Validation of models and tests for constrained combinatorial interaction testing

Paolo Arcaini  Angelo Gargantini  Paolo Vavassori
University of Bergamo- Italy

International Workshop on Combinatorial Testing 2014
Intro to validation

• GOAL: finding faults in combinatorial models and test suites
  • Better before test generation and test execution
  • Normally done by hand by domain experts
• We focus on some faults that are detected by the violation of some meta-properties that
  • must be true in any valid model and test suite
  • automatically checked (by an SMT solver)

• Example
  • *The constraints do not contradict each others*
Desired (meta)-properties

• For combinatorial models and tests

• **Consistency**
  • requires that there are no elements that conflict with each other.
  • e.g. the constraints should not be contradictory

• **Completeness**
  • e.g. every feasible requirement must be covered by at least one test

• **Minimality**
  • guarantees that the specification does not contain elements defined or declared in the model but never used
  • no over-specification
    • e.g. if a parameter value is never used, it could be removed from the parameter domain
    • e.g. the test suite is minimal
CitLab

- http://code.google.com/a/eclipselabs.org/p/citlab/
- Providing testers (and researchers) with some utilities
  - Language (syntax checker) and eclipse-based editor [IWCT12]
  - Test generation via external tools [ICST13]
  - Translators from feature models [IWCT13]

- IWCT14
  A “semantic” checker for validation
  + a guideline on how to solve these problems
Using SMT solver to prove meta-props

1. CCIT models and tests are translated in the SMT solver as SMT problem
2. Every meta-property is translated into the SMT solver
3. If the property holds (valid) then **OK** else **FAULT**
   • **We use the SMT as prover**: $\phi$ valid $\iff \neg \phi$ is sat

Why SMT ?????
A BDD or SAT is enough, but:
• in the future some new features to the CitLab language (functions, arrays ...)
• Some SMT utilities:
  • No need of CNF
  • Incremental resolution
VALIDATION OF CIT MODELS
Inconsistent constraints

A set of constraints \( C = \{c_1, \ldots, c_n\} \) is consistent iff 
\[ \bigwedge_{c \in C} c \text{ is satisfiable} \] 
(i.e. there is a model that satisfies them) 
A model is consistent if its constraints are consistent.

• **Example** of inconsistent constraints:
  \[ \{a \land \neg b, a \rightarrow b\} \]

• In order to discover if a model is consistent, we use the SMT solver by simply checking the satisfiability of the conjunction of the constraints.
How to deal with inconsistent constraints

• In the simplest case a single constraint $c_i$ is inconsistent by itself, i.e., it is a contradiction ($\models \neg c_i$).
  • Remove or correct the constraint
  • Example: $a=5$ and $a=6$

• In most cases there is not a single inconsistent constraint, but the inconsistency derives from the interaction of the constraints
  • Example: \{a \land \neg b, a \rightarrow b, a \lor b\}
  • In this case, the designer may be interested in finding a maximum subset of consistent constraints
    • Greedy algorithm (see paper)
Constraints Vacuity

• Classically a property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the designer originally had in mind or not
  • Example: the property $a \rightarrow b$ is vacuously satisfied by any model where $a$ is never true.

• Here not the same

We borrow the term vacuity to indicate a constraint or one of its subformulas which is useless
Constraints Total Vacuity

• Intuitively, a constraint is totally vacuous if it can be removed because it is always true

A constraint $c_i$ is totally vacuous iff

$$\models \left( \bigwedge_{k \in \{1,\ldots,n\} - \{i\}} c_k \right) \rightarrow (c_i \equiv true)$$

• With the SMT solver, check the validity of the formula above

• **Example**: set of constraints $C = \{\neg a, a \rightarrow b\}$
  • The constraint $a \rightarrow b$ is totally vacuous
  • $\neg a \rightarrow (a \rightarrow b)$ is valid.

