



A Combinatorial Test Suite Generator for Gray-Box Testing

Anthony Barrett
Daniel Dvorak



Generic Problem

As NASA missions become ever more complex and subsystems become ever more complicated, testing for correctness becomes progressively more difficult. Exhaustive testing is usually impractical, so how does one select a smaller set of test cases that is effective at finding/analyzing bugs?

This problem often addressed by performing Monte Carlo tests and analyzing the results. Unfortunately, this approach does not provide any coverage guarantees, does not provide any help in actually analyzing the results, and limits testing to small regions of the option space.

Example: CEV Launch Pad Abort Simulation

- Simulated Scenario
 - Initiate (abort & control motors ignite)
 - Abort motor burnout
 - Canard deploy & CM reorient
 - LAS jettison
 - Drogue deploy
 - Main parachute deploy
 - Retro fire
 - Touchdown
- Each simulation starts with the setting of 84 floating point parameters.



Image from Apollo



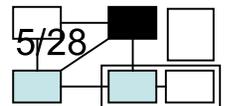
Outline

- Combinatorial testing
- Comparison with random testing
- Gray-box testing
- Components of a test model
- Testgen algorithm
- Experiments
- Conclusions



Combinatorial Testing

- The Challenge
 - Exhaustive testing is usually impractical, so how do you select a smaller set of test cases that is effective at finding bugs?
- A solution
 - Generate test cases automatically in a way that exercises interactions among the many test factors
- It only takes 216 tests to exercise all pairwise interactions among 20 ten-valued parameters



Pairwise Testing Example

Pairwise Interactions

A	B	C	A	B	C	A	B	C
0	0	-	0	-	0	-	0	0
0	1	-	0	-	1	-	0	1
1	0	-	1	-	0	-	1	0
1	1	-	1	-	1	-	1	1

Test Suite

Test Factor:	A	B	C
Test 1:	0	0	0
Test 2:	0	1	1
Test 3:	1	0	1
Test 4:	1	1	0

Only 4 tests to cover all pairwise interactions as opposed to 8 tests for exhaustive testing



Underlying Premise

- The simplest bugs in a program are generally triggered by a single input parameter
- The next simplest bugs are triggered by an interaction between two input parameters
- Progressively more obscure bugs involve interactions between more parameters
 - These are both progressively rarer and harder to test for
- Exhaustive testing involves testing all possible combinations of all inputs.
 - This blows up exponentially with the number of inputs



Does it work?

- Anecdotal evidence has suggested that this small number of tests is enough to catch most coding errors
 - D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, “The Combinatorial Design Approach to Automatic Test Generation.” *IEEE Software*, 13(5):83-87. 1996.
 - I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing.” *Proc. 19th International Conference on Software Engineering (ICSE '97)*. 1997.
 - K. Burr and W. Young, “Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage.” *Proc. International Conference on Software Testing, Analysis, and Review (STAR)*, San Diego, CA, October, 1998.
 - D. R. Wallace and D. R. Kuhn, “Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data.” *International Journal of Reliability, Quality and Safety Engineering*, 8(4):351-371. 2001.

Comparison with Random Testing

- Pairwise Testing Approach
 - Build minimal set test cases to cover all pairwise interaction of values from each set.



$$T_1 = \{low_1, \dots, high_1\}$$

$$T_2 = \{low_2, \dots, high_2\}$$

$$T_3 = \{low_3, \dots, high_3\}$$

...

Comparison with Random Testing

- Random Testing Approach
 - Build test cases by randomly selecting values from each set.



$$T_1 = \{low_1, \dots, high_1\}$$

$$T_2 = \{low_2, \dots, high_2\}$$

$$T_3 = \{low_3, \dots, high_3\}$$

...

