Verifying Safety Properties of Concurrent Java Programs Using 3-Valued Logic

Presenter: Javeria Iqbal
Tutor: Jörg Herter
Introduction

• Goal: Verification of concurrent Java programs
• Support the following:

  Java concurrency-model

  Dynamic allocation/de-allocation of objects

  Dynamic allocation/de-allocation of threads

  Verification/Detection of Safety Properties
Verification of Properties

- Deadlock avoidance
- No inconsistent updates
- Concurrent manipulation of linked list based ADT
- No run time errors due to illegal thread interactions
Why Verify Java Programs?

- Concurrency is hard to debug
  - Dead Locks
  - Interference
  - Failures hard to reproduce
  - Dependence on thread scheduling

- Concurrent Programming in Java
Java Concurrency

Threads and locks are just dynamically allocated objects

Synchronized implements mutual exclusion

Wait, Notify and NotifyAll coordinate activities across threads
Java Concurrency Challenges

Dynamic Allocation

Data and control are strongly related

```java
Thread t1 = new Thread();
Thread t2 = new Thread();
...
if (...) t = t1 else t = t2;
t.start();
```
Example: Mutual Exclusion

```java
l_0: while (true) {
  l_1: synchronized(sharedLock) {
    l_C: // critical actions
  }
  l_2:
}
```

Two threads: \((pc_1, pc_2, lockAcquired_1, lockAcquired_2)\)

- Allocate new lock?
- Allocate new thread?
State of the Art & Motivation
Model Checking Approach

• Explore the space of possible program configurations
• Find configurations that violate the desired safety property

How do you guarantee finiteness?
Existing Approaches for Dynamic Allocation

• dSPIN, Java pathfinder, Bandera, …
• All put an a priori bound on number of allocated objects
• Abstraction applied when model is extracted
Existing Approaches Failure

Example: http server

- Creates a thread for each http request
- Threads using exclusive shared resource
Challenges

Guaranteed correctness vs. assumed correctness

Correctly track thread states

Show Mutual Exclusion
Our Approach

Abstract Configurations represent Multiple Configurations

Compute a finite set of abstract configurations

Check property for computed abstract configuration
Agenda for Proceeding

• Concrete Program Model
• Safety Properties
• Abstract Program Model
• Prototype implementation (3VMC)
• Strengths and Weakness
• Summary
Concrete Program Model

- Represent program configurations
- Extract properties of configurations
- Manipulate configurations
Configuration’ Representation
Configurations

- A program configuration encodes:
  - Global Store
  - Program location of every thread
  - Status of locks and threads

- A First-order logical structures used to represent program configurations
Predicates for Java Semantics

- `is_thread(t)`: t is a thread
- `at[lab](t)`: lab ∈ Labels
- `rval[fld](o1,o2)`: fld ∈ Fields
- `held_by(l,t)`
- `blocked(t,l)`
- `waiting(t,l)`
- `idlt(l1,l2)`
Configurations
Extraction of Properties:
For Configurations
Extracting Properties of Configurations

- Program control-flow is not separately represented
- Extraction of properties for configurations

\[ \exists \text{Is\_thread}(t) \land \text{held\_by}(l,t) \]

\[ \forall t1,t2: (t1 \neq t2) \rightarrow (\text{at}[l\text{crit}](t1) \land \text{at}(l\text{crit}(t2)) \]

- Program location for each thread is encoded inside the configuration
Structural Operational Semantics: Actions
An action consists of:

- Precondition Formula
- Update Formula

Precondition formula may use a free variable $t_s$ for “currently scheduled” thread.
### Action: lock(v)

<table>
<thead>
<tr>
<th>lock(v)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>precondition</strong></td>
<td>$\neg \exists t \neq t_s: \text{rval}[v](t_s, l) \land \text{held_by}(l, t)$</td>
</tr>
</tbody>
</table>
| **predicate update** | \( \text{held\_by}'(l_1, t_1) = \text{held\_by}(l_1, t_1) \lor (l_1 = l \land t_1 = t_s) \)  
  
|  | \( \text{blocked}'(t_1, l_1) = \text{blocked}(t_1, l_1) \land ((l_1 \neq l) \lor (t_1 \neq t_s)) \) |
**Action: unlock(v)**

