Effective Testing of Factor Combinations

George B. Sherwood

AT&T Bell Laboratories
Lincroft, New Jersey 07738

ABSTRACT

A method and tool for system test case selection is described with examples. The Constrained Array Test System (CATS) is a system test tool which generates and analyzes possible test cases. It suggests a sequence of tests to minimize the number of untested combinations of test factor values. CATS improves system test productivity by reducing the number of test cases, thereby saving testing time. CATS also improves quality (the ability to find faults), through better coverage of combinations of test factor values. The tool easily adapts to real-world constraints which often preclude testing with certain combinations of test factors.

1. HOW MUCH TESTING IS ENOUGH?

As any software development project approaches completion, one inevitable tradeoff emerges: How much testing is enough? Too little testing leads to expensive fixes in the field and lost customers. Too much testing leads to delayed products and lost customers. So the tradeoff is how to balance the need for quality with the need for a timely product. Unless the testing program is effective and efficient, this tradeoff can eventually become a dilemma over which way to lose the intended market.

This paper describes a method and tool for test case selection developed at AT&T Bell Laboratories in early 1990. The motivation for the work was improvement of the system test process. At the time we were unable to find tools to make a very large test job manageable. We wanted to improve the way we chose test cases, and the Constrained Array Test System (CATS) was the result.

2. APPROACHES

The description of the Constrained Array Test System is best done in a context of other methods for software test coverage. For the purposes of this paper they are divided into two groups:

- Systematic Coverage Methods, and
- Test Factor Coverage Methods.

CATS falls into the latter group.

2.1 Systematic Coverage

Systematic coverage methods emphasize covering the states (or behavior) of the system under test. The methods require a good view into the system: Code coverage methods can be thought of as "white box" testing, with complete knowledge of what is inside the system. Model coverage methods can be considered "black box" testing, but a complete, formal model of the system is required. Generally systematic coverage methods employ software tools for assessing test coverage or for selecting test cases.

2.1.1 Code Coverage Tools Code coverage tools often are used for unit testing or subsystem testing. They may be part of a development package to be applied as individual units are built and integrated together. Typically code coverage tools give the tester statistics about usage of the code under test, including references to line numbers of source code which are not exercised by the tests in use. Using such tools the tester can create test cases until there is 100% coverage. Then additional test cases will not cover more code.

2.1.2 Model Coverage Tools Model coverage tools use formal models of what the system should do, to generate test cases. For example, the POSTMAN tool[1] uses deterministic finite-state models of systems to find test cases which cover all state transitions with a minimum run time. The tool has been successfully applied to the testing of communication protocol implementations. Using such a tool, the tester can run just enough test cases to determine if the system follows the model.

2.2 Limitations

Systematic coverage tools are very effective, but their applicability to system testing can be limited for any of several reasons:

- Source code may not be available for parts of the system.
- A formal model of the whole system may not be available.
- The system may be too large or complex for available tools.
- Available tools may be incompatible with some parts of heterogeneous systems.
- Additional test factors may be important -- e.g. environmental conditions, configurations, values for parameters and data.

These limitations suggest the need for other methods and tools to complement systematic coverage.

2.3 Definitions

The following definitions are provided for the discussion of test factor coverage methods below.

2.3.1 Test Factor A *test factor* is any variable whose values are to be controlled during the tests. A *test factor value* is a specific value taken by one of the factors during the tests.

An example of a test factor is the processor speed in MHz for a client computer -- *client_speed*. This factor could have as one of its values *33*, in those test cases where the client's processor runs at 33 MHz.

Generally test factors can be anything thought to be relevant to the test job. Examples are:

- Environmental conditions
- Characteristics of a larger system of which the test system is a part, e.g. hardware and operating system configurations
- Processor speed
- System load characteristics
- Modes of operation and options for the system under test
- Choice of communication configuration for interaction with another part of the system, e.g. interfaces, endpoints, etc.
- Characteristics of users and user interfaces
- Command line arguments
- Input data, e.g. text field contents, mouse position coordinates, etc.

2.3.2 Test Case A *test case* is a set of specific test factor values in which one allowed value is associated with each of the test factors. Thus, the set of factor values represents an individual test from a sequence to be performed.

For example, if there are seven test factors chosen, say *client_speed*, *server_os_ver*, and five others, a test case is a set of values for all of the seven factors. A test case could have *client_speed=33*, *server_os_ver=SVR4*, and five values for the other five factors.

2.3.3 Combination of Factor Values A *combination of test factor values* is an association of some number of factor values. E.g. *client_speed=33* with *server_os_ver=SVR4* is a combination of two factor values which could occur in a particular test case. Typically testers consider pairwise or two-at-a-time combinations when assessing test coverage.

