TAROT Summer School, Porto (PT), July 3, 2014

Lightweight Testing for Configurable Systems

Marcelo d'Amorim
Federal University of Pernambuco, Brazil

What is a Configurable System?

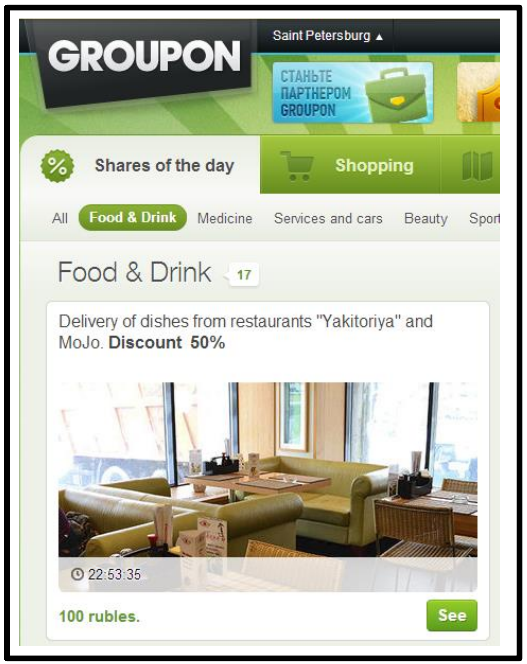
A system made up of several named parts, one of which is the base. Those parts often share functionality.

Why Configurable Systems?

Improve Productivity

Ability to add or remove features as new demands emerge.

Used in practice



GCC

MySQL

ORACLE
BERKELEY DB

Hot topic in research

- Several papers accepted in recent editions of ICSE, ASE, and FSE.
- Specialized venues. E.g.,
  - Modularity (previously AOSD)
  - Software Product Line Conference (SPLC)
  - Intl. Conference on Generative Programming (GPCE)

Contents

- Background
- Testing Configurable Systems
  - What to test and what configurations to test?
  - Test adequacy
  - Interpreting test results
  - Debugging configurations
  - GCC
- Research

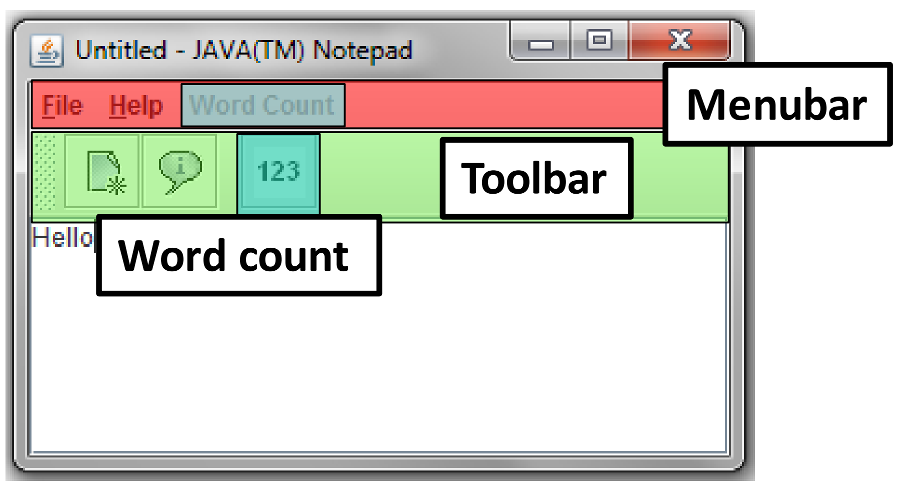
Material of this talk is available at
http://goo.gl/ctPcqe

BACKGROUND

Terminology

- **Feature**
  - Distinct system functionality
  - Example

Notepad



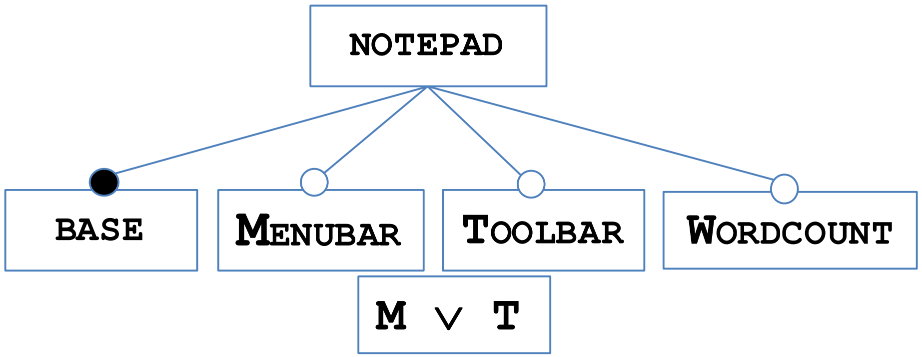
Terminology

- **Feature Option**
  - Features are controlled through input options
    - The value "true" indicates enable for boolean options
  - Options need not to be boolean
    - In eCos (embedded OS), most options are non-boolean
      - ~54% of options are non-boolean (e.g., number and string)
      - "A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System", Passos *et al.*, FOSD, 2011
    - In Apache Web Server, most options are boolean
      - ~92% (=158/172) of options are boolean
      - "Moving Forward with Combinatorial Interaction Testing", Yilmaz *et al.*, *IEEE Software*, 2014

