UNIVERSITY OF TEXAS ARLINGTON

Dynamic Symbolic Database Application Testing

Chengkai Li, Christoph Csallner

University of Texas at Arlington

June 7, 2010

Motivation

Maximizing code coverage is an important goal in testing.

- Database applications: input can be user-supplied queries.
- Query results will be used as program values in program logic.
- Different queries thus result in different execution paths.
- To maximize code coverage: we need to enumerate queries in an effective way.

Our Method

Generate queries dynamically by inverting branching conditions in existing program execution paths.

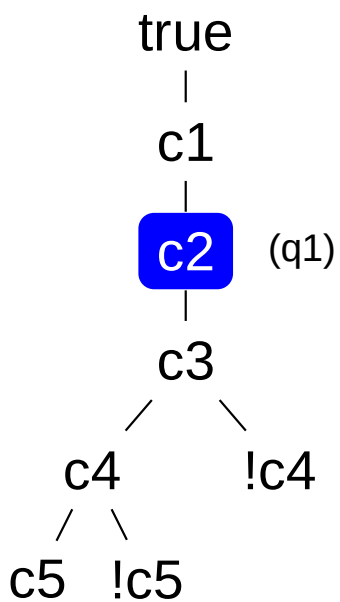
1. Monitor the program's execution paths by *dynamic symbolic execution* (e.g., Dart, Pex).
2. Invert a branching condition on some covered path $\rightarrow$ a new test query.
3. Execute the query, bring in new tuples.
4. The new tuples will cover new paths.
5. Do 1-4 iteratively.

Illustration of the Idea

After the initial query

$q_1 = c_1 \wedge c_2$

Execution tree (maintained by dynamic symbolic engine):
each path to a leaf node represents an execution path, encountered for tuples satisfying the branching conditions on the path.




```
if (z > 0) {          // c1
    if (z < 100)      // c2
        // ..
}
```

Illustration of the Idea

After the initial query, the candidate queries

Each dashed edge represents an inversed branching condition, thus a candidate query.