2.4 Test Factor Coverage

Test factor coverage methods aim to cover the values or combinations of values of specified test factors. The factors may or may not be related to the states (behavior) of the system under test. Since the test factors are general, test factor coverage methods can be more flexible and readily applicable than those of

systematic coverage. On the other hand, if the tester selects test factors related to the states of the system, he or she must select factors and values which correctly reflect how the system works, usually without the benefit of a formal model.[1] Section 6 below gives two examples in which test factor values are related to the state of the system under test.

The following sections outline five ways to select test cases for test factor coverage.

2.4.1 One Factor at a Time One straightforward approach is to select one value for each factor, say a most likely value or a default value. Then holding all but one of the factors at fixed values, the tester varies the remaining factor through all its allowed values. This process is repeated so that each factor takes on all of its allowed values. The number of resulting test cases is relatively small, but their ability to find faults is limited. While all of the individual factor values have been covered, all of the pairwise combinations have not. Consequently faults related to interactions between two factors may not be found.

2.4.2 All Combinations Another approach is to test all combinations of factor values. For example, with four factors having six values each, there would be $6 \cdot 6 \cdot 6 \cdot 6 = 1296$ test cases. This method should find all faults because all combinations are tested. However, unless the numbers of factors and values are very small, the method is not practical: It leads to too many test cases.

2.4.3 Random Test Case Selection Another approach is to select factor values randomly. Here the tester can choose an intermediate number of test cases, but their ability to find faults among the different factors is not readily predictable. Some values and combinations of values are likely to be covered more than others, because there is no control applied to the selection of values. In the author's experience, a larger number of randomly selected test cases is needed for coverage of all pairwise factor value combinations, compared to the number when using orthogonal arrays or constrained arrays.

2.4.4 Orthogonal Array Testing An orthogonal array (or Latin square) is an array of integers which may be used to select test cases. Each column of the array corresponds to the values for a test factor; each row of the array represents a test case. An orthogonal array has the property that for every pair of columns, all combinations of their values occur, and they occur an equal number of times.[2] Thus, if an orthogonal array representing the test problem can be found, the resulting test cases will cover all pairwise combinations of factor values. And the number of recommended test cases generally will be small. However at times it may be difficult or impossible to find an orthogonal array that accurately represents the real-world test problem.

The motivation to develop CATS came from our attempt to use orthogonal array testing to specify client test configurations for a local area network product. This particular job was large and labor intensive: The requirements called for 101 combinations of personal computers (PC) and operating system versions to be used with fifteen different network cards. We knew that the operating system version of the client PC was an important factor for finding software defects. However, the values selected for this factor were dependent on the values of two others -- the vendor of the PC and its processor. PC vendors would specify which versions of DOS or OS/2 were supported on any particular model; generally new machines were not supported with earlier operating systems. OS/2 could not be used with 8088 or 8086 processors, and only one of our required PC vendors supported all the required processors.

Matters got more complex when network cards were considered. Our requirements called for support for two different PC bus types. The newer of these was supported by only one of our PC vendors, and only on newer processors and newer operating system versions. I.e. the bus type factor was dependent on three other factors.

The orthogonal array tool we were using provided for some dependencies among test factors. However, the real constraints contained in our requirements could not be given to the tool without simplification. We

1. Systematic coverage and test factor coverage methods do not preclude each other. If a systematic coverage tool is applicable, it can be used with test factor coverage methods. E.g. CATS can be used to specify test configurations for a model coverage tool specifying the state transitions to be tested.

could *omit* some of the dependencies, but that led to some of the test cases being impossible. For example, anachronistic configurations like an 8088 processor with a Microchannel bus or an 80386 processor with DOS 3.1 were suggested. If these impossible test cases were left out, then some of the *desired* combinations of factor values would not be covered.

2.4.5 Constrained Array Testing A constrained array is an array similar to an orthogonal array in that its elements correspond to test factor values, with columns representing test factors and rows representing test cases. It differs from an orthogonal array in two ways:

- Constrained arrays do not cover factor value combinations which are excluded from consideration, e.g. because the combinations are not possible.
- Constrained arrays do not cover factor value combinations an equal number of times. Instead each combination is covered at least once.

Mathematically a constrained array is a generalization of a *t-covering array*. Several researchers have studied how to minimize the number of test cases resulting from *t*-covering arrays over the past several years.[3] [4] [5] Sloane has studied *3*-covering arrays and provided an extensive bibliography.[6] When CATS generates a constrained array, it does not attempt to find the absolute minimum number of test cases for the particular test problem. CATS does find the minimum number for some problems, but its purpose is to find a *small* set of test cases. The appendix to this paper gives more detailed information on the number of test cases CATS suggests.