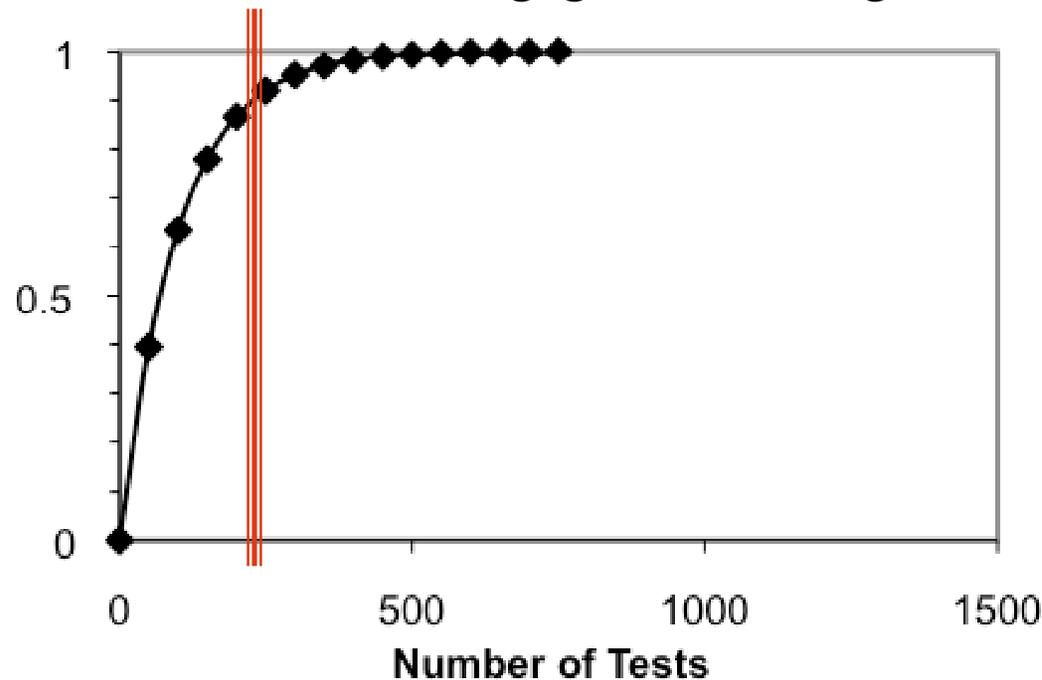


Issues With Random Approach

- How many tests are enough?
 - Typically just perform tests until some time limit is reached, resulting in large numbers of tests
- What kind of guarantee does this provide?
 - At best a probabilistic guarantee...

Probability of Finding Some Pairwise Problem

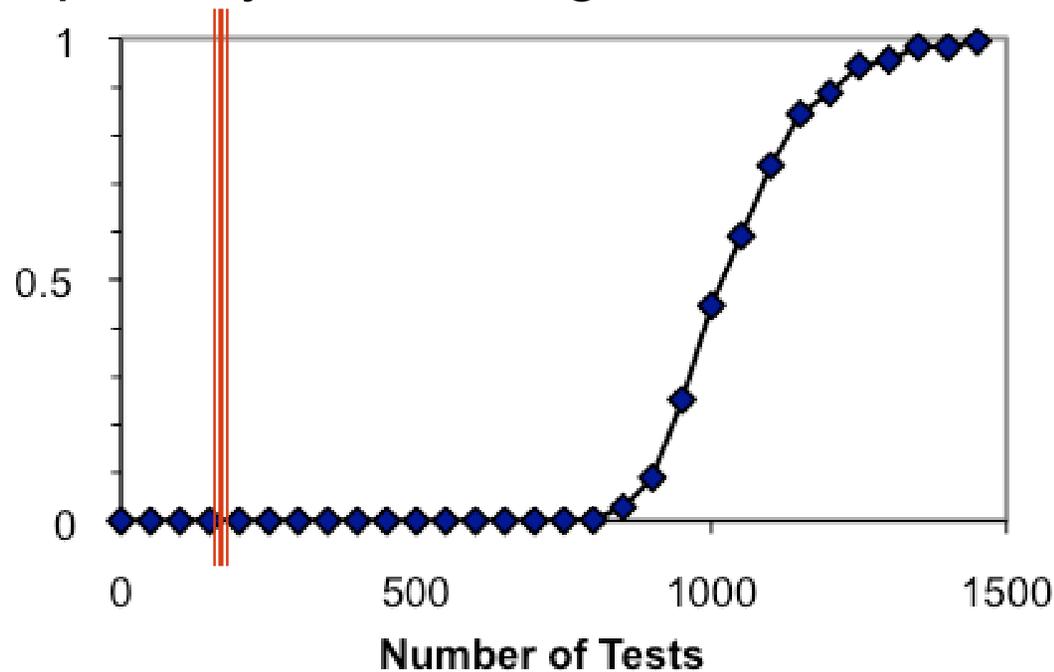
- For 20 ten-valued parameters (10^{20})
 - When is random testing good enough?