Terminology

- **Configuration**
  - A selection of features
  - Features may not be all independent
- **Feature Model**
  - Description of a set of acceptable configurations
  - Important for understanding and for **testing**
  - Unfortunately, often not documented

Feature Model (FM)

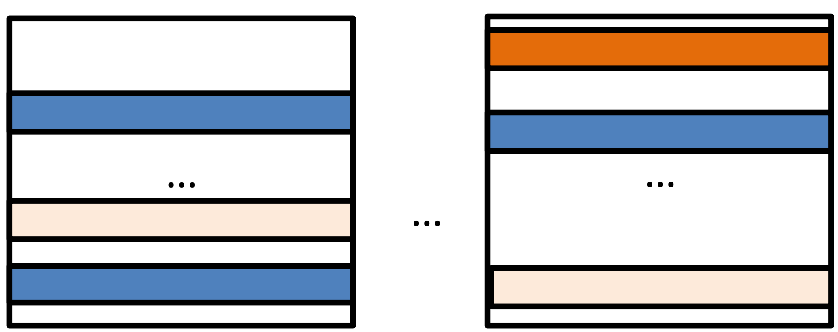


SMT-LIB encoding.
Follow example.

- Encode different forms of constraints
  - Mandatory (BASE) and optional (others)
  - Cross-feature
  - Alternative, etc.

Terminology

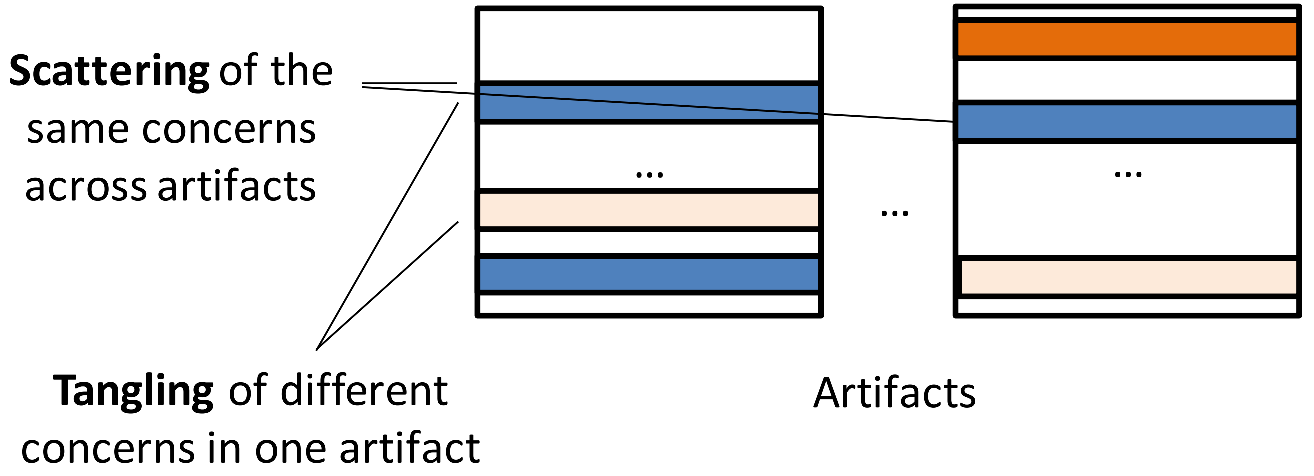
- **Variation**
  - Manifestations of features in artifacts
  - Scenario:
    - Feature is scattered across artifacts
    - Variations in artifacts collectively express feature



Artifacts

Terminology

- **Variation**
  - Manifestations of features in artifacts
  - Scenario:
    - Feature is scattered across artifacts
    - Variations in artifacts collectively express feature



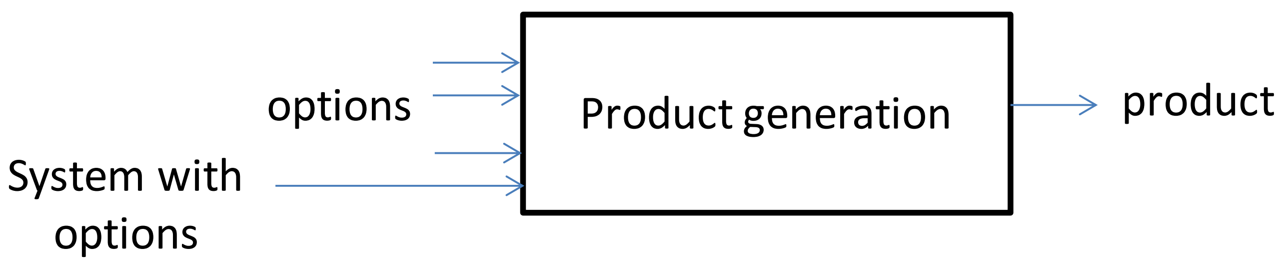
Terminology

- **Product**
  – Specialization of a configurable system for a particular configuration (set of features)

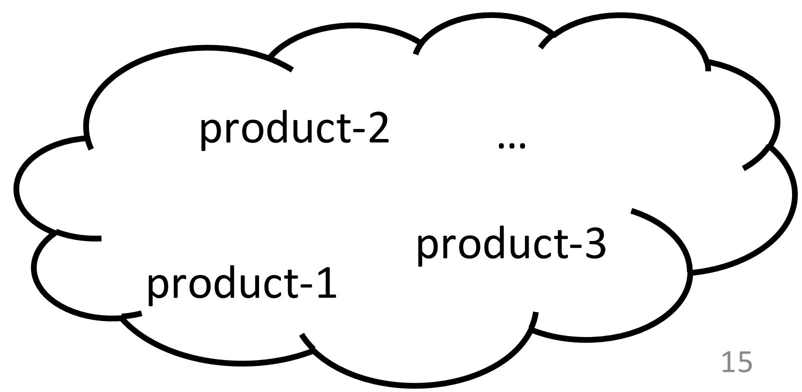
The term "configuration" is sometimes also used to denote the product that implements that configuration.

