Testing
Concurrent Java Programs
Using Randomized Scheduling

Scott D. Stoller
Computer Science Department
State University of New York at Stony Brook
http://www.cs.sunysb.edu/~stoller/
Goal

Pillars of Run-Time Verification

**monitoring**: identify interesting behavior in an execution
**control**: try to provoke interesting behavior

Java Language Spec. puts few constraints on thread scheduler. Scheduling may depend on load and run-time system.

During run-time verif., control the scheduling, at least partially, to help check robustness and portability.

**Approach**: Insert calls to *scheduling function* at selected points in bytecode. Sched fn causes context switches.
Outline

• How does sched fn choose which thread to switch to?

• Where to insert calls to sched fn?

**probabilistic completeness:** every error reachable according to the Java Language Spec. is reachable in the transformed program, independent of the underlying thread scheduler.

• Experiments

• Related work
Pseudo-Random Sched Fn

Semaphore Approach
Associate a semaphore sem(θ) with each thread θ. Pseudo-randomly choose a runnable thread θ (possibly self), call sem(θ).up(), and then call sem(currentThread).down().
+ probabilistic completeness
+ the probability distribution can easily be controlled

Loop Approach
while (nextFloat() < contextSwitchProb) contextSwitch();
where contextSwitch() calls yield or sleep.
+ easy to implement
+ probabilistic completeness if underlying sched is fair
  – less control over probability distribution
  ○ without loop, not probabilistically complete (e.g., FIFO sched)
Heuristics in Sched Fn

Start with a coverage metric or abstraction.

Sched fn remembers what has been explored (covered) in current and previous executions, always tries to explore something new, and chooses randomly when it can’t.

Sample coverage metrics [Edelstein et al. 2002]:
• set of methods in which context switches occurred
• set of pairs of methods \( \langle m_1, m_2 \rangle \) such that there was a context switch from a thread running \( m_1 \) to a thread running \( m_2 \).
Where to Insert Calls to Sched Fn?

**Easy Answer:** At every operation on potentially shared storage, namely:
- access to instance field of object (including arrays)
- access to static field
- synchronization operation
- class initialization (implicit shared state and sync.)

+ probabilistic completeness
- frequent calls to sched fn cause run-time overhead and complicate counterexamples
Synchronization Primitives in Java Bytecode

Each object has a **recursive lock** and a **condition variable**.

- **acquire o**    acquire o’s lock
- **release o**    release o’s lock
- **o.wait()**     release o’s lock; wait to be notified or interrupted; acquire o’s lock
- **o.notify()**   notify a thread waiting on o, if any
- **o.notifyAll()** notify all threads waiting on o
- **t.interrupt()** interrupt thread t

`wait()`, `notify()`, and `notifyAll()` require holding o’s lock; otherwise, they merely throw an exception.

Assume the program does not use real-time operations, e.g., `o.wait(timeout)`, `Thread.sleep(duration)`
Fewer Calls to Sched Fn: Unshared Locations

Identify **unshared** static and heap locations. **unshar**: set of classes (and pkgs) with all instances unshared
Do not insert calls to sched fn before operations on **unshar**.

How to determine **unshar**?
**Statically**: escape analysis
**Dynamically**: monitoring, iterative refinement
Insert calls to checkIfShared at operations on **unshar**.

```
\begin{tikzpicture}
  \node (guess) {guess \texttt{unshar}};
  \node (revise) [right of=guess, xshift=2cm] {revise \texttt{unshar}};
  \node (insert) [below of=guess, yshift=-1cm] {insert calls to sched fn & checkIfShared};
  \node (run) [right of=insert, xshift=2cm] {run \texttt{txfm (P,unshar)}};
  \node (diamond) [right of=run, xshift=2cm] {found bug or reached full coverage};

  \draw [->] (guess) -- node {\texttt{unshar}} (revise);
  \draw [->] (revise) -- node {\texttt{unshar}} (run);
  \draw [->] (run) -- node [near end, anchor=south] {otherwise} (diamond);
  \draw [->] (insert) -- node {\texttt{txfm (P,unshar)}} (run);
  \draw [->] (diamond) -- node {\texttt{txfm (P,unshar)}} (run);
  \draw [->] (insert) -- (run);
  \draw [->] (guess) -- (insert);
\end{tikzpicture}
```
Prob. Completeness for Finding Errors

Errors: We consider assertion violations and deadlocks.

\textbf{Exec}(P, sched): set of executions of program \textit{P} with scheduler \textit{sched}

\textbf{JLSched}: non-deterministic scheduler in Java Language Spec.

\textbf{Theorem}: Suppose locations in \textit{unshar} are unshared. For every program \textit{P} and scheduler \textit{sched}, every error that occurs in \textbf{Exec}(\textit{P}, \textit{JLSched}) also occurs in \textbf{Exec}(\textbf{txfm}(\textit{P}, \textit{unshar}), \textit{sched}), i.e., there exists a sequence of choices by the scheduling function that leads to it.

\textbf{Caveat}: We do not control non-determinism in \textit{notify}().

\textbf{Note}: \textit{sched} may cause add’l context switches at any point.
Prob. Completeness for Mis-Classifications

If $unshar$ is incorrect, can we say anything about executions of $\text{txfm}(P, unshar)$? Yes!

**Theorem:** $\text{Exec}(P, \text{JLSched})$ contains an execution in which a location in $unshar$ is shared iff $\text{Exec}(\text{txfm}(P, unshar), \text{sched})$ does.

