To appear in Proceedings of ICSE'99, May 1999 (ACM Press).

Model-Based Testing in Practice

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

B. M. Horowitz
Bellcore
6 Corporate Place
Piscataway NJ 08854, USA
bruceh@cc.bellcore.com

ABSTRACT
Model-based testing is a new and evolving technique for generating a suite of test cases from requirements. Testers using this approach concentrate on a data model and generation infrastructure instead of hand-crafting individual tests. Several relatively small studies have demonstrated how combinatorial test generation techniques allow testers to achieve broad coverage of the input domain with a small number of tests. We have conducted several relatively large projects in which we applied these techniques to systems with millions of lines of code. Given the complexity of testing, the model-based testing approach was used in conjunction with test automation harnesses. Since no large empirical study has been conducted to measure efficacy of this new approach, we report on our experience with developing tools and methods in support of model-based testing. The four case studies presented here offer details and results of applying combinatorial test-generation techniques on a large scale to diverse applications. Based on the four projects, we offer our insights into what works in practice and our thoughts about obstacles to transferring this technology into testing organizations.

Keywords
Model-based testing, automatic test generation, AETG software system.

1 INTRODUCTION

Product testers, like developers, are placed under severe pressure by the short release cycles expected in today's software markets. In the telecommunications domain, customers contract for large, custom-built systems and demand high reliability of their software. Due to increased competition in telecom markets, the customers are also demanding cost reductions in their maintenance contracts. All of these issues have encouraged product test organizations to search for techniques that improve upon the traditional approach of hand-crafting individual test cases.

Test automation techniques offer much hope for testers. The simplest application is running tests automatically. This allows suites of hand-crafted tests to serve as regression tests. However, automated execution of tests does not address the problems of costly test development and uncertain coverage of the input domain.

We have been researching, developing, and applying the idea of automatic test generation, which we call model-based testing. This approach involves developing and using a data model to generate tests. The model is essentially a specification of the inputs to the software, and can be developed early in the cycle from requirements information. Test selection criteria are expressed in algorithms, and can be tuned in response to experience. In the ideal case, a regression test suite can be generated that is a turnkey solution to testing the piece of software: the suite includes inputs, expected outputs, and necessary infrastructure to run the tests automatically.

While the model-based test approach is not a panacea, it offers considerable promise in reducing the cost of test generation, increasing the effectiveness of the tests, and shortening the testing cycle. Test generation can be especially effective for systems that are changed frequently, because testers can update the data model and then rapidly regenerate a test suite, avoiding tedious and error-prone editing of a suite of hand-crafted tests.

At present, many commercially available tools expect the tester to be 1/3 developer, 1/3 system engineer, and 1/3 tester. Unfortunately, such savvy testers are few or the budget to hire such testers is simply not there. It is a mistake to develop technology that does not adequately address the competence of a majority of its users. Our efforts have focused on developing methods and techniques to support model-based testing that will be adopted readily by testers, and this goal influenced our work in many ways.

We discuss our approach to model-based testing, including some details about modeling notations and test-selection algorithms in Section 2. Section 3 surveys related work. Four large-scale applications of model-based testing are presented in Section 4. Finally, we offer some lessons learned about what works and does not work in practice in Section 5.

