Taking symbolic execution to the libraries

Sarfraz Khurshid  Yuk Lai Suen
(now at Microsoft)

University of Texas at Austin

PASTE
6/sep/5
**assert-first programming**

Programmers have long used assertions to state crucial properties of code

- Various dynamic and static analyses make use of assertions

We believe we can squeeze more value from assertions and make them a viable form of program annotations

- Testing
- Repair

**Abstract symbolic execution** provides enabling technology

- Can unify software verification and resilient computing

Assert-first programming has the potential to provide the benefits of test-first programming but at a lower cost

- It is easier to write an assertion than to manually construct a high quality test suite or a correct repair routine
our take on symbolic execution

problem with traditional symbolic execution: it does not scale

proposed solution: try not to perform it fully symbolically

- treat a handful of fields symbolically
  - e.g., in repair, we selectively make fields symbolic
- provide direct support for symbolic execution of certain (commonly used) classes
  - give semantics for symbolic manipulations of objects and solve constraints in ensuing path conditions
  - alleviate the need to symbolically execute intricate implementations of library code
    - prevent path conditions from becoming too complex and choking underlying solvers
example

consider a red-black tree

- binary search tree
- red nodes have black children
- same number of black nodes on all paths from root to leaf

```java
class TreeMap {
    Entry root;
    int size;
    static class Entry {
        int key;
        Entry left, right, parent;
        boolean color;
    }
    ...
```

size: 9
assertion example

class invariant of TreeMap

    boolean repOk {
        if (root == null) return size == 0; // empty tree
        if (root.parent != null) return false; // root has no parent
        // check acyclicity and parent relation
        Set visited = new HashSet();
        List workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
            ...
        }
        if (visited.size() != size) return false; // check size
        ... // check colors
        ... // check keys
        return true;
    }
test generation example

korat: monitor executions of repOk to systematically enumerate inputs for which repOk returns true [boyapati+02, marinov05]
  • provides non-isomorphic generation
simple to implement using a model checker [khurshid+03]
efficient for enumerating a large number of small (~ a dozen nodes) structures
example: size=3, i.e., 3 nodes, 3 keys
repair example

juzi: on assertion violation, repair the state of the program and let it continue to execute [garcia05, khurshid+05, suen05] can be efficient for repairing large structures (~ 10K nodes) with a small number of corruptions

eexample
resilient computing background

fault-tolerance and error recovery have featured in software systems for a long time
most of the past work has been on specialized repair routines
  • file system utilities, such as fsck
  • commercial systems, such as IBM MVS operating system and lucent 5ESS switch
demsky and rinard’s framework is more generic [OOPSLA’03]
  • declarative constraints define desired structures
  • mapping defines data translations between abstract and concrete states
  • requires users to provide mappings and learn a new constraint language
outline

motivation

traditional symbolic execution
  • supporting references
supporting library classes
  • towards an implementation
discussion
traditional forward symbolic execution

technique for executing a program on symbolic input values
  • pioneered three decades ago [boyer+75, king76]
explore program paths
  • for each path, build a path condition
  • check satisfiability of path condition
various applications
  • test generation and program verification
traditional use focused on programs with fixed number of
  variables of primitive types
concrete execution path (example)

```c
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
        assert(false);
}
```

initial values: x = 1, y = 0

1 >?= 0
x = 1 + 0 = 1
y = 1 - 0 = 1
x = 1 - 1 = 0
0 - 1 >?= 0
symbolic execution tree (example)

```c
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x - y > 0)
        assert(false);
}
```
handling more general programs

how to handle programs with references or pointers?
  e.g., if (current.left.parent != current) ...

several recent approaches work with arbitrary java/C++ programs
  [khurshid+03, pasareanu+04, visser+04, xie+04, csallner+05, godefroid+05, cadar+05]

common theme: perform symbolic execution at **concrete representation level**
example algorithm

to symbolically execute a method m

• create input objects with uninitialized fields
• execute m
  • follow mainly Java semantics
  • systematically initialize fields on **first-access**
  • add constraints to path condition and check for feasibility
example field initialization

idea: on first access of a field, non-deterministically initialize it to explore all aliasing possibilities

when method execution accesses field \( f \)
  if (\( f \) is uninitialized) {
    if (\( f \) is reference field of type \( T \)) {
      non-deterministically initialize \( f \) to
      – null
      – a new object of class \( T \) (with uninitialized fields)
      – an object created during prior field initialization
    }
    if (\( f \) is numeric field)
      initialize \( f \) to a new symbolic value
  }
consider executing the statement
\[ \text{next} = \text{t}.\text{next}; \]
outline

motivation

traditional symbolic execution
  • supporting references

supporting library classes
  • towards an implementation

discussion
abstract symbolic execution (dianju)

basically the same algorithm as before except that objects and methods of supported classes are treated specially

