

Kex: how we are using SMT solvers to generate tests

Azat Abdullin

Nov 18, 2021

Kex — white-box fuzzer for JVM bytecode

- based on symbolic execution
- uses kfg for bytecode manipulation and transformation
- uses PredicateState for program representation
- currently works with z3, boolector and cvc4

Motivating example

```
class ListExample {  
  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")  
        }  
      }  
    }  
  }  
}
```

```
(  
  @S %1 = arg$0.size()  
  @S %3 = %1 != 2  
  @P %3 = false  
  @S %5 = arg$0.get(0)  
  @S %7 = (%5 as Point)  
  @S %9 = %7.getX()  
  @S %11 = %9 != 10  
  @P %11 = false  
  @S %13 = arg$0.get(1)  
  @S %15 = (%13 as Point)  
  @S %17 = %15.getY()  
  @S %19 = %17 != 11  
  @P %19 = false  
)
```

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?

Easy way: reflection

- 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

- Symstra¹
 - builds valid method sequence during analysis
- JBSE²
 - uses reflection utilities to create tests
- Sushi & Tardis³
 - use EvoSuite (search-based approach) to generate tests

¹Xie T. et al. Symstra: A framework for generating object-oriented unit tests using symbolic execution //International Conference on Tools and Algorithms for the Construction and Analysis of Systems. – Springer, Berlin, Heidelberg, 2005. – C. 365-381.

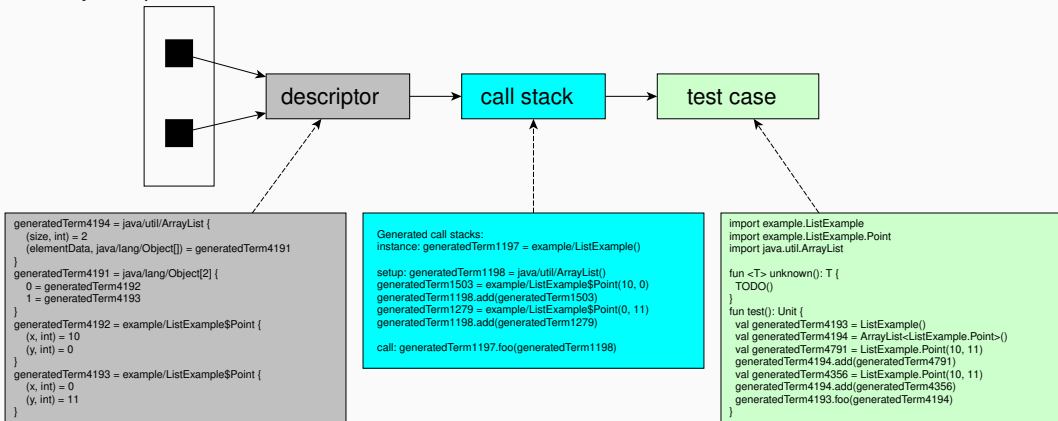
²Braione P., Denaro G., Pezzè M. JBSE: A symbolic executor for java programs with complex heap inputs //Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. – 2016. – C. 1018-1022.

³Braione P., Denaro G. SUSHI and TARDIS at the SBST2019 Tool Competition //2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST). – IEEE, 2019. – C. 25-28. 5/23

- 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 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

- generation of constants and arrays is stright forward
- objects are problematic:
 - there may be no direct access to fields
 - some states of an object are unreachable during normal execution