3. FEATURES OF CATS

This section outlines features that have been built into CATS. A detailed discussion of all of them is beyond the scope of this paper.

The Constrained Array Test System is a system test tool which generates and analyzes possible test cases. It suggests a sequence of tests to minimize the number of untested combinations of test factor values. CATS improves system test productivity by reducing the number of test cases, thereby saving testing time. CATS also improves quality (the ability to find faults), through better coverage of combinations of test factor values. The tool easily adapts to real-world constraints which often preclude testing with certain combinations of test factors.

- **Small Number of Test Cases** -- CATS generally recommends a dramatically reduced number of test cases compared to the number of possible test cases. As the test job gets bigger and more complex (i.e. as the number of factors increases), the number of recommended test cases increases very slowly: The number of test cases is observed to approximate a logarithmic function of the number of factor value combinations under consideration.
- **Capacity for Large Numbers of Factors and Values** -- CATS currently supports 63 or more test factors. The number of values allowed for each factor is limited by the number of test cases under consideration (typically up to a few thousand).
- **Conformance to Test Case Constraints** -- CATS can generate and analyze test cases with arbitrary constraints. That is, it does not matter if there are rules that make some test cases impossible; CATS can accommodate such constraints to analyze and recommend only possible test cases.

  CATS also supports the use of equivalence classes or "sliding levels."
- **Ordering of Test Case Sequence** -- CATS suggests a preferable order for test case execution, which gives early coverage to the most combinations of factor values. Thus more faults can be found sooner in the test process.

  The test case order suggested by CATS follows the priority order for the values of each factor, as specified by the user. So, for example, default values can be tested first, least used values last.
- **Interfaces for Factor Values and for Test Cases** -- Either of two interfaces can be used for test case analysis. The **expand** program takes factor values and generates test cases for analysis by the **cats** program. Alternatively, the **cats** program can be given test cases directly.

- **Detection of Higher Order Mode Failures** -- The process used by CATS is not restricted to work with factor values two at a time. It is easy to consider single, double, triple or higher mode failure detection if such faults are thought to be important to the test job.

4. HOW CATS WORKS -- SMALL PROBLEMS

Cats is a single program written in C language. It takes as input sets of test factor values which span the test cases under consideration. These cases can be derived directly from an analysis of requirements; they can be taken from the **expand** program or some other test case generation tool. **Cats** reorders these cases using an algorithm that looks for the next "best" test case, which minimizes the number of untested combinations of factor values. **Cats** output is the reordered test cases. With each test case is a running count of untested combinations of factor values. The user can judge how many of the test cases to execute, based on this information. **Cats** can analyze the combinations using an arbitrary number of factors at a time, limited only by the number of factors in the problem or the computing resources in use.

Cats uses two arrays to store and manipulate test cases -- an *experiment* array containing test cases assumed to have been performed, and a *vector* array containing all the remaining test cases, any one of which could be done next. Initially the experiment array is empty, and all the test cases are placed in the vector array.

On a trial basis, each test case from the vector array is placed in the experiment array, and the number of untested combinations in the experiment array is found. The best test case -- the one which if executed, results in the fewest number of untested combinations -- is selected. It is placed in the experiment array, to be done next; and it is removed from the vector array, as it no longer indicates a new direction.

The process of finding the next best test case is repeated, and each corresponding test case is moved from the vector array to the experiment array. This goes on until all the test cases are moved from the vector array to the experiment array, or there is no further improvement in the test coverage.

Essentially, **cats** follows the steepest gradient through the space of test cases, to reach the minimum number of untested combinations of factor values.

The **expand** preprocessor is a UNIX shell program which generates test cases for the **cats** program from a list of factor values to be analyzed. The preprocessor provides a simple way to generate test cases for **cats**: It echoes factor values in nested loops, so that all possible test cases are given to **cats**. For example, the **expand** input

```
rye     white
cheese  pnutbtr  jelly
```

enables the analysis of all combinations of rye or white bread with cheese, peanut butter or jelly. The individual test cases need not be enumerated.

Expand also has a simple mechanism for handling constraints and dependencies among test factors, which is illustrated in the example of the following section.

5. AN EXAMPLE

In this section we consider a test example with three factors. Each factor can take on five values:

- **Lunch Platform --**
  rye bread, white bread, wheat bread, macaroni, spaghetti
- **Topping Application --**
  cheese, peanut butter, jelly, clam sauce, tomato sauce
- **Beverage Program --**
  Coke, milk, coffee, tea, Chianti

There are constraints limiting which combinations of factor values are allowed with other values. These constraints are given to **expand** by listing all allowed groups of factor value combinations in the **expand** input.

```
rye       white
cheese    pnutbtr   jelly
coke      milk      coffee    tea
append
wheat
cheese    jelly
coke      milk      coffee    tea
append
macaron   spaghet
clams     cheese    tomato
milk      coffee    chianti
```

Each group of three lines separated by "append" represents allowed combinations of factor values, with each line containing the allowed values for one of the factors.