- building constraints on symbolic objects based on predicates
- updating state of symbolic objects based on state modifiers

path conditions may represent rich constraints, e.g.,

\texttt{string}\_0.equals("hello") and !\texttt{set}\_0.contains(\texttt{int}\_0)

dedicated constraint solvers, e.g., for strings, sets, and maps

- based on dedicated generators, e.g., for generating mathematical objects that represent sets (or maps)
  - can be focused to avoid/provide generation of certain values, e.g., a set must contain the value null

TestEra [ASE’01] had direct support for objects encapsulating primitives and arrays; GSE [TACAS’03] handled strings
example benefits in test generation

consider generating objects of class Test where field s is initialized to HashSet objects

```java
class Test {
    Set<Integer> s; // s != null
}
```

dianju does not require detailed class invariant
- e.g., s != null suffices; no need for invariant for HashSet

as an (extreme) example consider generating tests with 9 integers
- korat evaluates 3M candidates and generates 26K valid structures, while dianju evaluates $2^9 = 512$ candidates
- for systematic testing of library implementations, korat’s approach is necessary; for client code, dianju’s suffices
implementation via instrumentation

implementation has three basic components

- special libraries that implement basics of symbolic execution
- support for manipulation of symbolic objects
- constraint solvers, including use of off-the-shelf DP implementations, e.g., CVC-lite [barrett+04]
- a bytecode instrumentation engine that allows using a standard JVM to perform symbolic execution
  - introduces new fields and methods; replaces declarations and operations on supported types with special libraries
  - uses BCEL [dahm, bcel.sourceforge.net], javassist[chiba98]
- a systematic backtracking mechanism can be implemented using off-the-shelf model checkers
instrumentation example

add shadow fields to keep track of field accesses
  Entry left; boolean left_is_symbolic;

replace field accesses with invocations of new methods
  this.left → this.left()

where
  Entry left() {
    if (left_is_symbolic) {
      left_is_symbolic = false;
      left = ...; // non-deterministic initialization
    }
    return left;
  }

implemented using bytecode manipulation
  6:  getfield    #18; //Field left:Ldianju/examples/TreeMap$Entry;
  6:  invokevirtual #252; //Method left():Ldianju/examples/TreeMap$Entry;
nondeterministic initialization

the class Explorer allows emulating nondeterministic choice

- choose method returns an integer value nondeterministically

Explorer.initialize();
do {
    ...
    // i is systematically initialized to 0, 1, 2
    int i = Explorer.choose(2);
    ...
} while (Explorer.incrementCounter());

simple stateless search, similar to VeriSoft [Godefroid97]

- bounded depth-first
outline

motivation

traditional symbolic execution
  • supporting references

supporting library classes
  • towards an implementation

discussion
how symbolic execution enables testing

black-box [ISSTA 2002]
  • symbolically execute repOk; inputs for which it returns true are desired test inputs

white-box/hybrid [TACAS 2003, ISSTA 2004]
  • symbolically execute method under test; on field initialization, take into account preconditions
how symbolic execution enables repair

to repair structure \textit{s} [SPIN 2005]

• execute \texttt{s.repOk()} and monitor the execution
  • note the order in which fields of objects in \textit{s} are accessed

• when execution evaluates to false, backtrack and modify value of the \textbf{last} field that was accessed
  • modify the field value to a new (symbolic) value that is not equal to the original value

• re-execute \texttt{repOk}
role of assertions

efficient symbolic execution can unify software verification and resilient computing via the use of assertions

• systems can be systematically tested before deployment as well as ensured to behave as expected once deployed

applicability

• assertion-based techniques have minimal cost
  • assertion describes *what*; test generator or repair routine describes *how*

scalability

• it is possible to abstract away from irrelevant details

assertions are already immensely popular in hardware verification; the time has also come that we realize the potential benefits assertions have long offered in software