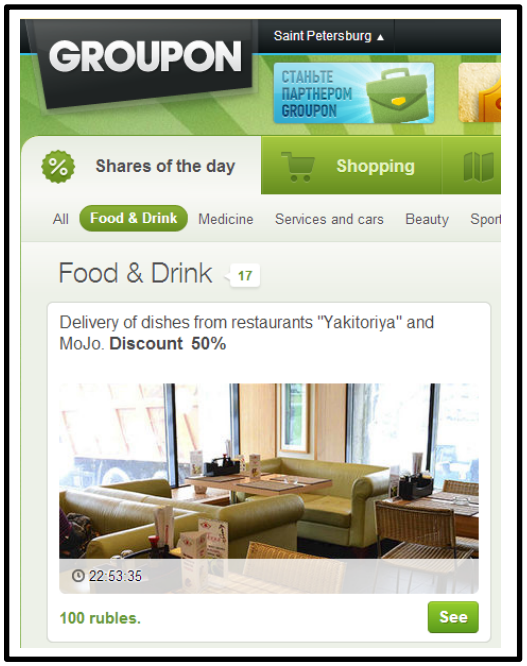
Lightweight Testing for Configurable Systems

Sabrina Souto
[sfs@cin.ufpe.br]
Informatics Center
Federal University of Pernambuco

Advised by:
Marcelo d'Amorim
[damorim@cin.ufpe.br]

Configurable System

- System behavior depends on **configuration** variables. Examples:

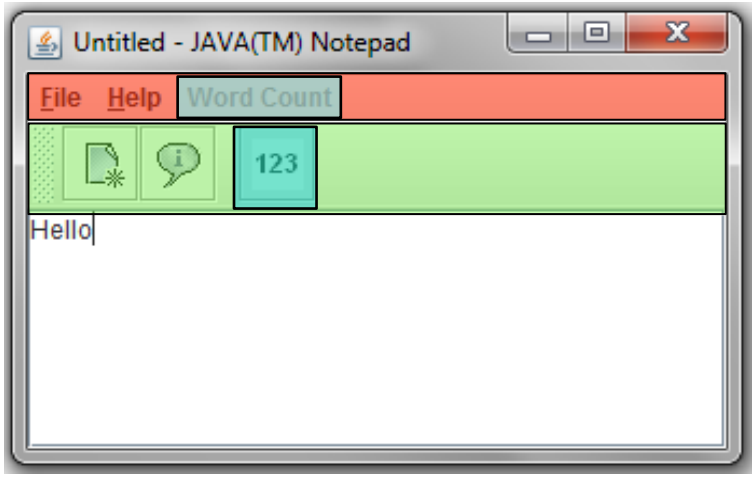


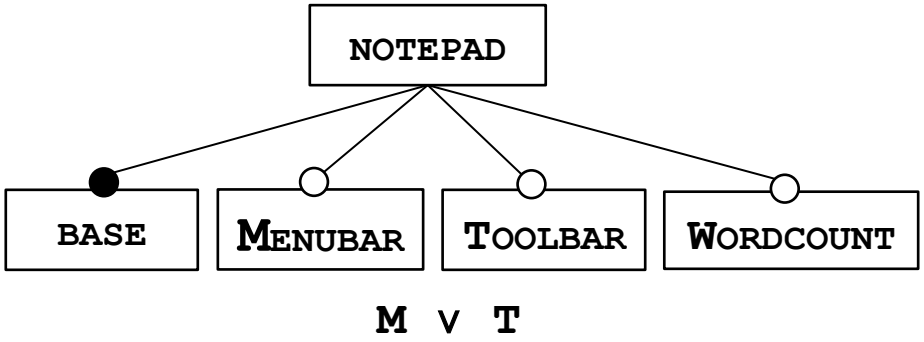
Basic Terminology

- **Feature**
  - Distinct functionality
- **Configuration**
  - A selection of features
- **Feature Model**
  - Description of a set of acceptable configurations of a system

  (Not always documented)

