

Wikiprint Book

Title: projects/jpf-mango/archive/FirstExample

Subject: Java Path Finder - projects/jpf-mango/archive/FirstExample

Version: 2

Date: 02/21/2013 01:55:34 PM

Table of Contents

Example 1: Hypotheses, Conjectures, and Loop Invariants	3
Source code	3
Specification	3

Example 1: Hypotheses, Conjectures, and Loop Invariants

Mango with JavaPathfinder, jpf-mango, has produced its first totally automated, not-completely-trivial specification.

There are two separate [embedded](#) jpf machines in the code base. The first is "standard" in the sense that it is driven by java byte code. The other one is "exotic", in that it traverses internally generated graphs, interpreting the nodes and edges as instructions from a custom [bytecode factory](#). Heavy use has been made of [choice generators](#) and [VM listeners](#) in both of these jpf embeddings.

The code and analysis for the "nested_blowup" example are included below. This is the first complete example with the new jpf backtrace mechanism for invariants in place. The code was written primarily to stress various mango systems, and so is not representative of anything particularly useful. Mango works from the innermost loop outwards.

Source code

```
32 public class nested_blowup {
33
34     public static void main (int x) {
35
36         while(true){
37             switch(x){
38                 case 1: x+=1;
39                 case 2: x+=2;
40                 default:{
41                     while(x>7){
42                         switch (x*x){
43                             case 1: x*=x;
44                             case 4: x*=4*x;
45                             default:{
46                                 int total=0;
47                                 for(int i=0;i<x;++i){
48                                     for(int j=0;j<x;++j){
49                                         total++;
50                                     }
51                                 }
52                                 x+=total;
53                             }
54                         }
55                     }
56                 }
57             }
58         }
59     }
60 }
61
```

Specification

```

Specifying loop at 48: for(int j=0;j<x;++j){
48: for(int j=0;j<x;++j){ [1]
Hypothesis: op0 is less than x
total = total + 1
j = j + 1
op0 = j + 1
Specifying loop at 47: for(int i=0;i<x;++i){
47: for(int i=0;i<x;++i){ [1]
Hypothesis: op0 is less than x
Conjecture: j^ is greater than or equal to x
i = i + 1
op0 = i + 1
Specifying blown up loop at 41: while(x>7){
41: while(x>7){ [1]
Hypothesis: 7 is less than op0
46: int total=0; [1.1]
Hypothesis: x * x is less than 1 OR 4 is less than x * x OR x * x equals 2 OR x * x equals 3
Conjecture: i^ is greater than or equal to x
x = total^ + x
op0 = total^ + x
43: case 1: x*=x; [1.2]
Hypothesis: x * x equals 1
Conjecture: i^ is greater than or equal to x * x * x * x * 4
x = total^_1 + x * x * x * x * 4
op0 = total^_1 + x * x * x * x * 4
44: case 4: x*=4*x; [1.3]
Hypothesis: x * x equals 4
Conjecture: i^ is greater than or equal to x * x * 4
x = total^_2 + x * x * 4
op0 = total^_2 + x * x * 4
Specifying blown up loop at 37: switch(x){
Cannot generate specification because code is hung at this point.
#Smango.rewriter.synthetic.SuperLoopSym "A!_B!_A!_Component_loopTests.nested_blowup.main(I)V"

```

The spec for each "module", meaning a loop or method body, consists of

Yellow hypotheses: condition to stay in the module

Red conjectures: condition to leave an embedded module

Blue state transition: the effect of the module on its input parameters

The "^" indicates values that are not loop invariant, so "lack" of a ^ is very significant. The "op0" quantity refers to an unknown input value on the operand stack. Typically, by the time a loop starts up, the counter is already loaded into the operand stack, and so its "context" is unknown to the loop. The conjectures are generated automatically and imply loop termination. Of course, the termination conjecture for the outermost loop is false, and this is essentially what causes Mango to give up at this point.

A far more detailed spec is produced in persistent internal form for subsequent referral by dependent modules. But the English spec is intended to be the primary vehicle for delivering meaning to the code assurance analyst.

Actually proving the conjectures, and, even harder, generating appropriate hypotheses for such proofs, are interesting research questions. Anyone interested in the so-called [Grand Challenge Verifying Compiler](#) is welcome to embed the mango model in a theorem prover and tackle this problem. I am always on the lookout for collaborators.

Lots more to do, and lots more examples to generate, but all the pieces are now in place. I should have a stable version of jpf-mango by Spring 2010.