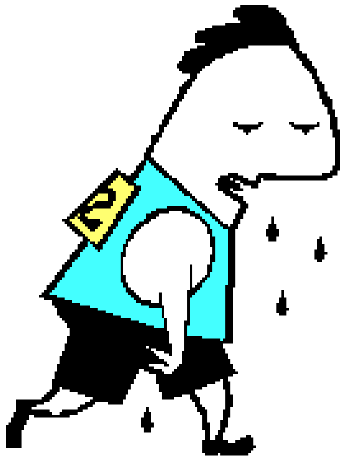
Combinatorial Test Design

Itai Segall

IBM Haifa Research Labs

The Problem

- Exhaustive testing is impossible: many, many billions of potential tests
- When people create tests by hand, they tend to produce tests which add little value in discovering defects
- “More tests = better” is a belief commonly held by project sponsors



The Goal

- We would like to:
  - Use our time efficiently
  - Control the risks we are taking
  - Know what we tested
  - … and what we did not test
- We need to:
  - Identify tests that are likely to expose defects
  - In a demonstrably smarter approach

Combinatorial Test Design (CTD)

- **Combinatorial Test Design (CTD)** is one such approach
- Uses advanced mathematics to:
  - Dramatically reduce the number of test cases needed
  - While ensuring coverage of conditions and interactions
- The ultimate shift left
  - Handle coverage concerns when defining the test plan
- Systematic planning of tests
- Maximizes the value of each tested scenario
  - Significant reduction in the number of tests
- Controlled risk
- Easy to review

  → Minimizes omissions

Case Study - High Availability Recovery Subsystem enhances testing with Combinatorial Test Design

Goals:

- Regain customer trust after two outages in six months
- Avoid further outages
- Shorten testing cycles in SIT

Solution:

- Cooperated with the development and test teams to create a model of the test space of the system
- Applied combinatorial test design to suggest 21 test (from the 7800 possible)

Results:

- Reduced testing cycle by an order of magnitude
- Improved defect discovery and decreased defect escapes into production – found 20 bugs
- No more outages in two years

CTD Pilots for FVT Teams

- Goal: Increase quality of testing while reducing test redundancies across FVT
- Expected impact: increased defect discovery rate during FVT testing, reduce defect escapes to later test phases and to the field
- CTD was deployed by FVT teams to replace manual test planning by systematic
- e
-
- s /
- on,

"Our team had the opportunity to take part in the pilot and adopt CTD methodology in our FVT work, which has no doubt brought fresh blood to our routine testing – not only refined the testing coverage and improved productivity, but also turned FVT work into a more interesting and exciting way.
CTD pilot is a valuable experience to us, we'll definitely continue it"

CTD has proven useful across multiple industries and project types

- **Insurance**
  - Modeling the claims adjudication process identified 41 test cases to close potential gaps in system test plan of 6,000 test case
- **Telco**
  - Reverse-engineered model present in 117 hand-written test cases that had ~70% coverage of pairwise interaction; could be replaced by 12 test cases with 100% coverage
- **Brokerage**
  - Manual test plan had 256 test cases with ~40% good-condition coverage and ~5% bad-condition coverage; CTD identified 71 replacement test cases that give 100% coverage of good and bad conditions, and reduced test data requirements by 25%
- **Manufacturing and sales**
  - Large ERP implementation needed to select optimal subset of test cases to meet implementation deadline; for one domain 12% of tests could be omitted without reducing coverage; for another domain, 58% of tests could be omitted without reducing coverage

So… How does this magic work ?

Toy Example – Online Shopping System

Step 1 – identify parameters (points of variability):

- Availability
- Payment Method
- Carrier
- Delivery Schedule
- Export Control

Toy Example – Online Shopping System – cont.

Step 2 – identify values:

Availability	**Payment**	**Carrier**	**Delivery Schedule**	**Export Control**
▪Available ▪Not in Stock ▪Discontinued ▪No Such Product	▪Credit ▪Paypal ▪Gift Voucher	▪Mail ▪UPS ▪Fedex	▪One Day ▪2-5 Working Days ▪6-10 Working Days ▪Over 10 Working Days	▪True ▪False

A test is represented by an assignment of exactly one value to each parameter

4 x 3 x 3 x 4 x 2 = 288 combinations

Do we really need to test all combinations?

