CHAPTER – 5
Model-Based Testing
Using Combinatorial Design Method

5.1 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. 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.

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 testing 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.

An example is testing a web-based application, where a user enters data and picks values in a form, then submits the form for processing. Combinatorial
testing will generate a set of cases that ensure every combination of values is tried at least once. This is especially relevant for the field values that are picked from a fixed list. However, combinatorial testing can also assist with values in user-specified fields such as the user’s name. In the case of a user’s name, empty, short, and long values can be tested in combination with empty, short and long values in other user-specified fields.

5.1.1 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.
5.1.1.1 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.

5.1.1.2 Model notation

The ideal model notation would be easy for testers to understand, describe a large problem as easily as a small system, which can be 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

```plaintext
# 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
}
}
```

Fig: 5.2 Example data model in RATGSpec notation
specification notation called RATGSpec. 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.

An example model written in RATGSpec notation appears in Figure 5.2. Besides the constructs shown in the example, RATGSpec 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.

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.

5.1.1.3 Test-generation algorithm

We use the AETG system to generate combinations of input values. This approach has been described extensively elsewhere [COHE 97], so we just summarize it here.
The central idea behind RATG is the application of experimental designs to test generation [DALA 98]. 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 pair-wise coverage of the input domain. Pair-wise 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 210 combinations can be tested with just 6 cases, assuming that all combinations are allowed. The generated cases are shown in Table 5.1 to illustrate pair-wise combinations of values. The combinatorial design technique is highly scalable; pair-wise 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 RATG 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
in pair-wise 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 pair-wise 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 pair-wise combinations occur, which would be difficult to attain by hand.

5.1.1.4 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.
5.2 Modeling System Requirements with the RATG

System requirements are modeled using a basic set of constructs described next. The fundamental RATG construct [GOPA1 06], a relation, is a table with columns for each input item, and rows for the values of each input item. See Table 5.2 for an example. An input item, called a field (or a parameter), is any discrete input to a system under test such as a field on a HTML input form, a parameter to a procedure, etc. Each field can have a different number of values, which are partitioned into valid and invalid values. A test generated from valid values is a valid test, and a test generated from valid and invalid values is an invalid test. Invalid tests are expected to fail before completion because of some error condition.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>IDE</td>
<td>Simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSW 2K</td>
<td>RAID</td>
<td></td>
<td>AGP 64M</td>
<td></td>
</tr>
<tr>
<td>MSW XP</td>
<td>SCSI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Firewire</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 RATG relation shown as a table

From a relation, the AETG system generates test cases, which are vectors of values, one per field. A generated vector of values is called a tuple. A single relation with fields and field values is enough in many cases for generating tuples that achieve thorough testing of the requirements. For the relation shown in Table 5.2 with 24 possible combinations, the AETG system finds 12 test cases to cover all possible pairs. The test cases are shown in Table 5.3.
Field view (or a parameter), is any discrete input to a system under test such as a field on a HTML input form, a parameter to a procedure, etc. Each field can have a different number of values, which are partitioned into valid and invalid values. In the below screen 5.1 the RATG Tool can append more fields as well as update the system test parameters.
A Relation defines the relationship between the systems test parameters (fields). In the below screen 5.2 the RATG tool allows the user to select the degree of interaction among values. The most commonly used degree of interaction is 2, which results in pair-wise combinations. Higher values can be used to obtain greater coverage of the input domain with accordingly larger test sets such as triple wise, is selected when the degree of interaction is 3 and so on. The other constructs in the RATG Tool are added, modify and delete the Relation.

Additional constructs make it possible to model complex situations. These constructs are also used in RATG relations.
First, to take advantage of existing test cases, or to ensure that a particular test case appears in the generated set, a seed test case can be defined in a relation. A seed is a tuple, just like a generated tuple, except that the user supplies it as input to the test generator.

The tester can also guarantee inclusion of their favorite test cases by specifying them as seed tests or partial seed tests for a relation. The seed tests are included in the generated test set without modification. The partial seed tests are seed test cases that have fields that do not have assigned values. The RATG system completes the partial test cases by filling in values for the missing fields as shown in the below screen 5.3.
Second, to reflect closely linked fields in a model, compound field values can be defined. A compound is a set of values for fields. Compounds are useful when pair-wise coverage is not needed within a set of fields (and their values), but pair-wise (or higher) coverage of each set of fields (their values) is desired with other fields. In the below screen 5.4 the RATG Tool can add, modify and delete the Compound.