- The first group indicates that rye or white bread is allowed with cheese, peanut butter or jelly and with coke, milk, coffee or tea.
- The second group indicates that wheat bread is allowed with cheese or jelly and with coke, milk, coffee or tea.
- The third group indicates that macaroni or spaghetti is allowed with clams, cheese or tomato and with milk, coffee or chianti.

With this feature it is easy for a user to accommodate complex factor value constraints in the generation of test cases for **cats**.

Expand generates all possible test cases for each of the three factor value groups: The first group yields the first $2 \cdot 3 \cdot 4 = 24$ test cases; the second group yields the next $1 \cdot 2 \cdot 4 = 8$ test cases; and the third group yields the last $2 \cdot 3 \cdot 3 = 18$ test cases.

The **expand** output for **cats** is as follows.

0	5	5	5
1	rye	cheese	coke
2	rye	cheese	milk
3	rye	cheese	coffee
4	rye	cheese	tea
5	rye	pnutbtr	coke
6	rye	pnutbtr	milk
7	rye	pnutbtr	coffee
8	rye	pnutbtr	tea
9	rye	jelly	coke
10	rye	jelly	milk
11	rye	jelly	coffee
12	rye	jelly	tea
13	white	cheese	coke
14	white	cheese	milk
15	white	cheese	coffee
16	white	cheese	tea
17	white	pnutbtr	coke
18	white	pnutbtr	milk
19	white	pnutbtr	coffee
20	white	pnutbtr	tea
21	white	jelly	coke
22	white	jelly	milk
23	white	jelly	coffee
24	white	jelly	tea
25	wheat	cheese	coke
26	wheat	cheese	milk
27	wheat	cheese	coffee
28	wheat	cheese	tea
29	wheat	jelly	coke
30	wheat	jelly	milk
31	wheat	jelly	coffee
32	wheat	jelly	tea
33	macaron	clams	milk
34	macaron	clams	coffee
35	macaron	clams	chianti
36	macaron	cheese	milk
37	macaron	cheese	coffee
38	macaron	cheese	chianti
39	macaron	tomato	milk
40	macaron	tomato	coffee
41	macaron	tomato	chianti
42	spaghet	clams	milk
43	spaghet	clams	coffee
44	spaghet	clams	chianti
45	spaghet	cheese	milk
46	spaghet	cheese	coffee
47	spaghet	cheese	chianti
48	spaghet	tomato	milk
49	spaghet	tomato	coffee
50	spaghet	tomato	chianti

When **expand** invokes **cats**, the following output results.

0	5	5	5	1:15	2:75	3:125	215
1	rye	cheese	coke	1:12	2:72	3:124	208
2	white	pnutbtr	milk	1:9	2:69	3:123	201
3	wheat	jelly	coffee	1:6	2:66	3:122	194
4	macaron	clams	chianti	1:3	2:63	3:121	187
5	spaghet	tomato	milk	1:1	2:60	3:120	181
6	rye	pnutbtr	tea	1:0	2:57	3:119	176
7	rye	jelly	milk	1:0	2:54	3:118	172
8	white	cheese	coffee	1:0	2:51	3:117	168
9	white	jelly	coke	1:0	2:48	3:116	164
10	wheat	cheese	milk	1:0	2:45	3:115	160
11	macaron	tomato	coffee	1:0	2:42	3:114	156
12	spaghet	clams	coffee	1:0	2:39	3:113	152
13	spaghet	cheese	chianti	1:0	2:36	3:112	148
14	rye	pnutbtr	coffee	1:0	2:34	3:111	145
15	white	cheese	tea	1:0	2:32	3:110	142
16	wheat	jelly	tea	1:0	2:30	3:109	139
17	macaron	clams	milk	1:0	2:28	3:108	136
18	rye	pnutbtr	coke	1:0	2:27	3:107	134
19	wheat	cheese	coke	1:0	2:26	3:106	132
20	macaron	cheese	milk	1:0	2:25	3:105	130
21	macaron	tomato	chianti	1:0	2:24	3:104	128
22	rye	cheese	milk	1:0	2:24	3:103	127
23	rye	cheese	coffee	1:0	2:24	3:102	126
24	rye	cheese	tea	1:0	2:24	3:101	125
25	rye	pnutbtr	milk	1:0	2:24	3:100	124
26	rye	jelly	coke	1:0	2:24	3:99	123
27	rye	jelly	coffee	1:0	2:24	3:98	122
28	rye	jelly	tea	1:0	2:24	3:97	121
29	white	cheese	coke	1:0	2:24	3:96	120
30	white	cheese	milk	1:0	2:24	3:95	119
31	white	pnutbtr	coke	1:0	2:24	3:94	118
32	white	pnutbtr	coffee	1:0	2:24	3:93	117
33	white	pnutbtr	tea	1:0	2:24	3:92	116
34	white	jelly	milk	1:0	2:24	3:91	115
35	white	jelly	coffee	1:0	2:24	3:90	114
36	white	jelly	tea	1:0	2:24	3:89	113
37	wheat	cheese	coffee	1:0	2:24	3:88	112
38	wheat	cheese	tea	1:0	2:24	3:87	111
39	wheat	jelly	coke	1:0	2:24	3:86	110
40	wheat	jelly	milk	1:0	2:24	3:85	109
41	macaron	clams	coffee	1:0	2:24	3:84	108
42	macaron	cheese	coffee	1:0	2:24	3:83	107
43	macaron	cheese	chianti	1:0	2:24	3:82	106
44	macaron	tomato	milk	1:0	2:24	3:81	105
45	spaghet	clams	milk	1:0	2:24	3:80	104
46	spaghet	clams	chianti	1:0	2:24	3:79	103
47	spaghet	cheese	milk	1:0	2:24	3:78	102
48	spaghet	cheese	coffee	1:0	2:24	3:77	101
49	spaghet	tomato	coffee	1:0	2:24	3:76	100
50	spaghet	tomato	chianti	1:0	2:24	3:75	99

