GENERATING MINIMAL FAULT DETECTING TEST SUITES FOR BOOLEAN EXPRESSIONS

ANGELO GARGANTINI
UNIVERSITY OF BERGAMO
ITALY

GORDON FRASER
UNIVERSITY OF SAARLANDES
GERMANY
1. Introduction about testing of (DNF) logic expressions
   • Boolean expressions: where to find them, how to test them
   • For boolean specification in DNF
     • Fault classes
     • Classical testing criteria

2. A new way of generating fault detecting tests
   • How to discover a fault
   • Using SAT solvers to generate tests
   • Optimizations

3. Experiments
LOGIC PREDICATES AND CLAUSES

A predicate is an expression that evaluates to a Boolean value.

Predicates can contain:

- boolean variables
- non-boolean variables that contain >, <, ==, >=, <=, !=
- boolean function calls

Internal structure is created by logical operators:

- ¬ the negation operator
- ∧ the and operator
- ∨ the or operator
- → the implication operator
- ⊕ the exclusive or operator
- ↔ the equivalence operator

A clause is a predicate with no logical operators.
EXAMPLES

(a < b) \lor f(z) \land D \land (m \geq n \times o)

Four clauses:

- (a < b) – relational expression
- f(z) – boolean-valued function
- D – boolean variable
- (m \geq n \times o) – relational expression
DISJUNCTIVE NORMAL FORM

Common Representation for Boolean expressions

- Slightly Different Notation for Operators
- Slightly Different Terminology

Basics:

- A literal is a clause or the negation (overstrike) of a clause
  - Examples: $a, \bar{a}$
- A term is a set of literals connected by logical “and”
  - “and” is denoted by adjacency instead of $\land$
  - Examples: $ab, \bar{a}b, \bar{a}b$ for $a \land b, a \land \neg b, \neg a \land \neg b$
- A predicate is a set of terms connected by “or”
  - “or” is denoted by $+$ instead of $\lor$
  - Examples: $abc + \bar{a}b + a\bar{c}$
  - Terms are also called “implicants”
    - If a term is true, that implies the predicate is true
FAULT CLASSES FOR BOOLEAN EXPRESSIONS

There exist typical errors done by programmers. Errors cause faults in the expression:

- Faults grouped in fault classes
- For DNF expressions there classical fault classes
DNF FAULT CLASSES

ENF: Expression Negation Fault
\[ f = ab + c \]
\[ f' = \overline{ab} + c \]

TNF: Term Negation Fault
\[ f = ab + c \]
\[ f' = \overline{ab} + c \]

TOF: Term Omission Fault
\[ f = ab + c \]
\[ f' = ab \]

LNF: Literal Negation Fault
\[ f = ab + c \]
\[ f' = \overline{a}b + c \]

LRF: Literal Reference Fault
\[ f = ab + bc \]
\[ f' = ad + bc \]

LOF: Literal Omission Fault
\[ f = ab + c \]
\[ f' = a + c \]

LIF: Literal Insertion Fault
\[ f = ab + c \]
\[ f' = ab + bc \]

ORF+: Operator Reference Fault
\[ f = ab + c \]
\[ f' = abc \]

ORF*: Operator Reference Fault
\[ f = ab + c \]
\[ f' = a + b + c \]

Key idea is that fault classes are related with respect to testing:

- Test sets should guarantee to detect certain fault classes
FAULT CLASS HIERARCHY

Not all the faults are equal

- Among the fault classes it may exist a **hierarchy**
- A class F1 subsumes another F2 if a test suite that is able to detect all the faults in F1 then it will also detect all the faults in F2.