The third construct for modeling complex situations is the constraint. This is simply an if-then statement that captures relationships among fields that must be honored in the set of generated test cases. For example, if generating
mathematical expressions, the maximum integer value can appear on both sides of the subtraction operator but not the addition operator. Constraints are converted into statements about what field values cannot occur together in a generated tuple. In the below screen 5.5 the RATG Tool can add the constraint to specify the testers particular condition.

Finally all the constructs are used based on the degree of interaction, the Test set or Test Result is generated as shown below in screen 5.6. In the RATG Tool the Test cases are reduced based on the degree of interaction and generates valid test cases and invalid test cases.
5.2.1 Scalability

A table of values can grow in the number of fields (table width) or the number of field values (table height). The number of test cases generated in combinatorial testing grows with the log of the number of fields but with the square of the number of values.

A combinatorial test generator should scale well with large numbers of fields. Each test case allows a large number of pair-wise combinations to be covered. In fact, it is possible to add one or more fields to a test model without
increasing the number of tests required to cover all pair-wise combinations in that model. A similar conclusion holds for triple and higher order coverage.

A combinatorial test generator cannot scale well with large numbers of values. Each field value must appear in a test case with every value for every other field. So a field with a large number of values will result in a large number of generated tests. This also means that adding a single value to a field may result in a significant increase in the number of generated tests. Often a model of this characteristic can be alternatively remodeled so that the field with the large number of values is broken into multiple fields with a small number of values each. This adjustment yields a substantial reduction in the number of test cases required to achieve the desired coverage.

Relations can also be highly constrained. Large numbers of constraints may significantly increase the amount of time required to find a solution. If a relation has sufficient constraints, it may be impossible to find a solution that covers all pairs. A non-obvious part of this problem is that a set of explicit constraints may give rise to other implicit constraints that may consequently rule out other pairs. For example, consider a relation with three fields, binary input \( \{0, 1\} \) for each field, and the following two constraints: (1) if Field 1 is 0, then Field 2 cannot be 2 (ignoring Field 3); and (2) if Field 1 is 1, then Field 3 cannot be 1 (ignoring Field 2). This implies an additional constraint that if Field 2 is 1, then Field 3 cannot be 1 irrespective of the value of Field 1. The logic is clear since if Field 3 is 1, then Field 1 cannot be 1 by constraint (2), so Field 1 can only be 0, and thus by constraint (1) Field 2 cannot be 1. Constraint identification and
solving is a major part of the algorithms for combinatorial test generation. Further, the use of constraints usually increases the number of test cases required for pair-wise and higher coverage.

5.2.2 Multiple relations in one model

System requirements can be modeled with multiple relations, as demonstrated in several examples below. All relations in a model use fields from some defined set. Relations can have identical fields (perfect overlap), can be disjoint (have completely different sets of fields), or can overlap partially in their fields. Because tests are generated separately for each relation, the set of results must be merged when a model has multiple relations.

Using multiple relations in a single model allows generation of tests using different coverage criteria. In other words, one relation may have a lower coverage and another a higher coverage criteria. For example, consider a test scenario that has at least eight inputs. A model can be constructed for this scenario with two relations: the first relation has half the fields and specifies three-way coverage, and the second relation has the remaining fields and specifies pair-wise coverage. Tests are generated according to the two relations and merged. Coverage can be reduced further by specifying a coverage value of one for a relation.

Next, we discuss how and whether generated results can be merged. For perfectly overlapping relations, no merge is required. The sets of tests are simply joined together. An issue to note here is that some pairs may be covered many times unless field values are chosen judiciously. To achieve a smaller number of
test cases, a model with perfectly overlapping relations can usually be rewritten as a single relation with constraints.

For disjoint relations, merging generated tests is relatively straightforward. All fields from all relations are expected to appear in the output, which is a simple union of fields. The only minor issue is how to merge results of different cardinality. For example, in a specification with two relations, one may yield 3 cases and the second may yield 4. The output set must have at least 4 cases. To fill out the tuple that has the fourth case from the second.

5.2.3 Hierarchy and hierarchical testing

A system often has several natural degrees of interaction between its fields. A few fields might be important and the tester may want to test their interactions with each other more intensively than their interactions with the rest of the system. One option is to have two relations. One which contains all the fields and which is tested for pair-wise combinations and another which contains only the most important fields and which is tested for a high-degree of interaction. However, that would be wasteful. A better solution is to use a subrelation.