• Any tautology is totally vacuous

• More often constraints contain parts that are useless:
  • Partial vacuity
Reducing a formula

- Given a predicate R, \textit{reduce} returns the set of all the formulas obtained from R by removing one occurrence of a subformula in R.

\begin{verbatim}
function reduce(R){
    if R is atomic then return Ø
    if R= a \land b then return a \land reduce(b) \cup reduce(a) \land b
    if R= a \lor b then return a \lor reduce(b) \cup reduce(a) \lor b
    if R= \neg a then return \neg reduce(a)
}
\end{verbatim}

Example:

\(reduce(a \land b) = \{a, b\}\)
\(reduce(a \land (b \lor c)) = \{a, b \lor c, a \land b, a \land c\}\)
Reducing a formula (tree)

- *reduce* can be represented as abstract syntax tree “cutting”
Vacuity (general)

A constraint $c_i$ is partially vacuous if there exists $\phi \in \text{reduce}(c_i)$ such that \((\bigwedge_{k \in \{1,\ldots,n\} - \{i\}} c_k) \rightarrow (c_i \equiv \phi)\)

- If this is true, then $\phi$ is equivalent to $c_i$ (assuming all the other constraints).
- Using $\phi$ instead of $c_i$ would give an equivalent simpler model.
- **Example:** $C = \{c_1, c_2\}$ with $c_1 = a \land b$ and $c_2 = (a \lor b) \land d$
  \[
  \text{REDUCE}(c_2) = \{a \land d, b \land d, a \lor b, d\}.
  \]

The constraint $c_2$ is partially vacuous because it is equivalent to $d$
  - $c_1 \rightarrow (c_2 \equiv d)$ is valid
How to deal with **vacuous constraints**

- The vacuity of a constraint may actually be caused by an error
  - Correct the error
- If the model and constraints are correct
  1. Simplify the constraints (to speed up the test generation)
     - *Totally vacuity* → remove one, check again
     - *Partial vacuity* → substitute one with an equivalent subformula and check again
       Or
  2. The user may be interested to keep extra **implied** constraints
     - Keep and introduce a new notation **property**?
Useless Values and Parameters

• A parameter p can contain in its domain some values which are never taken by p.

The value v of a parameter p is useless if, due to the constraints, p can never assume value v.

If the parameter p can assume only a value, then the whole parameter is useless.

• In SMT: v \in D domain of parameter p is useless, iff \( p = v \land \left( \land_{k \in \{1,...,n\} \setminus \{i\}} c_k \right) \) is unsatisfiable.

• Example: Parameter Enumerative a \{a1 a2 a3\};
  Constraints: # a == a.a1 #
How to deal with useless elements and parameters

- Uselessness of parameters and values can be caused by errors in the constraints: the test designer may have inadvertently introduced a restriction not present in the real system
  - In this case, the constraints should be revised.
- Otherwise useless parameters and values can be removed from the model
  - To speed up the test generation
  - However, other constraints may require to be modified accordingly.
VALIDATION OF CIT TEST SUITES
Test suite correctness

- A test suite is sound if every test is syntactically correct and valid:
  - an assignment of values to the parameters is a syntactically correct test if it satisfies the type definitions;
  - a test is valid if it does not violate any constraint
- A test suite is complete if every feasible test requirement is covered

Test suite correctness.
A test suite is correct if it is sound and complete.

Checking the completeness of a test suite requires a satisfiability solver, since in the presence of constraints it is not possible to judge if a test requirement is feasible by syntax checking
How to deal with incorrect test suites

• An unsound test suite must be fixed before it can be used.
  • either discard any invalid test or
    • It may reduce the coverage
  • Or modify it in order to make it valid
    • It may be difficult

• An incomplete test suite can be completed
  • For instance by using a test generator tool that accepts an existing possibly incomplete test suite (often called seeds)

• Incorrect test suites signal a fault of the test generation algorithm: useful for researchers when experimenting and implementing new techniques
Test suite minimality

• A test generally cover several other tuples covered by other tests
  • Redundancies
• Some tests that overlap may be eliminated without reducing the total coverage of the test suite.
• test suite reduction or minimization

A test suite TS is minimal if there exists no subset $TS' \subset TS$ such that $TS'$ satisfies all the testing requirements as the original set TS does, i.e., that all the tuples covered by TS are also covered by TS

• How to recognize a non-minimal test suite?
Test suite minimality check

• A test case $t_i$ is essential if it covers at least one tuple in TP (the set of all the tuples for a given n-wise coverage) not covered by other test cases of the test suite TS.

