

Lecture 11

Explicit State Checkers

Zvonimir Rakamarić
University of Utah

Last Time

- ▶ Loops and loops invariants
- ▶ Strategies for proving programs correct

Desugaring While Loop Using Invariant

► while E invariant J do S end

assert J;

check that the loop
invariant holds initially

havoc x; assume J;

jump to an arbitrary
iteration of the loop

(

where x denotes the
assignment targets of S

assume E; S; assert J; assume false

□

assume $\neg E$

check that the loop invariant is
maintained by the loop body

)

exit the loop

Dafny

- ▶ Simple “verifying compiler”
 - ▶ Proves procedure contracts statically for all possible inputs
 - ▶ Uses theory of weakest preconditions
- ▶ Input
 - ▶ Annotated program written in simple imperative language
 - ▶ Preconditions
 - ▶ Postconditions
 - ▶ Loop invariants
- ▶ Output
 - ▶ Correct or list of failed annotations

Proving Correctness: Strategies

- ▶ Read Chapter 6
- ▶ Heuristics, requires intuition and practice
- ▶ Loop invariants are key and typically hardest part
- ▶ Strategies
 - ▶ Include all basic simple facts
 - ▶ For example, loop counter should be between 0 and n
 - ▶ Come up with complex invariants using the “precondition method”
 - ▶ Figure out which fact is failing, and compute its weakest precondition up to loop header
 - ▶ Comes more naturally with practice

This Time

- ▶ Checking concurrent programs using explicit-state model checking

Concurrency is Pervasive

- ▶ Old problem of computer science
 - ▶ Ancient supercomputers
- ▶ Today
 - ▶ Multi-cores even in cell phones
 - ▶ Many-cores in desktops
- ▶ Most programs are concurrent
 - ▶ At least the ones you care about

Concurrency is Hard I

- ▶ Inefficient (dumb) concurrency is easy
 - ▶ Big fat lock around everything
 - ▶ Poor performance
- ▶ Efficient concurrency is hard
- ▶ A concurrent program should
 - ▶ Function correctly
 - ▶ Maximize throughput
 - ▶ Finish as many tasks as possible
 - ▶ Minimize latency
 - ▶ Respond to requests as soon as possible
 - ▶ While handling nondeterminism in the environment

Concurrency is Hard II

- ▶ Huge number of possible thread interleavings/schedules
- ▶ Concurrent program with n threads where each thread has k instructions has

$$(n \cdot k)! / (k!)^n \geq (n!)^k$$

interleavings

- ▶ Exponential in both n and k !
- ▶ Example: 5 threads with 5 instruction each

$$25! / 5!^5 = 6.2336074e+14$$

= 623 trillion interleavings

Concurrency is Hard III

- ▶ Concurrent executions (thread interleavings) are highly nondeterministic
- ▶ Stress testing
 - ▶ Trying to explore many different thread interleavings by creating hundreds of threads
- ▶ Stress testing is highly inefficient
 - ▶ Some concurrency bugs occur only in certain thread interleavings
 - ▶ Finding the “right” thread interleaving is pure luck
 - ▶ No notion of coverage
 - ▶ Running for days, even months

Concurrency Bugs

- ▶ Rare thread interleavings result in Heisenbugs
 - ▶ Difficult to find, reproduce, and debug
- ▶ Observing the bug can “fix” it
 - ▶ E.g., likelihood of interleavings changes when you add `printf` statements
- ▶ A huge productivity problem
 - ▶ Developers and testers can spend weeks chasing a single Heisenbug

Model Checking I

- ▶ Model checking is
 - ▶ checking whether a program satisfies a property by exploring its state space
 - ▶ systematic state-space exploration = exhaustive testing
 - ▶ checking whether a system satisfies a temporal-logic formula

Model Checking II

- ▶ Simple, automatic, and yet effective technique for finding bugs in high-level hardware and software models
- ▶ Invented in the early 1980s
- ▶ 2008 Turing Award
 - ▶ Edmund M. Clarke, E. Allen Emerson, Joseph Sifakis

Software Model Checking Evolution

- ▶ General model checkers
 - ▶ Examples: Spin, SMV, Murphi
 - ▶ Custom input specification languages
 - ▶ Require translation of the program into the input language of the model checker
 - ▶ Not automated
 - ▶ Ad-hoc simplifications and abstractions
- ▶ Specialized software model checkers
 - ▶ Work directly on source code
 - ▶ Input language is a programming language
 - ▶ Well-defined techniques for restricting the state space
 - ▶ Automated abstraction techniques

Simple Example

```
int x, y;
```

Thread 1:

- 1) `x = 1;`
- 2) `y = 2;`
- 3) `x++;`
- 4) `y++;`

Thread 2:

- 5) `y = 3;`
- 6) `x = 2;`
- 7) `y++;`
- 8) `x++;`

Explicit-State Model Checking of Programs

- ▶ Verisoft from Bell Labs
 - ▶ C programs
 - ▶ Handles concurrency, bounded search, bounded recursion
 - ▶ Uses stateless search and partial order reduction
- ▶ Java Path Finder (JPF) from NASA Ames
 - ▶ Java programs
 - ▶ Handles concurrency, bounded search, bounded recursion
 - ▶ Uses techniques similar to the ones in Spin
- ▶ CMC from Stanford for checking systems code written in C

