Using verification tools for test generation

Angelo Gargantini
Department of Information Technology and Mathematical Methods
Università di Bergamo- Italy
http://cs.unibg.it/gargantini
University of Bergamo

- public University
- rather young

<table>
<thead>
<tr>
<th>Schools</th>
<th>Established in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreign Languages and Literatures</td>
<td>1968</td>
</tr>
<tr>
<td>Economics and Business Administration</td>
<td>1974</td>
</tr>
<tr>
<td><strong>Engineering</strong></td>
<td>1991</td>
</tr>
<tr>
<td>Educational Studies (formerly Arts and Philosophy)</td>
<td>2001</td>
</tr>
<tr>
<td>Law</td>
<td>2004</td>
</tr>
<tr>
<td>Human Sciences</td>
<td>2006</td>
</tr>
</tbody>
</table>

**Information Technology** and Mathematical Methods:

- 1 Full Professor – Stefano Paraboschi
- 3 Associate Professors
- 6 Assistant Professors
- 3 post docs, 5

Security, DB, Robotics, ...
About me

- Master in Electronic Engineering @ Polytechnic of Milano
- PhD in Software Engineering @ Polytechnic of Milano
- PhD in Computer Science @ University of Catania
- Assistant professor @ UniBg since 2005

- I teach courses of basic and advanced programming
- Collaborations
  - Elvinia Riccobene @ Unimi, Milan, Italy
  - Angelo Morzenti @ Polimi, Milan, Italy
  - Connie Heitmeyer @ NRL, Washington DC
  - Gordon Fraser @ Sarrland University, Germany
    - (thanks for many parts of these slides)
  - Siamak Haschemi @ Humboldt Universität zu Berlin, Germany
My research

- Formal methods
  - Abstract State Machines .... See Elvinia’s slides
- Testing
  - Model-based test generation
- 3D vision therapy
  - Virtual rehabilitation
- More on my web site
USING VERIFICATION TOOLS FOR TEST GENERATION
Testing vs. Verification

• Testing
  • Execute subset of possible scenarios
  • Confidence in correctness - validation
  • Incomplete

• Formal verification
  • (Mathematical) proof of correctness
  • Complex
  • Tool supported/verified

• Tests and proofs seem competing techniques
  • **PROVER**: If correctness is proved, what do we need tests for?
    • Eventually we will get rid of testing
  • **TESTER**: proving is too limited in applicability and testing is the only true path to correctness
    • Eventually we won’t need to prove software correctness
Interesting tale:

• Taken from Daniel M. Berry, *Academic Legitimacy of the Software Engineering Discipline*, 1992

• Peter Naur published about proof-assisted structured programming in 1969.
  • Proof of correctness of 25 lines of Algol
  • Mainly by using natural language

• Burt Leavenworth in 1970 found a fairly trivial boundary condition fault that would have been found easily in a test.

• Ralph London in 1971 found another similar fault
  • He formally proved his version of the program correct
  • But not tested

• John Goodenough and Sue Gerhart in 1975 found three more faults by testing it!
Tests and Proofs, both are here to stay

• Ed Brinksma from his 2009 keynote at the Dutch Testing Day and Testcom/FATES:
  • ``Who would want to fly in an airplane with software proved correct, but not tested?''

• combination of both approaches:
  • modern test systems rely on techniques deeply rooted in formal proof techniques,
  • and testing techniques make it possible to apply proof techniques where there was no possibility previously.

• Example of combination:
  • USING VERIFICATION TOOLS FOR TEST GENERATION
Using Model Checking for Tests Generation

From:
Angelo Gargantini and Constance Heitmeyer

Using Model Checking to Generate Tests from Requirements Specifications

Software Engineering — ESEC/FSE ’99- Lecture Notes in Computer Science, 1999

310 citations

When visiting NRL in 1997/98

Abstract. Recently, many formal methods, such as the SCR (Software Cost Reduction) requirements method, have been proposed for improving the quality of software specifications. Although improved specifications are valuable, the ultimate objective of software development is to produce software that satisfies its requirements. To evaluate the correctness of a software implementation, one can apply blackbox testing to determine whether the implementation, given a sequence of system inputs, produces the correct system outputs. This paper describes a specification-based method for constructing a suite of test sequences, where a test sequence is a sequence of inputs and outputs for testing a software implementation. The test sequences are derived from a tabular SCR requirements specification containing diverse data types, i.e., integer, boolean, and enumerated types. From the functions defined in the SCR specification, the method forms a collection of predicates called focuskdes, which “cover” all possible software behaviors described by the specification. Based on these predicates, the method then derives a suite of test sequences by using a model checker’s ability to construct counterexamples. The paper presents the results of applying our method to four specifications, including a simple component of a contractor specification of a real system.