Terminology

- **Generation of a product**
  - Process of generating a product
    - Input: Selection of features, system
    - Output: Product that implements features



A configurable system is often called a family of systems.



Terminology

- Binding time of features
  - Static binding

    Often called Software Product Lines

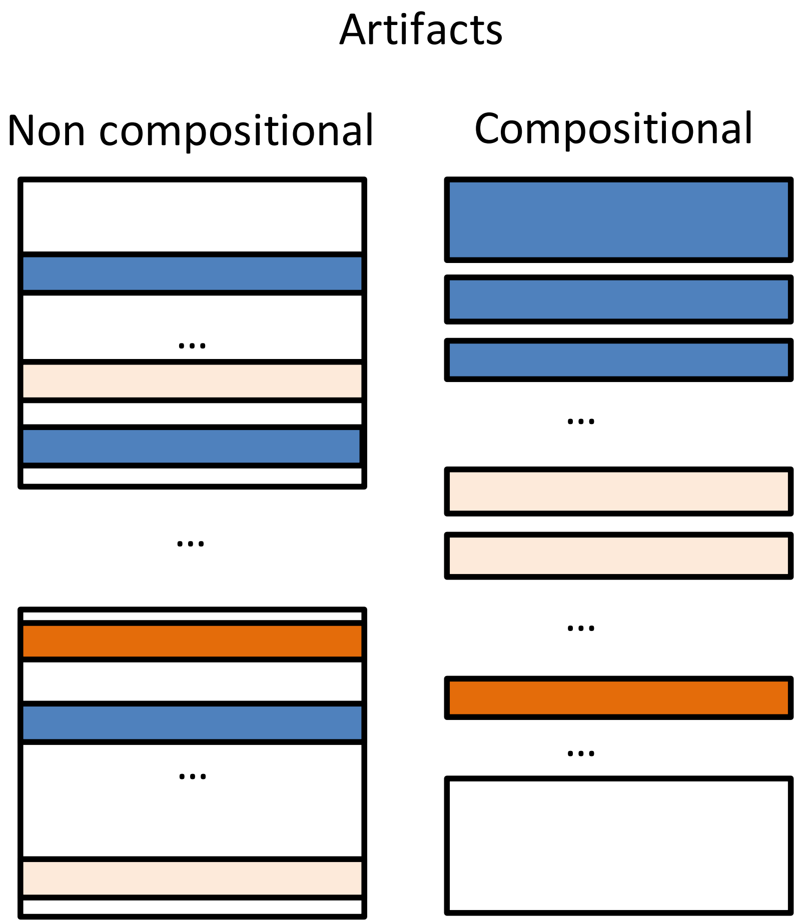
    - Annotative (e.g., #ifdefs)
      - Flexible but easy to introduce errors and hard to maintain
    - Compositional (e.g., AHEAD, AOP, etc.)
      - Easy to maintain but requires a new methodology for coding
  - Dynamic binding
    - Program state determines what features are enabled

Static (Annotative)

- Approach
  - Annotate program with preprocessor directives guarded by feature (boolean) expressions
    - E.g., `#ifdef FORMAT … #endif`
  - At build time, decide/bind value of each variable

See TankWar game example.

Static (Compositional)



- Partitions code w.r.t. features
  - Avoid scattering and tangling of concerns
- Several supporting languages. E.g., AHEAD, HyperJ, AspectJ, etc.

Dynamic

- Approach
  - Condition execution of code based on the evaluation of feature expressions

```
class Notepad {
  void toolBar() {
    if(T) {
      ...
      if(W)
        ...
    }
  }
  ...
}
```

Only executes this part if expression T evaluates to true.

T and W are program variables.

TESTING

What to test?

- Feature Testing
  - Analogous to Unit Testing
    - Example: Test the feature “Sound” in TankWar or the feature "Wordcount" in Notepad
- System Testing
  - As usual, but features are treated as inputs

What configurations to test? (1/3)

- Default configuration
  - Run test on one special (default) configuration
    - For example, consider default a configuration with the most popular set of features
- Random
  - Run test on a selection of random configurations

What configurations to test? (2/3)

- Exhaustive
  - Run test on all configurations
    - Potentially very expensive
    - Optimizations to address combinatorial explosion
      - Use feature model
      - Only consider reachable configurations from tests
    - SPLat (later discussed) builds on these optimizations

“SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems”, Kim *et al.*, *ESEC/FSE’13.*

What configurations to test? (3/3)

- Combinatorial Interaction Testing (CIT)
  - Run test on a selection of configurations
  - Generate covering arrays (e.g., 2-way covering arrays) that satisfy FM constraints

Example* of a traditional 2-way covering arrays (no constraints added)

A	B	C	D	E
0	1	1	2	0
0	0	0	0	0
0	0	0	1	1
1	1	1	0	1
0	1	0	0	2
1	0	1	1	0
1	1	1	1	2
1	0	0	2	1
1	0	0	2	2

A, B, and C are binary feature variables while D and E are ternary. Overall, there are 72 (=2^3*3^2) combinations.