Java Path Finder (JPF)

- ▶ Program checker for Java
- ▶ Properties to be verified
 - ▶ Program assertions
 - ▶ LTL properties
- ▶ Depth-first and breadth-first search, heuristics
 - ▶ Uses static analysis techniques to improve the efficiency of the search
- ▶ Requires a complete Java program
 - ▶ Cannot handle native code

JPF: First Version

- ▶ Translate from Java into the input language of Spin (Promela)
- ▶ Spin cannot handle unbounded data
 - ▶ Restrict the program to finite domains
 - ▶ Fixed number of objects from each class
 - ▶ Fixed bounds for array sizes
- ▶ Does not scale well when these fixed bounds are increased
- ▶ Java source code is required for translation

JPF: Current Version

- ▶ Implements its own virtual machine
 - ▶ Executes Java bytecode
 - ▶ Doesn't need source code
 - ▶ Stores visited states and current path
 - ▶ Exposes various “knobs” to the user to optimize verification
- ▶ Traversal algorithm
 - ▶ Traverses the state-graph of the program
 - ▶ Tells VM to move forward, backward in the state space, evaluate an assertion,...

Storing Program States

- ▶ JPF implements a systematic search on the state space of the given Java program
 - ▶ Systematic search requires storing visited states
- ▶ Program state consists of
 - ▶ Information for each program thread
 - ▶ Stack of frames, one for each called method
 - ▶ Static fields in classes
 - ▶ Locks and fields for classes
 - ▶ Dynamic fields in objects
 - ▶ Locks and fields for objects

Storing States Efficiently

- ▶ Intuition: different states have common parts
- ▶ Divide each state into a set of components and store them separately
- ▶ Keep a pool for each component
 - ▶ A table of field values, lock values, frame values
- ▶ Instead of storing the value of a component in a state, store an index at which the component is stored in the table in the state
 - ▶ The whole state becomes an integer vector
- ▶ JPF collapses states to integer vectors using this idea

State Space Explosion

- ▶ Major challenge in model checking
- ▶ Reduce the number of states that have to be visited during state space exploration

Combating State Space Explosion

- ▶ Symmetry reduction
 - ▶ Search equivalent states only once
- ▶ Partial order reduction
 - ▶ Do not search thread interleavings that generate equivalent behavior
- ▶ Static analyses
 - ▶ Reduce state space using static analyses
- ▶ User-provided restrictions
 - ▶ Manually bound variable domains, array sizes,...

Symmetry Reduction

- ▶ Some states of the program may be equivalent
 - ▶ Equivalent states should be searched only once
- ▶ Some states may differ only in their memory layout, the order objects are created, etc.
 - ▶ May not have any effect on program behavior

Symmetry Reduction in JPF

- ▶ Order in which classes are loaded shouldn't effect the state
 - ▶ There is a canonical ordering of classes
- ▶ Location of dynamically allocated heap objects shouldn't effect the state
 - ▶ If we store the memory location as the state, then we can miss equivalent states which have different memory layouts
 - ▶ Store some information about the new statements and the number of times they are executed

Simple Symmetry Example

```
int x, y;
```

```
Foo a, b;
```

Thread 1:

```
1) a = new Foo();
```

```
2) x = 1;
```

```
3) y = 2;
```

```
4) x++;
```

```
5) y++;
```

Thread 2:

```
5) b = new Foo();
```

```
6) y = 3;
```

```
7) x = 2;
```

```
8) y++;
```

```
9) x++;
```

Partial Order Reduction

- ▶ Statements of concurrently executing threads can generate many different interleavings
 - ▶ All these different interleavings are allowable behavior of the program
- ▶ Model checker checks all possible interleavings for errors
 - ▶ But different interleavings may generate equivalent behaviors
- ▶ Partial order reduction
 - ▶ It is sufficient to check just one representative interleaving

Simple POR Example

```
int x, y;
```

Thread 1:

```
int a;
```

```
1) a = 5;
```

```
2) a++;
```

```
3) x = 1;
```

```
4) y = 2;
```

```
5) x++;
```

```
6) y++;
```

Thread 2:

```
int b;
```

```
5) b = 10;
```

```
6) b--;
```

```
7) y = 3;
```

```
8) x = 2;
```

```
9) y++;
```

```
10) x++;
```

Static Analysis in JPF

- ▶ Using static analysis techniques to reduce the state space
 - ▶ Slicing
 - ▶ Partial evaluation

Static Analysis in JPF

▶ Slicing

- ▶ Remove program parts with no effect on the slicing criterion
 - ▶ A slicing criterion could be a program point
- ▶ Program slices are computed using dependency analysis

▶ Partial evaluation

- ▶ Propagate constant values and simplify expressions

User-Provided Restrictions

- ▶ To improve scalability, users can restrict domains of variables, sizes of arrays,...
- ▶ Restrictions under-approximate program behaviors
 - ▶ May result in missed errors
 - ▶ Still useful in finding bugs

Next Time

- ▶ Checking concurrent programs using symbolic techniques