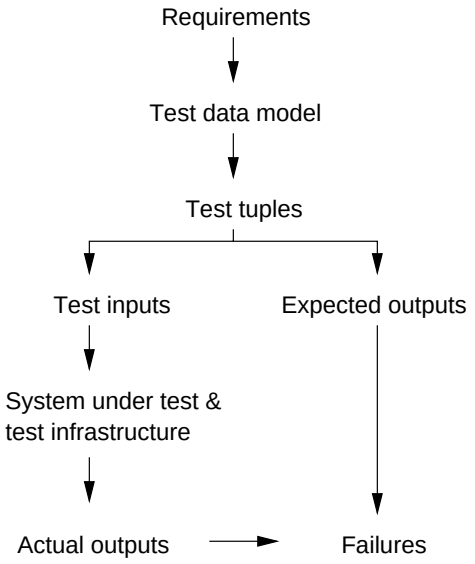

Figure 1: Architecture of a generic test-generation system

2 METHODS AND TOOLS FOR MODEL-BASED TESTING

Model-based testing depends on three key technologies: the notation used for the data model, the test-generation algorithm, and the tools that generate supporting infrastructure for the tests (including expected outputs). Unlike the generation of test infrastructure, model notations and test-generation algorithms are portable across projects. Figure 1 gives an overview of the problem; it shows the data flows in a generic test-generation system.

We first discuss different levels at which model-based testing can be applied, then describe the model notation and test-generation algorithm used in our work.

Levels of testing

During development and maintenance life cycles, tests may be applied to very small units, collections of units, or entire systems. Model-based testing can assist test activities at all levels.

At the lowest level, model-based testing can be used to exercise a single software module. By modeling the input parameters of the module, a small but rich set of tests can be developed rapidly. This approach can be used to help developers during unit test activities.

An intermediate-level application of model-based testing is checking simple behaviors, what we call a *single step* in an application. Examples of a single step are performing an addition operation, inserting a row in a table, sending a message, or filling out a screen and submitting the contents. Generating tests for a single step requires just one input data model, and allows computation of the expected outputs without creating an oracle that is more complex than the system under test.

A greater challenge that offers comparably greater benefits is using model-based testing at the level of complex system behaviors (sometimes known as flow testing). Step-oriented tests can be chained to generate comprehensive test suites. This type of testing most closely represents customer usage of software. In our work, we have chosen sequences of steps based on operational profiles [11], and used the combinatorial test-generation approach to choose values tested in each step. An alternate approach to flow testing uses models of a system's *behavior* instead of its inputs to generate tests; this approach is surveyed briefly in Section 3.

Model notation

The ideal model notation would be easy for testers to understand, describe a large problem as easily as a small system, and still be a form understood by a test-generation tool. Because data model information is essentially requirements information, another ideal would be a notation appropriate for requirements documents (i.e., for use by customers and requirements engineers). Reconciling these goals is difficult. We believe there is no ideal modeling language for all purposes, which implies that several notations may be required. Ideally the data model can be generated from some representation of the requirements.

In practice, a requirements data model specifies the set of all possible values for a parameter, and a test-generation data model specifies a set of valid and invalid values that will be supplied for that parameter in a test. For example, an input parameter might accept integers in the range 0..255; the data model might use the valid values 0, 100, and 255 as well as the invalid values -1 and 256. (We have had good experience with using values chosen based on boundary-value analysis.) Additionally, the model must specify constraints among the specific values chosen. These constraints capture semantic information about the relationships between parameters. For example, two parameters might accept empty (null) values, but cannot both be empty at the same time. A test-generation data model can also specify combinations of values ("seeds") that must appear in the set of generated test inputs. The use of seeds allows testers to ensure that well-known or critical combinations of values are included in a generated test suite.

Our approach to meeting this challenge has employed a relatively simple specification notation called AETGSpec, which is part of the AETG™ software system.[1] Work with product testers demonstrated to us that the AETGSpec notation used to capture the functional model of the data can be simple to use yet effective in crafting a high quality set of test cases. AETGSpec notation is not especially large; we have deliberately stayed away from constructs that would increase expressiveness at the expense of ease of use. For example, complex relational operators like *join* and *project* would have provided more constructs for input test specifications, but we could never demonstrate a practical use for such constructs.