  • Formally, $\exists tp \in TP: t_i \models tp \land (\neg \exists t_i \in TS: (i \neq j \land t_j \models tp))$

A test suite TS is non-minimal iff TS contains at least a not essential test.
How to deal with non minimal test suites

• Not essential tests can be removed
  • *Not at once*: removing a not essential test may make others essential...

• Test suite reduction (also known as test suite minimization) is often applied in regression testing
  • The problem of finding the minimal test suite that satisfies a set of test goals can be reduced, in polynomial time, to the minimum set covering problem which is NP-hard.

• A simple greedy heuristic for the minimum set covering problem defined can be adapted to the test suite minimization.
EXPERIMENTS
Experiment setup

• A set of 64 models with constraints taken from the literature
  • CASA (M. B. Cohen, M. B. Dwyer, and J. Shi)
  • FoCuS (I. Segall, R. Tzoref-Brill, and E. Farchi)
  • ACTS (Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence)
  • IPO-S (A. Calvagna and A. Gargantini)
  • used in most papers

• SMT solver yices http://yices.csl.sri.com/
Good news

• No benchmark contains inconsistent constraints
• No tool (ACTS, CASA, ATGT) produced incorrect test suites

• Somehow expected because the models are used in the literature and the tools are validated by other experiments
  • Future work: use during model design or to setup new algorithms
1. Vacuity detection

- 37 models presented at least one vacuity
  - More than 50% of the benchmarks
  - *bench_n*: randomly synthetized from real case studies [CASA]
  - But others as well
    - See table

<table>
<thead>
<tr>
<th></th>
<th>Vacuous subformalas</th>
<th>Vac. constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommProt</td>
<td>814</td>
<td>76%</td>
</tr>
<tr>
<td>Concurr</td>
<td>7</td>
<td>14%</td>
</tr>
<tr>
<td>gcc</td>
<td>6</td>
<td>5%</td>
</tr>
<tr>
<td>HealthC2</td>
<td>14</td>
<td>8%</td>
</tr>
<tr>
<td>ProcComm2</td>
<td>43</td>
<td>83.2%</td>
</tr>
<tr>
<td>Services</td>
<td>81</td>
<td>7%</td>
</tr>
<tr>
<td>SmartHome</td>
<td>33</td>
<td>77%</td>
</tr>
<tr>
<td>Storage1</td>
<td>205</td>
<td>43%</td>
</tr>
<tr>
<td>Telecom</td>
<td>5</td>
<td>5%</td>
</tr>
</tbody>
</table>
2. Useless values and parameters

- 23 models have at least one useless parameter value or one useless parameter
- 21 *bench_n*
  - Ok, but they should be fixed
- ProcComm2
  - “real-life test space instance generated by or for our customers’
- SmartHome
  - From a Feature Model: useless parameters are present because the model has been automatically obtained from a feature model without applying any optimization.

Are we sure they are good benchmarks?
3. Non minimal test suites

- ACTS in two cases (3% of all the cases) produced non minimal test suites
  - However, still very easy to find and fix
Conclusions

- Validation of combinatorial models and tests by proving some semantic meta-properties:
  - Consistency and not vacuity (total or partial) of constraints,
  - Utility of parameters and elements
  - Correctness and minimality of test suites
- Requires a solver (SMT in our case)
- How to deal with them? Still open problem.
- Useful for users and researchers

Thank you