Kex: how we are using SMT solvers to generate tests

Azat Abdullin

Nov 18, 2021

Kex

Kex — white-box fuzzer for JVM bytecode

- based on symbolic execution

Motivating example

```
class Point(
  val x: Int,
  val y: Int
)

fun foo(a: List<Point>) {
  if (a.size == 2) {
    if (a[0].x == 10) {
      if (a[1].y == 11) {
        error("a")
      }
    }
  }
}
```

How to create a test case from the model?

Problem of symbolic execution

```
Model {
  this = 131072
  arg$0 = 4
  %0.inlined0 = 2
  arg$0.size = 4
  %5 = false
  %0.inlined7 = 27
  (274)<3> = 121
  (260)<3> = 101
  type(0)<1> = 0
  length(258)<2> = 11
  String.value(273)<2> = 274
  type(0)<2> = 5
  ...
  type(1)<12> = 2
  ArrayList.elementData(4)<11> = 1
}
```

How to create a test case from the model?

Related work

- Symstra
  - builds valid method sequence during analysis
- JBSE
  - uses reflection utilities to create tests
- Sushi & Tardis
  - use EvoSuite (search-based approach) to generate tests

Related work: JBSE

- test are hard to comprehend and maintain
- can generate invalid objects

```
public class TestSuite {
    public void test4() {
      example.ListExample __ROOT_this = (example.ListExample)
         newInstance("example.ListExample");
      java.util.ArrayList __ROOT_a = (java.util.ArrayList)
         newInstance("java.util.ArrayList");
      new AccessibleObject(__ROOT_a)
        .set("java/util/ArrayList:size", 2L);
      new AccessibleObject(__ROOT_a)
        .set("java/util/ArrayList:elementData",
                newArray("java.lang.Object", 2L));
      new AccessibleObject(__ROOT_a)
        .set("java/util/ArrayList:elementData[0]", null);
      __ROOT_this.foo(__ROOT_a);
    }
  }
```

Related work: Sushi

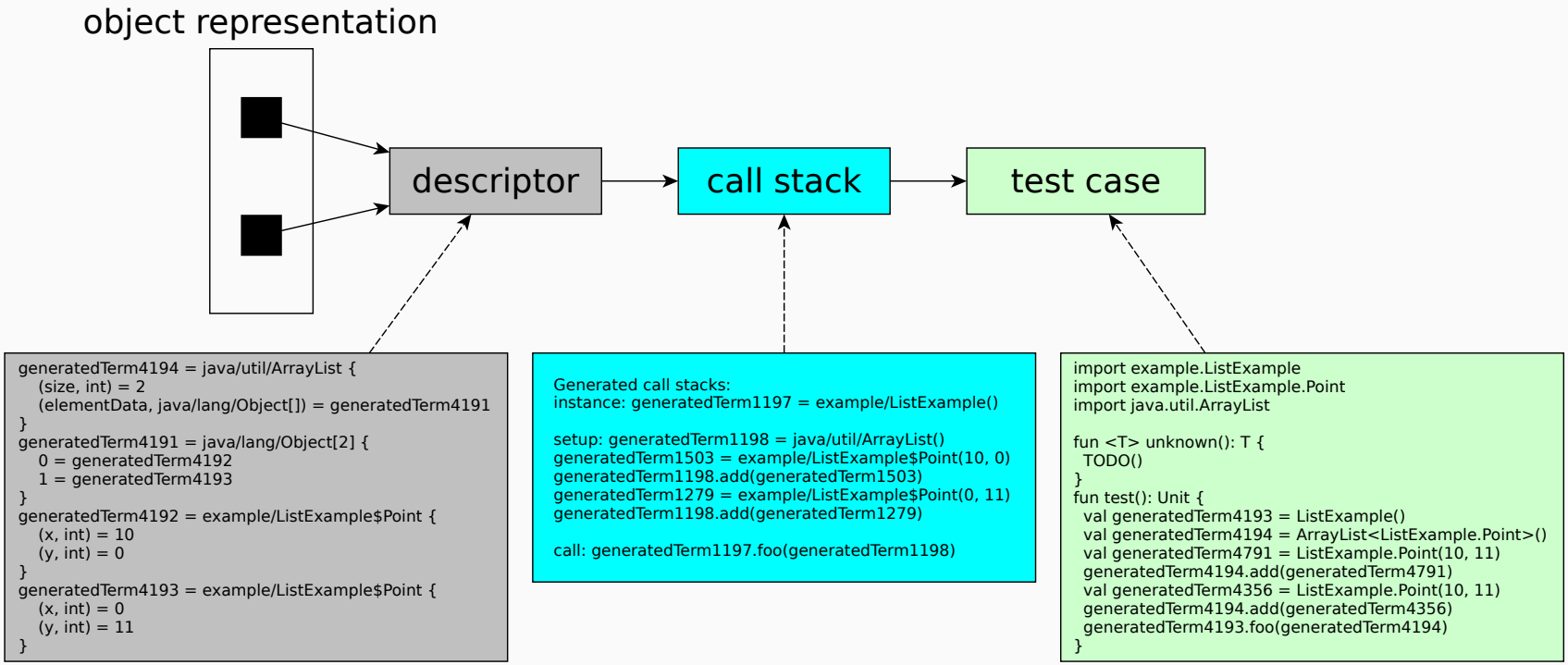
- converts path conditions into fitness function
- EvoSuite tries to generate satisfying test case
- very slow

```
public class JListExample_0_0_Test {
  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      ListExample listExample0 = new ListExample();
      ArrayList<Point> arrayList0 = new ArrayList<Point>();
      listExample0.foo(arrayList0);
  }
}
```