[1] AETG is a trademark of Bellcore.

```
# This data model has four fields.
field a b c d;

# The relation 'r' describes the fields.
r rel {

# Valid values for the fields.
    a: 1.0 2.1 3.0;
    b: 4 5 6 7 8 9 10;
    c: 7 8 9;
    d: 1 3 4;

# Constraints among the fields.
    if b < 9 then c >= 8 and d <= 3;
    a < d;

# This must appear in the generated tuples.
    seed {
         a  b  c  d
        2.1 4  8  3
     }
}
```

Figure 2: Example data model in AETGSpec notation

An example model written in AETGSpec notation appears in Figure 2. Besides the constructs shown in the example, AETGSpec supports hierarchy in both fields and relations; that is, a relation could have other relations and a field could use other fields in a model. The complete syntax of the language is beyond the scope of this paper.

Thanks to the relative simplicity of the notation, we have had good experience in teaching testers how to write a data model and generate test data. Experience discussed in Section 4 showed that testers learned the notation in about an hour, and soon thereafter were able to create a data model and generate test tuples.

After an input data model has been developed it must be checked. Deficiencies in the model, such as an incorrect range for a data item, lead to failed tests and much wasted effort when analyzing failed tests. One approach for minimizing defects in the model is ensuring traceability from the requirements to the data model. In other words, users should be able to look at the test case and trace it to the requirement being tested. Simple engineering techniques of including as much information as possible in each tuple reduce the effort associated with debugging the model. Still, defects will remain in the model and will be detected after tests have been generated. Incorporating iterative changes in the model without drastically altering the output is vital but difficult. Using "seed" values in the data model can help, but ultimately the test-selection algorithm will be significantly perturbed by introducing a new value or new constraint, most likely resulting in an entirely new set of test cases.

Test-generation algorithm

We use the AETG software system to generate combinations of input values. This approach has been described extensively elsewhere [4], so we just summarize it here.

Test	Parameters (factors)									
no.	1	2	3	4	5	6	7	8	9	10
1	a	a	a	a	a	a	a	a	a	a
2	a	a	a	a	b	b	b	b	b	b
3	b	b	a	b	b	a	b	a	b	a
4	a	b	b	b	a	a	a	b	b	b
5	b	a	b	b	a	b	b	a	a	b
6	b	b	b	a	b	b	a	b	a	a

Table 1: Test cases for 10 parameters with 2 values each

The central idea behind AETG is the application of experimental designs to test generation [6]. Each separate element of a test input tuple (i.e., a parameter) is treated as a *factor*, with the different values for each parameter treated as a *level*. For example, a set of inputs that has 10 parameters with 2 possible values each would use a design appropriate for 10 factors at 2 levels each. The design will ensure that every value (level) of every parameter (factor) is tested at least once with every other level of every other factor, which is called pairwise coverage of the input domain. Pairwise coverage provides a huge reduction in the number of test cases when compared with testing all combinations. By applying combinatorial design techniques, the example with 2^{10} combinations can be tested with just 6 cases, assuming that all combinations are allowed. The generated cases are shown in Table 1 to illustrate pairwise combinations of values. The combinatorial design technique is highly scalable; pairwise coverage of 126 parameters with 2 values each can be attained with just 10 cases.

In practice, some combinations are not valid, so constraints must be considered when generating test tuples. The AETG approach uses avoids; i.e., combinations that cannot appear.

The AETG algorithms allow the user to select the degree of interaction among values. The most commonly used degree of interaction is 2, which results in pairwise combinations. Higher values can be used to obtain greater coverage of the input domain with accordingly larger test sets.

The approach of generating tuples of values with pairwise combinations can offer significant value even when computing expected values is prohibitively expensive. The idea is using the generated data as test data. The generated data set can subsequently be used to craft high-quality tests by hand. For example, a fairly complex database can easily be modeled, and a large data set can be quickly generated for the database. Use of a generated data set ensures that all pairwise combinations occur, which would be difficult to attain by hand. The data set is also smaller yet far richer in combinations than arbitrary field data.

Initial work with product testers was facilitated by offering access to the AETG software system over the web. The service is named AETG Web. By eliminating expensive delays in installing and configuring software, testers could begin using the service almost immediately.