*From "Moving Forward with Combinatorial Interaction Testing", Yilmaz *et al.*, *IEEE Software*, 2014.

TEST ADEQUACY

Coverage

- Not well studied in this context
- Problem: Lack of mapping from features to code
  - See non-compositional impl. mechanisms
  - If mapping is available, it is possible to compute feature coverage
    - Related to the TAROT’14 talk of Breno Miranda on “Relative Coverage”

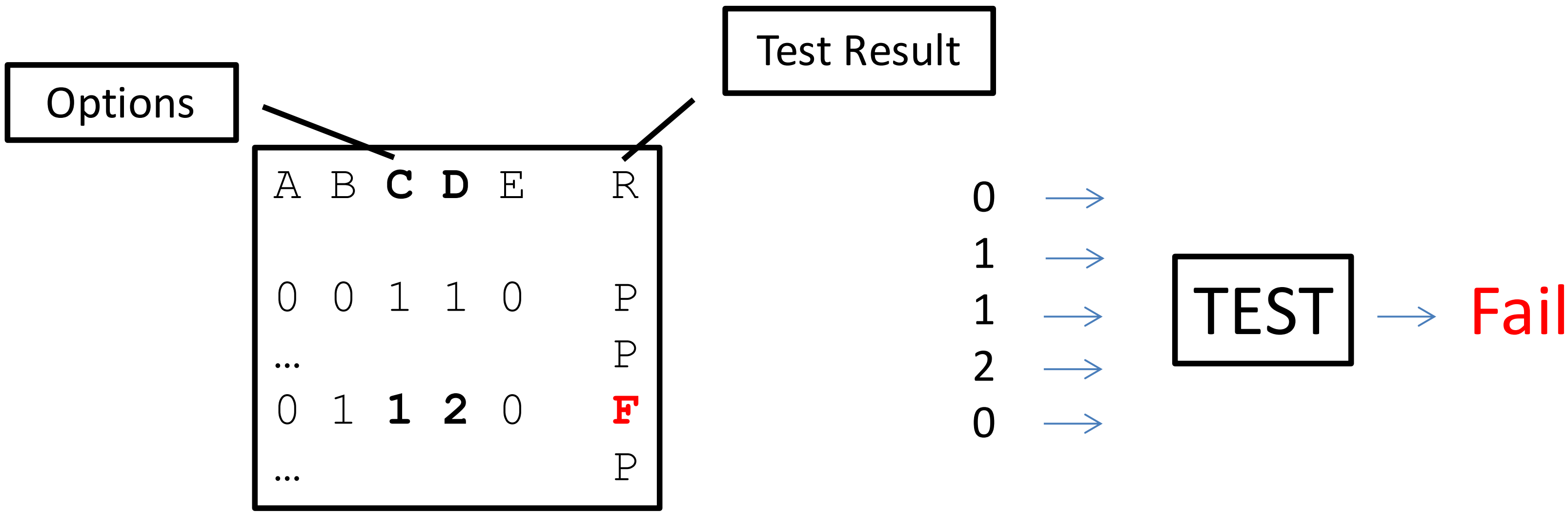
Mutation analysis

- Not very well studied too
- What mutants to apply?
  - "Feature Interaction Faults Revisited: An Exploratory Study", Garvin and Cohen, ISSRE'11.
    - E.g., modify feature expressions in #ifdef conditionals
- Problem: Even more expensive than mutation analysis on non-configurable systems
  - Tests x Configurations x Mutants

INTERPRETING TEST RESULTS

Feature Interaction

- Scenario: Used 2-way covering arrays and found exactly 1 failure



- Observation: Pair (C=1, D=2) is distinctly covered
- Hypothesis: Features C and D interact

Masking Effect

- Scenario: Found multiple failing executions
- Conjecture: Failures are due to the combinations of distinct features

Distinct pairs covered

A	B	**C**	**D**	E	R
…					P
0	**1**	1	2	0	**F**
1	**1**	0	1	0	**F**
1	1	0	2	**1**	**F**
…					P

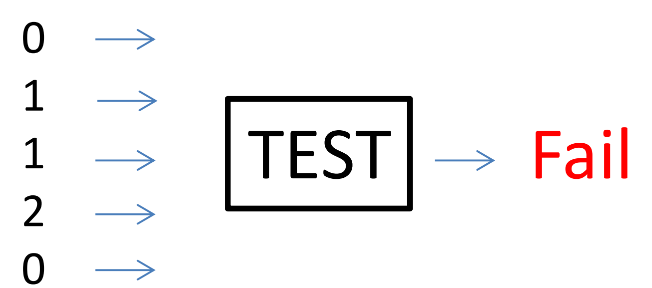
It can happen that this test will fail simply because B=1

DEBUGGING CONFIGURATIONS

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

A	B	C	D	E	R
0	1	1	2	0	F



Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
0 0 1 2 0   F
```

Pick one variable, alternate its value, observe results.

Refer to the discussion on the "Alternating Variable Method" from Gordon Fraser's talk.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

A	**B**	C	D	E	R
1	0	1	2	0	F

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
1 0 0 2 0   P
```

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

A	B	C	D	E	R
1	0	1	2	0	F

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
1 0 1 1 0   F
```

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
1 0 1 0 0   P
```

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
1 0 1 2 0   F
```

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