Illustrative Example: Notepad



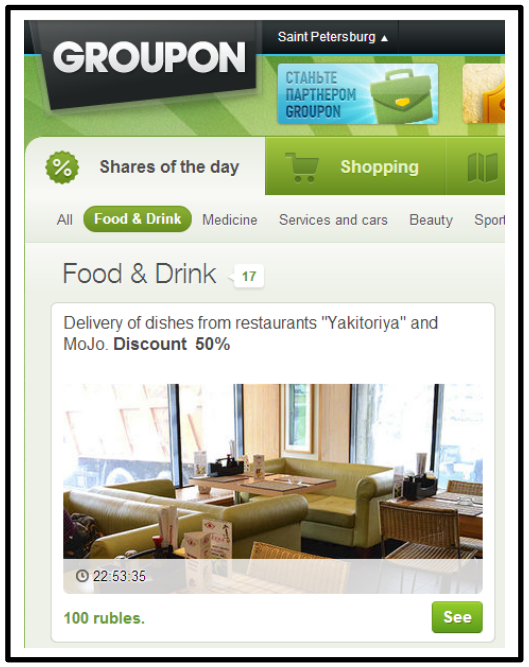



```
class Notepad {
  void toolBar() {
    if(T) {
      ...
      if(W)
        ...
    }
  }

  ...

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

Forbidden configurations: MTW=001, MTW=000

Testing Configurable Systems is Challenging!

Problem 1: High Dimensionality



www.groupon.com

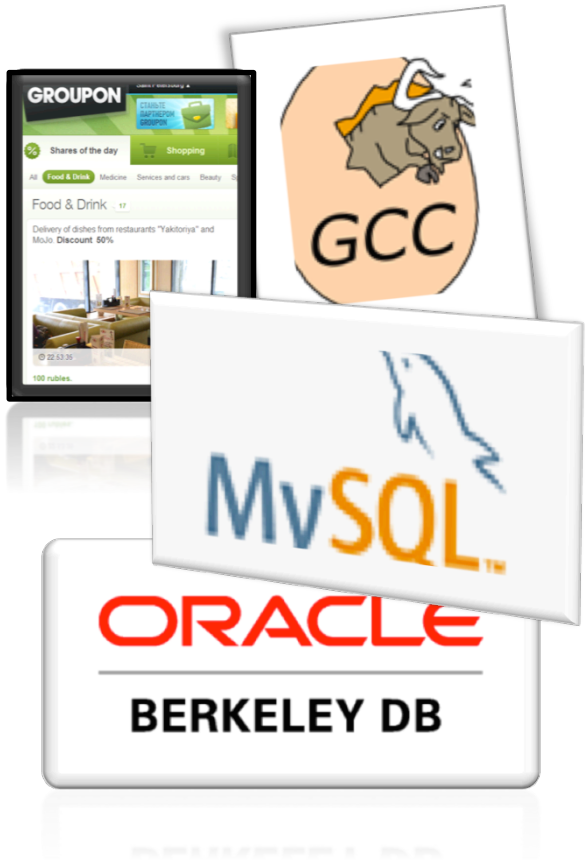
170+ boolean variables

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

Problem 2: Lack of Feature Models



- Feature Models are important!
- But often are not documented
  - One reason: Features emerge and submerge in short periods under highly-dynamic environments

Problem 1
High Dimensionality

Our Solution
-- SPLat --

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
- **Proposal**
  - Only consider…
    - Features dynamically reachable from a test
    - Configurations consistent with feature model
- **Assumptions**
  - Feature model exists
  - Test can be run on multiple configurations

Insight

- Only reachable features
  - E.g. If **T** is false, combinations of **W** and **M** yield identical program traces
- SPLat produces:
  - T=false, W=?, M=?
  - T=true, W=false, M=?
  - T=true, W=true, M=?
- Only consistent configurations are explored
  - When **T** is false, **M** must be true

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

  ...

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

Constraint: T v M

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

- 1st run

Stack

T	false

Configurations Executed

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

- 2nd run

W	false
T	true

TWM=<true, false, ?>

- 3rd run

W	true
T	true

TWM=<true, true, ?>

- 4th run

W	true
T	true

Nothing to execute