Strengths, Weaknesses, and Applicability

The major strengths of our approach to automatic test generation are the tight coupling of the tests to the requirements, the ease with which testers can write the data model, and the ability to regenerate tests rapidly in response to changes. Two weaknesses of the approach are the need for an oracle and the demand for development skills from testers, skills that are unfortunately rare in test organizations. The approach presented here is most applicable to a system for which a data model is sufficient to capture the system's behavior (control information is not required in the model). In other words, the complexity of the system under test's response to a stimulus is relatively low. If a behavioral model must account for sequences of operations in which later operations depend on actions taken by earlier operations, such as a sequence of database update and query operations, additional modeling constructs are required to capture control-flow information. We are actively researching this area, but it is beyond the scope of this paper.

3 RELATED WORK

Heller offers a brief introduction to using design of experiment techniques to choose small sets of test cases [8]. Mandl describes his experience with applying experiment design techniques to compiler testing [10]. Dunietz et al. report on their experience with attaining code coverage based on pairwise, triplet-wise, and higher coverage of values within test tuples [7]. They were able to attain very high block coverage with relatively few cases, but attaining high path coverage required far more cases. Still, their work argues that these test selection algorithms result in high code coverage, a highly desirable result. Burr presents experience with deriving a data model from a high-level specification and generating tests using the AETG software system [2].

Other researchers have worked on many areas in automated test data and test case generation. Ince offers a brief survey [9]. Burgess offers some design criteria that apply when constructing systems to generate test data [1].

Ostrand and Balcer discuss closely related work to ours [12]. As in our approach, a tester uses a modeling notation to record parameters, values, and constraints among parameters; subsequently, a tool generates tuples automatically. However, their algorithm does not guarantee pairwise coverage of input elements.

Clarke reports on experience with testing telecommunications software using a behavioral model [3]. This effort used a commercially available tool to represent the behavioral model and generate tests based on paths through that model. Although Clarke reports impressive numbers concerning the cost of generating tests, no indicators are given about the tests' effectiveness at revealing system failures.

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 2: Manipulators tested in project 1

```
field type1 type2 type3;
field value1 value2 value3;
field op1 op2;

a rel {
    type1 type2 type3: int float hex ;
    value1 value2 value3:  min max nominal ;
    op1 op2: "+" "*" "/" "-";
}
```

Figure 3: AETGSpec data model for an expression with 3 operators

4 CASE STUDIES

We present experience and results from four applications of our technology to Bellcore products.

Project 1: Arithmetic and table operators

The first project addressed a highly programmable system that supported various basic operators [5]. This work had many parallels to compiler testing, but the focus was very narrow. Test were generated for arithmetic and table operators, as shown in Table 2.

The data model was developed manually. Individual data values were also chosen manually, with special attention to boundary values. The data model included both valid and invalid 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 manipulator tests were run using test infrastructure that was written in the language provided by the programmable system. This infrastructure ("service logic") performed each operation, compared the result to an expected value, and reported success or failure. The effort to create the required service logic required more time than any other project element.

Testing arithmetic/string manipulators

Figure 3 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 exam-

Type	Val	Op	Type	Val	Op	Type	Val
float	min	–	float	nom	+	float	min
int	nom	–	int	min	–	hex	nom
hex	max	/	hex	min	+	int	max
int	min	+	int	max	/	float	max
hex	max	*	float	max	*	hex	min
float	nom	+	hex	nom	*	int	nom
hex	nom	/	float	max	–	float	nom
int	min	*	hex	min	–	int	min
float	max	/	int	nom	/	hex	max
hex	min	–	hex	max	/	int	nom
float	nom	*	float	min	/	float	max
int	max	*	int	nom	+	hex	nom
int	min	/	int	min	*	hex	min
hex	max	+	int	nom	–	hex	min
float	max	–	hex	max	*	float	max
int	nom	+	float	max	+	int	min
float	max	+	int	min	–	int	max
float	max	–	hex	nom	/	hex	min

Table 3: Cases for a 3-operator expression with pairwise coverage

ple 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 (shown in Table 3) 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. A set of invalid test cases was also generated using invalid values for the parameters. Appropriate logic was appended to the test cases so they would check and report their own results.