```
A B C D E   R
1 0 1 2 1   P
```

Pick one variable, alternate its value, observe results.

Debugging Configurations

- Scenario
  - Test fails on a particular configuration (see below), which options are relevant and which are not?

A	B	C	D	E	R
?	?	1	2	0	F

Options C, D, and E are relevant to induce failure.

Pick one variable, alternate its value, observe results.

Further Reading

- Delta Debugging (DD). Zeller *et al.*
  - https://www.st.cs.uni-saarland.de/dd/
- "Locating errors using ELAs, covering arrays, and adaptive testing algorithms", Martinez *et al.*, *SIAM Journal of Discrete Mathematics*, 23(4):1776–1799, 2009.
- "Spectrum-based Fault Localization in Embedded Software", Rui Abreu, PhD thesis, Delft University, November 2009.

GCC

GNU Compiler Collection (GCC)

- Supports several front-ends and back-ends
- Both static (annotative) and dynamic bindings
- Uses DejaGnu for Testing
  - DejaGnu is the testing framework of GNU
    - Git access:
      - git clone git://git.sv.gnu.org/dejagnu.git

DejaGnu

- Important features
  - Supports testing of interactive systems
    - Think of testing a shell command like “`ls`”
  - Language independent
    - Black-box interaction
    - Assertions defined with string matching
- Written in Expect, which is written in Tcl
  - Expect acts as a programmable shell
  - See http://www.nist.gov/el/msid/expect.cfm

DejaGnu

See Calc example

This example has no code variations. The purpose is to illustrate DejaGnu at use.

GCC DejaGnu test

This test will only compile on GCC using the C compiler front-end.

ext-4.c

```
/* Test for scanf formats.  %a extensions. */
/* Origin: Joseph Myers <jsm28@cam.ac.uk> */
/* { dg-do compile } */
/* { dg-options "-std=gnu89 -Wformat" } */

#include "format.h"

void foo (char **sp, wchar_t **lsp) {
  /* … */
  scanf ("%as", sp);
  scanf ("%aS", lsp);
  scanf ("%a[bcd]", sp);
}
```

Options passed to the compiler. Many other exist; default values used.

RESEARCH

Research Problems

- Testing
  - High Dimensionality
  - Lack of Feature Models
- Design & Implementation
  - Safe Composition
  - (Safe) Decomposition

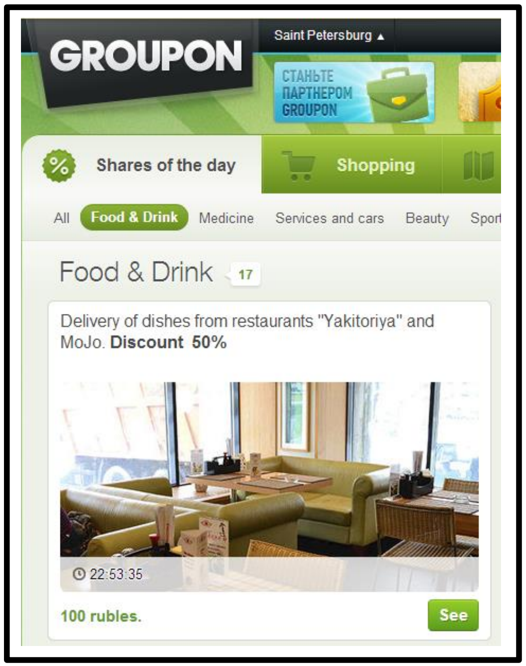
Work led by PhD student
Sabrina Souto
(sfs@cin.ufpe.br)

High Dimensionality

Our Solution
-- SPLat --

Kim *et al.*, **SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems.** ESEC/FSE'13: 257-267

High Dimensionality



www.groupon.com

170+ boolean variables

2^{170+} configurations

The same test needs to be run against many configurations

- E.g. The same Ruby on Rails test for Groupon needs to be run against all configurations

Existing Techniques

- **Sampling** [Cohen *et al*. ISSTA'07, Perrouin *et al.,* ICST'10, Garvin and Cohen ISSRE'11, Song *et al.* ICSE'12, Shi *et al.* FASE'12]
  – Heuristically sample the configuration space
    - Fast! But can miss errors or produce redundant tests
- **Exhaustive** [d'Amorim *et al.* ISSTA'07, Rhein *et al.* JPF'11, Kim *et al.* AOSD'11, Kastner *et al.* FOSD'12, Kim et al. ISSRE'12, Apel *et al.* ICSE'13]
  – Static/dynamic analysis for pruning redundant configurations
    - Safe! But slow and often doesn't scale

Proposal: SPLat

- Observation
  - Each test exercises a small portion of code
- Assumption
  - Feature variables can be easily identified in code
- Proposal
  - Explore all combinations of features dynamically reachable from a test
  - Can be optimized to only consider configurations consistent with feature model

SPLat in a Nutshell

1. Determine reachable configurations ***during*** execution
2. Set feature value when feature is encountered
3. Keep a stack of encountered features
4. Repeat until explore all legal combinations of encountered features

SPLat on Notepad

- 1^{st} run

Configurations Executed

Stack

T	false

TWM= <false, ?, true>
(M=true due to T∨M)

- 2^{nd} run

W	false
T	true

TWM=<true, false, ?>

- 3^{rd} run

W	true
T	true

TWM=<true, true, ?>

- 4^{th} run

W	true
T	true

Nothing to execute

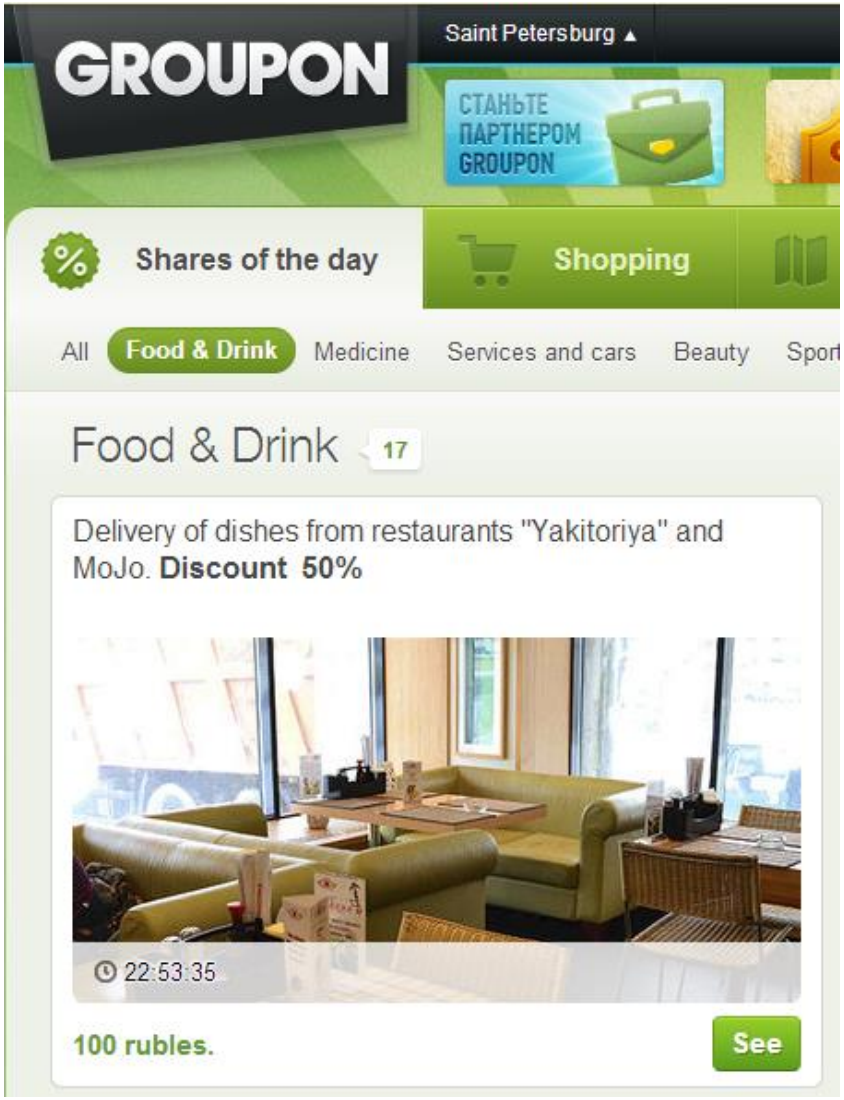
```
class Notepad {
  void toolBar() {
    if(T) {
      ...
      if(W)
        ...
    }
  }

