Modeling Requirements for Combinatorial Software Testing

C. Lott, A. Jain
Applied Research Area
Telcordia Technologies, Inc.
One Telcordia Drive
Piscataway, NJ 08854

S. Dalal
Imaging and Services Technology
Xerox Corporation
800 Phillips Road
Webster, NY 14580

ABSTRACT

The combinatorial approach to software testing uses models to generate a minimal number of test inputs so that selected combinations of input values are covered. The most common coverage criteria is two-way, or pairwise coverage of value combinations, though for higher confidence three-way or higher coverage may be required. This paper presents example system requirements and corresponding models for applying the combinatorial approach to those requirements. These examples are intended to serve as a tutorial for applying the combinatorial approach to software testing. Although this paper focuses on pairwise coverage, the discussion is equally valid when higher coverage criteria such as three-way (triples) are used. We use terminology and modeling notation from the AETG[1] system to provide concrete examples.

1. INTRODUCTION TO COMBINATORIAL SOFTWARE TESTING

Combinatorial testing is a special kind of black-box testing[2, 4]. It is a relatively low-level functional-test technique that can be applied broadly across the software development lifecycle, from unit testing to customer acceptance testing. As in all functional testing approaches, a set of test inputs is presented to a system, and the result is evaluated for conformance to the requirements.

The combinatorial testing approach is used to generate sets of test inputs from models of system requirements. Instead of testers choosing test inputs manually, testers develop models of the requirements, and an algorithm builds sets of test values automatically. Principles from design of experiments (DOE) are applied to choose a representative sample, thereby minimizing the size of the generated test sets. However, the designs that are used in the context of combinatorial design testing are a new class of designs that select samples for efficacy, not for reliability in statistical estimation.

For example, a computer manufacturer must test different system configurations. Testing every configuration is impossible, but the manufacturer wants to ensure that each operating system works with each storage system, with each display system, etc. Stated differently, the goal is to cover each pair, such as an operating system-storage system pair, at least once. Some time and money is needed to build and test each configuration, so minimizing the number of systems to be tested is interesting. The problem of testing system configurations is ideal for the combinatorial test approach. Given an appropriate model, a combinatorial test generator can select a minimal number of system configurations (i.e., test cases) such that every operating system occurs at least once with every storage system, at least once with every display system, etc. In other words, the set of configurations covers all pairwise combinations of system components. Note that each test case covers several pairwise combinations, which is the key to minimizing the number of cases.

Another 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.

In this paper, we present example system requirements and sample models of those requirements that can be used to generate suitable test cases. These examples are intended to serve as a tutorial for applying the combinatorial approach to software testing. We use terminology and modeling notation from the AETG system to provide concrete examples. However, the modeling guidelines given here are not limited to the AETG system.

2. MODELING SYSTEM REQUIREMENTS WITH AETG

System requirements are modeled using a basic set of constructs described next. The fundamental AETG construct, a **relation**, is a table with columns for each input item, and rows for the values of each input item. See Table 1 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.

[1]AETG is a trademark of Telcordia Technologies, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
A-MOST '05, May 15-16, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-59593-115-5/00/0004 $5.00.

	Fields		
	Operating sys.	Storage sys.	Display sys.
Values	Linux	IDE	Simple
	MSW_2k	RAID	AGP_64M
	MSW_XP	SCSI	
		Firewire	

Table 1: AETG 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 Figure 1 with 24 possible combinations, the AETG system finds 12 test cases to cover all possible pairs. The test cases are shown in Table 2.

Test number	Operating sys.	Storage sys.	Display sys.
1	MSW_2k	Firewire	AGP_64M
2	Linux	Firewire	Simple
3	MSW_2k	SCSI	Simple
4	Linux	IDE	AGP_64M
5	MSW_2k	RAID	Simple
6	MSW_XP	IDE	Simple
7	Linux	RAID	AGP_64M
8	MSW_XP	SCSI	AGP_64M
9	Linux	SCSI	AGP_64M
10	MSW_2k	IDE	Simple
11	MSW_XP	RAID	AGP_64M
12	MSW_XP	Firewire	Simple

Table 2: Generated tests from AETG

Additional constructs make it possible to model complex situations. These constructs are also used in AETG 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.