The hierarchy is useful when generating tests
TESTING CRITERIA

To target these faults, several testing criteria have been (and are continuously) introduced

A testing criteria must define an algorithm to derive the tests

- It analyzes the structure of the expression
- It finds the right truth values for the clauses

simplest: implicant Coverage

- Make each implicant evaluate to “true”
OTHER TESTING CRITERIA

MAX-A and MAX-B
  • Weyuker, Goradia, and Singh

Multiple Unique True Points (MUTP)

Multiple Near False Points (MNFP)

Corresponding Unique True Point Near False Point (CUTPNFP)

\[ \text{MUMCUT} = \text{MUTP} + \text{MNFP} + \text{CUTPNFP} \]

• Chen, Lau, and Yu

It has been proved that MUMCUT criteria detect all the faults in the hierarchy

• Very efficient (faults/number of tests)
• Several variations to reduce number of tests
• New criteria with different fault detection capability
A NEW WAY TO GENERATE FAULT DETECTING TESTS
Instead of introducing a new testing criterion, a generation methods that targets explicitly the fault classes:

- new fault classes can be added if needed
- or removed

Testing and proving become complementary: tools, methods and techniques generally used for property verification can be efficiently employed to solve testing problems.
DETECTION CONDITION

Let $\varphi$ be a Boolean expression and $\varphi'$ be one faulty implementation.

**Definition** detection condition $dc = \varphi \oplus \varphi'$

- where $\oplus$ is the exclusive or.

$dc$ is is true only if $\varphi'$ evaluates to a different value than the correct predicate $\varphi$.

- $\varphi$: true, $\varphi'$: false
- $\varphi$: false, $\varphi'$: true

The fault can be discovered only when there exists a test case $t$ in which the condition $\varphi \oplus \varphi'$ evaluates to true, i.e., $t \models \varphi \oplus \varphi'$

This $dc$ is also called Boolean **difference** or **derivative**

$dc$ represents our test goal -> test predicate
DETECTION CONDITION
EXAMPLE

• Literal Omission Fault
• If the Boolean predicate $\phi = ab$
• is implemented as $\phi' = a$
• The detection condition is $dc = ab \oplus a$
  • $\equiv a \land \neg b$
  • Is true only if $a$ is true and $b$ is false
• Our test predicate is $a \land \neg b$
DETECTING ALL THE FAULTS IN A CLASS

Let \( \phi \) be a predicate and \( \mathcal{C} \) a fault class.

\[ F_{\mathcal{C}}(\phi) \text{ the set of all the possible faulty implementations of } \phi \text{ according to the fault class } \mathcal{C} \]

- \( F_{\mathcal{C}}(\phi) \) can be obtained by applying the mutation operator \( \mu_{\mathcal{C}} \) that represents the fault class \( \mathcal{C} \) to \( \phi \).

The set of test predicates to discover the \( \mathcal{C} \) faults in \( \phi \) are the expressions

\[ TP_{\mathcal{C}}(\phi) = \{ \phi \oplus \phi' | \forall \phi' \in F_{\mathcal{C}}(\phi) \} \]

Example

- Consider the expression \( a \land b \) and let the fault class \( \mathcal{C} \) be TOF,
- \( F_{TOF} = \{a, b\} \)
- \( TP = \{ab \oplus a, ab \oplus b\} \)
- \( \{a \land \neg b, b \land \neg a\} \)
ADEQUACY OF A TEST SUITE

Let $\phi$ be a predicate and $C$ a fault class.

Let $TP_C(\phi)$ be the set of the test predicates for $\phi$ and $C$.

Definition A test suite $T$ is adequate to test the predicate $\phi$ with respect to a fault class $C$ if it covers every test predicate in $TP_C(\phi)$. Formally:

$$\forall tp \in TP_C(\phi) \exists t \in T \ t \vDash tp$$

Finding a test suite = finding models for test predicates

• a SAT solver can be used for this

Example:

• To find the TOF in $a \land b$ we have to find the models of the two test predicates: $a \land \neg b, b \land \neg a$

• Easy (a,b): [TF], [FT]
SAT-BASED TEST GENERATION METHOD

Fault Classes

Bool Spec

Test Predicate Generator

Model = test

Test Suite Generator

Test Suite

Very naive: a lot of tests and time

SAT

Test predicate

Very naive: a lot of tests and time

A. Gargantini

Generating minimal fault detecting test suites for Boolean expressions
UNFEASIBLE TEST PREDICATES

Not all faulty implementations can be distinguished from the original Boolean predicate:

- Some are “equivalent”

Equivalent faults of Boolean predicates can be detected by the SAT solver:

\[ \phi \oplus \phi' \text{ is unsatisfiable iff } \phi \text{ is equivalent to } \phi'. \]
Generating minimal fault detecting test suites for Boolean expressions

A test covers other test predicates?
MONITORING COVERAGE

A test case generated for one test predicate may satisfy a number of further test predicates.