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. Figure 4 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

```
field type1 type2 type3 ;
field key1 key2 key3 ;
field sel1 sel2 sel3 ;

a rel {
    # Data type of columns 1, 2, 3
    type1 type2 type3: hex int float string date ;
    # Is column 1, 2, 3 used as a key?
    key1 key2 key3 : yes no ;
    # Is column 1, 2, 3 used as selection criteria?
    sel1 sel2 sel3 : yes no ;
}
```

Figure 4: AETGSpec data model for testing 3-column tables

in Figure 4, the AETG software system creates 24 table and selection schemas instead of approximately 8000 in the exhaustive case. Due to an environmental constraint of 15 columns maximum per table, tables were modeled with 15 columns only. Instead of exhaustively testing $5^{15} * 2^{30}$ test cases, only 45 test cases were created.

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.

Results
The models were used to generate 1,601 tests cases, of which 213 (approximately 15%) failed. 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.

Several of the failures were revealed only under certain combinations of values. For example, if a table had a compound key, (i.e., the key consisted of more than one column), 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. This would lead to the wrong rows being updated, deleted, or selected.

After developing the test-generation system for one release of the 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.

Because the project never changes the functionality of the basic manipulators, there is no need to regenerate the suite. One major implication of this stability was that it was straightforward to transfer the test suite to the testing organization; they only needed to understand the tests, not the generator. In this project, the main benefit of generated tests was the discovery of failures that otherwise would not have been detected before reaching the customer.

Project 2: Message parsing and building

This project generated tests that exercised message parsing and building functionality for a telephone network element [5]. 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). 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.

On this project, the data model was extracted from a specification of a message set that had been created by the project in order to guarantee traceability from the requirements down to the code. Three valid values and one invalid value were selected automatically based on the specification for each scalar. Valids were selected by focusing on boundary values; the empty (null) value was also shown for all optional parameters. Included in the generated tests were deliberate mismatches of values to rule out false positive matches.

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. In invalid messages, exactly one parameter value was invalid.

The strategy for testing an outgoing message was to build the message in the network element, 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 network element 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, logic, and test specifications.

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.

Results

The data models were used to generate approximately 4,500 test cases, of which approximately 5% revealed system failures. The failures were revealed both while developing the test-generation system and running the generated tests. After analysis of all problems, 27 distinct failure classes were identified and submitted for repair. Just 3 of the failures that were revealed had been detected concurrently by the test team. The model-based testing approach revealed new failures that otherwise would have been delivered to the field.

Following the transfer of this technology to the testing organization, the 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.

Project 3: A rule-based system

The system under test helps manage a workforce. It uses rules encoded in a proprietary language to assign work requests to technicians. A database stores information about work requests (e.g., job location, estimated time for the job, skills required, times when the work can and can't be done, etc.) and information about technicians (e.g., working hours, skills, overtime policies, work locations, meetings, etc.). As the day progresses, the data reflects work assignments that are completed on-schedule, ahead of schedule, or that are running late with respect to the estimated times. When run during a work day, the system extracts information from the database, assigns work requests to technicians, and stores the assignments back in the database. The assignment run is affected by several control parameters, such as the priority given to work requests due that day.

Modeling and testing challenges

Five separate models were used to generate tests. Three data models were used to establish the initial state of the database: one model for work requests (20 parameters), one for technicians (23 parameters), and one for external assignments

("locks") of technicians to specific jobs (4 parameters). Actual job times (travel, setup, work duration) were modeled in relation to the estimated times for the scheduled work requests using a percentage of the estimated times (a fourth data model). The fifth model had information about control parameters that affect the assignment of work requests to technicians (e.g., how many technicians would be considered for each work request; 12 parameters).