The test cases now appear in a different order (with different numbering) and with four new columns on the right side. CATS recommends this new order for test case execution. Each of the new columns gives a running count of the number of untested combinations of factor values as the test cases are performed.

The first new column (which starts with 1:) lists the number of untested combinations of factor values when the factors are considered one at a time. The zeroth row shows that there are 5 + 5 + 5 = 15 such combinations before any testing is done. After the first test case is done, four values of each factor remain untested, so 1:12 appears. After the sixth test case is done, all of the individual values have been tested, so 1:0 appears.

The second new column (which starts with 2:) lists the number of untested combinations of factor values when the factors are considered two at a time. The zeroth row shows that there are $5 \cdot 5 + 5 \cdot 5 + 5 \cdot 5 = 75$ pairwise combinations before any testing is done. After the first test case is done, 72 combinations of values remain untested, so 2:72 appears. After the twenty-first test case is done, 24 of the combinations have not been tested, so 2:24 appears.

Similarly, the third new column gives a count of the uncovered three-at-a-time combinations. The fourth column simply sums the other three columns for a total number of uncovered combinations.

In this example CATS suggests

- 6 test cases to cover all factor values individually, or
- 21 test cases to cover all *allowed* factor value pairs, or
- 50 test cases to cover all *allowed* factor value triples.

If the tester selects the first 21 test cases, he or she will test all the allowed factor values individually and in pairs. There are $125 - 75 = 50$ allowed factor value triples. Of these, the tester will cover $125 - 104 = 21$ combinations, for $21/50 = 42\%$ coverage of the allowed three-at-a-time combinations.

There are 24 out of 75 pairwise combinations and 75 out of 125 three-at-a-time combinations which are not covered due to constraints imposed in the problem.

6. TEST CASES THAT MASK COMBINATIONS

When we work with configuration factor values for hardware and operating system versions, our constraints may have a simple meaning: The cases we do not consider are really impossible. But in other problems it may be necessary to apply constraints to avoid test cases that mask coverage of desired combinations. Typically these are problems in which factor values imply a change of state in the test system. The next two sections illustrate the point with two examples.

6.1 Single Values that Mask

The simplest example of masking combinations is the use of error values. Testers need to test not only "sunny day" values of factors, which demonstrate that the system operates normally, but also the "rainy day" values, which show that the system detects and responds to abnormal conditions. Now if the tester simply mixes normal and error values in the CATS analysis, it is quite likely that when an error value occurs, it will cause the system to follow an error leg of the program logic. And this change of state may mean that a combination of two normal values which is *supposed* to be covered in the same test case really is not covered as the tester intended. We call this occurrence a masked combination.

The tester can avoid masked combinations by applying different CATS runs to different states, i.e. by constraining each CATS run to tests of a single state. For testing error values there are a couple of simple approaches:

- If the tester is concerned only with covering individual error values, and not combinations with other, normal values, then the CATS analysis can be done with only normal factor values. Individual error values then can be substituted for normal values in the resulting test cases.
- If the tester is concerned with covering combinations of error values with other, normal values, then the tester can do a separate CATS run for each factor which has error values. Each run would include all the error values for one factor with the normal values for all the other factors.