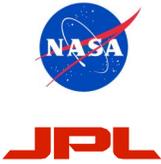




Probability of Finding Any Pairwise Problem

- For 20 ten-valued parameters (10^{20})
 - Not quite if you want a guarantee





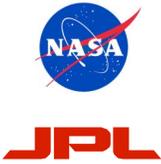
Performance Issues

- While it takes 216 tests to cover pairwise interactions among 20 ten-valued parameters, it takes:
 - 1000 tests to cover all 3-way interactions of just 3 ten-valued parameters.
 - 10000 tests to cover all 4-way interactions of just 4 ten-valued parameters.
- The number of tests gets exponentially large for higher combinatorial tests.



Targeted Interaction Testing

- Pairwise testing is too limiting.
 - Blindly using it is not a best practice.
- There is no replacement for a cognizant test engineer.
 - The objective of combinatorial test suite generation is to take a set of test space coverage requirements and compute a conforming test suite containing as few tests as possible.



Gray-Box Testing

- Black Box Testing
 - Build a test suite with no knowledge of system internals
 - e.g. Monte Carlo and Pairwise testing
- Gray Box Testing
 - Build a test suite given partial knowledge of system internals
 - Requires mechanisms to tune test generation



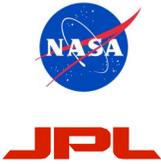
Testgen Modeling Language Features

- Explicitly include particular combinations
- Explicitly exclude particular combinations
- Require different m -factor combinatorial coverage of specific subsets of factors
- Nest factors by tying the applicability of one factor to the setting of another



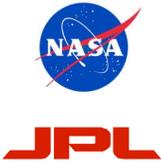
Testgen Model Features

- Input Factors
 - $[T_1 \dots T_k]$ – k enumerated sets denoting possible levels for k factors
 - Pairwise Example
 - $\{\{0,1\} \{0,1\} \{0,1\}\}$
- Output Tests
 - M – a set of k -element test vectors
 - Pairwise Example
 - $[0,0,0], [0,1,1], [1,0,1], [1,1,0]$



Testgen Model Features

- Nested factors denote factor interactions
 - $NEST \subseteq \{(N(i), c(i), i) \mid 1 \leq N(i) < i \leq k \text{ and } c(i) \in T_{N(i)}\}$, where $N(i)$ and $c(i)$ denote that the i^{th} factor applies only when the $N(i)^{\text{th}}$ factor is level $c(i)$.
- e.g. Can only test i^{th} factor if the earlier $N(i)^{\text{th}}$ factor is not an illegal value.



Testgen Model Features

- Seed Test Cases
 - $SEEDS \subseteq (T_1 \cup \{“*”\}) \times \dots \times (T_k \cup \{“*”\})$, conforming to NEST and denoting specific combinations that must occur in returned tests.
 - Pairwise example
 - $SEEDS = \{[0,0,0]\}$ -- all generated test suites will have this vector.