Levels of interaction

- Suppose there is a bug, and Credit does not work well with One Day delivery
- Any test that includes Credit and a One Day delivery will expose that bug
  - There are 24 such tests

- Suppose Credit does not work well with a One Day delivery, but only with Fedex
- Any test that includes Credit, a One Day delivery, and Fedex will expose that bug
  - There are 8 such tests

- We call the first case a level two interaction, and the second case a level three interaction

Do we really need to test all combinations?

The root cause analysis of many bugs shows they depend on a value of one variable (20%-68%)

Most defects can be discovered in tests of the interactions between the values of two variables (65-97%)

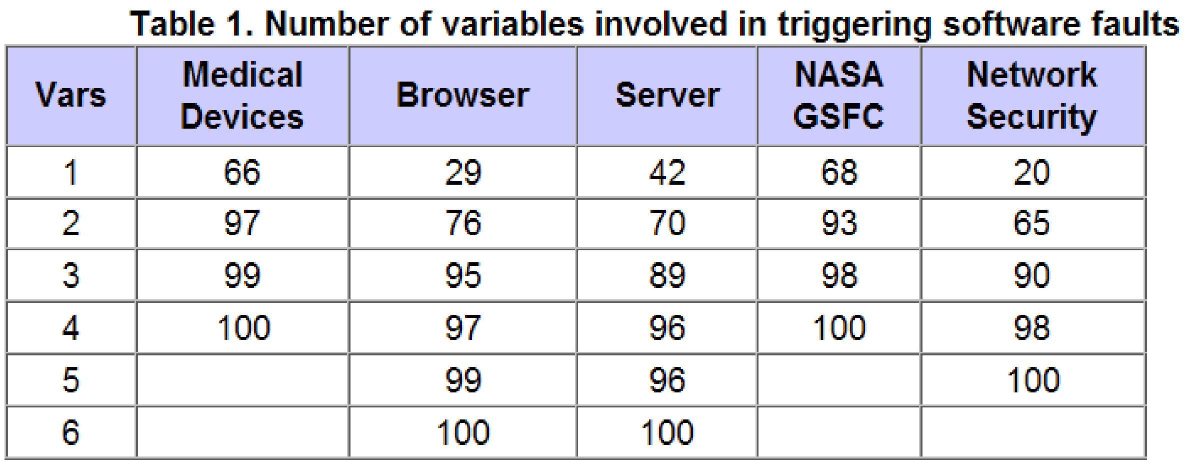


Table 1. Number of variables involved in triggering software faults

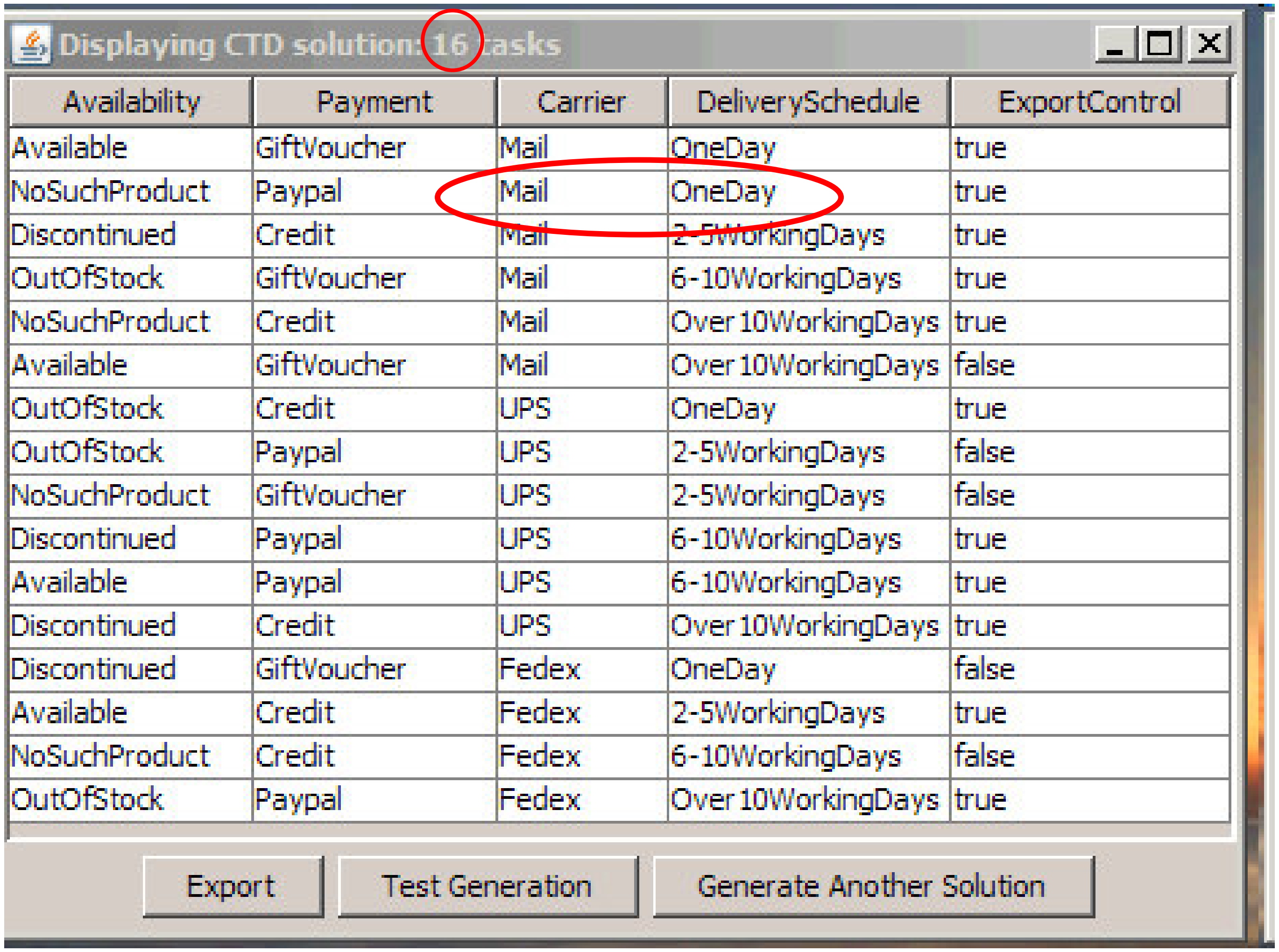
Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security
1	66	29	42	68	20
2	97	76	70	93	65
3	99	95	89	98	90
4	100	97	96	100	98
5		99	96		100
6		100	100		

- Source http://csrc.nist.gov/groups/SNS/acts/ftfi.html

Combinatorial Test Design (CTD)