Two types of tests were generated and run. The first tested the initial assignment of jobs based on the initial state of the database. These tests could be created rapidly because the initial state had been generated. The second type of test checked the assignment of jobs during the course of a day; i.e., after the estimated time for jobs had been replaced by actual times. The second type of test was much more difficult to create because updates had to correlate perfectly with existing jobs; i.e., the existing state of the database had to be extracted and used during the test-generation step.

This system differed sharply from the other systems to which we applied model-based testing in that multiple valid outputs could result from a single dataset. For example, in the trivial case of two simple work requests and two skilled technicians, the only criteria on the result is that both jobs must be assigned to *someone*.

Evaluation of the results was, in short, difficult. Because all data in the application is related, a change to one piece of input data (e.g., the start time of a work request), can change every output record. The only case that is easy to detect automatically is a crash (i.e., no output at all). It is difficult to determine that each work request is assigned optimally to each technician, given all the business rules, the other technicians, and the work requests. Computation of expected results is the primary weakness of model-based testing. Our experience demonstrated the need for creative solutions to constructing an oracle without reimplementing the system under test.

Because we had no oracle, work assignments were analyzed manually in several ways. A broad analysis checked whether work requests were assigned reasonably and in conformance with business rules. A deep analysis was performed for selected technicians to determine whether the tech's time was used efficiently, or to determine why a tech was assigned no jobs when work requests were still available. Similar deep analyses were performed for particular jobs that were not scheduled despite availability of technicians.

Results

The database was loaded and updated using generated data from the four data models. A total of 13 tests were run, where a test consisted of requesting assignments for all outstanding work requests. The 13 tests were generated from the model of the system's control parameters (9 valid, 4 invalid). Measurements of code coverage using a code-coverage tool developed by Bellcore showed relatively high block coverage (84%) after running the test sets. While very good, this may not be significant for a object oriented system with a rule processing mechanism.

A total of 4 failures were revealed and submitted for repair, of which just one was found concurrently by testers. Two of the failures were due to combinatorial problems in time relations, and one of the failures resulted from a combination that stressed the priority parameter; the appropriate combinations of values apparently had never been tried before. The relatively small number of problems detected was directly caused by the difficulty of analyzing the large amount of output that results from generated test inputs.

Based on this experience, we recommend that test sets be generated by product test and given to developers to run prior to delivering the application to product test. This is because the developers have trace facilities and other tools to check the output more carefully than is possible by a product tester. We would also recommend that once the minimum test sets have been generated, the parameter interaction values in the data models be increased to create large numbers of tests that could be used for performance, load, and stress testing. We also recommend that the product testers use the generated test data as a starting point for creating hand-crafted test cases in which the input data is modified in small, controlled ways so that the output may be more easily understood in relation to the change.

We found that using the AETGSpec data model for application modeling was easy, and can recommend its use by testers. The analysis of effort shows that modeling and updating the models for new releases took only a few days whereas harness development took 17% of the total time spent and the analysis of the large amounts of test output took 40% of the total time. Further research is necessary to develop methods for decreasing the analysis time, to develop methods for easily creating test harnesses, and to develop more automated test analysis techniques. With the increase in tests that test automation brings, it is imperative that there be robust automated test output analysis techniques and test harnesses to run the tests automatically.

Project 4: A user interface

The objective of this project was to model and generate tests for a single GUI window in a large application. The tests would verify that the application responded appropriately to inputs. Related work focused on discovering the model for the window automatically, but that is beyond the scope of this paper.

Challenges in modeling

The window had 6 menus, each with several items. Also displayed in the window were two "tabs" frequently seen in the Windows GUI; i.e., the window was really a frame that housed two pages of information, only one of which could be displayed at a time. Clicking on one or the other tab completely changed the window's controls and behavior.