A subrelation is a relation that is used as a part of another relation. The tester can put the most important fields into a subrelation and give it a high degree of interaction testing. The tester can then use the subrelation inside relations that are tested for a lower degree of interaction. When generating tests, the RATG system will first generate tests that cover the subrelation's specified degree of interaction and then use those tests as partial seed test cases when generating tests for the containing relation.

5.3 Example of Model Based Testing for the RATG System
This example addresses 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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, subtract, multiply</td>
</tr>
<tr>
<td>String</td>
<td>eqv, or, xor</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, xor</td>
</tr>
<tr>
<td>Time and date</td>
<td>datestr, timestr, date1, time1</td>
</tr>
<tr>
<td>Table</td>
<td>addrow, delrow, selrow</td>
</tr>
</tbody>
</table>

Table 5.4 Manipulators tested in project 1

5.3.1 Scope of the tests to be generated

Table 5.4 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 RATG software system to achieve pair-wise 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 to an embedded expected value, and reported success or failure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of operand 1</td>
<td>int, float, hex</td>
</tr>
<tr>
<td>Value of operand 1</td>
<td>min, max, nominal</td>
</tr>
<tr>
<td>Operator 1</td>
<td>+, -, *, /</td>
</tr>
<tr>
<td>Type of operand 2</td>
<td>int, float, hex</td>
</tr>
<tr>
<td>Value of operand 2</td>
<td>min, max, nominal</td>
</tr>
<tr>
<td>Operator 2</td>
<td>+, -, *, /</td>
</tr>
<tr>
<td>Type of operand 3</td>
<td>int, float, hex</td>
</tr>
<tr>
<td>Value of operand 3</td>
<td>min, max, nominal</td>
</tr>
</tbody>
</table>

Table 5.5 RATG software system relation for an expression with 3 operators

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.

5.3.2 Testing arithmetic/string manipulators

Table 5.5 shows a model (an RATG 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 RATG software system creates 18 test cases for covering all the pair-wise interactions as compared to 11,664 test cases required for exhaustive testing. We created expressions with 5 operators. Instead of exhaustively testing $3^{12} \times 4^5$ combinations, the RATG 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.

5.3.3 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column 1 data type</td>
<td>hex, int, float, string, date</td>
</tr>
<tr>
<td>Used as key?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Use in sel. criteria?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Column 2 data type</td>
<td>hex, int, float, string, date</td>
</tr>
<tr>
<td>Use in sel. criteria?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Used as key?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Column 3 data type</td>
<td>hex, int, float, string, date</td>
</tr>
<tr>
<td>Use in sel. criteria?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Used as key?</td>
<td>yes, no</td>
</tr>
<tr>
<td>Use in sel. criteria?</td>
<td>yes, no</td>
</tr>
</tbody>
</table>

Table 5.6 Relation for testing 3-column Tables

A selection schema states which columns will participate in a query. Table 5.6 gives a relation for creating table and selection schemas for three-column tables.

Except for the add row operation, all the other operations have to specify a selection criteria. For the example given in Table 5.3, the RATG 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 $5^{15} \times 2^{30}$ test cases, the RATG 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 that 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 was
used to populate the table instance. The newly generated table instance by a
program; rows at was used to populate the table instance. The beginning, middle,
and end of the table were favored. The three unsuccessful queries were generated
by invalidating the three successful cases.

5.3.4 Results and Payoff

Table 5.7 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.

<table>
<thead>
<tr>
<th>Basic manipulators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total test cases</td>
<td>1601</td>
</tr>
<tr>
<td>Failed test cases</td>
<td>213</td>
</tr>
<tr>
<td>Failure classes</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 5.7 Results from testing manipulators

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.

5.4 Related Work

Heller offers a brief introduction to using design of experiment techniques to choose small sets of test cases [HELI 95]. Mandl describes his experience with applying experiment design techniques to compiler testing [MAND 85]. Dunietz et al. report on their experience with attaining code coverage based on pairwise, triplet-wise, and higher coverage of values within test tuples [DUNI 97]. 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 [BURR 98].

Other researchers have worked on many areas in automated test data and test case generation. Ince offers a brief survey [INCI 87]. Burgess offers some design criteria that apply when constructing systems to generate test data [BURG 93].

Ostrand and Balcer discuss closely related work to ours [OSTR 88]. 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 [CLAR 98]. 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.

5.5 Summary

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 RATG 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.

Our Thesis work reports 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.