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

  ...

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

Constraint: T ∨ M

Why is SPLat Lightweight?

- Inexpensive instrumentation
  - Only feature variables need instrumentation
- Uses efficient SAT solver for checking path feasibility
  - We used SAT4J

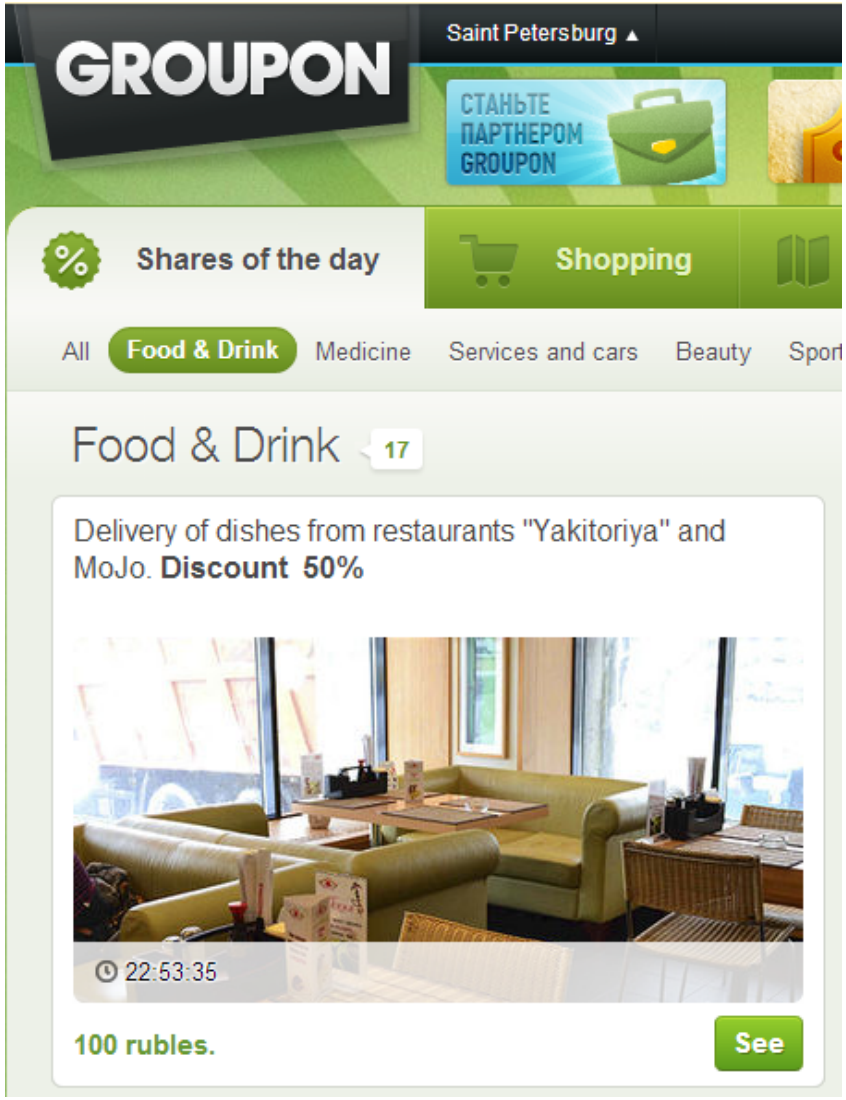
Java Evaluation: Setup

- **Questions**
  - How does SPLat compares against?
    - Conventional execution: running every configuration
    - Static analysis [Kim et al., AOSD'11]
  - What is the overhead of SPLat?
- **Experiment**
  - 10 SPLs previously used
    - 5–25 features, 20–192 configurations, 580–14,480 LOC
  - Tests for exercising low, medium, and high number of reachable configurations

Java Evaluation: Results

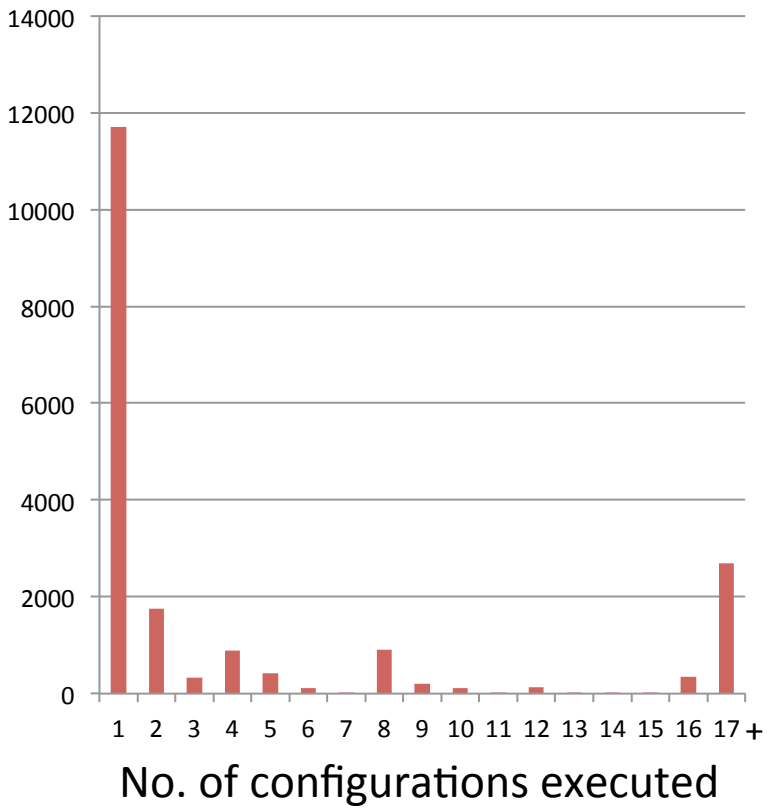
- SPLat is faster than conventional execution 83% of the time
- SPLat is faster than static analysis all the time and up to 2 orders of magnitude faster
- Overhead
  - IdealTime: Time of ideal execution
  - SPLatTime: Time of SPLat execution
  - Overhead = SPLatTime - IdealTime
    - <50% overhead in 73% of tests, small for long-running tests

Groupon Evaluation: Setup

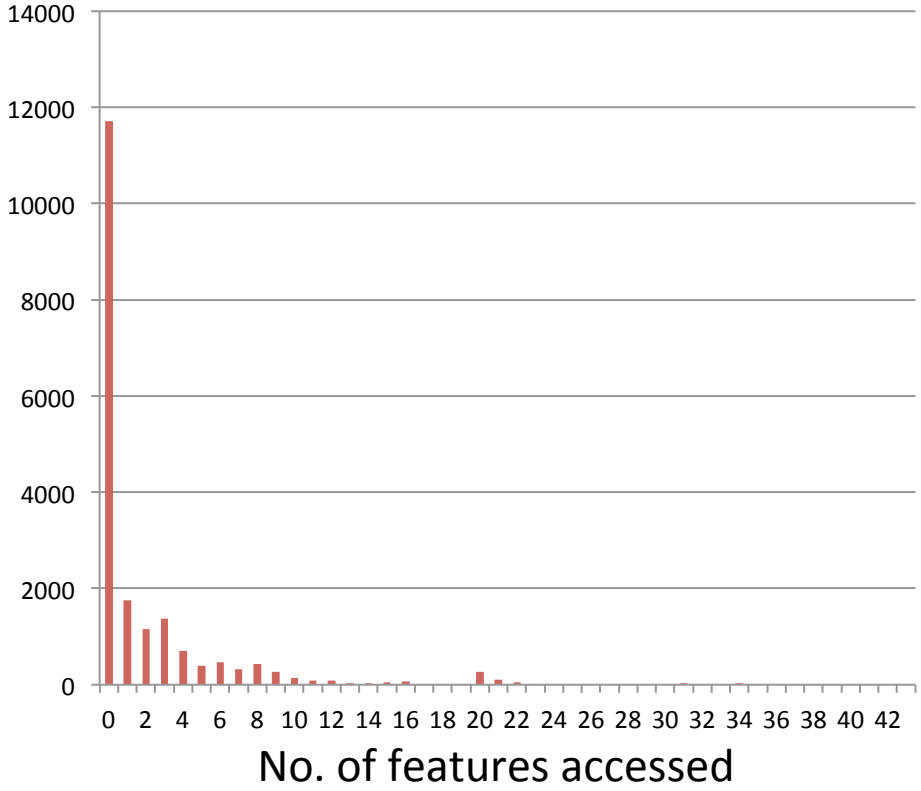


- How well does SPLat scale?
- Experiment
  - Ruby on Rails implementation of SPLat
  - Applied against the Groupon code base
    - 4.5 years of work from 250+ engineers
    - 400K+ LOC (171K LOC of server side, 231K lines of tests)
    - 19K tests
    - 170 boolean feature variables (up to 2^{170})

Groupon Evaluation: Results



No. of tests



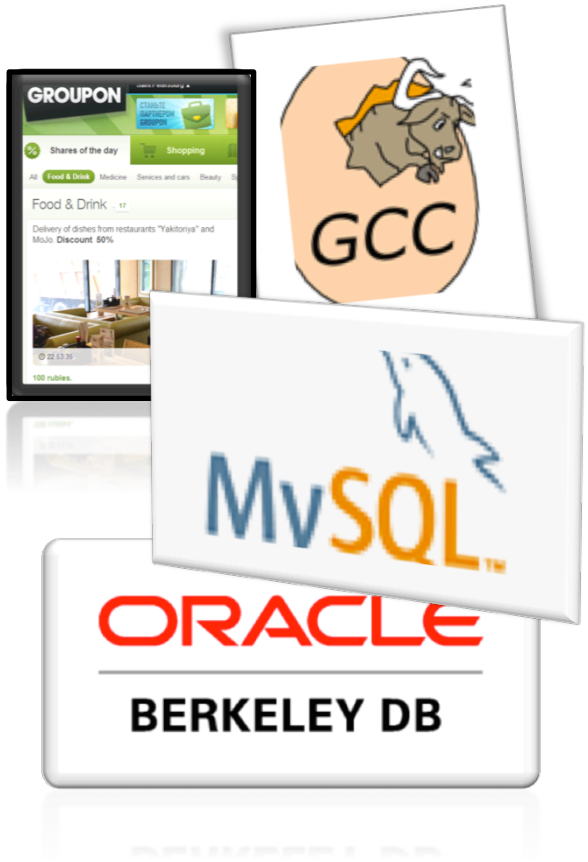
- Most tests exercise small number of features (<170) and configurations (<2^{170})

Problem 2
Lack of Feature Models

Our Solution
-- SPLif --