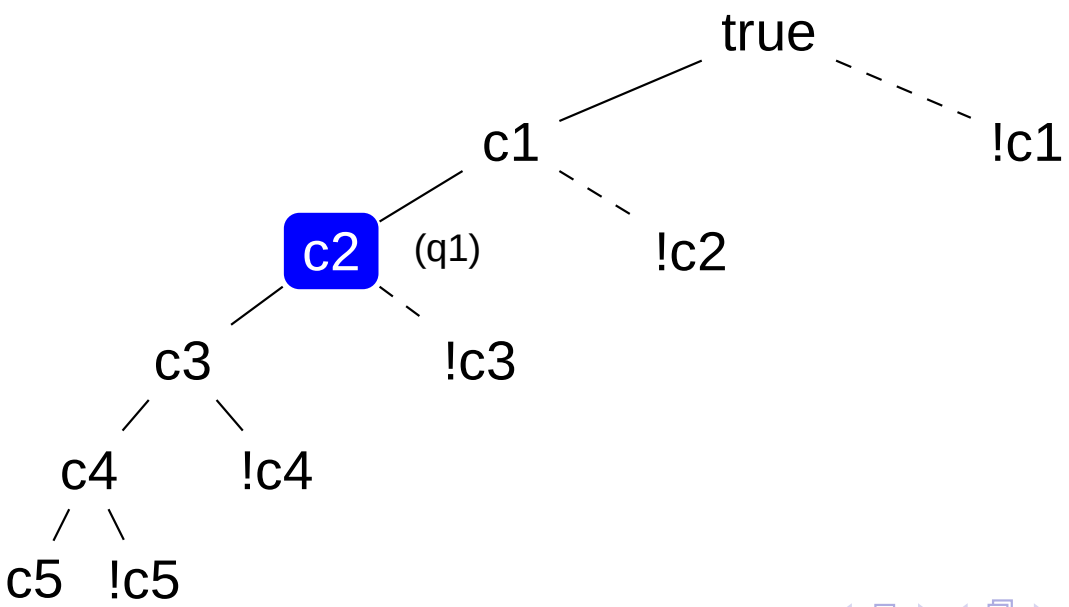


Illustration of the Idea

The second test query

q_2=!c_1

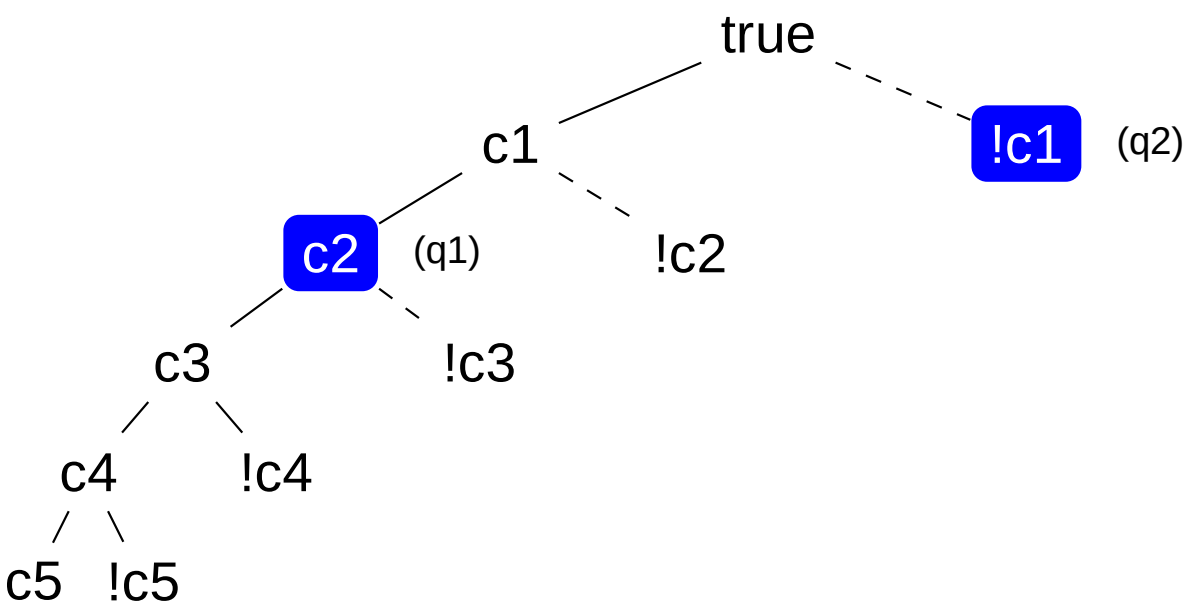


Illustration of the Idea

After the second test query

q_2=!c_1
candidate queries are again dashed.

