# Finite State Verification

Dr. Paul West

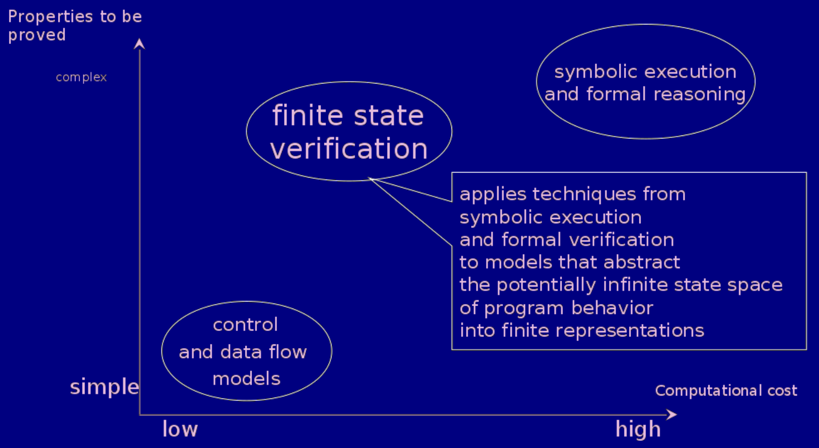
Department of Computer Science
College of Charleston

January 30, 2014

# Limits and Trade-offs

- Most important properties of program execution are undecidable in general
- Finite state verification can automatically prove some significant properties of a finite model of the infinite execution space
  - balance trade-offs among
    - generality of properties to be checked
    - class of programs or models that can be checked
    - computational effort in checking
    - human effort in producing models and specifying properties

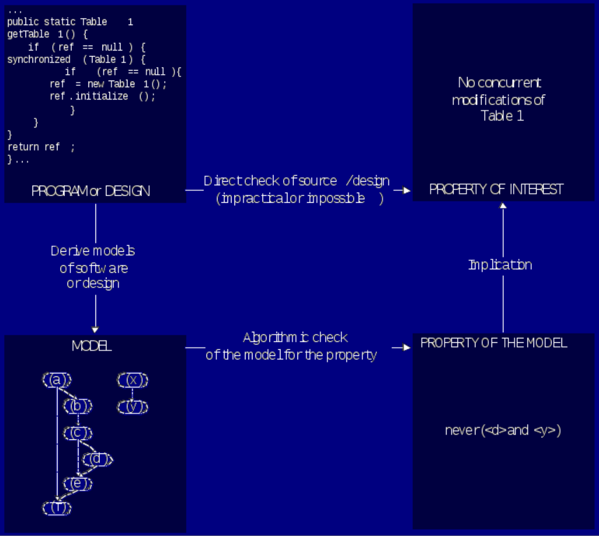
# Resources and Results



# Cost Trade-offs

- Human effort and skill are required
  - to prepare a finite state model
  - to prepare a suitable specification for automated analysis
- Iterative process:
  - prepare a model and specify properties
  - attempt verification
  - receive reports of impossible or unimportant faults
  - refine the specification or the model
- Automated step
  - computationally costly
    - computational cost impacts the cost of preparing model and specification, which must be tuned to make verification feasible
  - manually refining model and specification less expensive with near-interactive analysis tools

# Analysis of Models



# Application for finite State Verification

- Concurrent (multi-threaded, distributed, ...)
  - Difficult to test thoroughly (apparent non-determinism based on scheduler); sensitive to differences between development environment and field environment
  - First and most well-developed application of FSV
- Data models
  - Difficult to identify “corner cases” and interactions among constraints, or to thoroughly test them
- Security
  - Some threats depend on unusual (and untested) use

# Defining the global state space - Concurrent system example

- Deriving a good finite state model is hard
- Example: finite state machine model of a program with multiple threads of control
  - Simplifying assumptions
    - we can determine in advance the number of threads
    - we can obtain a finite state machine model of each thread
    - we can identify the points at which processes can interact
  - State of the whole system model
    = tuple of states of individual process models
  - Transition = transition of one or more of the individual processes, acting individually or in concert

# State space exploration - Concurrent system example

- Specification: an on-line purchasing system
  - In-memory data structure initialized by reading configuration tables at system start-up
  - Initialization of the data structure must appear atomic
  - The system must be reinitialized on occasion
  - The structure is kept in memory
- Implementation (with bugs):
  - No monitor (Java synchronized): too expensive*
  - Double-checked locking idiom* for a fast system

*Bad decision, broken idiom ... but extremely hard to find the bug through testing.

