JAVA VIRTUAL MACHINE WITH ROLLBACK PROCEDURE ALLOWING SYSTEMATIC AND EXHAUSTIVE TESTING OF MULTI-THREADED JAVA PROGRAMS

Pascal Eugster <pe@student.ethz.ch>

10th April 2003

EXAMPLE 1

- rollback procedure

- sample sequence of setMilestone, rollback and removeMilestone

- stm sequ A -
  1. setMilestone

- stm sequ B -
  2. setMilestone

- stm sequ C -
  3. rollback

- stm sequ D -
  4. rollback

- removeMilestone

- stm sequ E -
  5. rollback

- stm sequ F -
  6. rollback
EXAMPLE 2

- three threads each containing a single synchronized region
- each thread consists of exactly one atomic block

T1:
1: synchronized (x) {}

T2:
2: synchronized (x) {}

T3:
3: synchronized (x) {}
EXAMPLE 3

- Two threads, both containing nested synchronized regions
- How does the depth-first-search handle situations, where a lock cannot be get

```java
T0:
1: t1 = new LockAB (A, B);
2: t1.start();
3: synchronized (B) {
4:    synchronized (A) {
5:    }
6: }

T1:
7: synchronized (A) {
8:    synchronized (B) {
9: }
10: }
```

- Blocked paths are aborted
- Lock cycle deadlock detection algorithm started
EXAMPLE 4

- two threads with a condition deadlock occurring when notify is performed prior to wait
  \(\rightarrow\) schedule \((1, 5, 6, 7, 2)\)

\[
\begin{align*}
T0: \\
1: & \text{t1.start();} \\
2: & \text{synchronized(a) {} } \\
3: & \text{a.wait();} \\
4: & \}
\]

\[
\begin{align*}
T1: \\
5: & \text{synchronized(a) {} } \\
6: & \text{a.notify();} \\
7: & \}
\]

- to ensure behaviour-complete testing, a milestone has to be created after each thread creation
  \(\rightarrow\) otherwise, only the schedule \((1, 2, 5, 6, 7, 3, 4)\) is executed, where wait and notify are executed in the right order, i.e. deadlock does not occur

EXAMPLE 4, SCHEDULE TREE

- compare Petri net model and its (reduced) reachability graph
EVENT HANDLER

- remark: stop, resume, suspend - deprecated

ALGM1
HANDLER TRIGGERED ON A LOCK EXIT

handleLockExit() {

enabled_set = collectEnabledThreads();

enabled_set = enabled_set \ cur_thread;

if (number_enabled_threads >= 1)
    /* avoids empty milestones */
    setMilestone( enabled_set );
}

- when a lock exit occurs, ExitBlock is notified by the JVM's lock manager

- set of enabled threads
  - -> runnable threads not scheduled yet from the current milestone
  - -> since the current thread continues running, it is not member of the enabled set
  - -> used by the rollback

- on rollback, the state of the whole VM is restored and a new branch/schedule is created scheduling one thread from the enabled set
**ALGM2**

**HANDLER TRIGGERED ON DEATH OF CURRENT THREAD**

milestone = getCurrentMilestone();
enabled_set = enable_set \ {cur_thread};
next_thread = reschedule();

if (next_thread == NULL )
  /** rollback continues down the stack, selects a next thread there. Returns NULL, if stack becomes empty. */
  next_thread = rollback();

if (next_thread == NULL )
  /** empty stack of milestones */
  terminate();
else
  switchThread (next_thread);

- an enabled thread to be scheduled next must be selected

- if there is no enabled thread left, the end of the current schedule is reached
  -> all threads have terminated
  -> rollback to explore next schedule in a new branch

**ALGM3**

**IMPLEMENTATION OF ROLLBACK**

thread rollback() {
    rollbackVM();

    /** elect thread */
    milestone = getCurrentMilestone();
    next_thread = first( milestone->enabled_set );
    milestone->enabled_set = milestone->enabled_set \ next_thread;

    if (next_thread == NULL )
    {
      /** no further branch from this milestone. So remove this milestone and rollback again, continuous down the stack. */
      removeMilestone( milestone );
      next_thread = rollback();
    }

    return next_thread;
}
**ALGM4**

**HANDLER TRIGGERED WHEN A LOCK COULD NOT BE OBTAINED**

next_thread = rollback();  
switchThread (next_thread);

- lock manager notifies ExitBlock, when a lock could not be obtained by the current thread  
  -> *current thread is aborted by ExitBlock*  
  -> *rollback*

**ALGM5**

**HANDLER TRIGGERED FOR JOIN AND WAIT**

next_thread = reschedule();

if (next_thread == NULL )  
{
    /** condition deadlock detected !!!*/
    next_thread = rollback();
}

if (next_thread == NULL )  
    terminate_depth_search();
else  
    switchThread(next_thread);

- current thread falls into sleep, until  
  -> occurrence of notify  
  -> joined thread terminates

- find a new thread that is still enabled
**ALGM 6**

**HANDLER TRIGGERED ON INVOCATION OF NOTIFY**

```c
/** retrieve the wait set of the object on which notify was performed */
wait_set = getWaitSet( object );

/** notify first thread which removes notified thread from wait set */
notifyNext (wait_set);

/** create a milestone if there are still threads waiting on this object that could also be notified */
if (count (wait_set) > 0) {
    handleLockExit() /* see Algorithm 1 */
    milestone = getCurrentMilestone();

    /** threads from the wait set needs to be notified later. */
    milestone->notify_set = copy(wait_set);

    /** remember enabled set */
    milestone->saved_enabled_set =
        copy(milestone->enabled_set);
}
```

**ALGM 7**

**IMPLEMENTATION OF ROLLBACK WITH ADDITIONS FOR NOTIFY HANDLING.**

```c
thread rollback() {
    milestone = getCurrentMilestone();
    enabled_set = getEnabledSet(milestone);
    next_thread = pop( enabled_set );

    if (isNotifyMilestone( milestone )) {
        milestone->enabled_set =
            copy(milestone->saved_enabled_set);
        thread = first(milestone->notify_set);
        milestone->notify_set =
            milestone->notify_set \ {thread};
        if (thread)
            notify( thread );
        else
            next_thread = cur_thread;
    else
        next_thread = NULL;

    if (next_thread == NULL) {
        removeMilestone( milestone );
        next_thread = rollback();
    }

    return next_thread;
}
```