Paper under submission

Problem Reminder



- Feature Models are **important** but often not documented.

> ⚠ **False positives**!
> A test can fail due to a configuration that is not in the (missing) model is meaningless.

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

- Configuration
  - A selection of features

- Each feature can assume 3 values:
  - 0: the feature is disabled (=false)
  - 1: the feature is enabled (=true)
  - ?: the feature has no value yet (=unknown)

Basic Terminology

- **Partial** vs. **Complete** Configuration

MTW=0?1 (partial or incomplete)
MTW=010 (complete)

Recall Notepad Features:
Menubar, **T**oolbar, and
Wordcount

- **Consistent** vs. **Inconsistent** Configuration

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

Recall Notepad Constraint:
M v **T** (Undocumented)

Insight

- Run each test against many products, by using a modified version of SPLat
- Use the profile of passing and failing runs to help developers prioritize their inspection of failures in order to distinguish
  - Failures in products, due to invalid combinations of features
  - Failures in the code, e.g. a bug.

Proposal: SPLif

- Revise the feature model during Testing
  - Ask the user to label configurations
    - If configuration is consistent, inspect the test!
- 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):

011 ✖

Select failing configurations

10? ✖

00? ✖

SPLif Example (1 test)

- Configurations (MTW):

00**?**
10**?**
011

Rank configurations for inspection

SPLif Example (1 test)

- Configurations (MTW):

00?

10?

011

Inconsistent!

SPLif Example (1 test)

- Configurations (MTW):

00?

10?

011

Inconsistent!

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

In this case c_i=(`!M ∧ !T`) and PFM=