Reanimator

- an approach to generate valid code snippets using only public API
  - can’t produce invalid objects
- works in reasonable time
- applicable in any automatic test generation tool
- can be used in any programming language

Reanimator


object representation
descriptor
call stack
test case
generatedTerm4194 = java/util/ArrayList {
(size, int) = 2
(elementData, java/lang/Object[]) = generatedTerm4191
}
generatedTerm4191 = java/lang/Object[2] {
0 = generatedTerm4192
1 = generatedTerm4193
}
generatedTerm4192 = example/ListExample$Point {
(x, int) = 10
(y, int) = 0
}
generatedTerm4193 = example/ListExample$Point {
(x, int) = 0
(y, int) = 11
}
Generated call stacks:
instance: generatedTerm1197 = example/ListExample()
setup: generatedTerm1198 = java/util/ArrayList()
generatedTerm1503 = example/ListExample$Point(10, 0)
generatedTerm1198.add(generatedTerm1503)
generatedTerm1279 = example/ListExample$Point(0, 11)
generatedTerm1198.add(generatedTerm1279)
call: generatedTerm1197.foo(generatedTerm1198)
import example.ListExample
import example.ListExample.Point
import java.util.ArrayList
fun <T> unknown(): T {
TODO()
}
fun test(): Unit {
val generatedTerm4193 = ListExample()
val generatedTerm4194 = ArrayList<ListExample.Point>()
val generatedTerm4791 = ListExample.Point(10, 11)
generatedTerm4194.add(generatedTerm4791)
val generatedTerm4356 = ListExample.Point(10, 11)
generatedTerm4194.add(generatedTerm4356)
generatedTerm4193.foo(generatedTerm4194)
}

Descriptor example

```
instance: generatedTerm4193 = example/ListExample {}
args: generatedTerm4194 = java/util/ArrayList {
  (size, int) = 2
  (elementData, java/lang/Object[]) = generatedTerm4191
}
generatedTerm4191 = java/lang/Object[2] {
  0 = generatedTerm4192
  1 = generatedTerm4193
}
generatedTerm4192 = example/ListExample$Point {
  (x, int) = 10
  (y, int) = 0
}
generatedTerm4193 = example/ListExample$Point {
  (x, int) = 0
  (y, int) = 11
}
```