- test predicates can be skipped because they are covered by tests already generated

Checking whether a test predicate $tp$ is covered by a test case $t$ simply requires evaluating the test predicate with the model that $t$ represents, i.e.

$$ t \models tp $$

it is computationally less expensive then finding a model for $tp$
A. Gargantini

Generating minimal fault detecting test suites for Boolean expressions
ORDERING TEST PREDICATES

When monitoring is applied the order in which test predicates are selected may impact the size of the resulting test suite.


**Random order**

- randomly take the next tp

**Subsuming order**

- If the subsuming relation between fault classes is known, or at least a subsumption relationship is suspected to be in place due some empirical data, it can be used to order tps
  - Start with the test predicates coming from top classes in the hierarchy
  - LIF, LRF, LOF, TOF, LNF, ORF+, ORF*, TNF, and ENF.
Instead of ONE tp, take many TPs, so the test will cover them all.
COLLECTING TEST PREDICATES

Instead of one test for every tp, collect the tps to build a conjoint

Model = test that covers all the test predicates collected

Test suite builder

Collected test predicates\ntp1 \land tp2 \land tp3

Note: When collecting, infeasible tps must be ignored, incompatible tps must be skipped
REDUCTION

Test Predicate

Coverage Evaluator

Test + Cov Info

Test Suite Generator

Model = test

Test suites

Test + Cov Info

Sat

Reduction

Test + Cov Info

TP Ordering

Any unnecessary test in the test suite?

Test predicate

TP Collecting

Ordering

Test Suite

A. Gargantini

Generating minimal fault detecting test suites for Boolean expressions
A test suite is minimal with regard to an objective if removing any test case from the test suite will lead to the objective no longer being satisfied.

- Some tests may be useless

**Simple greedy heuristic to the minimum set covering problem for test suite minimization**

**Note:** Monitoring and minimization can behave very differently:

- Minimization requires existing, full test suites
- while monitoring checks test predicates on the fly during test case generation
EXPERIMENTS
EXPERIMENTS

Benchmark: 20 Boolean expressions in a traffic collision avoidance system (TCAS).

• Introduced for MAX-A and MAX-B (Weyuker et al.)
• Used by MUMCUT (Chen, Lau, and Yu)
• And minimal-MUMCUT (Kaminksy and Ammann)

GOAL: reduce the test suite size
## COMPARISON AMONG OUR STRATEGIES

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Reduction of the test suite size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
</tr>
<tr>
<td>Subsumption order instead of random order</td>
<td>5%</td>
</tr>
<tr>
<td>Reduction</td>
<td>6%</td>
</tr>
<tr>
<td>Collection</td>
<td>24%</td>
</tr>
</tbody>
</table>

The smallest test suites are generated with monitoring, ordering by subsumption, collecting, and minimizing.
COLLECTION IS EXPENSIVE

<table>
<thead>
<tr>
<th></th>
<th>NO COLL</th>
<th>COLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>RND</td>
<td>190.0</td>
<td>45821.2</td>
</tr>
<tr>
<td>SUB</td>
<td>44.2</td>
<td>18380</td>
</tr>
</tbody>
</table>

328 times the time required by the strategy without collection

Collecting test predicates is effective at reducing the number of test cases, but computationally expensive.
COMPARISON WITH MUMCUTS

• Much better than the original MUMCUT strategy
• Always better than the new MUMCUT strategy
• Comparable w.r.t Minimal MUMCUT
Our method reduces the number of test cases necessary to cover all faults of these classes in comparison to MinimalMUMCUT.
CONCLUSIONS

It is possible to generate tests explicitly targeting faults

- SAT solvers can be employed

Several optimizations can be applied

- Monitoring, ordering, collecting, minimization

In comparison to *MUMCUT, it reduces the number of test cases necessary to cover all faults of these classes

Future work:

- Not only DNF
- Improve efficiency: reducing the number of runs of the SAT