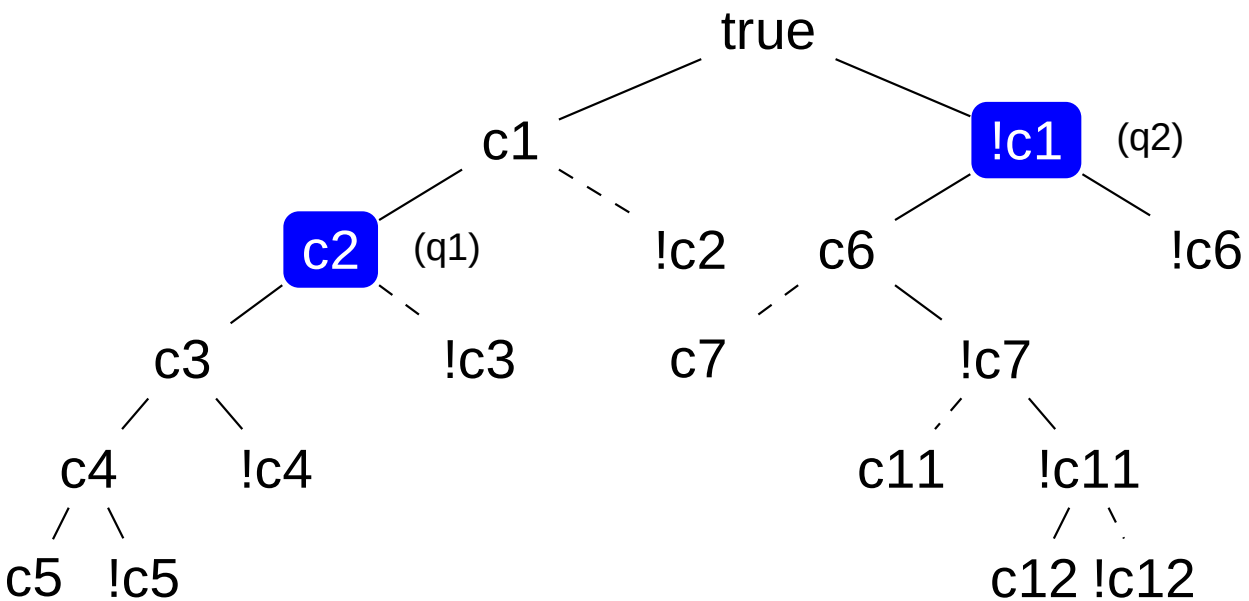


Illustration of the Idea

The third test query

$q_3 = !c_1 \wedge c_6 \wedge c_7$

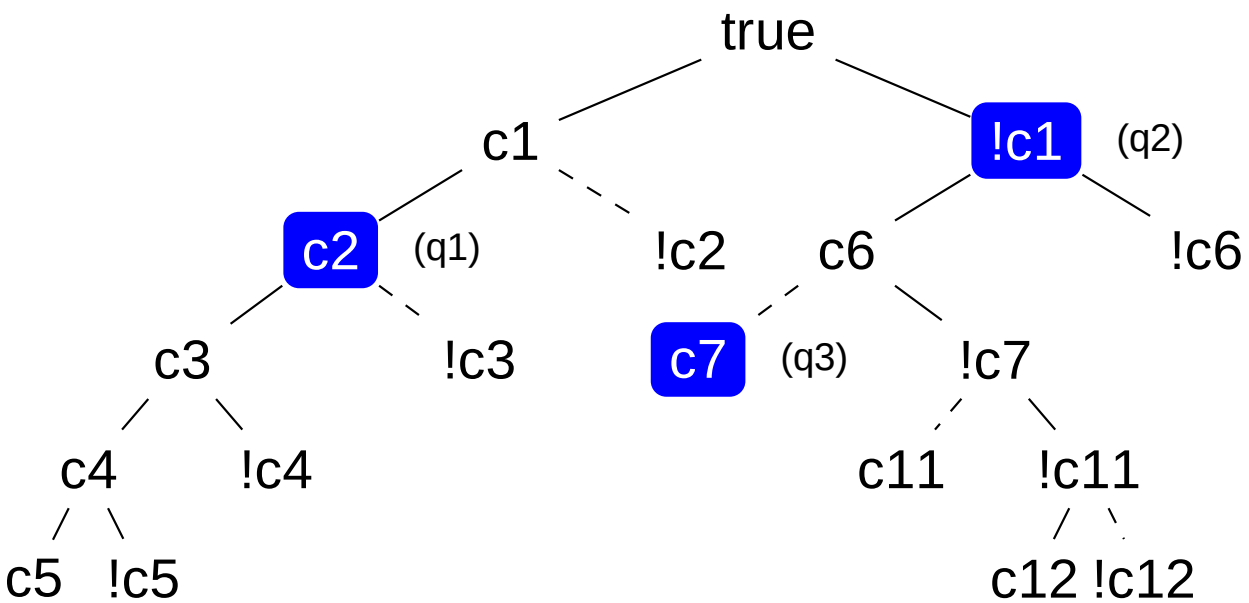


Illustration of the Idea

After the third test query

$q_3 = !c_1 \wedge c_6 \wedge c_7$

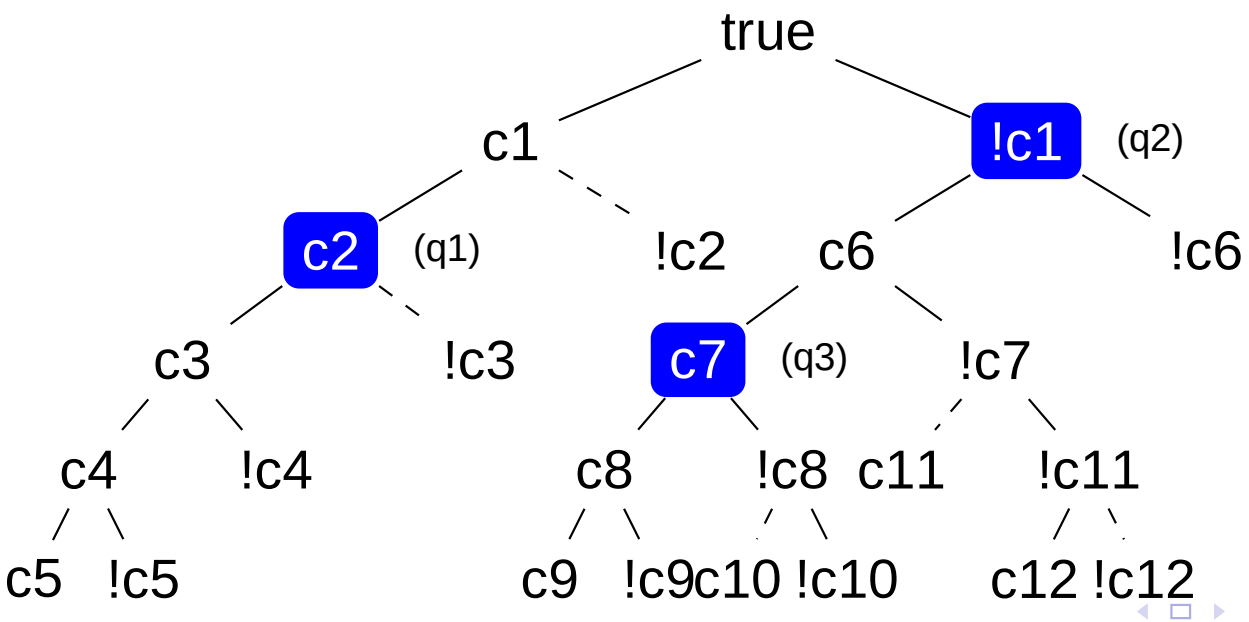


Illustration of the Idea