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 pairwise coverage is not needed within a set of fields (and their values), but pairwise (or higher) coverage of each set of fields (their values) is desired with other fields. An example is presented in the next section.

The third and last 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.

2.1 Data generation

A combinatorial test generator can be used to generate sample data, not just test-case inputs. A generated data set covers pairwise combinations of values in that data, which can be a rich source of examples for testing. Tests that query or update the generated data can be written manually or generated automatically. For example, Dalal et al. used AETG to generate rows for a relational database [3]. That paper also describes a project where the combinatorial test approach was used to generate data for populating five separate tables in a system under test, after which test cases were built by hand.

2.2 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 pairwise 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 pairwise 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. An example of this type is given in Section 3.9.

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 pairwise and higher coverage.

2.3 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 relation, some case from the first relation must be repeated.

Merging is a significant problem for partially overlapping relations. Does field "f1" in one relation mean the same thing as the field "f1" in another relation? What about constraints, which are local to a relation? Because the answers to these questions require further research, and because users do not seem to require this feature, the current version of the AETG system does not support partially overlapping relations.

2.4 Flow Testing

When doing system or product test, testing scenarios typically involve more than a single step of providing an input and verifying the output. Instead, multi-step scenarios or flows are needed.

Consider testing of a student self-registration system. The system allows students to browse the catalog of courses, check the availability of a particular course, and register for a course. The system has a database that keeps track of prerequisites and gives appropriate errors when a student tries to register for a course for which the prerequisites are not met. One flow for system test can consist of the steps of login, selecting the catalog browser, searching for a course using keyword search, selecting a course and a term, and registering for the course. In this flow there is a dependency between steps. For example, the keyword search may return no results, or registration may fail for a selected course because the prerequisites were not met.

AETG can be used for designing efficient test scenarios for flow testing such as this example. Several flows can be modeled using a state-transition diagram (state machine), as shown in Figure 1. Two special states are the initial state and the final state. For example, for the above system, the states would be "Initial," "Login," "Browsing," "Empty search," "Non-empty search," "Successful registration," "Unsuccessful registration," and "Final." States sometimes logically correspond to screens on the user interface, but that may not always be the case.

The input variations in each state can then be modeled as an AETG relation. These inputs may correspond to the inputs which the real user would supply. For example, when in "Browsing" state, the user interface may have several fields for the term, year, course number, course name, etc. For each of these fields we can have a field in an AETG relation where we can assign valid values, invalid values, and constraints among these values. A state transition happens when the user performs certain actions on the user interface like submit, select, click, etc. Besides the action, the transition from one state to another depends upon the input values entered in the immediate prior state or another state that occurred in the path leading to this state. Thus, transition between states is a Boolean condition involving fields in the AETG relations. We use a special field named "Action" with each state. For example, the state transition from "Browsing" to "Empty search" can be a statement like "Browsing.CourseName = 'Course Does Not Exist' ".

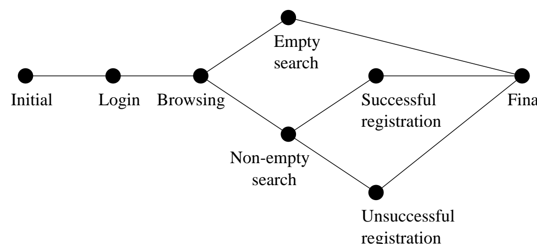

Figure 1: State-transition diagram for a registration system

A more concise formalism of the above is as follows. States are represented using $S(i)$ where $S(0)$ = "Initial," $S(n)$ = "Final," and n denotes the total number of states. For each state $S(i)$ there is a AETG model $m(i)$ consisting of a set of relations. Thus, fully qualified path to a field name is $S(i).f(j)$. Transitions between states are Boolean expressions on the fields. Another piece of information each transition must capture is the action needed to perform to enable that transition. This is not important for AETG modeling but is required for creating automated scripts.