  ...

  void test() {
    toolBar();
  }
}
```

Constraint: T ∨ M

Evaluation

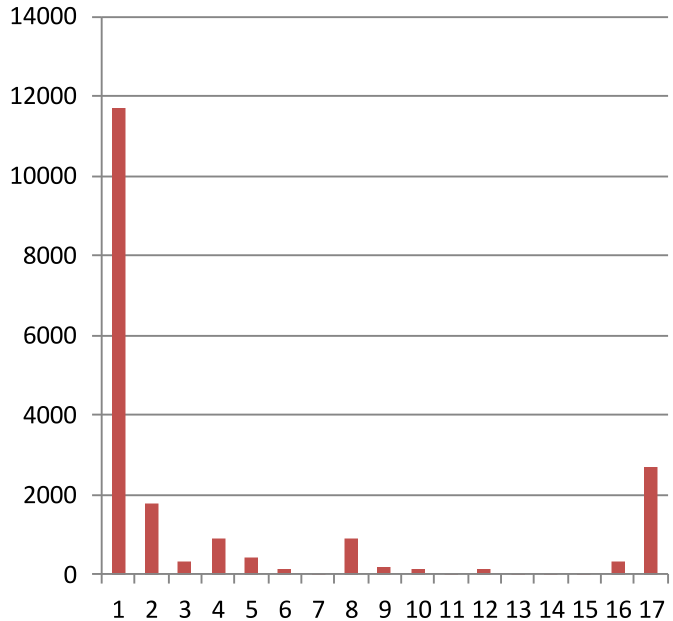
- Run SPLat on 10 SPLs
- Baselines
  – Exhaustive (worst case)
  – Static Reachability
  – Ideal (best case)
- SPLat was better for almost all cases
  – Overhead was high for short-running executions

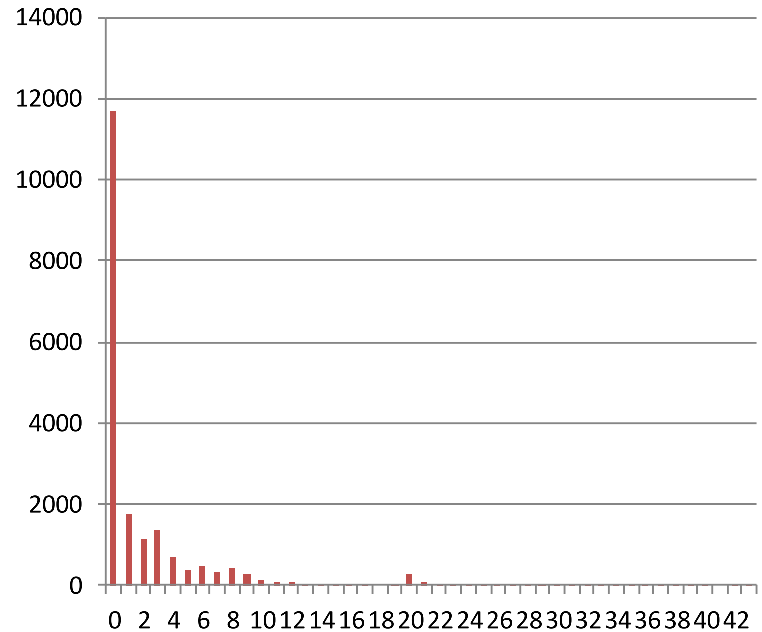
Groupon Evaluation: Setup



- How well does SPLat scale?
- Experiment
  - Ruby on Rails implementation of SPLat
  - Applied against the Groupon code base
    - 4.5 years of work from 250+ engineers
    - 400K+ LOC (171K LOC of server side, 231K lines of tests)
    - 19K tests
    - 170 boolean feature variables (up to 2^{170})

Groupon Evaluation: Results





Summary of SPLat

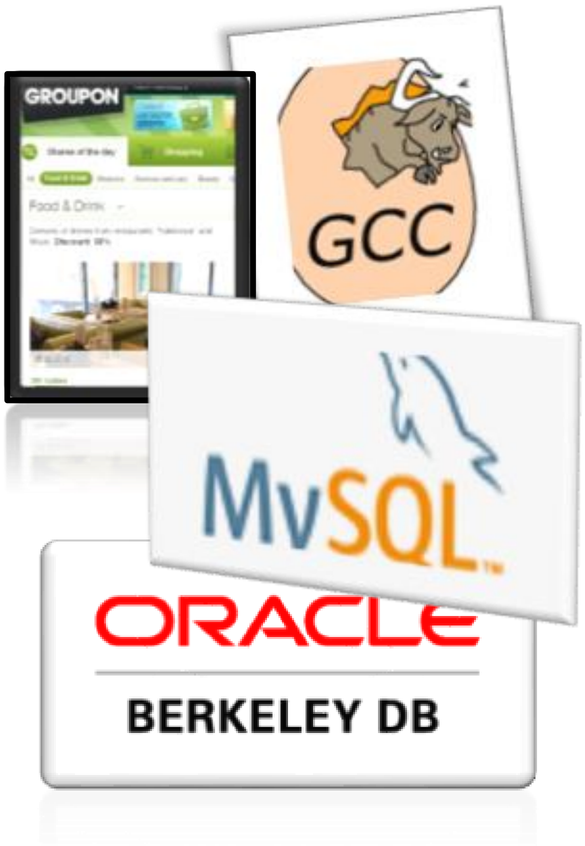
- Hypothesis: most tests exercise a relatively small number of configurations
  - Confirmed with Groupon case study
- It misses no configurations
- Low overhead compared to running selected configurations with no instrumentation
- Limitations
  - SPLat is not able to find equivalent states during executions (merging)

Lack of Feature Models

Our Solution

-- SPLif --

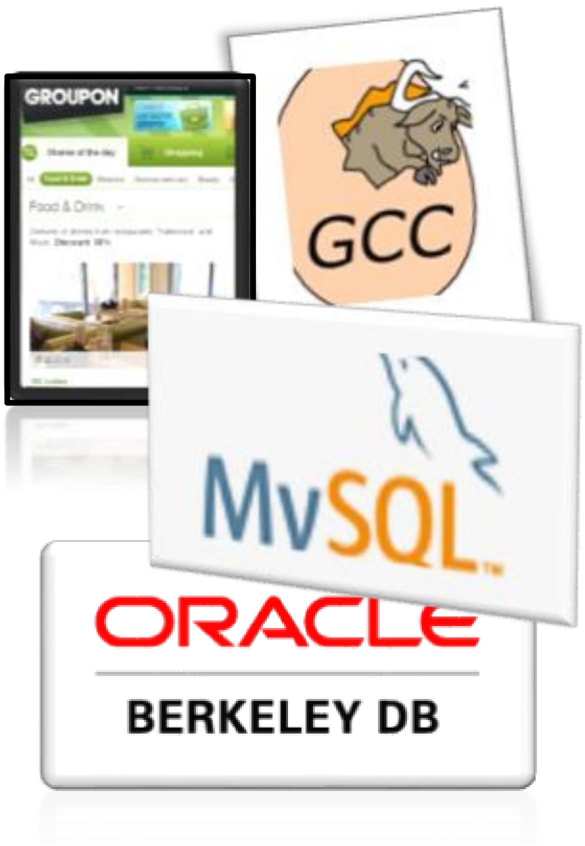
Lack of Feature Models



