Detecting Likely Maintained Invariants

Daniel Ellsworth and Michal Young
{dellswor,michal}@cs.uoregon.edu

Let's say we need to modify an unfamiliar program…

We could make changes more safely if we knew the pre- and post- conditions each function maintains, but they are seldom documented.

Dynamic invariant detectors, such as Daikon, produce this kind of data.

A dynamic invariant detector instruments a target program to capture the values in variables visible at interesting program points. After execution, it creates, for selected program points, a set of relationships (potential invariants) that held on all observations.

After running Daikon, we have output like the following…

16899 Invariants Detected

```
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- Simulation.VISUAL_SIZE == 5
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- Simulation.VISUAL_SIZE != 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- Simulation.VISUAL_SIZE >= 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- 0 <= Simulation.VISUAL_SIZE <= 63
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.BLASTER_RANGE == 20.0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.BLASTER_RANGE != 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.BLASTER_RANGE >= 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world.getClass() == SimulationObject[][].class
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[..] contains no nulls and has only one value, of length 78
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[..] elements != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.width == 78
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.width != 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.width >= 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.width >= 64
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.height == 31
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.height != 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.height >= 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- 0 <= arg0.height <= 63
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.critters has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.critters != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg1.toString == "critters.dat"
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- size(arg0.world[..])-1 == 77
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- size(arg0.world[..])-1 != 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- size(arg0.world[..])-1 >= 0
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- size(arg0.world[..])-1 >= 64
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[Simulation.VISUAL_SIZE] has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[Simulation.VISUAL_SIZE] != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[Simulation.VISUAL_SIZE-1] has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[Simulation.VISUAL_SIZE-1] != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.width-1] has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.width-1] != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.height] has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.height] != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.height-1] has only one value
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- arg0.world[arg0.height-1] != null
Simulation.readCritterFile(SimulationBoard, java.lang.String):::EXIT73 -- Simulation.VISUAL_SIZE == Simulation.VISUAL_SIZE
```

16,899 invariants! There are only 850 lines of program text including whitespace and comments in the original source. Are there really 20 invariants maintained for each line of source code?

The 3 line bittenBy method is reporting 87 invariants…

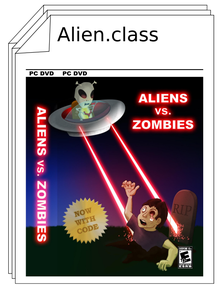
87 useful invariants generated/maintained by these 3 lines? Seems unlikely…

bittenBy can't possibly maintain most of these invariants. They involve variables that are independent on every execution path. Some aren't even read or written by the method.

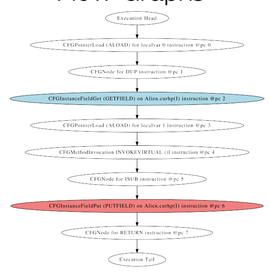
What if we tighten up the definition of a likely invariant to include a predicate only if it relates dependent variables, including one that is modified by the method. Dependency includes control and data dependency through any number of intermediate variables.

```
public void bittenBy(Zombie z) {
        curhp-
=z.getBiteDamage();
```

```
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp == this.curhp
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp - this.curhp == 0
Alien.bittenBy(Zombie):::EXIT42 -- this.name.toString == "THX432"
Alien.bittenBy(Zombie):::EXIT42 -- this.maxhp == 50
Alien.bittenBy(Zombie):::EXIT42 -- this.maxhp != 0
Alien.bittenBy(Zombie):::EXIT42 -- this.maxhp >= 50
Alien.bittenBy(Zombie):::EXIT42 -- this.maxhp <= 50
Alien.bittenBy(Zombie):::EXIT42 -- this.maxhp >= 0
Alien.bittenBy(Zombie):::EXIT42 -- 0 <= this.maxhp <= 63
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie == 0
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie >= 0
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie <= 0
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie == 0
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie >= 0
Alien.bittenBy(Zombie):::EXIT42 -- this.turnsToZombie is boolean
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp <= orig(this.curhp)
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp != 0
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp >= -31
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp <= 46
Alien.bittenBy(Zombie):::EXIT42 -- 0 <= this.turnsToZombie <= 63
Alien.bittenBy(Zombie):::EXIT42 -- Zombie.rand has only one value
Alien.bittenBy(Zombie):::EXIT42 -- Zombie.rand != null
Alien.bittenBy(Zombie):::EXIT42 -- arg0.name.toString == "Bob"
Alien.bittenBy(Zombie):::EXIT42 -- arg0.maxBite == 25
…
```

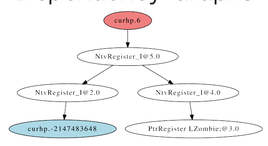


Compute Control Flow Graphs[1]



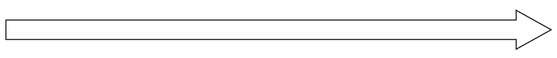
[1]The current implementation uses an intraprocedural analysis for computation of control and data flow. For many interesting programs this is inadaquate since functions are often used to encapsulate non-trivial checks and computations, potentially adding additional control and data dependencies. Current work is directed at extending the analysis for interprocedural analysis.

Compute Variable Dependency Graphs



Filter Invariants

Dynamically detected invariants are only admitted when a connected subgraph of the variable dependency graph contains all variables referrenced in the invariant.



Filtering on dependence reduces the 87 invariants to 5.

```
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp == this.curhp
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp - this.curhp == 0
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp <= orig(this.curhp)
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp != 0
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp >= -31
Alien.bittenBy(Zombie):::EXIT42 -- this.curhp <= 46
```

Dynamic invariant detection has been criticized for producing a large number of shallow invariants. We have demonstrated that filtering on dependence can reduce the noise.

Special thanks to the University of Oregon Computer Science Department for funding this work during Summer 2012.