Generating Tests for Detecting Faults in Feature Models

Paolo Arcaini¹, Angelo Gargantini², Paolo Vavassori²

¹ Charles University in Prague, Czech Republic
² University of Bergamo, Italy
Outline

• Feature models (FM)
• Fault-based testing approach for FMs
  • Common faults and mutation operators for FMs
  • Distinguishing configuration (dc)
• Generation of \textit{dcs}
  • Test suite that guarantees the detection of faults
• Experiments
  • Usage of \textit{dcs}
• Conclusions
Feature models

- A feature model is a compact representation of all the products of the Software Product Line (SPL) in terms of "features" and relations among them.
Configurations and products

- A configuration of a feature model $M$ is a subset of the features in $M$ that must include the root.

- A valid configuration is called a product, since it represents a possible valid instance of the feature model.

Product: 
\{Mobile Phone, Calls, Screen, Basic, Media, MP3\}

Invalid configuration: 
\{GPS, Screen, Media\}

Configurations used as tests: exhaustive coverage is unpractical. Which configurations to select?
FM as Boolean formula

- Feature models semantics can be rather simply expressed by using propositional logic

Let $BOF$ be a function that, given a feature model $M$, returns its representation as propositional formulae

$BOF(M)$:

root mandatory optional

\[ a \land a \leftrightarrow b \land c \rightarrow a \]
Fault-based testing for Feature Models
Fault-based testing approach

- **Goal**: demonstrate that prescribed faults are not in the artifact under test

- **Assumption**: the model can only be incorrect in a limited fashion specified by a relatively small set of common mistakes

1. Fault model
   - Fault classes and mutation operators

2. A **definition** of the conditions able to expose the faults

3. A way to **generate** test that satisfy those conditions
1. Fault model - Mutation operators

- We focus on faults that can be detected by a configuration
  - may change the set of products (valid configurations)

**AlToOr**: Alternative to Or

**ManToOpt**: mandatory to optional

**AlToAnd**: Alternative to And

**OptToMan**: optional to mandatory

**OrToAl**: Or to Alternative

**MF**: a feature f is removed

**OrToAnd**: Or to And

**MC**: a constraint is removed

**AndToOr**: And to Or

**ReqToExcl**: requires to excludes

**AndToAl**: And to Alternative

**ExclToReq**: excludes to requires

**OrToAnd**: Or to And

**OrToAnd**: Or to And

**AndToOr**: And to Or

**AndToAl**: And to Alternative

**Mutant**:

- **ManToOpt**: a feature f is removed
- **MC**: a constraint is removed
- **ReqToExcl**: requires to excludes
- **ExclToReq**: excludes to requires

Angelo Gargantini - Generating Tests for Detecting Faults in Feature Models
2. Distinguishing configuration

- Given a feature model $M$ and one of its mutants $M'$

A configuration $c$ distinguishes $M$ from $M'$ IFF $c$ is valid in $M$ and not in $M'$ or vice versa

Example

Configuration $\{a, b\}$ is valid in both
It is not distinguishing

Configuration $\{a, c\}$ is valid in $M'$ but not in $M$
It is distinguishing
Some notes

- a distinguishing configuration
  - does not need to be a valid product

- may not exist:
  - **Equivalent mutants**

\[ \equiv \]

Equivalent mutant, with the same product set

\[ \equiv \]

Equivalence

\[ \equiv \]

Constraint removal

\[ \equiv \]

Constraint

\[ \equiv \]

Product set
3. Generation of distinguishing configurations

• Boolean difference or derivative:
  • the erroneous implementation $\varphi'$ of a Boolean expression $\varphi$ can be discovered only when the expression $\varphi \oplus \varphi'$ is evaluated to true
  • where $\oplus$ denotes the logical exclusive or (xor) operator

• The generation of a distinguishing configuration can be performed by:
  1. translating $M$ and its mutant $M'$ by BOF
  2. using a SAT/SMT solver to get a model of the Boolean derivative
Given a FM $M$

1. Apply the mutation operator
2. By BOF, get the Boolean formulas for $M$ and $M'$
3. Build the Boolean difference
4. Call a SAT solver

$$\varphi \oplus \varphi'$$

model = **distinguishing configuration** = test

$M \equiv M'$ eq mutant
Example

\( M \)

\[ a \land a \leftrightarrow b \land c \rightarrow a \]

\( \oplus \)

\[ a \land a \leftrightarrow b \land c \rightarrow a \]

\( a \land a \rightarrow a \land c \rightarrow a \)

\( \oplus a \land b \rightarrow a \land c \rightarrow a \)

\( M' \)

\[ a \land b \rightarrow a \land c \rightarrow a \]

\{a = true, b = false, c = true\}

BOF

Valid in \( M \) but not in \( M' \)

SAT/SMT SOLVER
Generation of a test suite

• To get a set of configurations able to detect all the faults for a given feature model:

1. generate all the mutants for the original feature model by applying one mutation at a time
2. generate a test predicate for every mutant
3. use the solver to get the model for every test predicate (or to prove that the mutant is equivalent)

• This basic process can be optimized (for producing more compact test suites and possibly save time)
Optimizations

• Process
  • **monitoring**: it checks whether a test produced for a test predicate is also a model for other test predicates
  • **collecting**: it looks for a model of a conjunction of test predicates, instead of a model for each test predicate
    • Very expensive in terms of SAT calls
  • **prioritizing**: it considers the test predicates in a particular order
  • **post reduction**: after the test generation, it removes unnecessary tests, i.e., tests that only cover test predicates that are also covered by other tests

• Logical
  • **xor simplification**: \((\alpha \land \beta) \oplus (\alpha \land \gamma) \equiv \alpha \land (\beta \oplus \gamma)\)
Experiments
Settings

- FeatureIDE
- SMT solver Yices
- 53 models from SPLOT repository
  - From 9 to 287 features (2.48 \times 10^{86} configurations)
  - From 2 to 8.53 \times 10^{47} products
- 1600 artificially generated models distributed with FeatureIDE
  - having 10, 20, 50, 100, 200, 500, 1000, and 2000 features

Angelo Gargantini - Generating Tests for Detecting Faults in Feature Models
Some preliminary research questions

- **RQ1** How many mutants are generated?
  - Not so many – linear increase with the number of feature

- **RQ2** How many mutants are generated by each mutation operator?
  - Half of the mutants are for missing features

- **RQ3** How are mutants distributed in the Feature Models edit classes \{Refactoring, Specialization, Generalization, Arbitrary edit\}?
  - Details in the paper

- **RQ4** Is the simplification of the xor expressions effective?
  - A lot
RQ5 How big are the fault-detecting test suites? How much time is required to generate them?

- Artificial models - for models up to 200 features:
  - Average time 38 seconds
  - Average size is 60 tests (instead of $2^{200}$ configurations)
RQ5 How big are the fault-detecting test suites? How much time is required to generate them?

- SPLOT models
  - From 9 to 287 features (2.48 X 10^{86} configurations)

<table>
<thead>
<tr>
<th></th>
<th>TS Size</th>
<th>Time (ms)</th>
<th>#Infeasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>4</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>Mean</td>
<td>19.9</td>
<td>5,264</td>
<td>0.49</td>
</tr>
<tr>
<td>Max</td>
<td>134</td>
<td>137,029</td>
<td>8</td>
</tr>
</tbody>
</table>

Much fewer infeasible tests than artificial models: fewer eq mutants means better quality?
Example of use for distinguishing configuration

```c
#ifdef HELLO
char* msg = "Hello!\n";
#endif

#ifdef BYE
char* msg = "Bye bye!\n";
#endif

main() {
    printf(msg);
}
```

Alternative configurations:
- `{HELLO}` is valid
- `{HELLO,BYE}` is invalid

Confirmed by the compiler (msg is declared twice)

- How to detect faults in the feature model for C code:
  - generate the distinguishing configurations and check that the oracle (compiler) confirms its validity value
RQ6 Are distinguishing configurations useful?

• **Experiment of typical use of distinguishing configurations:**

  • We have taken 6 small programs from the literature regarding the synthesis of feature models from preprocessor directives of C/C++ source code

  • We have instructed 6 students and asked them to build the 6 feature models

  • Each configuration represents an option set to be given to the compiler

  • We can use the *compiler as oracle* to judge if a configuration is valid (product) or not

  • We have generated the distinguishing configurations and checked with the compiler
# Results - comparison w.r.t. other techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>#tests</th>
<th>#failing tests</th>
<th>#faulty models (fm)</th>
<th>tests/fm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinguishing configurations</td>
<td>109</td>
<td>12</td>
<td>10</td>
<td>10.9</td>
</tr>
<tr>
<td>Pairwise</td>
<td>165</td>
<td>12</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>All Products</td>
<td>285</td>
<td>23</td>
<td>5</td>
<td>57</td>
</tr>
<tr>
<td>All configurations</td>
<td>516</td>
<td>119</td>
<td>17</td>
<td>30.4</td>
</tr>
</tbody>
</table>

- Our approach produces the minimum number of tests and it is able to discover 59% of the faulty models
  - much more effective than other techniques
  - but not all the faults were detected

- The coupling effect does not hold in our case
  - The component programmer hypothesis may not be true
  - New mutation operators or HOM
Conclusions

• a configuration is distinguishing (dc) if it can characterize a feature model w.r.t. a faulty version of it

• generation of dc by:
  • translating FMs to Boolean formulas,
  • making the derivative (xor) and
  • using SAT for test generation

• the technique is effective
  • but some faults may be undetected

Thank you