The fourth test query

$q_4 = !c_1 \wedge c_6 \wedge !c_7 \wedge !c_{11} \wedge !c_{12}$

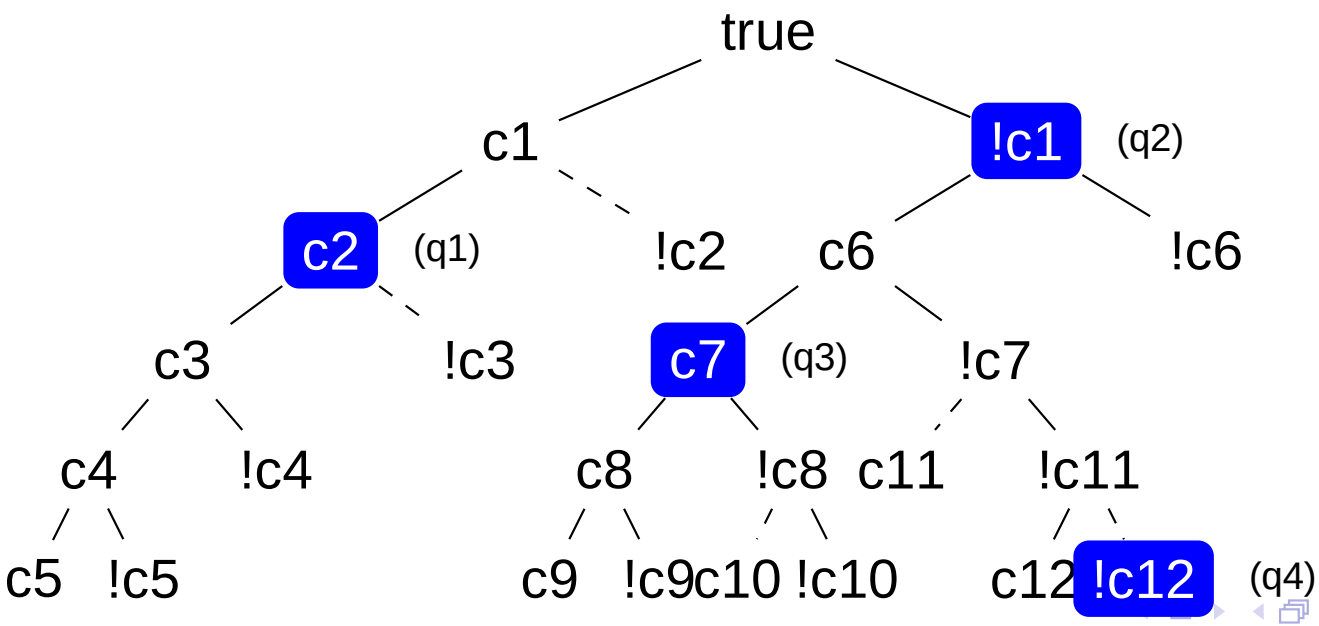
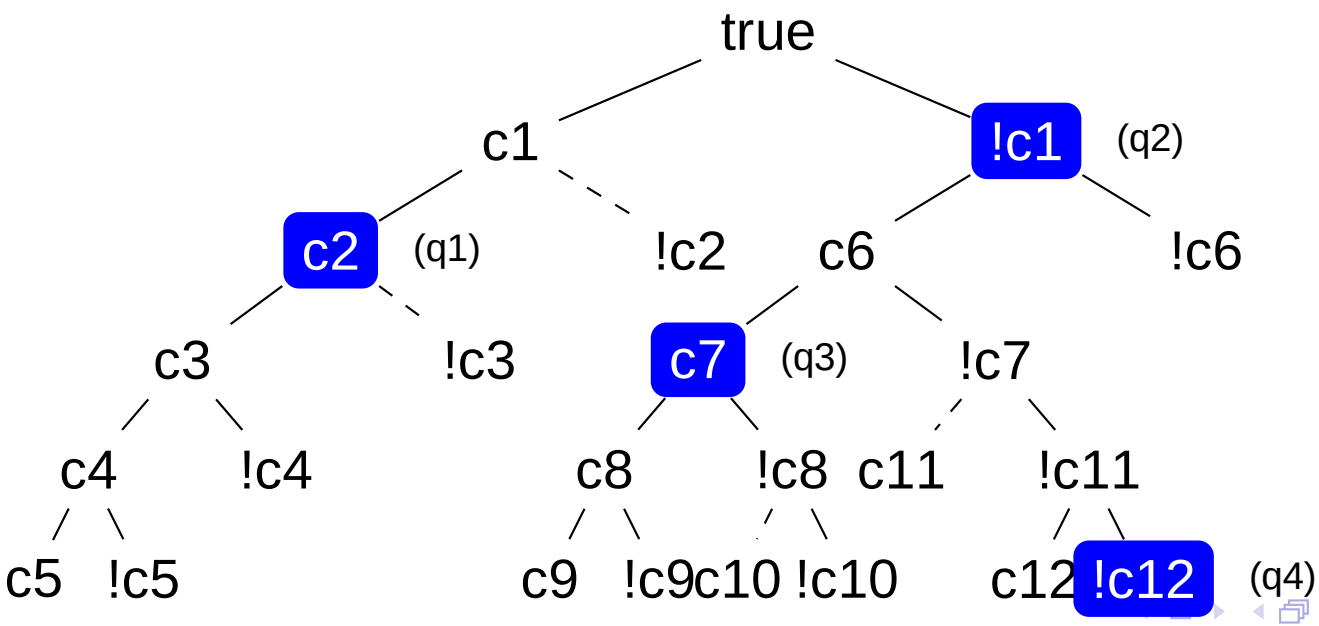


Illustration of the Idea

After the fourth test query

$q_4 = !c_1 \wedge c_6 \wedge !c_7 \wedge !c_{11} \wedge !c_{12}$



Advantages of the Proposed Method

- Real data, no mock database (which can be hard to generate).
- No need to worry about if the mock database is representative.
- Given large space of possible program paths, we only test those that can be encountered for real data.
- This is especially useful for applications that only read existing data.