Given the state machine description as above, there are two methods for designing the test cases. The first approach requires creating the test cases for each state using the AETG system. Assume that $T(i)$ is the set of test cases for state i. This method then requires that a script be created to traverse the state machine using the following basic process:

1. Initialize all states as unselected and each test case as unselected. Initialize "Flow" as empty. Start in current state = $S(0)$.
2. Repeat Loop
   (a) Using the flow, identify all the transitions that are enabled as per the condition on the transition. Randomly select a transition that was enabled. Give more weight to transitions leading to an unselected state. Set the current state to $S(i)$. Mark the new state $S(i)$ as visited.
   (b) Randomly select a test case $t_j \in T(j)$ in state $S(i)$. Give preference to unselected test cases. Flow = Flow U t_j. Mark the test case as selected.
   (c) If the current state is $S(n)$ (the final state) then output Flow. Reset Flow to empty. Set current state = $S(0)$. Repeat until all states and test cases have been visited.

The above process is not guaranteed to terminate. However under the following conditions it is guaranteed to terminate:

1. Each transition condition is memoryless (Markovian); that is, it depends only upon the input in the current state and not on any of the prior states (this can be ensured through careful definition of states);
2. The condition specified is consistent with the condition of the AETG model for that state; and
3. The condition does not refer to more fields than the degree of interaction specified in the AETG model for $S(i)$.

Note that as per the above algorithm, besides every possible pairwise or higher field coverage, we also guarantee that all adjacent states in the flow hierarchy are traversed; i.e., all possible pairs of states are covered. In general there is no guarantee that every path is traversed; when the Markovian property holds one also has a guarantee that every path is covered.

Alternately, the other process is:

1. Without regard to conditions, traverse the state machine and decompose it into a set of paths such that all states and all transition are covered at least once.
2. For each path, create one AETG model by taking a cartesian product of the corresponding AETG models and adding the conditions in the transition to the conditions in the model.
3. For each path, create efficient test cases.

2.5 Post-processing and scripts

Tests from a combinatorial test-generation system generally require post processing to format the input values for the system under test. Depending on that system, some scripting infrastructure can be extremely helpful to execute the generated tests. For example, a basic test harness can apply generated sets of test values to a system under test, collect the results, and possibly provide a rudimentary evaluation of the result.

2.6 Using multiple models together

Multiple data sets may be generated and used together. In an example discussed above, several data sets may be generated. Alternately, database requirements may modeled first and used to generate a sample data set, after which other functional requirements (e.g., updates) may be modeled and used to generate test cases.

3. EXAMPLE AETG SYSTEM MODELS

This section presents sample system requirements and corresponding models for testing those requirements. These examples are intended to serve as a tutorial for applying the AETG system.

3.1 Basic billing system

Consider requirements for a billing system that processes telephone-call data with the following four call properties. Values for the properties are also shown in Table 3. A basic black-box test consists of presenting the system with a call record (i.e., a set of values for all the properties shown above), and observing the system's response. Stated abstractly, this system has four inputs, and each input has three possible values.[2] In this example, all combinations are valid, and no invalid values are used. Testing all possible combinations of values requires $3^4 = 81$ tests, but all *pairwise* combinations of values can be covered with 10 tests.