Call stack generation algorithm

```
fun generate(d: Descriptor, limit: Int): CallStack {
  if (limit == 0) return Unknown()
  val calls = mutableListOf<Action>()
  when (d) {
    is ConstantDescriptor -> calls += PrimaryValue(d.value)
    is ArrayDescriptor -> {
      val array = NewArray(d.elementType, d.length)
      for ((idx, elem) in d.elements)
        calls += ArrayWrite(array, idx, generate(elem, limit - 1))
    }
    is StaticFieldDescriptor -> {
      val value = generate(d.value, limit - 1)
      calls += StaticFieldSetter(d.field, value)
    }
    is ObjectDescriptor -> calls += generateObject(d, limit - 1)
  }
  return CallStack(calls)
}
```

Object generation

- each field of the descriptor imposes new constraints
- more fields means more complex generation
- by gradually reducing descriptor we can find a constructor-like call to create an object
    - at each step try to find a method that initializes one or more fields

Object generation algoritnm

```
fun generateObject(d: ObjectDescriptor, limit: Int): List<Action> {
  concretize(d)
  val (d, setters) = generateSetters(d)
  val queue = queueOf(d to setters)
  while (queue.isNotEmpty()) {
    val (desc, stack) = queue.poll()
    if (stack.size > limit) return Unknown()
    for (ctor in desc.ctors) {
      val (nDesc, args) = execAsCtor(desc, ctor)
      if (isFinal(nDesc))
        return stack + CtorCall(desc, genArgs(args))
    }
    for (method in desc.relevantMethods) {
      val (nDesc, args) = execAsMethod(desc, method)
      if (nDesc < desc)
        queue.push(nDesc to calls + MethodCall(method, genArgs(args)))
    }
  }
}
```

Concretization

Replace all non-instantiable types in descriptor:

- abstract classes or interfaces
- non-static inner classes

Setter generation

- preprocess all available classes and find setters
- if an object descriptor contains fields with available setters, reduce the descriptor and add corresponding setter calls

Original descriptor:

```
generatedTerm4194 = Point {
  (x, int) = 2
  (y, int) = 43
}
```

Descriptor:

```
generatedTerm4194 = Point {
  (y, int) = 43
}
```

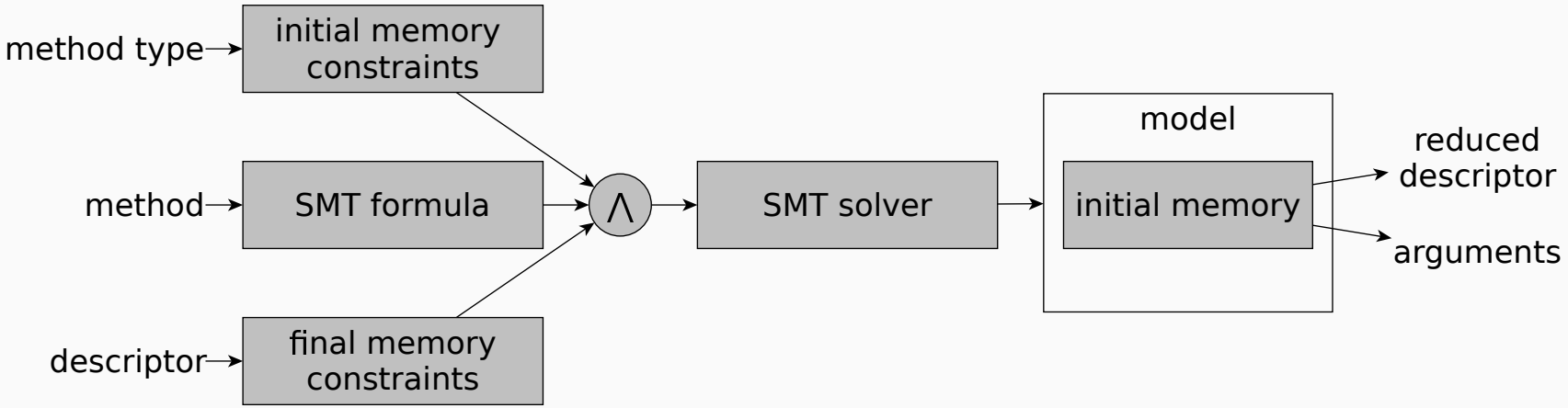
Call stack:

```
generatedTerm4194.setX(2)
```

