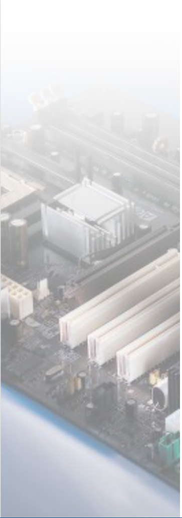


Test Case Generation from Mutants using Model Checking Techniques

Heinz Riener[*] Roderick Bloem[†] Görschwin Fey[*]

[*]Institute of Computer Science, University of Bremen, Germany
[†]Institute for Applied Information Processing and Communications, Graz University of Technology, Austria

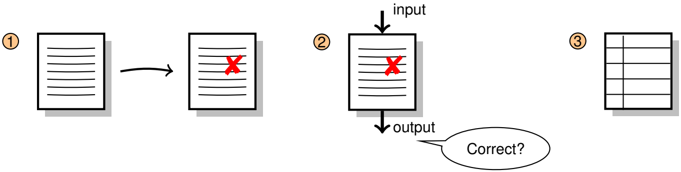
Mutation Analysis 2011
Berlin, Germany

24th March 2011

Test Case Generation with Mutation Testing

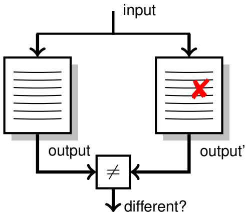
How does mutation testing work?

1. Seed artificial faults.
2. Write test cases that catch the faults.
3. Collect the test cases in a database.



The Problem

But: writing test cases that catch the faults is tedious [GSZ09]!



A test case has to satisfy three conditions to catch a fault [DO91].

1. Reach the location of the fault.
2. Infect the program state when the fault is executed.
3. Propagate the infected state to an output.

Help from Formal Methods?

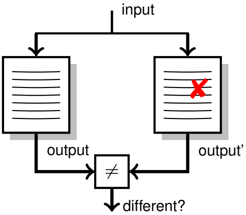
Bounded Model Checking [BCCZ99]

- Given: a state-transition graph (model) and a safety property
- Search for a counterexamples of finite length k

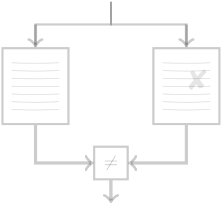
Satisfiability Modulo Theories (SMT) [BHvMW09]

- Efficient decision procedure for constraint satisfaction
- Selects a specific background theory
- SMT solvers are implemented on top of SAT solvers
- Solving the constraints is NP-complete

Test Case Generation with Formal Methods

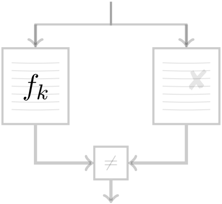


Test Case Generation with Formal Methods



Use BMC to encode all paths up to length k

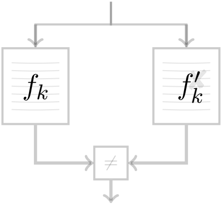
Test Case Generation with Formal Methods



Use BMC to encode all paths up to length k

- Encode the program into formula f_k

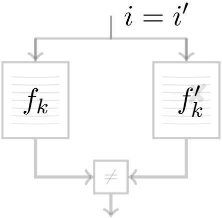
Test Case Generation with Formal Methods



Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k

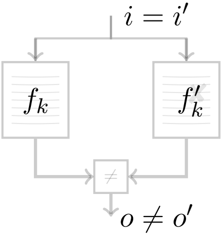
Test Case Generation with Formal Methods



Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'

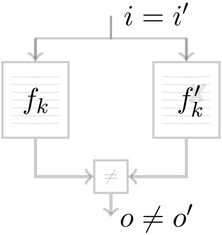
Test Case Generation with Formal Methods



Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'
- Assert different outputs o, o'

Test Case Generation with Formal Methods



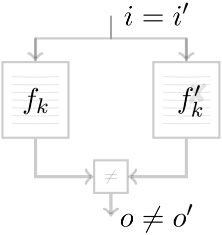
Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'
- Assert different outputs o, o'

Use an SMT solver over bitvector theory

- Solve $f_k \wedge f'_k \wedge (i = i') \wedge (o \neq o')$

Test Case Generation with Formal Methods



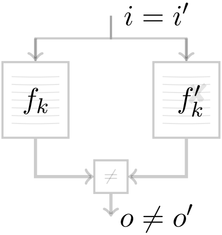
Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'
- Assert different outputs o, o'

Use an SMT solver over bitvector theory

- Solve $f_k \wedge f'_k \wedge (i = i') \wedge (o \neq o')$
- A satisfying assignments is a test cases.

Test Case Generation with Formal Methods



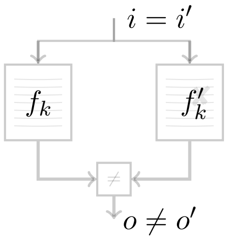
Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'
- Assert different outputs o, o'

Use an SMT solver over bitvector theory

- Solve $f_k \wedge f'_k \wedge (i = i') \wedge (o \neq o')$
- A satisfying assignments is a test cases.
- The mutant is equivalent with respect to k if the formula is unsatisfiable.