	Fields			
	Access	Billing	Call type	Status
Values	Loop	Caller	Local	Success
	ISDN	Collect	Long_Distance	Busy
	PBX	800	International	Blocked

Table 3: AETG relation shown as a table

Translating these requirements to a model for a combinatorial test generator is straightforward. In the AETG system, each call property is considered a *field*, and each of the possible values for each property is a *field value*. Here is the model:

```
 1  field Access;
 2  field Billing;
 3  field Call_Type;
 4  field Result;
 5
 6  Bill rel 2 {
 7    Access : "Loop1" "ISDN" "PBX" ;
 8    Billing : "Caller" "Collect" 800 ;
 9    Call_Type : "Local" "Long Distance" "International" ;
10    Result : "Success" "Busy" "Blocked" ;                  }
```

3.2 Invalid tests

An invalid test is a tuple that violates some requirement. The AETG system generates two kinds of invalid tests: invalid values and violated constraints. This section focuses on tests with invalid values.

Invalid values can be specified in AETG relations on a per-field basis just like valid values. If invalid values are available, invalid tests are built by choosing an invalid value for exactly one field and then choosing valid values for all remaining fields.

Invalid values are denoted in the specification by appending an exclamation mark. After extending the AETG model shown above with a few invalid values, the test specification looks like this:

```
 1  field Access;
 2  field Billing;
 3  field Call_Type;
 4  field Result;
 5
 6  Bill rel 2 {
 7    Access : "Loop1" "ISDN" "PBX" "Radio"! ;      # CHANGED
 8    Billing : "Caller" "Collect" 800 "Free"! ;    # CHANGED
 9    Call_Type : "Local" "Long Distance" "International" ;
10    Result : "Success" "Busy" "Blocked" ;                  }
```

3.3 Using existing or critical tests

This example considers the problem of guaranteeing that a certain combination is included among a set of generated cases that yield pairwise coverage. A certain combination may reflect some test case that already exists or cover a fault that was recently fixed. No matter the reason, testers want to cover these combinations as part of the generated test suite. Tests can always be added to a generated set, but since a combination achieves pairwise coverage of a certain set of values, it is a nice optimization for the generator to take advantage of the required combination.

To extend the previous example, we consider the case of a call with the following properties: access is "ISDN," billing is "Collect," call type is "Long Distance," and result is "Blocked."

[2] In the terminology of statistical experiment design, there are four *treatments* with three *levels* each.

In AETG terminology, a required combination is called a *seed* test case. The test-generation algorithm takes into account the pairwise value coverage achieved by these combinations. After extending the AETG model shown above with the seed case, the test specification looks like this:

```
 1  field Access;
 2  field Billing;
 3  field Call_Type;
 4  field Result;
 5
 6  Bill rel 2 {
 7    Access : "Loop1" "ISDN" "PBX" "Radio"! ;
 8    Billing : "Caller" "Collect" 800 "Free"! ;
 9    Call_Type : "Local" "Long Distance" "International" ;
10    Result : "Success" "Busy" "Blocked" ;
11    seed {                                                # NEW
12        Access  Billing    Call_Type        Result        # NEW
13        "ISDN"  "Collect"  "Long Distance"  "Blocked" }  }  # NEW
```

The example billing system must be for unlimited-length calls, since it completely ignores call duration! This is corrected next.

3.4 Related fields

This example adds call duration to the previous billing-system example. In this system, call duration must be computed from a pair of values: start time and stop time. (Date information should be included in the times, to allow a call to span a day boundary, but is omitted here just to keep things short.) These input values can be modeled as two additional inputs to the system, with some valid values for each. A pairwise generator will yield some combinations, and the duration can be computed from each combination.

However, unlike the previous inputs, there is a strong dependency between start and stop times. It probably makes sense to try 1-minute, 1-hour, and possibly 1-day calls. While it is possible to test all possible or pairwise combinations of a set of start and stop times, it probably isn't helpful.