$\Rightarrow$: Consider the first shared access to a location in $unshar$ in an execution in $\text{Exec}(P, \text{JLSched})$. Show that it can also occur in an execution of $\text{Exec}(\text{txfm}(P, unshar), \text{sched})$.

$\Leftarrow$: straightforward

**Similar results:** [Holzmann & Peled 1994], [Stoller 2000].
Fewer Calls to Sched Fn: Protected Locations

A location $o$ is **protected** if, after **initialization** of $o$,
- all accesses to $o$ are reads (“$o$ is read-only”)
- some lock $\ell$ is hold at every access to $o$ ("$\ell$ protects $o$")

**Initialization** of a **static** field $C.f$ ends when the class initializer for $C$ terminates.

**Initialization** of a **heap** location $o.f$ ends before $o$ escapes from the thread that allocated it.

*prot*: set of classes (and pkgs) with all instances protected

Do not insert calls to sched fn at accesses to *prot*. Insert calls to sched fn at acquire, wait, and notify. Do not insert calls to sched fn at release and notifyAll.
Prob. Completeness with Protected Locations

Prob. Completeness for Errors:
Suppose unshar and prot are correct. Every error that occurs in \( \text{Exec}(P, \text{JLSched}) \) also occurs in \( \text{Exec}(\text{txfm}(P, \text{unshar}, \text{prot}), \text{sched}) \), i.e., there exists a sequence of choices by the scheduling function that leads to it.

Similar results: [Lipton 1975], [Cohen and Lamport 1998], [Stoller 2000].

Prob. Completeness for Mis-Classifications:
Exec\((P, \text{JLSched})\) contains an execution in which a location in unshar or prot is mis-classified iffExec\((\text{txfm}(P, \text{unshar}, \text{prot}), \text{sched})\) does.

Similar results: [Stoller 2000].
Prob. Completeness for Errors: Proof Sketch

Show: Context switch at access to a protected location \( x \) is unnecessary. Context switch at acquire of protecting lock suffices.

Consider an execution with a context switch at an access to \( x \). Repeatedly swap adjacent transitions to eliminate that context switch, while preserving errors.

Suppose \( t_3 \) accesses \( x \) and is not an acquire. In states \( s_3 \) and \( s_2 \), \( \theta_{\text{black}} \) owns the lock \( \text{lk} \) that protects \( x \). In \( s_2 \), \( \theta_{\text{red}} \) does not own \( \text{lk} \) or access \( x \). So \( t_2 \) and \( t_3 \) commute.
How to Identify Protected Locations?

**Statically:** race-free type systems [Flanagan & Freund 2000], [Boyapati & Rinard 2001]. Types are augmented with parameters indicating how objects are protected.

**Example:**
```java
class C<thisOwner> {
    Data<this> a = new Data<this>;
    Data<thisOwner> b;
    Data<readOnly> c;
}
```

**Dynamically:** monitoring, iterative refinement. Start with a guess. Use lockset algorithm [Savage et al. 1997] to check whether objects in prot are protected.
Lockset Algorithm

\( o.\text{lockSet} \): set of locks that protected \( o \) so far after init.
\( o.\text{readOnly} \): whether all accesses to \( o \) after init were reads.

**At end of initialization of \( o \):**
\( o.\text{readOnly} = \text{true} \)
\( o.\text{lockSet} = \{ \text{all locks} \} \)

**At each subsequent access to \( o \):**
\( o.\text{readOnly} = o.\text{readOnly} \land (\text{current access is a read}) \)
\( o.\text{lockSet} = o.\text{lockSet} \cap \text{heldLocks(currentThread)} \)

\( o \) is **protected** iff \( o.\text{readOnly} \lor o.\text{lockSet} \neq \emptyset \)
Experiments

Current implementation supports dynamic checking of unshar, not prot.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean [Brat et al. '00]</td>
<td>57 LOC</td>
<td>deadlock</td>
</tr>
<tr>
<td>Fund Managers [JDC '00]</td>
<td>123 LOC</td>
<td>assertion violation</td>
</tr>
<tr>
<td>Xtango Animation Library</td>
<td>1.3 KLOC</td>
<td>minor oddities</td>
</tr>
<tr>
<td>ArgoUML design envir.</td>
<td>5+ MB of .class</td>
<td>exception in AWT event handler thread</td>
</tr>
</tbody>
</table>

Future experiments:

- Evaluate reduction in # of context switches in counterexamples
- Dynamic checking: compare cost of checkIfShared and lock-set alg with cost of sched fn. (Static checking is a pure win.)
- Experiment with heuristics in sched fn.
Related Work

Traditional Model Checkers
Special run-time system captures and stores entire state. Examples: SMV, Spin, Murphi, Java PathFinder
Applying them to large Java or C programs typically requires significant effort for translation or abstraction, even with tool support.

State-less Model Checkers and Testers
Use standard run-time system.
Insert code to (partially) control non-determinism.
Related Work: State-less Checkers and Testers

VeriSoft [Godefroid 1997]
Controls non-determinism in **inter-process scheduling**. 
**Systematic** exploration of possible transition sequences. 
Not targeted at control of thread scheduling.

JavaChecker [Stoller 2000]
Controls non-determinism in **thread scheduling**. 
**Systematic** search, as in VeriSoft. 
rstest is “JavaChecker lite”.

ConTest [Edelstein et al. 2002]
Controls non-determinism in **thread scheduling**. 
**Randomization** and **heuristics**, not systematic search. 
Calls sched fn at **all** operations on shared objects.