The first modeling attempt used a single relation with 41 constraints, but the constraints were difficult to understand and eventually found to prevent valid cases. Ultimately a 4-relation data model with 15 constraints was created. The main difficulty was that all menu items are essentially different values of the same parameter: only one menu item can be clicked per test. This meant that the relations all had to have the same parameters; however, in the case of the cut, copy, and paste items we did not care about the values of any parameter except the one that indicated which text field was the target of the cut, copy, or paste operation. Thus in that relation for cut, copy, and paste, all parameters had only one value except the two in question.

The most difficult part of the modeling effort was reading the requirements and determining the constraints. In fact, some of the constraints were not covered in the requirements and we had to run the application to see exactly how it had been implemented. This is to be expected, since we have found that on many applications we model, the requirements are not complete and the developers are forced to make decisions about the missing cases.

Results

Test generation from the model yielded 159 test cases that covered 2423 pairwise interactions. Subsequent work developed a test harness so the tests could be executed automatically. Output analysis was easy because it consisted solely of determining that the correct window was displayed after an action key was clicked. A test for the correct window was implemented in the test harness and the harness checked after each test to determine if the correct window had appeared.

A total of 6 failures were revealed and reported. Of those failures, 3 had been detected concurrently by the test team. Again, the model-based testing approach revealed new failures that would not have been caught by the test team.

One interesting problem was found thanks to the automatic execution of tests. The problem occurred because the code that processed the manipulation of the cut, copy, and paste icons leaked resources each time characters were selected in a text box. The problem was found because the windows would invariably disappear after running about 70 of the 159 automated tests. A close look at a trace file showed that the cut and copy icons were called frequently, and it was then simple to construct a script that repeatedly selected and unselected text in a text field. The loop ran for 329 times until the failure was triggered. This type of problem would not have been found by a manual test approach.

Summary of results

Table 4 gives an overview of the results from the four case studies. A failed test case is any deviation from the expectation. A failure class aggregates similar failures. An example of a failure class is incorrectly handling a null parameter in a message; this class might be represented by hundreds of failures.

Project	Total test cases	Failed test cases	Failure classes
1: Basic manipulators	1,601	13%	43
2: Messaging	4,500	5%	27
3: Rule-based system*	13	23%	4
4: User interface	159	2%	6

*Multiple models were used, and each test case was large, so the numbers from Project 3 are misleading.

Table 4: Results from testing manipulators

5 LESSONS LEARNED

We offer some lessons from developing a system that generates, documents, executes, evaluates, and tracks thousands of test cases based on a data model, starting with implications for model-based testing.

The model of the test data is fundamental. The model must be developed carefully and must include constraints among data values. It is important that the model properly maps to the software under test. We found domain experience to be crucial to developing and quality-checking the model, and we interacted heavily with project developers and testers.

Iteration is required. Our method of operation when building a layer of the testing system was first to build sample data artifacts manually (such as script files or test specification text segments) and next to write software that produced similar artifacts. We would iterate again to incorporate updated or more complete models. We found the iterative approach effective and would apply it to similar projects.

Find abstractions. It takes effort and experience to layer the testing software and to interface the layers. We developed a number of intermediate data artifacts between software layers, but having multiple layers allowed us to encapsulate functionality within layers to help with software readability and maintenance.

Use intermediate files. Managing artifacts in the testing system is a large part of a test-generation effort. We chose a file interface between software layers; this allowed every layer of files (such as test specification documents) to be placed under revision control, and altered manually if necessary. The payoff from this effort was that we could execute layers of software in succession using a traditional "Makefile."

Minimize effort by managing change. Support for managing change is critical to minimizing human effort. Both during testing system development and when analyzing test results, we found both data model and testing system problems. We found it non-trivial to require minimal regeneration and/or test case execution when a layer of data or software changed. Using 'make' helped in a limited way, but nomi-

nally, changes required both regeneration of all downstream file and re-execution of the test suite (costly). Revision control, manual updates, and tracking test results were challenging to handle as well with respect to regeneration. Prototype testing systems can ignore some of these issues, but at the expense of maintainability or of not conforming to auditable quality standards.