6.2 Combinations that Mask

The second example of masking combinations involves a "normal" state change of the system under test. We suppose an electronic messaging system is under test, and it has two features of interest:

- automatic forwarding on the basis of the received message's subject and
- automatic deletion on the basis of the received message's subject.

Also, the autodeletion feature takes precedence over autoforwarding if they are to act on the same message.

(It would be deleted and not forwarded.) There may be several test factors relating to how these features are configured and to the properties of test messages. However, if we do not avoid the autodeletion feature taking precedence over autoforwarding, then there may be a desired autoforwarding combination which is masked. Fortunately CATS provides a solution here: The tester can impose constraints so that the autodeletion feature does not delete a message to be forwarded. (Of course a separate CATS run could be used for cases where a message to be forwarded is deleted.)

7. HOW CATS WORKS -- LARGE PROBLEMS

For systems with large numbers of test factors (for which **expand** would generate too many test cases for **cats** to process at once), the **expand** program has been designed to work with **cats** in an iterative mode. In this situation **expand** starts with the largest possible subset of the test factors, and **cats** analyzes the resulting test cases. Then **expand** removes the unnecessary test cases (which do not reduce the number of uncovered combinations) by truncating the **cats** output file. Using the remaining test cases with one or more additional factors, **expand** generates new test cases for **cats** to analyze. This iteration process continues until all the factors are included, and their test cases are analyzed. Thus, large problems are broken into smaller pieces, enabling CATS to handle large numbers of test factors easily. For example, using this procedure CATS has found 240 test cases to cover all pairwise combinations among twenty factors with ten values each.

APPENDIX
HOW MANY TEST CASES DOES CATS RECOMMEND?

This appendix gives formulas for estimating the number of test cases CATS will recommend for a particular test problem. These are empirical results based on experience with CATS.

Formulas for Cases Without Constraints

This section estimates upper and lower bounds for test problems without constraints. The upper bound τ^+ approximates the "worst case" in which all factors have an equal number of values.

Definitions:

n = *number of factors in problem.* $1 \le n$.
m_j = *number of allowed values for* j^{th} *factor.*

List factors in decreasing order:

$$m = \max_{all\ j}(m_j) = m_1 \ge m_2 \ge \cdots \ge m_n$$

l = *number of factors to consider in combination.* $1 \le l \le n$.

$$\binom{n}{l} = \frac{n!}{l!(n-l)!} = \frac{n\cdot(n-1)\cdots(n-l+1)}{l\cdot(l-1)\cdots 1},$$

= *number of combinations of n factors taken l at a time.*

τ = *number of test cases from* CATS *(without constraints).*
τ^- = *minimum number of test cases.*
τ^+ = *approximate maximum number of test cases from* CATS.

$$\tau^- \le \tau \le \tau^+.$$

Formulas:

- Minimum Number of Test Cases:

$$\tau^- = \prod_{j=1}^{l} m_j = m_1 \cdot m_2 \cdots m_l.$$

- Approximate Maximum Number of Test Cases:

$$\tau^+ = A_{l,m}\left\{\sqrt{\ln^2\binom{n}{l} + 1} - 1\right\} + m^l,$$

$$A_{l,m} = \sqrt{\pi^{l-1}(l-1)^{1/2}(m-1)^{l+1/2}}.$$

- Total Possible Test Cases:

$$\prod_{j=1}^{n} m_j = m_1 \cdot m_2 \cdots m_n.$$

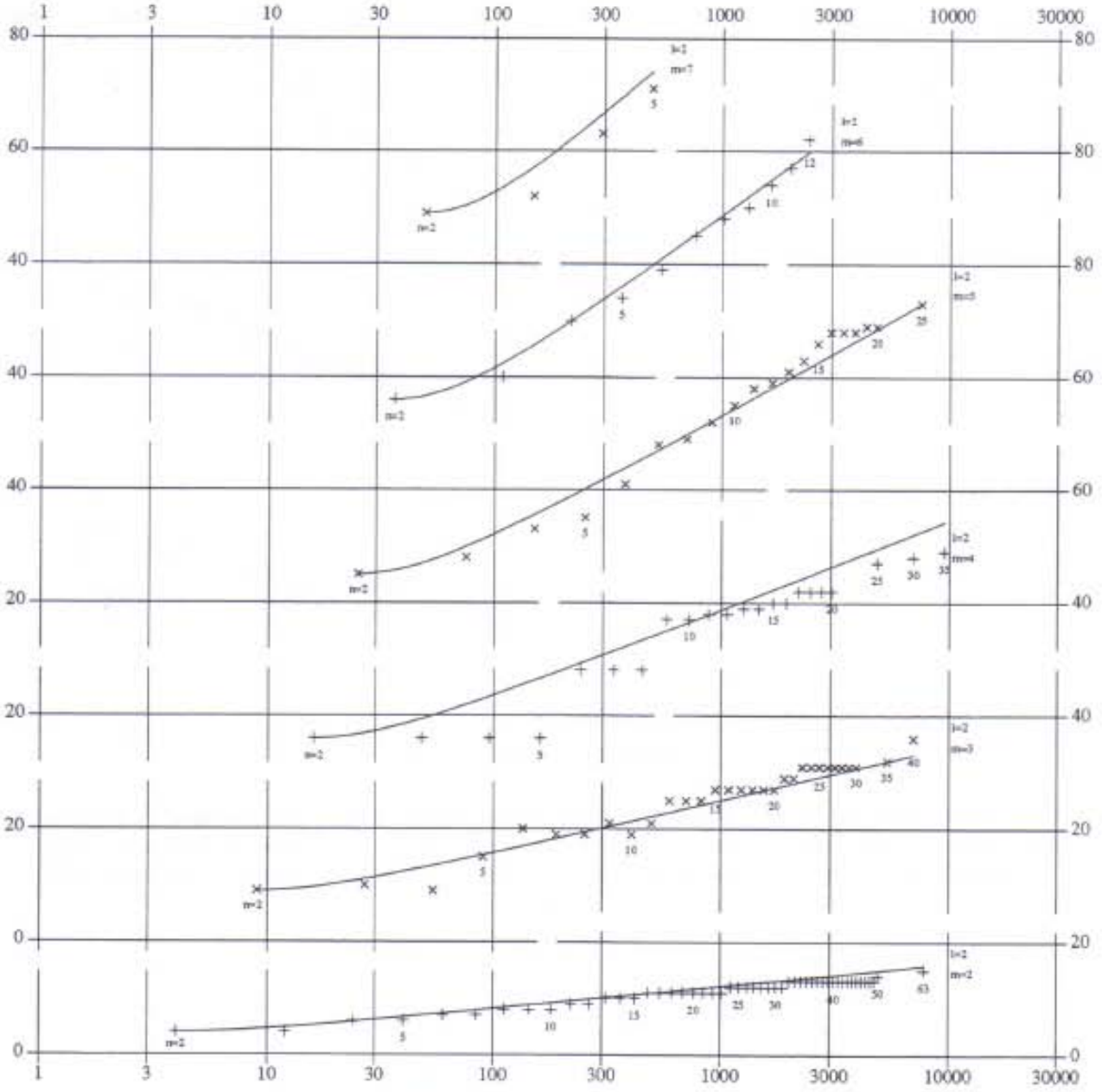
Examples

The table below shows three examples for $n = 4$ factors, considered $l = 2$ at a time, with $m = 6$. In the first two examples CATS suggests the minimum number of test cases possible, because the numbers of factor values m_j fall off from m fairly quickly. The third example illustrates the "worst case." Here all the factors have the same number of values, and the approximate maximum number of test cases is reached.

Numbers of Test Factor Values				Number of Test Cases			
				Minimum	Actual	Approximate Maximum	Total Possible
m_1	m_2	m_3	m_4	τ^-	τ	τ^+	
6	2	2	2	12	12	50	48
6	4	4	2	24	24	50	192
6	6	6	6	36	50	50	1296