Testgen Model Features

- Excluded combinations
 - $EXCLUDE \subseteq (T_1 \cup \{‘*’\}) \times \dots \times (T_k \cup \{‘*’\})$, consistent with elements of SEEDS and denoting specific combinations that cannot occur in returned tests.
 - e.g. an illegal combination of SEEDS and EXCLUDE
 - $[1\ 0\ 2\ 3\ *\ * \ 2\ *\ * \ 7\ *\ * \ *\ * \ 3\ *\ *] \in SEEDS$
 - $[*\ 0\ 2\ *\ * \ *\ 2\ *\ * \ *\ * \ *\ * \ *\ * \ *\ *] \in EXCLUDE$

Testgen Model Features

- Mixed Strength Coverage
 - $\text{COMBOS} \subseteq \{(n:t_1\dots t_j) \mid n \leq j \ \& \ 1 \leq t_1 < \dots < t_j \leq k\}$
denoting the required n -way combinations for specific subsets of n or more factors.
 - e.g. $\{(2:1 \ 2 \ 3)\}$ gives us pairwise of first three of six factors
 - $[0 \ 0 \ * \ * \ * \ *] [0 \ 1 \ * \ * \ * \ *] [1 \ 0 \ * \ * \ * \ *] [1 \ 1 \ * \ * \ * \ *]$
 - $[0 \ * \ 0 \ * \ * \ *] [0 \ * \ 1 \ * \ * \ *] [1 \ * \ 0 \ * \ * \ *] [1 \ * \ 1 \ * \ * \ *]$
 - $[* \ 0 \ 0 \ * \ * \ *] [* \ 0 \ 1 \ * \ * \ *] [* \ 1 \ 0 \ * \ * \ *] [* \ 1 \ 1 \ * \ * \ *]$

Testgen algorithm

Testgen($[T_1 \dots T_k]$, SEEDS, NEST, EXCLUDE, COMBOS)

1. $M \leftarrow$ SEEDS.
2. For $i \leftarrow 1$ to k do:
 3. $\pi_i \leftarrow$ {combinations that end with T_i , conforming with COMBOS, NEST, and EXCLUDE};
 4. If π_i is not empty then
 5. **Grow tests in M to cover elements of π_i**
 6. **Add tests to M to cover leftover elements of π_i**
 7. For each test $m \in M$ do:
 8. For $i \leftarrow 1$ to k do:
 9. If $m[i] = '*'$ then
 10. Randomly set $m[i]$ to a value from T_i
(conforming with EXCLUDE and NEST).
 11. Return the test suite M .



Growing existing tests

To grow tests in M to cover elements of π_I

1. For each $c \in T_i$ in random order do:
 2. Find $m \in M$ where $m[i] \in \{ '*', c \}$ & let $m[i] \leftarrow c$
(conforming with EXCLUDE and NEST).
 3. Remove elements from π_i that are covered by tests.
 4. For each test $m \in M$ if π_i not empty do:
 5. If $m[i] = '*'$ then
 6. Set $m[i]$ to a level covering the most elements of π_i
(conforming with EXCLUDE);
 7. Remove covered elements from π_i



Adding new tests

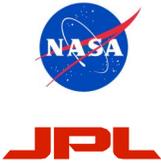
To add tests to M to cover leftover elements of π_I

1. For each P left in π_i do:
2. Try to set '*' entries of some $m \in M$ to cover P (avoiding EXCLUDE);
3. If P still uncovered add a new test to M for P .



Experiments

- Cannot compare with other algorithms in the field due to none having the same modeling capabilities.
- Also, testgen is much faster, facilitating the handling of over 1000 factors.
 - Solved pairwise problem of 1000 3-level factors in 22 seconds with 48 tests



Experiments

- Pairwise quality is comparable with existing algorithms.

Problem	IPO	AETG	PICT	Testgen
3^4	9	11	9	9
3^{13}	17	17	18	19
$4^{15}3^{17}2^{29}$	34	35	37	35
$4^{13}3^{39}2^{35}$	26	25	27	29
2^{100}	15	12	15	15
10^{20}	212	193	210	212



Conclusions

- While random testing is simpler, combinatorial testing provides coverage guarantees
- Simple pairwise testing suffers the same limitations as any other black-box approach
- Testgen facilitates tuning test suite generation to take advantage of gray-box information.
- The Testgen algorithm is fast enough to enable the exploration/comparison of multiple test-suite specification models



Copyright

- Copyright 2009 California Institute of Technology. Government sponsorship acknowledged.