```
!(!M ∧ !T)
!!M ∨ !!T
M ∨ T
```

SPLif Example (1 test)

- Configurations (MTW):

00?

10?

011

Inconsistent!

Partial Feature Model (PFM) = $!(\bigcup c_i)$, where c_i is an inconsistent configuration

In th

!(!M ∧ T

!!M ∨ .T

M ∨ T

Configurations that violate this constraint will not be inspected!

SPLif Example (1 test)

- Configurations (MTW):

00?

10?

011

Consistent

Partial Feature Model:

M v T

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

SPLif Example (1 test)

- Configurations (MTW):

00?

10?

011 Consistent

Partial Feature Model:

M v T

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

SPL[at,if] collaborators

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

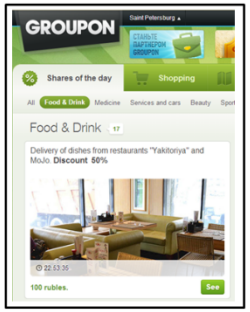
Lightweight Testing for Configurable Systems

Marcelo d'Amorim
Federal University of Pernambuco

THE UNIVERSITY OF TEXAS AT AUSTIN

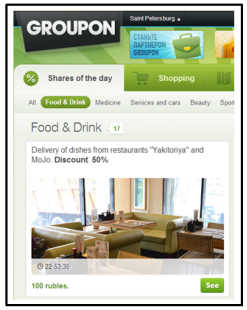
Configurable System

- System behavior depends on **configuration** variables. Examples:



ORACLE
BERKELEY DB

Problem 1: High Dimensionality



www.groupon.com

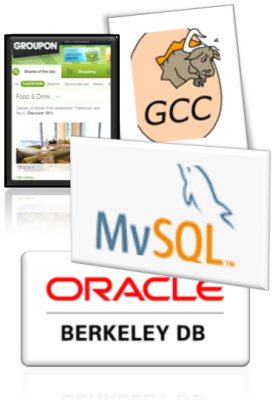
170+ boolean variables
2^{170+} configurations

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

--SPLat--

Problem 2: Lack of Feature Models



- Feature Models are important!
- But often are not documented
  - One reason: Features emerge and submerge in short periods under highly-dynamic environments

--SPLif--