1 Introduction

During the last decade, numerous formal methods have been proposed to improve software quality and to decrease the cost of software development. One of these methods, the SCR (Software Cost Reduction) method, is based on a user-friendly tabular notation and offers several automated techniques for detecting errors in software requirements specifications, including an automated consistency checker to detect missing cases and other application-independent errors [14]; a simulator to symbolically execute the specification to ensure that it captures the users’ intent [15]; and a model checker to detect violations of critical application properties [21,12]. Recently, groups at NASA and Rockwell Aviation as well as our group at NRL have used the SCR techniques to detect serious errors in requirements specifications of real-world systems [7,11,12]. By exposing defects in the requirements specification, such techniques help the users improve the specification’s quality. This improved specification provides a solid foundation for the later phases of the software development process.

While high-quality requirements specifications are clearly valuable, the ultimate objective of the software development process is to produce high-quality software...
Model Checker

• Model checking problem
  • $M \models \varphi$?
    • Model $M$ satisfies the property $\varphi$?
    • $M$: given as machine (normally kripke structure or similar)
    • $\varphi$: given in some form of (temporal logic) like LTL, CTL

Counter example: a possible behavior of the machine $M$ that violates the property $\varphi$. Called also witness.
Model checker details

- LTL, CTL – linear vs. branching time logics
  - Safety: "something bad never happens"
  - Liveness: "something good eventually happens"
- Explicit model checking
  - Like Spin
- Symbolic model checking
  - BDD encoding (NuSMV)
- Bounded model checking
  - Based on SAT

Test generation as model checker problem

• Test generation as model checker problem
  • MC is used only as “engine”

• Basic ideas:
  1. Using the counter example as test
  2. Using as property to be verified a trap property:
  • not a real property but property with the goal of obtaining the right test
  • derived from our test requirements
  • as negation of the test goal
  • to force the generation of a counter example that cover the test goal
Example

Test goal – test predicate: \( x = 10 \)

- Trap property: \( \text{never}(x = 10) \)
  \( AG(x \neq 10) \): “\( x \) is always (AG) different to 10”

- Counter example (if found)
  - A sequence of states that falsifies the trap property
  - Behavior that shows that \( x \) can be equal to 10
  - A test case for \( x = 10 \)
  - \textit{It covers the test goal} \( x = 10 \)
How to produce a test suite

**SPECIFICATION**

- Test predicates
- Coverage Criteria

Test Suite generator

- Trap property `never(tp)`
- Model checker

- Cex = *witness* of `tp = test`
- Counter example
- Test + coverage info

Test suite
Some limits

1. Trap property proved false
   Cex found

2. Trap property proved true
   Test predicate infeasible

3. MC does not complete the proof
   • Out of memory (state explosion problem)
   • Out of time (timeout reached)
   • Wrong abstractions used by MC
Coverage Criteria

• How to derive test goals?
  • We can create any number of test cases
  • Which test cases do we really need?
  • Test goals are generated according a criterion

• Examples of criteria:
  1. Coverage of the structure of the machine
     • (structure-based)
     • Similar to code coverage
  2. Coverage of combinations of values for inputs
     • Combinatorial Interaction testing (in presence of constraints)
  3. Coverage of faults
Example NuSMV

MODULE main
VAR
accelerate: boolean;
brake: boolean;
velocity: { stop, slow, fast };
ASSIGN
init(velocity) := stop;
next(velocity) := case
accelerate & !brake & velocity = stop : slow;
accelerate & !brake & velocity = slow : fast;
!accelerate & !brake & velocity = fast : slow;
!accelerate & !brake & velocity = slow : stop;
brake: stop;
TRUE : velocity;
esac;
Graphic notation

velocity = stop

accelerate & !brake

velocity = slow

! accelerate & !brake

accelerate & !brake

velocity = fast

brake

! accelerate & !brake

brake
1. Structure-based Testing Criteria

• Covers the structure of the model
• Similar to program-based testing criteria
• Example:

1. State coverage
   • Every variable takes every possible its value

2. Decision Coverage
   • Every decision is taken (true and false)

3. MCDC: modified condition decision coverage
   • Each condition is shown to independently affect the value of the final decision
   • Mandatory on commercial aircraft software (FAA)
Example: coverage of values for velocity

• We want to cover all the values for velocity

• Three test goals:
  • velocity = stop,
  • velocity = fast,
  • velocity = slow

• Take one, e.g.
  • velocity = slow

• Trap property:

  $\text{AG}(\text{velocity} \neq \text{fast})$

• Counter example:

  $\rightarrow \text{State: 1.1} \leftarrow$
  $\text{accelerate} = \text{FALSE}$
  $\text{brake} = \text{FALSE}$
  $\text{velocity} = \text{stop}$

  $\rightarrow \text{State: 1.2} \leftarrow$
  $\text{accelerate} = \text{TRUE}$

  $\rightarrow \text{State: 1.3} \leftarrow$
  $\text{velocity} = \text{slow}$

  $\rightarrow \text{State: 1.4} \leftarrow$
  $\text{accelerate} = \text{FALSE}$
  $\text{velocity} = \text{fast}$
Other criteria

- Decision/guard coverage

- MCDC
  - A little bit more complex
  - Formal definition of MDCD ...
Open issues

• How to compare several testing criteria?
  • Which one is “better” than others
    • Test suites sizes, time effort, test suite length
    • Subsumption ....