We need to link the two time fields together so that the test generator considers them as one compound field, not two separate fields. We could do something relatively crude, like declare a field "start_stop_time" and use compound values like "7:49-7:50", "8:00-9:00", and so on, but that would require some post processing. Ideally the model will reflect the reality of linked fields without crude tricks. In this example, we want to treat pairs of time values as a single field value during test generation. We want pairwise coverage for time-pair 1 and value 1 of field 1, time-pair 1 and value 2 of field 1, and so on. The compound nature of the pair of time values should be ignored during generation, and then the result should show the values separately.

The AETG system provides a modeling construct for this kind of compound field called a *compound*. A compound can can be used to gather any number of linked values together into a set (or tuple).

In this example, we will use three pairs as compounds: "8:00" & "8:10," "9:00" & 9:01," and "15:00" & "16:45." After extending the AETG model shown above with these compounds, the test specification looks like this:

```
 1  field Access;
 2  field Billing;
 3  field Call_Type;
 4  field Result;
 5
 6  compound Times {                                # NEW
 7    label    Start_Time  Stop_Time                # NEW
 8    eight    "8:00"       "8:10"                  # NEW
 9    nine     "9:00"       "9:01"                  # NEW
10    three    "15:00"     "16:45"    }             # NEW
11
12  Bill rel 2 {
13    Access : "Loop1" "ISDN" "PBX" "Radio"! ;
14    Billing : "Caller" "Collect" 800 "Free"! ;
15    Call_Type : "Local" "Long Distance" "International" ;
16    Result : "Success" "Busy" "Blocked" ;
17                                                  # NEW
18    Times : "eight" "nine" "three";               # NEW
19                                                  # NEW
20    seed {
21      Access  Billing    Call_Type        Result
22      "ISDN"  "Collect"  "Long Distance"  "Blocked"  }   }
```

This model still has some problems, namely that as of this writing, an international call can't be a collect call nor a toll-free call. This model deficiency is rectified below by incorporating constraints.

3.5 Field-value constraints

This example extends the telephone-billing example to recognize that international calls have special requirements. Specifically, international calls are never billed as a toll-free (i.e., 800) call nor as a collect call. In the model, the parameter "Billing" cannot have the values "Collect" or "800" in a tuple where the parameter "Call Type" has the value "International." The model must reflect these constraints among field values so the generator never chooses a test case with the impossible pairs of values.

3.5.1 Using if-then statements to model constraints

The AETG system allows these constraints to be entered as "if-then" statements: if "Billing" is "Collect" or "800" then "Call Type" cannot be "International." Stated differently, the test generator must *avoid* the value pair (Billing is "Collect", Call Type is "International") and the pair (Billing is "800", Call Type is "International"). Here is the model:

```
 1  field Access;
 2  field Billing;
 3  field Call_Type;
 4  field Result;
 5
 6  compound Times {
 7    label    Start_Time  Stop_Time
 8    eight    "8:00"       "8:10"
 9    nine     "9:00"       "9:01"
10    three    "15:00"     "16:45"
11  }
12
13  Bill rel 2 {
14    Access : "Loop1" "ISDN" "PBX" "Radio"! ;
15    Billing : "Caller" "Collect" 800 "Free"! ;
16    Call_Type : "Local" "Long Distance" "International" ;
17    Result : "Success" "Busy" "Blocked" ;
18
19    Times : "eight" "nine" "three";
20
21    if Call_Type = "International"              # NEW
22        then Billing != "Collect" 800;          # NEW
23                                                # NEW
24    seed {
25      Access  Billing    Call_Type        Result
26      "ISDN"  "Collect"  "Long Distance"  "Blocked" }  }
```

Invalid tests are generated by AETG along with valid tests. In addition to simple invalid-value tests that were described in the previous section, because constraints are available, a set of invalid-constraint tests are also generated. These are tuples with valid values chosen to violate constraints of the relation.