Object generation

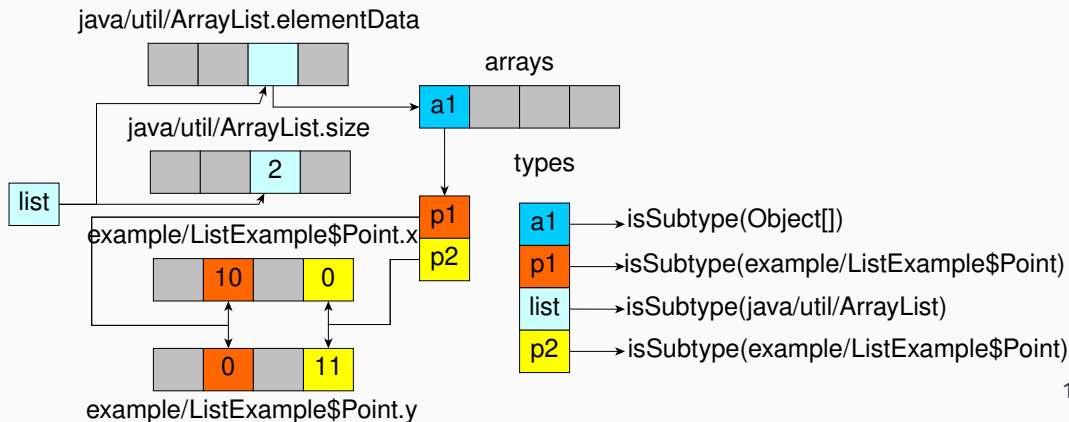
- each field of the descriptor imposes new constraints
- more fields means more complex generation
- by gradually reducing descriptor we can find a constructor-like call to create an object
 - at each step try to find a method that initializes one or more fields

Object generation algorithm

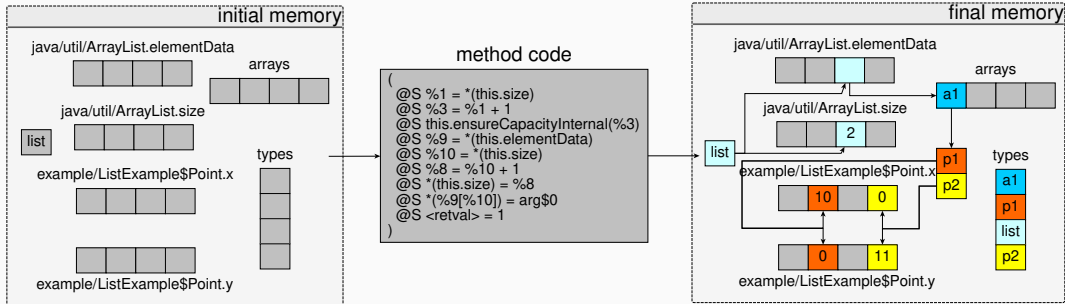
```
fun generateObject(d: ObjectDescriptor, limit: Int): CallStack {  
    val queue = queueOf(d to CallStack())  
    while (queue.isNotEmpty()) {  
        val (desc, stack) = queue.poll()  
        if (stack.size > limit) return Unknown()  
        for (ctor in desc.ctors) {  
            val (nDesc, 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)))  
        }  
    }  
}
```

Memory model in SMT

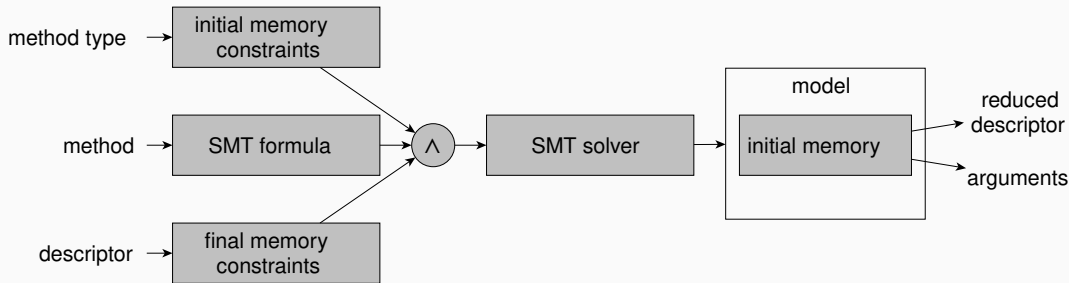
- primitive types encoded as corresponding SMT theories
- references are represented as bitvectors
- arrays are encoded as SMT arrays
- object properties are encoded through *property memories*



Method is a memory transformation



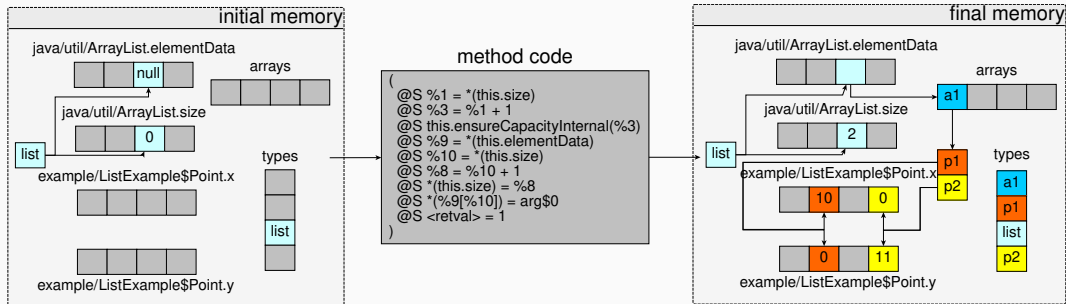
Symbolic execution



Method types:

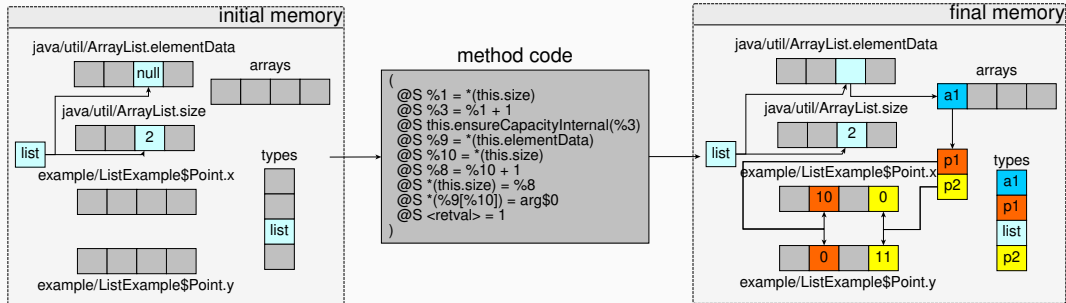
- constructor — initial memory is uninitialized
- setter — setted fields are uninitialized
- method — no constraints for initial memory

Examples: constructor call



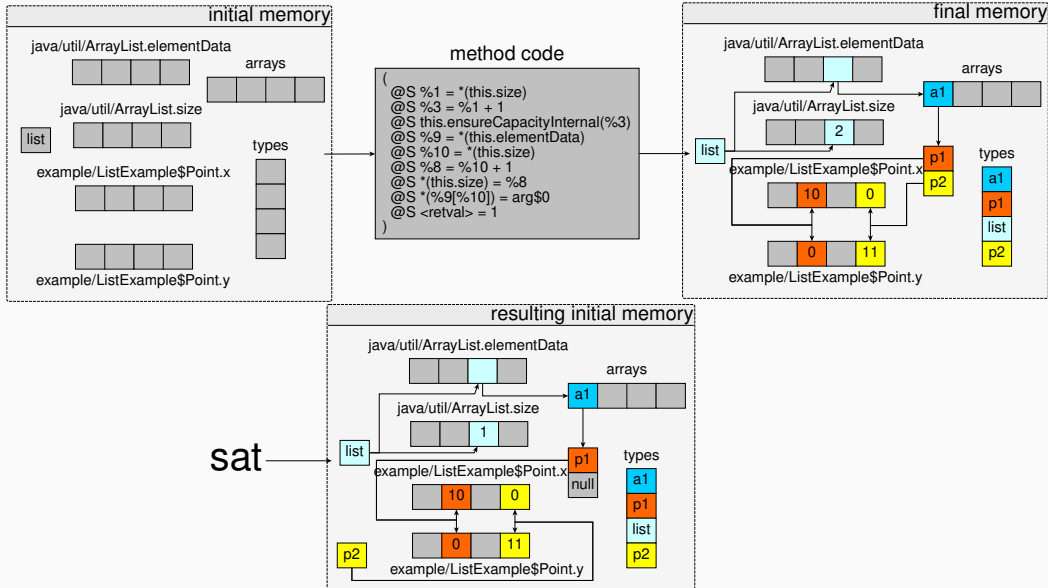
unsat

Examples: setter call



unsat

Examples: method call



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)
}
```

Limitations

- incomplete program model
- built-in types (collections, files etc.)
- search termination

Evaluation on SBST 2021 benchmark⁴

	60s	120s	300s	600s
tardis	13.96%	15.71%	18.50%	19.60%
tardis + reanimator	13.84%	15.99%	17.84%	19.30%
kex + reanimator	24.57%	25.29%	25.43%	27.61%

⁴<https://github.com/JUnitContest/JUGE>

- an approach for test generation based on symbolic execution
- compatible with state of the art alternatives

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>