- To balance cost and risk, we select a subset of tests that covers all the interactions of variables at some level of interaction (pairs, three-way, etc.)
- A combinatorial test design (CTD) algorithm finds a small test plan that covers 100% of a given interaction level

Complete pairwise coverage (one of many) for shopping system



Availability	Payment	Carrier	DeliverySchedule	ExportControl
Available	GiftVoucher	Mail	OneDay	true
NoSuchProduct	Paypal	Mail	OneDay	true
Discontinued	Credit	Mail	2-5WorkingDays	true
OutOfStock	GiftVoucher	Mail	6-10WorkingDays	true
NoSuchProduct	Credit	Mail	Over10WorkingDays	true
Available	GiftVoucher	Mail	Over10WorkingDays	false
OutOfStock	Credit	UPS	OneDay	true
OutOfStock	Paypal	UPS	2-5WorkingDays	false
NoSuchProduct	GiftVoucher	UPS	2-5WorkingDays	false
Discontinued	Paypal	UPS	6-10WorkingDays	true
Available	Paypal	UPS	6-10WorkingDays	true
Discontinued	Credit	UPS	Over10WorkingDays	true
Discontinued	GiftVoucher	Fedex	OneDay	false
Available	Credit	Fedex	2-5WorkingDays	true
NoSuchProduct	Credit	Fedex	6-10WorkingDays	false
OutOfStock	Paypal	Fedex	Over10WorkingDays	true

Interaction level 2 has 16 test cases – 95% reduction from 288

Why do we need restrictions?