Test Case Generation with Formal Methods



Functional equivalence checking with respect to a bounded model is decidable!

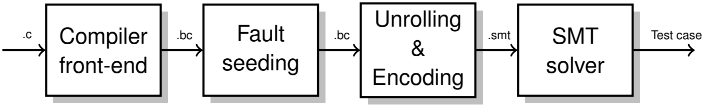
Use BMC to encode all paths up to length k

- Encode the program into formula f_k
- Encode the mutant into formula f'_k
- Assume equal inputs i, i'
- Assert different outputs o, o'

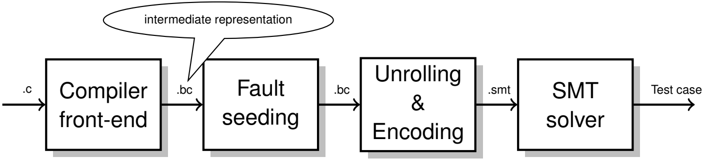
Use an SMT solver over bitvector theory

- Solve $f_k \wedge f'_k \wedge (i = i') \wedge (o \neq o')$
- A satisfying assignments is a test cases.
- The mutant is equivalent with respect to k if the formula is unsatisfiable.

Our Approach — Big Picture

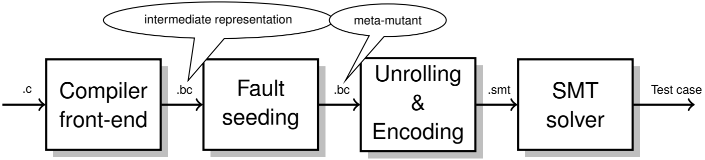


Our Approach — Big Picture



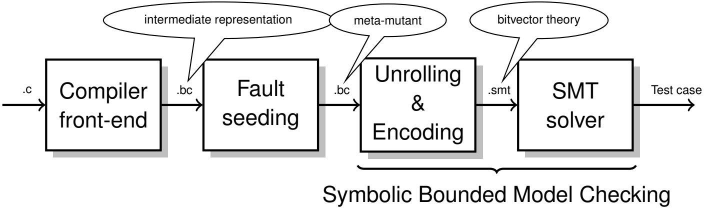
1. Seed faults into the compiler's intermediate representation
   - Independent from the front-end's language
   - Eases later translations!

Our Approach — Big Picture



2. Construct a meta-mutant [UOH93] containing all faults

Our Approach — Big Picture



3. Unroll loops k-times and encode the program as logic formula

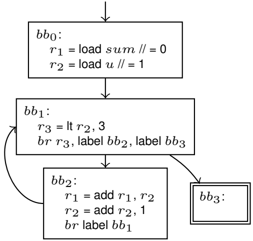
Simple Example

```
sum = 0;
for ( int u = 1; u < 3; ++u ){
    sum += u;
}
```

1. Consider a fragment of a C program.

Simple Example

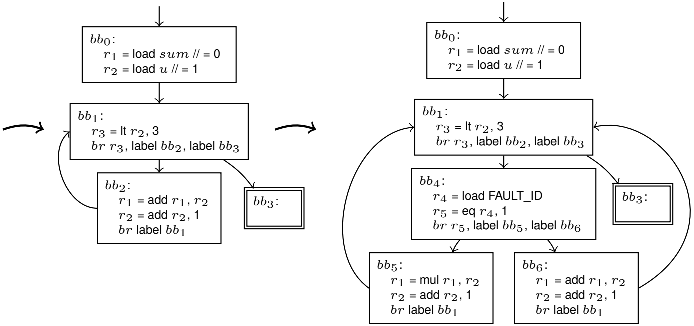
```
sum = 0;
for ( int u = 1; u < 3; ++u ){
    sum += u;
}
```



2. Transform the C code into intermediate code.

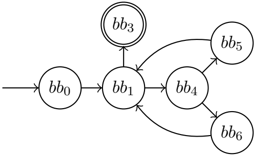
Simple Example

```
sum = 0;
for ( int u = 1; u < 3; ++u ){
    sum += u;
}
```



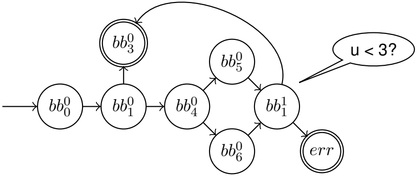
3. Seed faults and construct a meta-mutant.

Simple Example



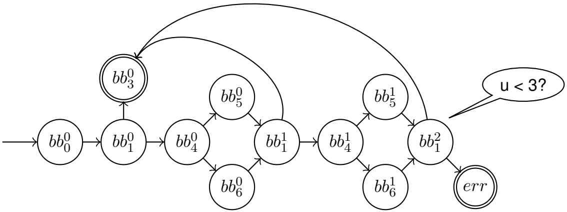
(Redraw the graph with less detail . . .)