Restart in the middle. When running thousands of generated test cases, invariably the machine will be restarted, the power will fail, or some other unfortunate event will interrupt the session. The tester must be able to continue execution of tests following these interruptions. One technique that helps is making tests independent of each other; i.e., starting each test from a known base state instead of chaining them.

Obstacles to technology transfer

Model-based testing represents a significant departure from conventional testing practice, and researchers face a number of problems in transferring this approach into a testing organization.

Tests are mysterious. The testing objectives of a particular test case are not as clearly defined as in a typical, manually authored test case. In other words, there is no comment such as "test feature X." Instead, the testing objective of covering the input domain is spread across a large number of cases. This can lend an unwanted element of mystery to each test.

Development expertise wanted. A test-generation system can be readily applied by testers, but the development of that system requires staff who have experience with both the software quality profession and software development. The mix of skill sets is imperative for making the tradeoffs involved in designing data flows and implementing tools, yet is difficult to find in most testing organizations.

Local practices may hinder automation. Local practices may directly conflict with automation, but any generation system must respect them. For example, a test specification document may be required to establish an audit trail. Tracking the number of test cases run, failed, and passed may be required; without an interface to the tracking system, the generated tests will be invisible to project management. These issues can dramatically increase the effort required to develop the testing system, but are vital to acceptance.

Test process must be reengineered. A substantial initial investment is required to benefit from model-based testing. Appropriate changes must be made in test strategies, test planning, etc. Additionally, a large development effort is required to establish the needed support infrastructure for running and logging thousands of test cases. However, brute force test data generation approaches immediately drive themselves out of contention when automation is lagging. Fine tuning of the generation process to go from a select few tests with critical coverage to a highly redundant, high coverage set of tests can help.

6 CONCLUSION AND FUTURE WORK

In the four case studies presented here, generated test cases revealed numerous defects that were not exposed by traditional approaches. Many of the defects could only be observed given certain pairs of values, which offers considerable support for the efficacy of the AETG software system's approach.

We believe our modeling and test-generation approach satisfies the goal of usability by testers. In our experience, testers found the activity of specifying a software component's inputs to be natural and straightforward. By using the AETG software system, testers required minimal training (about two hours) to write their first data model and generate test tuples with pairwise combinations. As noted above, these tuples offer immediate value when used as test data sets (inputs for hand-crafted tests). Of course a significantly greater investment, mostly in software and script development, is required to develop the infrastructure such as an oracle that will allow the tests to be run wholly automatically. These investments revealed significant numbers of failures in our pilot projects, which is the ultimate justification for investments in testing technology.

We have identified the following questions for future work:

- What are the challenges of applying model-based testing during different phases of testing?
- Can the same modeling language be used for different phases of testing?
- What benefits will model-based testing deliver at different phases of testing?
- How can commonalities in successive versions of a generated test suite be identified in order to avoid unnecessary regeneration?

Future work will also explore combining behavioral models (covering possible paths) with input models (covering pairwise combinations of inputs) to reach new heights in test effectiveness and input domain coverage.

Free trials of the AETG software system are offered via a secure server on the world-wide web. Please visit:
`https://aetgweb.tipandring.com`

REFERENCES

[1] 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.

[2] K. Burr. Combinatorial test techniques: Table-based, test generation, and code coverage. In *Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR'98 West)*, Oct. 1998.

[3] J. M. Clarke. Automated test generation from a behavioral model. In *Proceedings of the Eleventh International Software Quality Week*, San Francisco, CA, May 1998. Software Research, Inc.

[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] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C. M. Lott. Model-based testing of a highly programmable system. In F. Belli, editor, *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 174–178, Nov. 1998.

[6] S. R. Dalal and C. L. Mallows. Factor-covering designs for testing software. *Technometrics*, 40(3):234–243, Aug. 1998.

[7] 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.

[8] 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.

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

[10] R. Mandl. Orthogonal latin squares: An application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, Oct. 1985.

[11] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, 1987.

[12] 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.