- Impossible or irrelevant combinations, for example:
  - Mail Carrier with One Day Delivery Schedule
  - Fedex Carrier with Over 10 Working Days Delivery Schedule
- Naturally we cannot create and run actual tests that contain impossible combinations, so we need to state in advance what should be excluded

Why not just skip tests that contain impossible/irrelevant combinations?

- Each test in the CTD test plan may cover multiple unique legal combinations

- By skipping a test we will lose all these combinations, and no longer have 100% interaction coverage

What are restrictions?

- Restrictions are rules that determine which combinations are included and which are excluded from the model
- Combinations that are excluded from the model will never appear in the test plan
  - So it is important to define them carefully
- Many ways to define restrictions:
  - Force remodeling to separate unconstrained inputs
  - Explicit enumeration of illegal tests
  - Excluding combinations
  - Boolean expressions/predicates
  - If-then-else statements
  - Trees
- Related research challenges:
  - Supporting restrictions in CTD algorithms
  - Reviewing the effect of restrictions (e.g., a cross-product view)
  - Debugging restrictions
  - Simplifying restrictions by introducing new constructs

Test Plans vs. Actual Tests

- CTD tools generate a test plan, not actual tests
- Extracting actual tests from the generated test plan may be a laborious task – generate data, generate test environments, etc.
- However, in many cases the test plan scheme can be easily used as input to a data driven test automation framework
- Alternative approach – test/data selection (discussed shortly)

The History of Combinatorial Test Design

- Evolved from Design of Experiments (DoE) as an application for software testing
  - DoE is a methodology to change values of a number of test factors, and measure corresponding change in the output to learn about a cause-effect system
  - Useful for determining important factors
- Classical DoE started in the 1920s
- In the mid-40s, orthogonal arrays were introduced by C. R. Rao to set the experiment plan
- From the mid-80s, orthogonal arrays were used for combinatorial testing of software and systems
  - R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing", Communications of the ACM, 1985
  - K. Tatsumi, "Test Case Design Support System", ICQC, 1987

Orthogonal and Covering Arrays

- An orthogonal array, $OA(N; t; k; v)$, is an $N \times k$ array on v symbols with the property that every $N \times t$ sub array contains each ordered subset of size t from the v symbols the same number of times
- A covering array, $CA(N; t; k; v)$, is an $N \times k$ array on v symbols with the property that every $N \times t$ sub array contains each ordered subset of size t from the v symbols at least once
- Fixed-value versus mixed-values
- t-way CTD ≡ covering array of strength t

Current Research on Combinatorial Test Design

- www.pairwise.org lists 36 tools for combinatorial testing

- Active research areas: [NL 11]
  - Test case generation (CTD algorithms) – over 50 papers on this topic alone
  - Applying CTD to various types of applications – 17 papers (mostly experience reports)
  - Proper handling of restrictions to avoid invalid test cases – 6 papers
  - Identifying parameters, values and restrictions – 5 papers
  - Contribution of CTD to improvement in software quality – 5 papers
  - Test case prioritization (scheduling) – 4 papers
  - Failure diagnosis – 4 papers

[C. Nie, H.Leung, “A Survey of Combinatorial Testing”, ACM Computing Surveys, 2011]

Approaches for Generating a Combinatorial Solution

3 Main approaches: [CDS 07]

- Mathematical Construction
  - Efficient, usually optimal
  - Not applicable to the general case (e.g., in the presence of restrictions)
- Greedy Algorithms
  - Heuristic-based incremental construction
  - Sub-Optimal
- Meta-Heuristic Search
  - Search-based algorithms, e.g., simulated annealing, genetic algorithms and tabu search
  - Converge to near-optimal

[M. Cohen, M. Dwyer, J. Shi, “Interaction testing of highly-configurable systems in the presence of constraints”, ISSTA, 2007]

An Example of A Greedy Algorithm – In-Parameter-Order (IPO)

- Implemented in FireEye software (a.k.a ACTS), developed by NIST, University of Texas at Arlington, and George Mason University
- Based on IPO algorithm developed by Kuo-Chung Tai and Jeff Lei
- Start with a covering array of k – 1 columns
- Horizontal Growth: add a column to best preserve the t-way coverage property
- Vertical Growth: restore the t-way coverage property by adding rows