Alternative Method 1: Brute force

Test for every tuple in database.

- Too costly
  - Limited resources in testing.
  - Many tuples result in the same execution path. Thus efforts wasted.
- May not be possible to get all the tuples
  - Security constraint.
  - Query capability constraint. (e.g., deep-Web databases)

Alternative Method 2: Sample the existing database

Do sampling first, then test for every tuple in the sample.

- A presentative database sample may not trigger a set of program execution paths that is representative of the paths encountered in production use.
- E.g., a column with 1 million distinct values; several particular values will trigger some paths.
- Ours can be viewed as a sampling technique that is aware of the program structure.

Alternative Method 3: Generate custom mock databases

Generate a mock database such that its data will expose a bug in the program

- Will expose potential program bugs.
- But users may not care about them.
- Because many “bugs” will never occur in practice.
- Because the mock database generator typically cannot generate fully realistic databases.

Alternative Method 4: Static Analysis

Static program analysis is typically:

(+) Fast
(-) Imprecise: misses bugs and gives false alarms

Our approach: Test = execute the program (dynamic analysis)

(+) Fully precise: no false alarms
(-) Resource-hungry, will still miss bugs

Our (dynamic) analysis reasons about program + existing database contents. We are not aware of any static analysis that does that.

Assumptions/Limitations

Queries

- single-relation conjunctive selection query.
- Each conjunct is $a \odot v$, where a is an attribute, v is a constant value, and $\odot$ can be $<$, $\leq$, $>$, $\geq$, $=$, or $\neq$.
- no grouping, aggregation, join, insertion, deletion, updates.

Programs

- follow tuple-wise semantics.
- if a branching condition depends on a database tuple, the condition can be rewritten to the same form of the query conjuncts: $a \odot v$.

Iterative Testing Method

1: $q \leftarrow$ define an initial test query; $\mathcal{Q} \leftarrow \{q\}$
2: **repeat**
3: $\quad \mathcal{T} \leftarrow$ run q and get the first n_q result tuples
4: $\quad$ **for** each tuple t in $\mathcal{T}$ **do**
5: $\quad\quad$ run the program over t and update the execution tree $tree_{\mathcal{Q}}$ with encountered new execution paths
6: $\quad \overline{tree_{\mathcal{Q}}} \leftarrow$ the complement tree of $tree_{\mathcal{Q}}$
7: $\quad \mathcal{Q}_c \leftarrow$ get the candidate queries based on $\overline{tree_{\mathcal{Q}}}$
8: $\quad q \leftarrow$ select a query from $\mathcal{Q}_c$
9: $\quad \mathcal{Q} \leftarrow \mathcal{Q} \cup \{q\}$
10: **until** stopping criteria satisfied

Challenges

How to

- decide how many tuples to retrieve for a query?
- choose the next test query?
- design stopping condition for testing?

Optimization Goals

Given program $\mathcal{P}$ and a set of test queries $\mathcal{Q}=\{q_i\}$

maximize coverage

$Path(\mathcal{P},\mathcal{R},\mathcal{Q}) = \{Path_t | t \in \bigcup_{\mathcal{T}_i}\}$, where $\mathcal{T}_i$ is the first n_i tuples for query q_i.

minimize cost

$cost(\mathcal{Q}) = \sum_i cost(q_i)$
$cost(q_i) = q\_cost(q_i) + t\_cost(q_i) = w + c \times n_i + t \times n_i$

- t_cost: t is test cost per tuple.
- q_cost: w is query cost to get first result tuple, c is query cost to get each additional tuple.

Why only n_i tuples for a query q_i?

Multiple tuples will result in the same program execution path. After a certain number of initial tuples, most or all distinct paths may have been encountered.

Less retrieved/tested tuples means both less testing cost and less query execution cost.