Checking methods

- `execAsCtor` and `execAsMethod` check how the method affects the descriptor using SMT solver
- method need to be represented as SMT formulae
  - predicate state
- method basically transforms memory state
- descriptor defines final memory
- *need to find initial memory*

Symbolic execution



Method types:

- constructor — initial memory is uninitialized
- setter — setted fields are uninitialized
- method — no constraints for initial memory

Call stack example

```
Generated call stacks:
instance: generatedTerm1197 = example/ListExample()

setup: generatedTerm1198 = java/util/ArrayList()
generatedTerm1503 = example/ListExample$Point(10, 0)
generatedTerm1198.add(generatedTerm1503)
generatedTerm1279 = example/ListExample$Point(0, 11)
generatedTerm1198.add(generatedTerm1279)

call: generatedTerm1197.foo(generatedTerm1198)
```

not enough type information

Test case example

```
import example.ListExample
import example.ListExample.Point
import java.util.ArrayList

fun <T> unknown(): T {
  TODO()
}

fun test(): Unit {
  val generatedTerm4193 = ListExample()
  val generatedTerm4194 = ArrayList<ListExample.Point>()
  val generatedTerm4791 = ListExample.Point(10, 11)
  generatedTerm4194.add(generatedTerm4791)
  val generatedTerm4356 = ListExample.Point(10, 11)
  generatedTerm4194.add(generatedTerm4356)
  generatedTerm4193.foo(generatedTerm4194)
}
```

Experimental setup

- implemented Reanimator as a part of Kex
- using Z3 for query solving
- custom string generator
- generation depth is limited to 5

Evaluation: Reanimator as part of Kex

Project	Coverage	Desc. gen.
authforce	25.80%	70.65%
exp4j	49.63%	84.71%
exposed	44.15%	72.90%
fescar	44.85%	83.34%
imixs	55.49%	88.61%
karg	56.21%	99.32%
kfg	34.72%	45.04%
koin	50.46%	68.81%
kotlinpoet	38.25%	71.74%

Evaluation: generating random objects

Project	Valid objects	All objects	Avg. depth
`authforce`	59.30%	13.48%	2.98
`exp4j`	50.10%	5.97%	2.31
`exposed`	60.90%	13.47%	2.36
`fescar`	68.30%	24.24%	2.11
`imixs`	87.30%	42.26%	2.65
`karg`	71.00%	51.75%	3.41
`kfg`	97.40%	31.62%	3.12
`koin`	46.90%	11.52%	2.97
`kotlinpoet`	95.90%	31.15%	1.40

Evaluation: manual analysis of failures

- 36/150 — complexity of object
- 10/150 — higher order functions
- 13/150 — builder pattern
- 24/150 — enum elements
- 67/150 — uninstantiable objects

Evaluation: efficiency

Project	Kex mode, ms	Random mode, ms
authforce	88.05	174.41
exp4j	1278.78	20.85
exposed	648.79	47.22
fescar	69.04	13.49
imixs	776.65	49.84
karg	103.25	81.43
kfg	96.39	50.17
koin	66.09	33.05
kotlinpoet	70.26	86.54

Conclusion

- an approach for generating valid code snippets to create a given target object
- can successfully and efficiently generate 70% of target objects on average

Future work

- more thorough investigation of Reanimator failures
- improved support of built-in types, such as collections
- higher order functions

Contact information

abdullin@kspt.icc.spbstu.ru

https://github.com/vorpal-research/kex

POLYTECH
Peter the Great
St. Petersburg Polytechnic
University