0	0	0	**0**
0	1	1	**1**
1	0	1	**1**
1	1	0	**0**
1	**1**	**1**	**0**

- Improvement for horizontal growth:
  - Instead of fixed row order, allow row order to be greedily determined
  - Use dynamic programming

Source: http://www.mit.edu/~miforbes/forbes_SURF-2007.pdf

A Different View of the Combinatorial Test Design Problem – A Subset Selection Problem

- The set of all valid tests is denoted by *S*(*P;V;R*)
  - P are the parameters, V the values, R the restrictions
- Our goal: select a small subset S' of S, so that:
  combination of size t appears in S → it appear also in S'
- In other words, S' preserves all interactions of size t that occur in the set of valid tests

[I. Segall, R. Tzoref-Brill, E. Farchi, "Using Binary Decision Diagrams for Combinatorial Test Design", ISSTA'11]

BDD-based Approach for Solving Combinatorial Test Design

- Viewing CTD as a subset selection problem assumes we can represent all valid tests
- To enable this for real-life test spaces with many billions of valid tests we use Binary Decision Diagrams (BDDs) [STF 11]
- BDDs are a compact data structure for representing sets
- Set operations are performed efficiently, directly on the compressed representation
  - Intersection, Unification, Complementation, Existential quantification, Universal quantification

[I. Segall, R. Tzoref-Brill, E. Farchi, “Using Binary Decision Diagrams for Combinatorial Test Design”, ISSTA’11]

Test Space Representation

$$\boxed{\begin{array}{l} \text{Restriction}_i \rightarrow \text{Allowed}_i \\ \text{Valid} = \bigcap_i \text{Allowed}_i \\ \text{Invalid} = \text{Valid}^c \end{array}}$$

"Is a test t valid?" $\equiv$ $t \cap \mathit{Valid} \neq \emptyset$

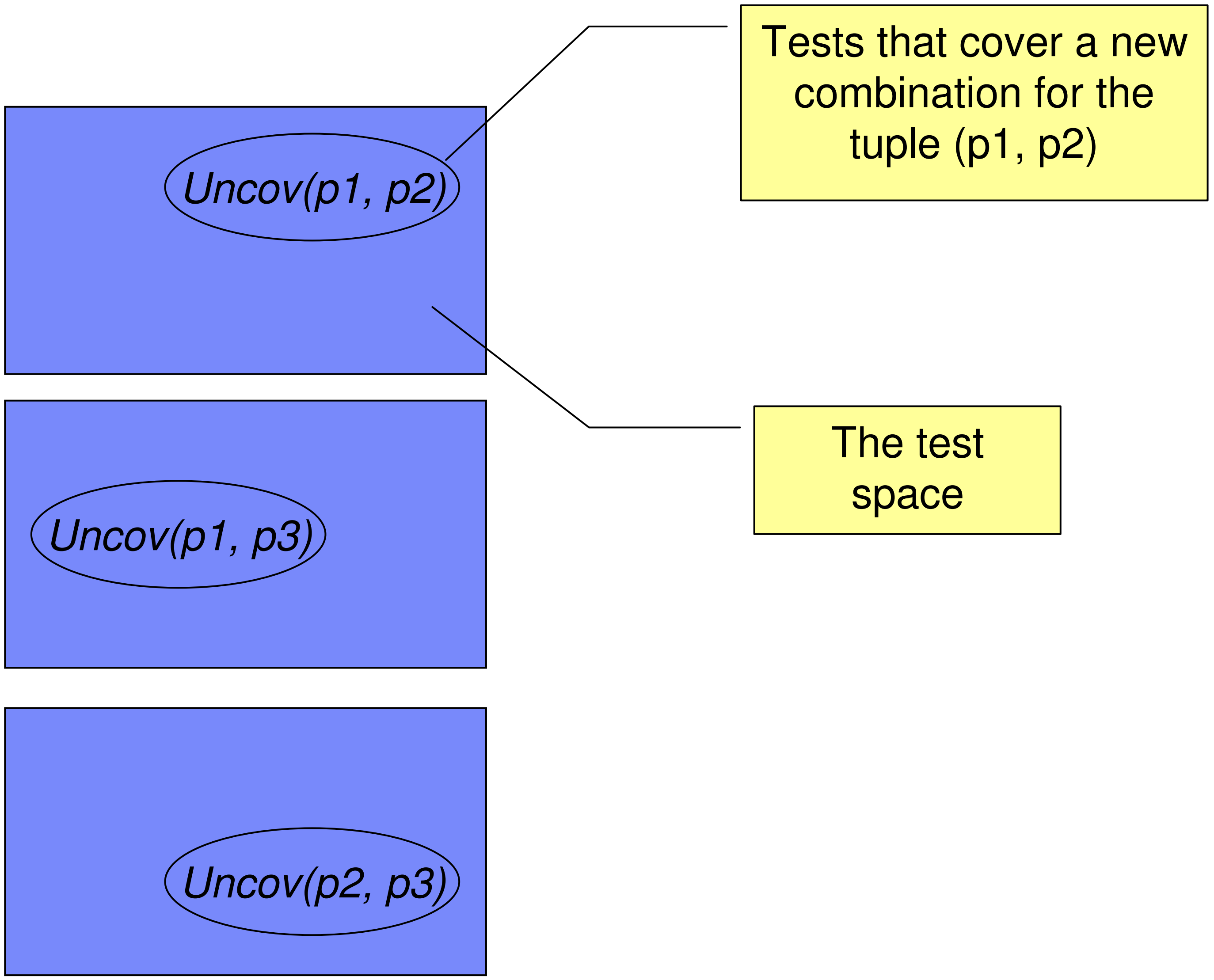
Also useful for reviewing the effect of restrictions on the test space