- Feature Models are important but often are not documented

Why important? A test failure due to a configuration that is not in the (missing) model is meaningless.

Lack of Feature Models



- Feature Models are important but often are not documented

Why not documented?

Existing Reverse Engineering Techniques

- Static Analysis [She *et al.* ICSE'11]
- Information Retrieval [Alves *et al.* SPLC'08, Davril *et al.,* FSE'13]
- Evolutionary Search [Lopez-Herrejon *et al.* SSBSE'13]
- Custom solutions [Haslinger *et al.* FASE'13]

No prior work builds on tests and their executions

Basic Terminology

- Partial vs. Complete Configuration

MTW=0*1 (partial)
MTW=010 (complete)

Recall Notepad Features:
Menubar, **T**oolbar, and **W**ordcount

- Consistent vs. Inconsistent Configuration

MTW=0*1 (consistent)
MTW=00* (inconsistent)

Recall Notepad Constraint:
$M \vee T$ (Undocumented)

Proposal: SPLif

- Revise the feature model during Testing
  - Ask the user to label configurations
    - If configuration is consistent, inspect!
- Assumptions
  - User is aware about many feature relationships
  - User makes no mistake :-(

SPLif Example (1 test)

- Configurations (MTW):

  111
  011
  110
  010
  10*
  00*