3.5.2 Using multiple relations to model constraints

An alternate way to reflect the requirements on international calls is to model the requirements in two parts: one part for local and long-distance calls, and one part for international calls. We can do this by using two relations. The first relation models the requirements for national calls; the second relation models the requirements for international calls. The relations have the same fields, but use different values. This example is primarily here to introduce the idea

of building a model using multiple tables (relations). The AETG system generates tests for each relation separately, then combine the tests in the final output.

While this formulation of the model honors the requirements for international calls, and prevents certain combinations of values from occurring together, it generates more cases since some combinations are tested twice. The generator covers combinations of "Access" and "Result" in both relations, resulting in more test cases as compared to the model that used a single relation with appropriate constraints. Finally, it sacrifices the ability to generate invalid-constraint tests. Here is the model:

```
 1  field Call_Type;
 2  field Billing;
 3  field Access;
 4  field Result;
 5  field Start_Time;
 6  field Stop_Time;
 7
 8  compound Times {
 9    label    Start_Time  Stop_Time
10    eight    "8:00"       "8:10"
11    nine     "9:00"       "9:01"
12    three    "15:00"     "16:45"
13  }
14
15  Bill_national rel 2 {                       # CHANGED
16    Access : "Loop1" "ISDN" "PBX" "Radio"! ;
17    Billing : "Caller" "Collect" 800 "Free"! ;
18    Call_Type : "Local" "Long Distance" ;   # CHANGED
19    Result : "Success" "Busy" "Blocked" ;
20    Times : "eight" "nine" "three" ;
21    seed {
22      Access  Billing    Call_Type        Result
23      "ISDN"  "Collect"  "Long Distance"  "Blocked"  }   }
24
25  Bill_international rel 2 {                  # NEW
26    Access : "Loop" "ISDN" "PBX" "Radio\"!;# NEW
27    Billing : "Caller" ;                      # NEW
28    Call_Type : "International" ;             # NEW
29    Result : "Success" "Busy" "Blocked" ;   # NEW
30    Times : "eight" "nine" "three" ;    }    # NEW
```

3.6 Factoring out commonality with auxiliary aggregates

The two-relation version shown above has three fields that appear in both relations with identical values. This commonality can be factored out by use of an AETG modeling construct called an *auxiliary aggregate*. This is just a way of capturing some fields and values for use in relations (relation aggregates). The model below shows an auxiliary aggregate:

```
 1  # Fields
 2
 3  field Call_Type;
 4  field Billing;
 5  field Access;
 6  field Result;
 7  field Start_Time;
 8  field Stop_Time;
 9
10  # Compounds
11
12  compound Times {
13    label Start_Time Stop_Time
14    eight "8:00" "8:10"
15    nine "9:00" "9:01"
16    three "15:00" "16:45"    }
17
18  # Auxilliary aggregate                      # NEW
19                                              # NEW
20  Bill_common {                               # NEW
21    Access : "Loop1" "ISDN" "PBX" "Radio"! ; # NEW
22    Result : "Success" "Busy" "Blocked" ;    # NEW
23    Times : "eight" "nine" "three" ;   }     # NEW
24
25  # Relations
26
27  Bill_international rel 2 {                 # CHANGED
28    Billing : "Caller" "Free"! ;
29    Call_Type : "International" ;
30    Bill_common include_aggregate_Bill_common ;   }
31
32  Bill_national rel 2 {                      # CHANGED
33  Billing : "Caller" "Collect" 800 "Free"! ;
34  Call_Type : "Local" "Long Distance" ;
35  Bill_common include_aggregate_Bill_common ;
36  seed {
37    Billing Call_Type
38    "Collect" "Long Distance"  }  }
```

3.7 Multiple input sets

Some test scenarios have two or more sets of unrelated inputs that must be presented together as test input. For example, in a calendar-printing program, the choice of date range, etc. may be presented together in a dialog with the printer to be used. A set of inputs must have values for the date and printer choices. To achieve thorough testing we should try pairwise combinations of values in the date-choice area of the dialog, and pairwise combinations in the printer-choice area. But it probably is not necessary to try every pairwise combination of date and printer.