BDD-Based CTD Algorithm

- A greedy algorithm
- At every step, choose a test out of all valid tests that covers as many new combinations as possible, and adds it to the solution

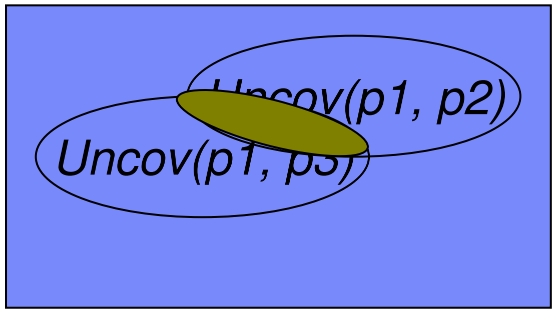
- Consider a toy example
  - 3 parameters p1, p2, p3
  - Pairwise coverage
- The set of parameter tuples for which coverage is required:
  T = { {p1, p2}, {p2, p3}, {p1, p3} }
- *uncov(p1, p2)* = The set of all valid tests that cover a new combination for {p1, p2}
  - Easily computed and maintained

BDD-Based CTD Algorithm – The Data



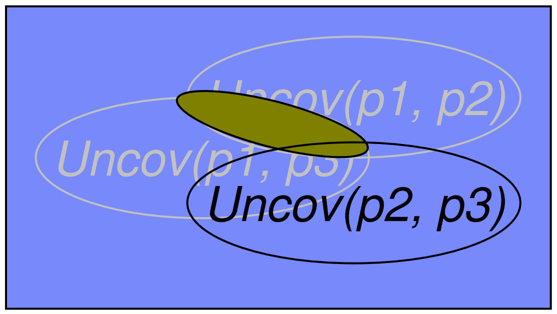
BDD-Based CTD Algorithm – The Main Iteration

Reminder: Goal – Find a test that covers many new combinations



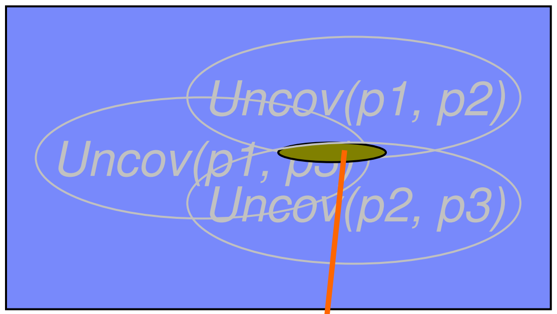
BDD-Based CTD Algorithm – The Main Iteration