# Concurrent system example - implementation

```
class Table1 {
    private static Table1 ref = null;
    private boolean needsInit = true;
    private ElementClass [ ] theValues;
    private Table1() { }

public static Table1 getTable1() {
    if (ref == null){
        synchedInitialize();
    }
    return ref;
}

private static synchronized
    void synchedInitialize() {
    if (ref == null) {
        ref = new Table1();
        ref.initialize();
    }
}
```

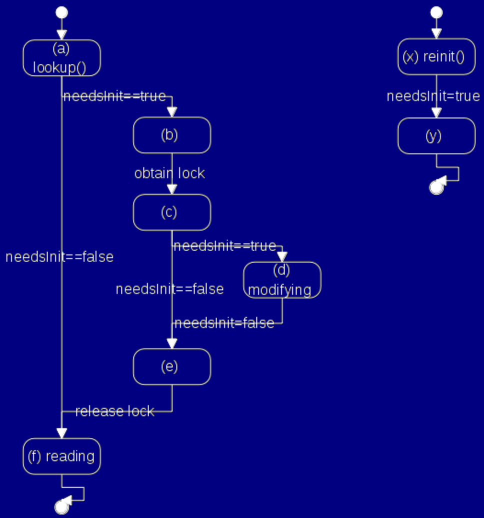
```
public void reinit(){
    needsInit = true;
}

private synchronized void initialize() {
    . . .
    needsInit = false;
}

public int lookup(int i) {
   if (needsInit) {
      synchronized(this) {
       if (needsInit) {
           this.initialize();
       }
      }
   }
   return theValues[i].getX()
     + theValues[i].getY();
}
. . .
}
```

# Analysis

- Start from models of individual threads
- Systematically trace all the possible interleavings of threads
- Like hand-executing all possible sequences of execution, but automated

... begin by constructing a finite state machine model of each individual thread ...

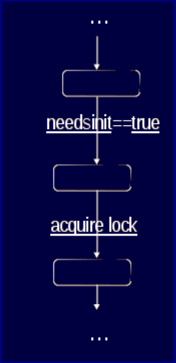
# Finite State Machine Model for each Thread



# Analysis

- Java threading rules:
  - when one thread has obtained a monitor lock
  - the other thread cannot obtain the same lock
- Locking
  - prevents threads from concurrently calling initialize
  - Does not prevent possible race condition between threads executing the lookup method
- Tracing possible executions by hand is completely impractical

# Express the model in Promela




```
proctype Lookup(int id ) {
  if :: (needsInit) ->
     atomic { ! locked  -> locked = true; };
     if  :: (needsInit) ->
        assert (! modifying);
        modifying = true;
        /*  Initialization happens here */
        modifying = false ;
        needsInit = false;
      :: (! needsInit) ->
        skip;
      fi;
      locked = false ;
   fi;
   assert  (! modifying);}
```

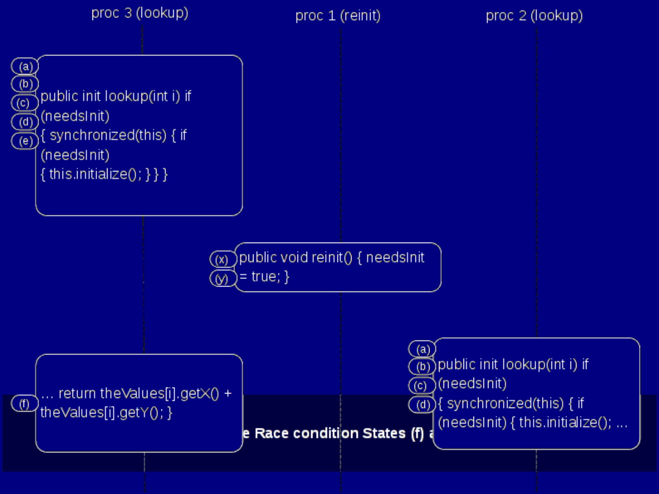
# Run Spin; Inspect Output

Spin

- Depth-first search of possible executions of the model
- Explores 10 states and 51 state transitions in 0.16 seconds
- Finds a sequence of 17 transitions from the initial state of the model to a state in which one of the assertions in the model evaluates to false

```
Depth=10 States=51 Transitions=92 Memory=2.302
pan: assertion violated  !(modifying) (at depth 17)
pan: wrote pan_in.trail
(Spin Version 4.2.5 -- 2 April 2005)
0.16 real        0.00 user        0.03 sys
```

# Interpret the Trace




Read/write Race condition States (f) and (d)

# The Promela (Spin) Modeling Language

- A set of processes described by process types
  - Can model threads (Java), processes (Unix), devices, resources, etc.
- C-like syntax
  - expression -> statements
  - atomic statements
    - treat as a single, atomic step (without interleaving)
  - do ... od, if ... fi
    - with multiple :: alternatives, chosen non-deterministically

# Safety and Liveness Properties

- Safety: bad things should not happen
  - e.g., two processes should not modify a variable at the same time.
  - Easy to specify in Promela with assert( ... )
- Liveness: good things should eventually happen
  - e.g., if I push the button, eventually the elevator should arrive
  - Can be specified in temporal logic; more expensive to check
  - Fairness (I should get lucky now and then) is an important and common class of liveness properties

# The state explosion problem

## Dining philosophers - looking for deadlock with SPIN

```
5 phils+forks     145 states
                  deadlock found
10 phils+forks      18,313 states
                  error trace too long to be useful
15 phils+forks      148,897 states
                  error trace too long to be useful
```