• Which model checker is better ?
  • BDD?
  • Explicit state?
  • BMC: SAT – SMT?
Open issues

• Order of test predicates

• Impact over the test length
  • Fraser e Gargantini, Experiments on the Test Case Length in Specification Based Test Case Generation in Fourth International Workshop on the Automation of Software Test (AST09) - ICSE (2009)
Further reading

• Background
  • A. Gargantini and C. Heitmeyer
    *Using Model Checking to Generate Tests from Requirements Specifications*
    in Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference,
  • A. Gargantini and E. Riccobene
    *ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation*
  • Gordon Fraser, Paul Ammann, and Franz Wotawa. Testing with Model Checkers: A Survey.
2. Combinatorial testing

- Failures often result from an interaction between components
- Faults are caused by the interaction of inputs or parameters
- Which combination of parameters?
- Every combination may be infeasible

- Combinatorial testing
Example of “Too many”!

- Suppose we have a system with on-off switches:

Every combination?
34 switches = $2^{34} = 1.7 \times 10^{10}$ possible inputs = $1.7 \times 10^{10}$ tests
N-wise coverage

• What if we knew that one single switch always causes the fault?
  2 tests would be enough to find if the system is correct:
  - all off, all on

• What if we knew no failure involves more than 2 switch settings interacting?
  • PAIRWISE coverage
  • Every possible combination of two values is tested at least once

• Generalizing:: n-wise combination
  • If only 3-way interactions, need only 33 tests
  • For 4-way interactions, need only 85 tests

• Further info:
  • http://csrc.nist.gov/groups/SNS/acts/
  • http://www.pairwise.org
MC and combinatorial testing

- Model checking / SAT - SMT solving can be used also for combinatorial testing

- Example
- Test goal: the combination
  - velocity = stop and brake = true
- Trap property
  - $\text{AG}(\neg (\text{velocity} = \text{stop} \land \text{brake}))$
Open issues, current research

• Compacting test suites
  • Competition among several groups (and companies) to find small test suites, faster

• Generation in presence of constraints
  • Constrained combinatorial testing
  • Constraints over the parameters (forbidden tuples, ...)
  • Model checkers and SMT/SAT solving can be applied

• Andrea Calvagna and Angelo Gargantini
  *A Formal Logic Approach to Constrained Combinatorial Testing*
3. Fault based testing

• If we knew what kind of errors programmers make?
  • Competent programmer, coupling effect, ...
• Definition of fault classes
• Can we generate tests detecting particular faults?

• Fault-based testing

• See D. Richard Kuhn, *Fault classes and error detection capability of specification-based testing*. TOSEM 99

• Model checking /SAT/SMT can be used too!
Detection condition

- Let $\phi$ be an expression
- and $\phi'$ be one faulty implementation

**Definition** detection condition $= \phi \oplus \phi'$

- where $\oplus$ is the exclusive or

- $\phi \oplus \phi'$ is true only if $\phi'$ evaluates to a different value than the correct predicate $\phi$
  - $\phi: true, \phi': false$
  - $\phi: false, \phi': true$

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

- $\phi \oplus \phi'$ represents our test goal
Detection condition example

- **Fault:** Literal Omission Fault
- If the predicate \( \phi = \text{accelerate} \land \neg \text{brake} \)
  is implemented as \( \phi' = \text{accelerate} \)
- The detection condition is
  - \( \text{accelerate} \land \neg \text{brake} \oplus \text{accelerate} \equiv \)
  - \( \text{accelerate} \land \text{brake} \)

- With \( \text{accelerate} \land \text{brake} \)
  - \( \phi \) is false, \( \phi' \) is true
- Our trap property:
  - \( \text{AG}(!(\text{accelerate} \& \text{brake})) \)
Open issues, current research

• How to obtain test suites that perform better than MCDC and other classical criteria (like MCDC, MUMCUT)
  • Better: **smaller** test suites
  • Angelo Gargantini and Gordon Fraser

• Techniques to make the use of SMT/SAT solving scalable enough
  • Paolo Arcaini and Angelo Gargantini and Elvinia Riccobene

• Application to program-based testing
Practical uses

- Model checking is widely used in HW verification
  - Often used for ATPG: test pattern generation (simulation)
    - Gerth, Rob, Model checking if your life depends on it: a view from Intel’s trenches, LNCS 2057, 2001
- Embedded systems
  - Example: Reactis and Simulink
- For software testing
  - Model based testing
  - Program-based
    - Microsoft

SAGE is running 24/7 on 100s machines: “the largest usage ever of any SMT solver” N. Bjorner + L. de Moura (MSR, Z3 authors)
Conclusions

• Testing and proving can work together

• In particular:
  • Tools for verification can be used for test generation
  • Model checking/SAT/SMT solvers can be used

• Some open issues still remain!