This can be modeled effectively with two relations. The first relation has fields, values, constraints, etc. for the date choice; the second relation has field, values, constraints, etc. for the printer choice. Tests are generated and optimized separately for each relation. When it is time to merge the results, tuples are constructed as described in the section on multiple relations above.

3.8 Field groups

While testing messaging systems we have frequently encountered the requirement to support optional groups of values. This means that if any one field from the group is present, then all fields from the group must be present; however, the entire group is optional. (Users of XML may recognize this scenario as an optional sequence with mandatory members.)

We use constraints to model this requirement. Each field in the message is modeled, including those in field groups. Fields that are optional get a special null value. For each group, we include an additional field in the model to switch the group on and off. This so-called control field gets the values "present" and "null." The control field is not a system input; it is used to ensure that either all fields in the group have null values or all fields in the group have non-null values.

Here is simplified model to demonstrate:

```
 1  field reg1;
 2  field reg2;
 3  field reg3;
 4  field group1_control;
 5  field group1_field1;
 6  field group1_field2;
 7  field group1_field3;
 8
 9  # Relations
10
11  r rel 2 {
12    reg1 : "r1" "r2" "r3" ;
13    reg2 : "r4" "r5" ;
14    reg3 : "r6" "r7" "r8" "r9" ;
15    group1_control : "PRESENT" "NULL" ;
16    group1_field1 : "value1" "value2" "value3" "NULL" ;
17    group1_field2 : "value4" "value5" "value6" "NULL" ;
18    group1_field3 : "value7" "value8" "value9" "NULL" ;
19    if group1_control =  "NULL" then group1_field1 =  "NULL";
20    if group1_control != "NULL" then group1_field1 != "NULL";
21    if group1_control =  "NULL" then group1_field2 =  "NULL";
22    if group1_control != "NULL" then group1_field2 != "NULL";
23    if group1_control =  "NULL" then group1_field3 =  "NULL";
24    if group1_control != "NULL" then group1_field3 != "NULL";   }
```

3.9 Large value sets and optional fields

This example presents a scenario in which the system requirements allow many valid values in a modest set of fields. As discussed above, combinatorial test generators yield a large set of tests when fields have many possible values.

We use the example of a request for a quotation for a custom-built PC system. After choosing the basic system configuration, a potential customer can request zero, one, or many additional features such as a graphics card, a sound card, an additional CD burner, etc.

Thorough testing should try every pairwise combination of feature codes to make sure they work together. We present several models to achieve this goal.

To use round numbers, we allow a quotation request to have 0, 1, or up to 10 additional feature requests. Each feature request can be modeled as a field; i.e., a system input. Each feature request field can hold a single feature code; we model 30 different features (i.e., field values). We also model one additional value to mean that the field is empty (i.e., has no feature code).

3.9.1 Simple model

The simplest combinatorial model for testing these requirements is a table of 10 inputs with 31 values each, and no constraints. This would allow field 1 to have a null code, fields 2–7 to have real codes, field 8 to have a null code, and so on. This simple scenario has 43,245 pairs, and well over a thousand test cases are required to cover all the value pairs (AETG finds about 1,300).

3.9.2 Simple model with constraints

We extend the simple model with a set of constraints to bunch feature request codes into the first set of feature-request fields. This prevents the first field from being empty when other fields are not empty, an input that might well be rejected by the order-processing system. The model must enforce the constraints that if field n is empty, then fields $n+1, .., 10$ should also be empty. We need the following 9 constraints:

```
if field1 = "null" then field2 = "null";
if field2 = "null" then field3 = "null";
..
if field9 = "null" then field10 = "null";
```

This model yields even more cases than the previous model because the constraints make it more difficult to cover the value pairs (AETG finds about 1,450). Still, the usage of empty fields conforms to convention.