```
class Notepad {
  void toolBar() {
    if(T) {
      ...
      if(W)
        ...
    }

    if (M) { ... }
  }

  ...

  void test() {
    toolBar();
  }
}
```

SPLif Example (1 test)

- Configurations (MTW):

111

011 ✖

110

010

10* ✖

00* ✖

Execution of some tests fails!

SPLif Example (1 test)

- Configurations (MTW):

Select failing configurations

011 ✖

10* ✖

00* ✖

SPLif Example (1 test)

- Configurations (MTW):

00*
10*
011

Rank configurations for inspection

SPLif Example (1 test)

- Configurations (MTW):

00*

Inconsistent!

10*

011

SPLif Example (1 test)

- Configurations (MTW):

00*

Inconsistent!

10*

011

Partial Feature Model (PFM) = !(⋃ c_i),
where c_i is an inconsistent configuration

In this case c_1=(`!M ∧ !T`) and PFM=

`!(!M ∧ !T)`

`!!M ∨ !!T`

`M ∨ T`

SPLif Example (1 test)

- Configurations (MTW):

00*

Inconsistent!

10*

011

Partial Feature Model (PFM) = !(U c_i),
where c_i is an inconsistent configuration

Configurations that violate this constraint will not be inspected!

In th

! (!M ∧

! !M ∨

M ∨ T

SPLif Example (1 test)

- Configurations (MTW):

00*

10* Consistent

011

Partial Feature Model:

M ∨ **T**

The test failed on a configuration where no inconsistency has been observed. Tester should inspect!

SPLif Example (1 test)

- Configurations (MTW):

00*

10* Consistent

011

Partial Feature Model:

$M \vee T$

Feature model obtained is complete in this case. But that is not always the case.

Evaluation Setup

- Asked students to generate tests for 5 SPLs
  - 212 tests in total
    - Of these 85 tests fail for some configuration (~40%)
  - 7378 configurations in total
    - Of these 1220 fail (~16%)
    - Of these 154 are consistent (~12%)
- SPLif ranks tests likely to contain consistent configurations and configurations on each test

test1 →	configuration-1
test2	configuration-2
…	…

Evaluation Setup

- We inspected tests and failing configurations
- Configuration inspection
  - Consistent configuration found => Bug in test or code
  - Inconsistent configuration found => Update in model

Evaluation Results

- # of configuration inspections smaller than # failing configurations
  - SPLif uses set of concrete configurations (due to ?)
- No bug in code found
- Few test repairs needed
  - Most cases only one change needed in test

Design & Implementation
Safe Composition

Safe Composition

- Problem
  - Are there inconsistencies in code?
  - This is a well studied problem
    - “Safe composition of product lines”. Thaker *et al.,* GPCE’07
    - “Safe composition of knowledge-based software product lines”, Teixeira *et al.,* JSS’13
    - …

One Approach

- Assume Feature Model (FM) is available
- Infer feature constraints from code and check those against FM using a constraint solver

"Safe composition of product lines". Thaker *et al.,* GPCE'07

Example

```
class Notepad {
  void toolBar() {
    if(T) {
      x
      if(W)
        y
    }
  }

  ...

  void test() {
    toolBar();
  }
}
```

FOF: Member --> Feature Expression

Consider uninterpreted function FOF as the mapping from members to features

Example

```
class Notepad {
  void toolBar() {
    if(T) {
      x
      if(W)
        y
    }
  }

  ...

  void test() {
    toolBar();
  }
}
```

Feature constraints extracted from code:

```
T => FOF(x)
(T AND W) => FOF(y)
```

Use a constraint solver to find contradictions between these constraints and those expressed in the FM.

Design & Implementation (Safe) Decomposition

Problem

- How to decompose features into modules?

alternatively,

- What is the binding of features to members?
  - Existing solutions are imprecise
    - E.g., information retrieval

Example

- What are the possible valuations for…
  - FOF(x), FOF(y), and FOF(toolBar)?

```
class Notepad {
  void toolBar() {
    if(T) {
      x
      if(W)
        y
    }
  }
}
```

Thanks to…

- Paulo Barros (UFPE)
- Don Batory (UT Austin)
- Divya Gopinath (UT Austin)
- Sarfraz Khurshid (UT Austin)
- Peter Kim (now Oxford then UT Austin)
- Darko Marinov (Illinois)
- Sabrina Souto (UFPE)