<table>
<thead>
<tr>
<th>unlock(v)</th>
<th>precondition</th>
<th>predicate update</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rval<a href="t_s,l">v</a></td>
<td>held_by’(l_1,t_1) = held_by(l_1,t_1) ∧ (t_1 ≠ t_s ∨ l_1 ≠ l)</td>
</tr>
</tbody>
</table>
### Action: wait(v)

<table>
<thead>
<tr>
<th>wait(v)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>precondition</strong></td>
<td>rval<a href="t_s,l">v</a></td>
</tr>
<tr>
<td><strong>predicate update</strong></td>
<td>held_by’(l_1,t_1) = held_by(l_1,t_1) ∧ (t_1 ≠ t_s ∨ l_1 ≠ l)</td>
</tr>
<tr>
<td></td>
<td>waiting’(t_1,l_1) = waiting(t_1,l_1) ∨ ((t_1 = t_s ∧ l_1 = l))</td>
</tr>
</tbody>
</table>
### Action: notify(v)

<table>
<thead>
<tr>
<th>notify(v)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>precondition</strong></td>
<td>rval<a href="t_s,l">v</a> ∧ ¬∃ tw : waiting(tw, l)</td>
</tr>
</tbody>
</table>
| **predicate update** | waiting’(t_l1) = waiting(t_l1) ∧ (t_l1 ≠ tw ∨ l1 ≠ l)  
blocked’(t_l1) = blocked(t_l1) ∨ ((l1 ≠ l) ∧ (t_l1 ≠ t_s)) |
Abstract Program Model

- Conservative representation of the concrete model
- Use 3-valued logical structures to conservatively represent multiple 2-valued structures
- Conservatively apply actions on abstract configurations
Safety Properties

- Read Write Interference Formula
- Write Write Interference Formula
- Resource Ordering Criteria
- Nested Monitors Formula
- Missing Ownership Formula
- Total Deadlock Formula
- Shared ADT
- Thread Interactions
Read Write Interference

\[
\neg \exists \ tr, \ tw, \ o : \text{is\_thread}(tr) \\
\wedge \text{is\_thread}(tw) \\
\wedge (tr \neq tw) \\
\wedge \text{at}[lr] (tr) \wedge \text{at}[lw] (tw) \\
\wedge \text{rvalue}[xw] (tw,o) \wedge \text{rvalue}[xr] (tr,o)
\]
\[\exists \, tw1, tw2, o :\]
\[\text{is\_thread}(tw1) \wedge \text{is\_thread}(tw2) \wedge (tw1 \neq tw2) \wedge \text{at}[lw1] (t1) \wedge \text{at}[lw2] (t2) \wedge \text{rvalue}[xw1] (tw1,o) \wedge \text{rvalue}(xw2) (tw2,o)\]
∃ t, l1, l2 : is_thread(t)
  ∧ blocked (t , l1)
  ∧ held_by (l2 , t)
  ∧ ¬idlt(l2 , l1)
Total Deadlock

∀ t: is_thread(t) → ∃ l: blocked (t, l)
Abstract Configurations

• First-order 3-valued logical structures are used to represent abstract program configurations

• Optimization of Old State Space Algorithm
  – Membership
  – Rewrite

• 3-valued logic
  – 1 = true
  – 0 = false
  – 1/2 = unknown
  – A join semi-lattice, 0 or 1 = $\frac{1}{2}$
Initialize($C_0$) {
  for each $C$ $C_0$
    push(stack,$C$)
}

explore() {
  while stack is not empty {
    $C$ = pop(stack)
    if not member($C$,stateSpace) {
      verify($C$)
      stateSpace' = stateSpace $\{C\}$
      for each action ac
        for each $C'$ such that $C \Rightarrow_{ac} C'$
          push(stack,$C'$)
    }
  }
}
Canonic Abstraction

- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values
- Converts a configuration of arbitrary size into a 3-valued abstract configuration of bounded size
- Use same state-space exploration algorithm to explore the abstract state space
Abstract Configuration
Example 2

- Queue Class
- Producer Class
- Consumer Class
- Approver Class
- Main Class
- QueueItem Class
Concrete Configuration

- **<u0> r_by[head]**
  - **rval[next]**
- **<u1> r_by[next]**
  - **rval[next]**
- **<u2> r_by[next]**
  - **rval[next]**
- **<u3> r_by[next]**
  - **rval[next]**
- **<u4> r_by[next]**
  - **r_by[tail]**

- **<q> r_by[this]**
  - **rval[head]**
  - **rval[q]**
  - **held_by**
  - **rval[q]**

- **<prd> is_thread at[lp_6]**
  - **rval[q]**
  - **rval[q]**
  - **rval[q]**

- **<m1> r_by[x_i]**
  - **rval[x_i]**

- **<a1> is_thread at[a_1]**
  - **rval[q]**

- **<a2> is_thread at[a_1]**
  - **rval[q]**
  - **blocked**

- **<cns> is_thread at[lt_1]**

- **<a3> is_thread at[a_1]**
  - **rval[q]**
Instrumentation Predicate

- Use `is_acquired(l)` for some thread
- Use `is_blocked(t)` for some lock
- Use `wait_for(t1,t2)` for cyclic dependency
- Use `slock(t)` to track resource ordering
- Thread State Errors
  - Use `ts_created(t)`, `ts_running(t)`, `ts_blocked(t)`, `ts_waiting(t)`, `ts_dead(t)`
Example 3

• List.h
  Declaration of Linked List data type
• Print.c
  Prints all the elements of the list pointed to by Temp Parameter
Abstract Configuration
Instrumentation Predicate
Shared ADT & Unlimited Threads

- Shared Abstract Data Types
- Thread State Errors
- Unbounded Number of Threads
Thread State Errors: Producer Consumer Deadlock
Mutual Exclusion: Unbounded Number of Threads

Initial configuration
Thread Enters Critical Section

Diagram:
- rval[q] connected to is_thread
- held_by connected to is_thread
- rval[q] connected to is_thread
- rval[q] connected to is_thread
Other Threads Blocked

Diagram showing relationships between is_thread at[l_C], is_thread at[l_0], and is_thread at[l_1], with connections labeled "rval[q]" and "held_by".
Prototype Implementation

- 3VMC based on 3-Valued Logic Engine TVLA
- It is used to verify several small example programs
- Running times were measured using JVM1.2.2 for Windows NT with 600MHZ memory on PIII
### Computational Analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>Configurations</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>Swap Ordering</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Stack</td>
<td>184</td>
<td>304</td>
</tr>
<tr>
<td>Synchronized Stack</td>
<td>104</td>
<td>330</td>
</tr>
<tr>
<td>Mutex</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>NestedMontior</td>
<td>42</td>
<td>7</td>
</tr>
<tr>
<td>ProducerConsumer</td>
<td>416</td>
<td>68</td>
</tr>
<tr>
<td>Synchronized (PC)</td>
<td>195</td>
<td>48</td>
</tr>
<tr>
<td>DinningPhilosopher</td>
<td>514</td>
<td>23</td>
</tr>
</tbody>
</table>

Maximum Lines: 64
Strength & Weakness

- Infinite objects allowed
- Infinite threads allowed
- Deadlocks avoidance
- Instrumentation Predicates

- No Inter-procedural Analysis
- No Recursive Procedures
- No Optimization
Further Work

- Partial-order reduction
- Symbolic representations

“My ☺pinion”
Putting All Together

State Space Exploration

Shape Analysis
• Explore state space configurations and convert it to abstract model
• 3 Valued first order logic is applied
• Infinite number of thread and objects are entertained
• Various safety properties are verified
• Instrumentation predicates enhance capabilities of framework
Thank You