3.9.3 Refactored model

In this example, the position of a feature-request code has no special meaning. For example, a customer request for “DVD burner” that appears in feature-code field 3 means exactly the same thing as a request for “DVD burner” in field 7. So instead of allowing any feature code to appear in any field as was done previously, the codes can be split up evenly across the fields. Modeling the requirements this way reduces the number of test cases dramatically and still achieves full pairwise coverage. The model still has 10 fields, but each field has only four values. I.e., field 1 gets the first 3 codes plus a “null” code, field 2 gets the next 3 codes plus a “null” code, etc. We use the same constraints from the simple model. This model has 1,179 possible pairs, and can be covered in about one hundred test cases (AETG finds 106).

3.10 Arithmetic expressions

We present a model for testing simple arithmetic expressions that was briefly discussed in [3]. This model was applied to an internal programming language to exercise computation and type conversion/coercion of expressions passed as arguments to functions. Tests generated from this model achieve pairwise coverage of computations applied to constant values and values stored in variables. The following model captures the four math operations addition, subtraction, multiplication and division, and has constraints to prevent overflow and division by zero.

```
 1  field arg1type;
 2  field arg1;
 3  field operator;
 4  field arg2type;
 5  field arg2;
 6  single rel 2 {
 7
 8    arg1type: "variable" "constant";
 9    arg1: "minint" "minfloat" "maxint" "maxfloat" 9 9.0 0 -9 -9.0 ;
10    operator: "+" "-" "x" "/" ;
11    arg2type: "variable" "constant" ;
12    arg2: "minint" "minfloat" "maxint" "maxfloat" 9 9.0 0 -9 -9.0 ;
13
14    if operator = "/"
15       then arg2 != 0 ;
16    if operator = "x"
17       then arg1 != "minint" "minfloat" "maxint" "maxfloat" ;
18    if operator = "x"
19       then arg2 != "minint" "minfloat" "maxint" "maxfloat" ;
20    if arg1 = "maxint" "maxfloat" and operator = "+"
21       then arg2 != "maxint" "maxfloat" 9 9.0 ;
22    if arg1 = "maxint" "maxfloat" 9 9.0 and operator = "+"
23       then arg2 != "maxint" "maxfloat" ;
24    if arg1 = "minint" "minfloat" and operator = "-"
25       then arg2 != "maxint" "maxfloat" 9 9.0 ;
26    if arg1 = "maxint" "maxfloat" 9 9.0 and operator = "-"
27       then arg2 != "minint" "minfloat" -9 -9.0 ;          }
```

4. RELATED WORK

Dalal and Mallows discuss the principles of applying statistical designs to software testing [4]. Dunietz et al. discuss the relative efficacy of two-way, three-way, and higher coverage criteria to attain different levels of code coverage [5]. The IBM Corporation offers a library that can be used for generating combinatorials test, but does not include a modeling language such as the one presented here [6]. Bryce et al. analyze greedy methods for finding minimal-sized combinatorial test sets [1].

5. CONCLUSION

The combinatorial approach to software testing holds considerable promise for revealing failures by covering field-value combinations that hand-crafted test suites miss. We hope this discussion and especially the examples serve to deepen understanding of this approach and make it easier for testers to apply combinatorial testing.

6. REFERENCES

[1] Renée C. Bryce, Charles Colbourn, and Myra Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 2005 International Conference on Software Engineering*. ACM Press, May 2005.

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

[3] S. Dalal, A. Jain, C. Lott, G. Patton, N. Karunanith, J. M. Leaton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the* 21st *International Conference on Software Engineering*, pages 285–294. ACM Press, May 1999.

[4] Siddhartha R. Dalal and Colin L. Mallows. Factor-covering designs for testing software. *Technometrics*, 40(3):234–243, August 1998.

[5] 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.

[6] IBM. Combinatorial Test Services. http://www.alphaworks.ibm.com/tech/cts, 2004.