Simple Example



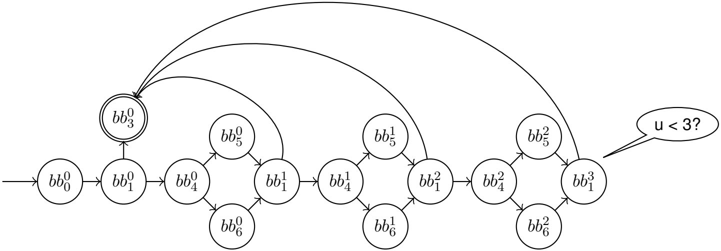
4. Unroll the control flow graph ($k = 1$).

Simple Example



Unroll again ($k = 2$).

Simple Example



Unroll again ($k = 3$).

Simple Example

5. Transform the unrolled meta-mutant into SSA form.

Simple Example

bb_0:
r_1 = load sum // = 0
r_2 = load u // = 1

⋮

$$bb_0 = true \wedge$$
$$bb_0 \rightarrow (r_1 = sum \wedge$$
$$r_2 = u) \wedge$$
$$\ldots$$

6. Encode the meta-mutant into a logic formula.

Test Case Generation

- Given: the meta-mutant M and an unrolling bound k
- The meta-mutant is k-times unrolled and encoded twice into bitvector formulae f_k and f'_k
- We constrain the fault id $id = 0$ in f_k but keep the fault id in f'_k unconstrained
- We assert equal input variables and different output variables
- An SMT-solver incrementally solves $f_k \wedge f'_k$
  - We collect a satisfying assignment as test case, constrain the fault id, and re-solve the formula.
  - When the formula becomes unsatisfiable, all undetect faults are proved to be equivalent.

Experimental Results

Table: Results of the test case generation with Boolector

Name	Instr. (Program)	Instr. (Meta-Mutant)	Faults	Test Cases	Time [s]
min	24	71	17	16	0.48
isl	20	80	19	18	0.14
fmin3	40	137	33	23	7.49
fmin5	58	203	49	37	34.38
fmin10	103	368	89	72	213.65
mid	52	194	46	43	6.82
tri	116	819	206	196	246.80

Experimental Results

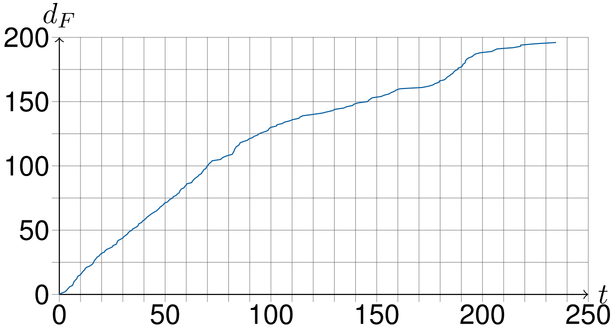


Figure: Detected faults for `tri` over time.

Experimental Results

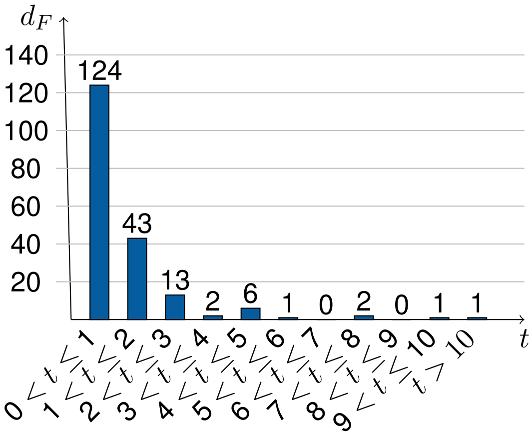


Figure: Test case generation for benchmark `tri` over time.

Conclusions & Future Works

- Our approach is complete with respect to the unrolling bound
  1. It detects all non-equivalent faults in the unrolled model and
  2. proves equivalence with respect to the bound of all undetected faults
- We have not considered simulation, which improves runtime
- Under development:
  - Support for pointers and arrays
  - Case-Study

Thank you for your attention!

Questions?

- A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu.
  Symbolic model checking without BDDs.
  In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207, 1999.
- A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
  Handbook of Satisfiability.
  IOS Press, Amsterdam, 2009.
- R. A. DeMillo and A. J. Offutt.
  Constraint-based automatic test data generatoin.
  IEEE Transactions on Software Engineering, 17(9):900–910, 1991.
- B. J. M. Gruen, D. Schuler, and A. Zeller.
  The impact of equivalent mutants.
  In *International Workshop on Software Testing, Verification and Validation Workshops*, pages 192–199, 2009.

R. H. Untch, A. J. Offutt, and M. J. Harrold.
Mutation analysis using mutant schemata.
ACM SIGSOFT Software Engineering Notes, 18(3):139–148, 1993.