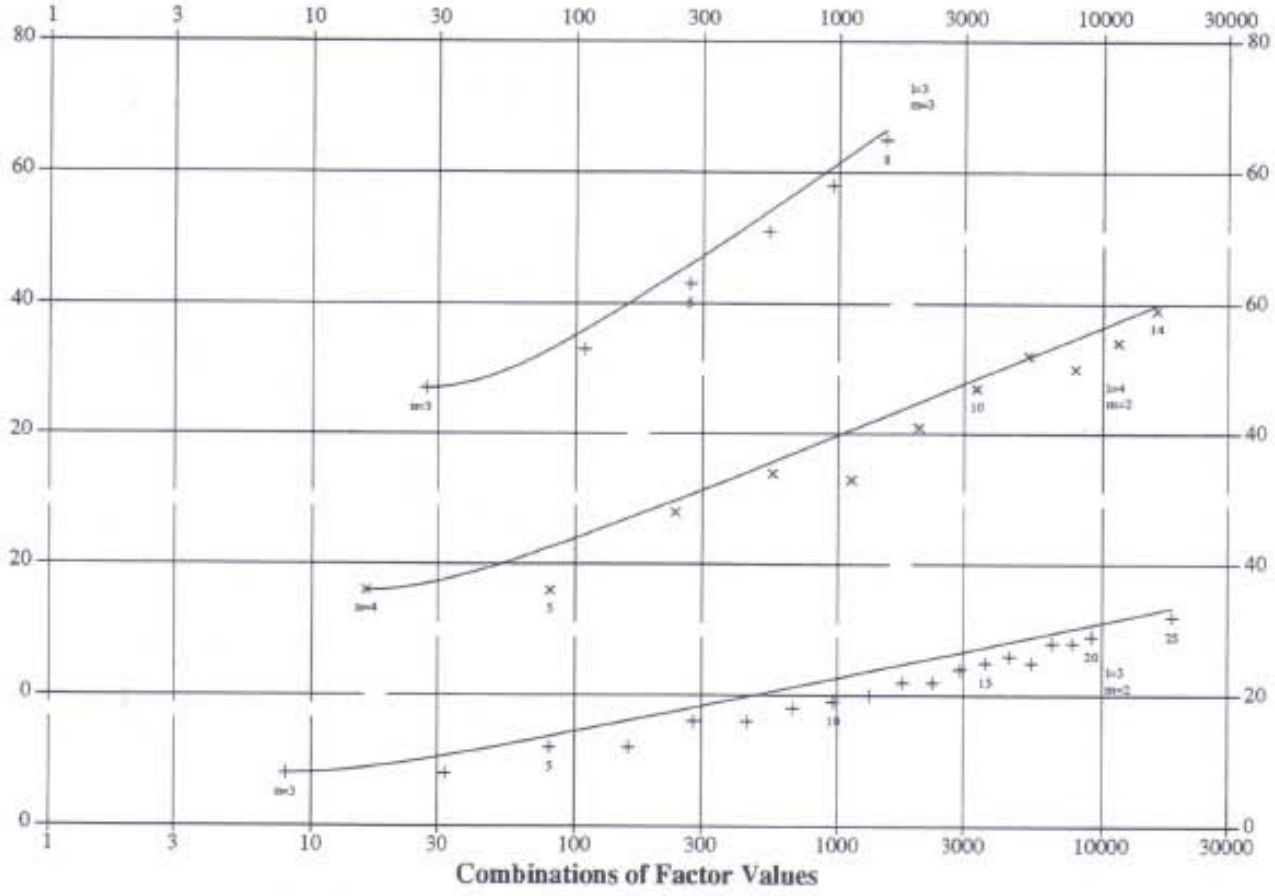
The figures on the following pages compare the approximate upper bound τ^+ with observed numbers of CATS test cases, for several examples in which all factors have the same number of values.

Number of Test Cases When l is 2 and m_j is Constant



Combinations of Factor Values

Number of Test Cases When t is 3 or 4 and m_j is Constant



These figures show that for large numbers of factors n, the number of test cases generated by CATS is approximated by a logarithmic function of the number of factor value combinations to be covered, $m^t\binom{n}{t}$. This means that as the number of factors n gets large, the number of test cases τ increases very slowly with n.

Formulas for Cases With Constraints

When a test problem has constraints, the number of test cases for each group of factor values can be estimated using the formulas of the previous section. Then the number of test cases for the whole problem can be estimated to be between the largest number of test cases for an individual group and the sum of the test cases for all the groups.

τ^c = *Number of Test Cases With Constraints.*
τ_i = *Number of Test Cases for* i^{th} *factor value group in* **expand** *input.*

First Group:	Factor 1 Values Factor 2 Values ⋮ Factor *n* Values **append**	$\rightarrow \tau_1$ cases
Second Group:	Factor 1 Values Factor 2 Values ⋮ Factor *n* Values **append**	$\rightarrow \tau_2$ cases
⋮	⋮	
i^{th} Group:	Factor 1 Values Factor 2 Values ⋮ Factor *n* Values **append**	$\rightarrow \tau_i$ cases
⋮	⋮	

- Minimum and Maximum Numbers of Test Cases With Constraints:

$$\max_{all\ i}(\tau_i) \leq \tau^c \leq \sum_{all\ i} \tau_i$$

REFERENCES

1. Anton T. Dahbura, Krishan K. Sabnani, and M. Umit Uyar, "Algorithmic Generation of Protocol Conformance Tests," *AT&T Technical Journal* **Vol. 69**, No. 1, pp. 101-118 (1990).

2. M. S. Phadke, "Quality Engineering Using Robust Design" (Chapter 3). Prentice Hall (1989).

3. A. Renyi, "Foundations of Probability," Wiley (1971).

4. G. O. H. Katonah, "Two applications (for search theory and truth functions) of Sperner type theorems," *Periodica Math. Hung.* **Vol. 3**, pp. 19-26 (1973).

5. D. J. Kleitman and J. Spencer, "Families of k-independent sets," *Discrete Math.* **Vol. 6**, pp. 255-262 (1973).

6. N. J. A. Sloane, "Covering Arrays and Intersecting Codes," *Journal of Combinatorial Designs* **Vol. 1**, pp. 51-63 (1993).