Reminder: Goal – Find a test that covers many new combinations



BDD-Based CTD Algorithm – The Main Iteration

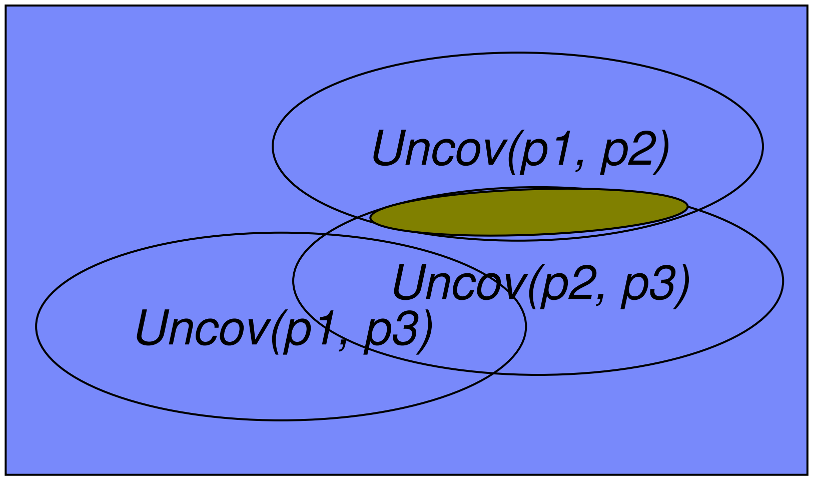
Reminder: Goal – Find a test that covers many new combinations



Any test here covers one new combination for each tuple

Choose one randomly
Update the *Uncov* sets

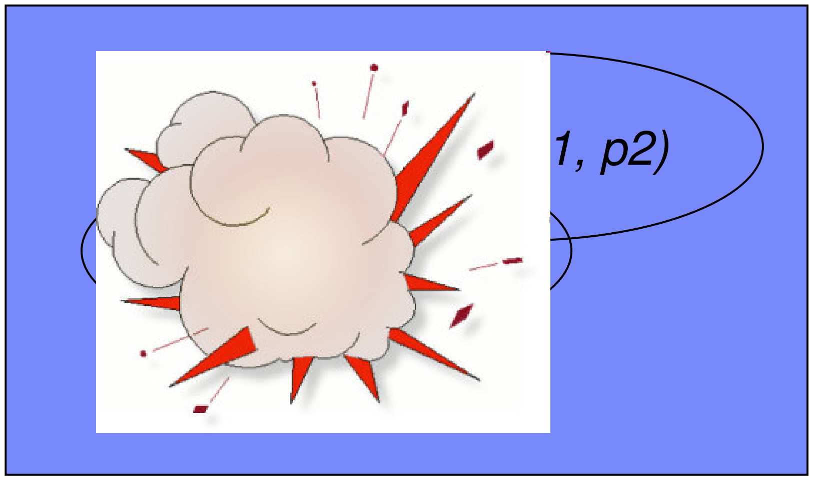
BDD-Based CTD Algorithm –
Variations of the Main Iteration (I)



Order Matters !

First sort the tuples by decreasing size of their *Uncov* set

BDD-Based CTD Algorithm – Variations of the Main Iteration (II)



BDD Operations Might "Explode"

If intermediate result becomes too large:

1. Stop conjuncting
2. Draw multiple random tests from the result, and take the best one

Further Benefits from the BDD Representation

- As opposed to extending partial tests, this approach removes the need to handle restrictions throughout the algorithm
  - –Handled only at the initial stage of constructing the set of all valid tests
  - –Never step out of the valid tests domain

- Same algorithm can run on different "worlds" of tests to select from
  - –"Classical" – set of all valid tests
  - –Test Selection – set of all existing tests
  - –Combinations of the two, and other crazy definitions

Additional Challenges in Real-Life Deployment of CTD

- How to define optional attributes? – A common pitfall
- How to reflect multiple selections from a category?
- How to deal with continuous domains?
- How to reflect symmetry in the model?
- How to capture possible orders?

