hand-crafted test case, it can be difficult to understand why a particular test was generated (i.e., what exactly is being tested), which lends an unwanted element of mystery to each test. For these reasons, professional testers should concentrate more on the underlying test models rather than a specific test case and the reason it was created.

Summary. The generated test cases revealed numerous defects that were missed by traditional approaches. Tests using pairwise combinations of valid values revealed multiple defects that could only be observed given certain pairs of values, which proved the efficacy of the approach supported by the AETG software system. The generated test suites did not replace existing tests but rather augmented them, making the calculation of cost-benefit ratios difficult. The investment in test-generation techology yielded concrete benefits, and the ISCP project plans to apply this technology to new areas. The ISCP project is now able to generate test suites following changes in certain areas at low cost, which is expected to improve reliability in the field significantly.

Acknowledgements

Many thanks to Gene Cracovia, George Hartley, and especially Isaac Perelmuter for sponsoring our work.

References

[1] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[2] C. J. Burgess. Software testing using an automatic generator of test data. In M. Ross, editor, *First International Conference on Software Quality Management (SQM93)*, pages 541–556, Southampton, UK, Apr. 1993. Comput. Mech.

[3] M. Camuffo, M. Maiocchi, and M. Morselli. Automatic software test generation. *Information and Software Technology*, 32(5):337–346, June 1990.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[6] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 205–215. ACM Press, May 1997.

[7] E. Heller. Using DOE structures to generate software test cases. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, pages 33–39, Washington, DC, June 1995.

[8] D. C. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, 1987.

[9] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[10] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[11] I. M. Perelmuter and D. M. Marks. A proven methodology for testing IN customer services. In *Proceedings of the Fifth International Conference on Intelligence in Networks*, Bordeaux, France, May 1998.

[12] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, Dec. 1976.