How to choose next q and n

Greedy Approach

Given candidate query q,

$$score(q) = \frac{cost'(q)}{|Path'(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q}\cup\{q\})|-|Path(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q})|}$$

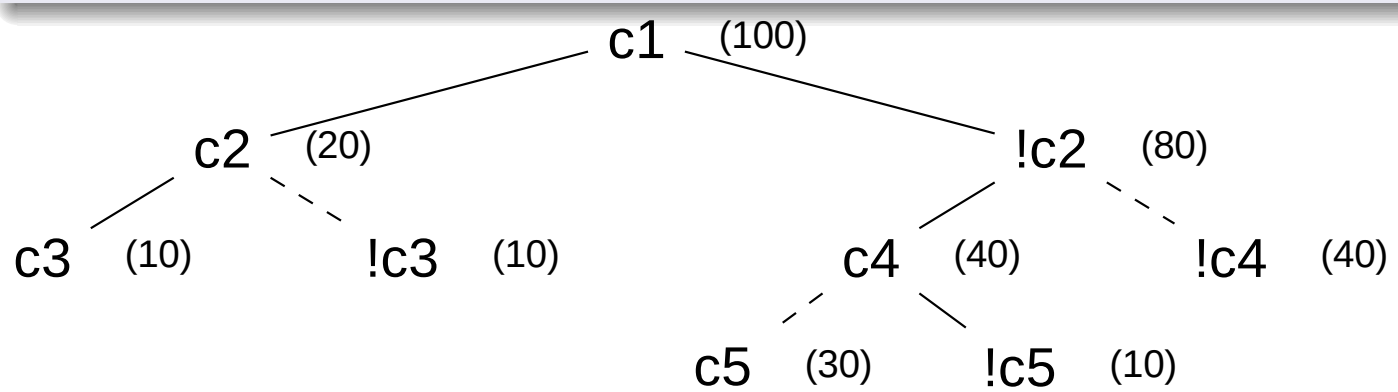
$|Path'(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q}\cup\{q\})|$: estimate of $|Path(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q}\cup\{q\})|$
$cost'(q)$: estimate of $cost(q)$
(both are functions of n)

find q that minimizes $score(q)$

Estimating the Coverage and Cost

Estimating the Coverage

- Estimate the query result size of leaf node (query).
- The result sizes for intermediate nodes are accumulated.



Estimating the Cost

- both initial tuple cost and total cost.

EXPLAIN (supported by major DBMSs)

Stopping Condition for Testing

- testing resource limit reached
- no more candidate queries
- no candidate query can return non-empty result
- total number of encountered tuples (associated with distinct paths) equals the table size

Implementation

Overview

- Fully automated tool
- Analyze Java bytecode programs (any Java program, no need for source code)
- Rewrite application bytecode at load-time: after each application bytecode instruction, insert a call to our dynamic symbolic engine
- Use inserted calls to maintain an accurate symbolic representation of program state
- Treat calls to database (e.g., Jdbc) differently: Represent returned values as symbolic variables and track how the program uses them, i.e., in path conditions

Implementation

Details

- Use Java 5 instrumentation facilities
- Use third-party open source bytecode instrumentation framework ASM
- Implement on top of new dynamic symbolic engine Dsc:
- Allows handling of regular (non-query) program inputs
- Solve constraints on regular program inputs with powerful third-party satisfiability modulo theories (SMT) constraint solver Z3

Ongoing and Future Work

Several directions

- Finish prototype implementation
- Evaluate on realistic applications
- Compare with mock-database generation techniques + compare with traditional database sampling techniques:
- Can we achieve higher coverage of the application code that is reachable with the existing database contents?
- How to deal with database insert, update, delete?

Thank you!

Contact

cli@uta.edu, csallner@uta.edu

References

Dynamic Symbolic Execution Systems

- Dart: C programs, by Godefroid et al. [PLDI'05]
- jCute: Java programs, by Sen et al. [CAV'06]
- Klee: C programs, by Cadar et al. [OSDI'08]
- Pex: .Net programs (C#, etc.), by Tillmann et al. [TAP'08]

Database application testing via mock database generation

- jCute extension: Java programs, by Emmi et al. [ISSTA'07]
- Qex (Pex extension): .Net programs (C#, etc.), by Veanes et al. [ICFEM'09]

References

Main tools used by our prototype implementation

- ASM: http://asm.ow2.org/
- Z3: http://research.microsoft.com/en-us/um/redmond/projects/z3/