Captured by Modeling Patterns [STZ 12] :

- Cases that often repeat in different CTD models of different types and from different domains
- Solutions for how to translate them into parameters, values and restrictions already exist and can be reused

[I. Segall, R. Tzoref-Brill, A. Zlotnick “Common Patterns in Combinatorial Models”, IWCT 2012]

Modeling Patterns Address Common Pitfalls in Combinatorial Modeling

Correctness

Failing to capture the intention correctly

Completeness

Omitting an important part of the test space from the model

Redundancy

Explicitly enumerating different cases that are actually equivalent

Modeling Patterns – Sample

- The paper describes 5 common patterns
- We'll sample 2 here:
  - –Optional and conditionally-excluded values (addresses correctness and completeness)
  - –Multiplicity and Symmetry (addresses redundancy)

Optional and Conditionally-Excluded Values (completeness, correctness)

- Example I: online form with fields email (mandatory) and home address (optional)
- Naïve model:
  - Email – valid / invalid
  - HomeAddr – valid / invalid

Incomplete !
Does not distinguish between empty and invalid home address

- Much better:
  - Email – valid / invalid
  - HomeAddr – valid / **empty** / invalid

Optional and Conditionally-Excluded Values – cont.

- Example II: online form with email address, home address, and “is billing by email”
  - Home address used only if not billing by email

Optional and Conditionally-Excluded Values – cont.

- Naïve model:
  - Email – valid / invalid
  - HomeAddress – valid / invalid
  - BillByEmail – true / false

  - Not allowed:
    - HomeAddress == “valid” && BillByEmail
    - HomeAddress == “invalid” && BillByEmail

Incorrect !

Billing by email was entirely excluded

Optional and Conditionally-Excluded Values – cont.

- Much Better:
  - Email – valid / invalid
  - HomeAddress – valid / invalid / **empty** / **NA**
  - BillByEmail – true / false

  - Not allowed:
    - **HomeAddr != “NA” && BillByEmail**
    - **HomeAddr == “NA” && ! BillByEmail**

Multiplicity and Symmetry (redundancy)

- Applicable when:
  - Multiple elements of the same type
  - Their interactions with other elements are equivalent
- Example: two hosts (x86 / PowerPC), two storage devices (Enterprise / Mid-Range / Low-End)

Multiplicity and Symmetry – cont.

- Naïve Model:
  - Host1 – x86 / PowerPC
  - Host2 – x86 / PowerPC
  - Storage1 – Enterprise / MidRange / LowEnd
  - Storage2 – Enterprise / MidRange / LowEnd

Warning: Possible Redundancy !

xample, (host1=x86, Storage1=LowEnd)
to (host2=x86, Storage1=LowEnd)

Warning: Possible Redundancy !

For example, (host1=x86, host2=PowerPC)
may be equivalent to (host2=x86, host1=x86)

Multiplicity and Symmetry – cont.

- Possible Solution:
  - numX86Running – 0 / 1 / 2
  - numPowerPCRunning – 0 / 1 / 2
  - numEnterpriseRunning – 0 / 1 / 2
  - numMidRangeRunning – 0 / 1 / 2
  - numLowEndRunning – 0 / 1 / 2

  - Not allowed:
    - numX86Running + numPowerPCRunning != 2
    - numEnterpriseRunning + numMidRangeRunning + numLowEndRunning != 2

Multiplicity and Symmetry – cont.

- An alternative approach:
- A tool and algorithm that supports designating equivalent parameters
  - And whether internal interactions are equivalent
- Requires a tool and algorithm that support dynamically changing coverage requirements

Open Challenges

- Much better handling of existing data/tests
  - Existing test plans are almost never in the form we want them
  - In fact, often they're not even structured
- Lifecycle of combinatorial models
  - Software systems evolve
  - The combinatorial models used for testing them should evolve respectively
  - Strive for minimal changes to the test suites
- Model reuse, multi-model support
- High-level combinatorial modeling language

Summary

- Combinatorial Test Design is an effective test planning technique
  - Shift left: Handle your coverage concerns as early as possible
- Reduced redundancies, controlled risks
- Applicable at any stage of the development life cycle
  - Unit testing, function testing, system testing, integration testing
- Interesting combinatorial problem
- Many challenges for real-life deployment
  - Restrictions, negative testing, modeling patterns, concrete test case generation, utilization of existing test cases, software evolution, and more…
- We believe that experience from real-life deployme research on CTD
  - Identifies the pain points that limit wide deployment by indu
  - Enables suitable solutions that can be adopted by industry

Thank You For Listening