

Incremental Checkpointing with Application to Distributed Discrete Event Simulation

Thomas Huining Feng and Edward A. Lee
`{tfeng, eal}@eecs.berkeley.edu`

Center for Hybrid and Embedded Software Systems
EECS, UC Berkeley

Winter Simulation Conference 2006
Monterey, CA
December 3-6, 2006

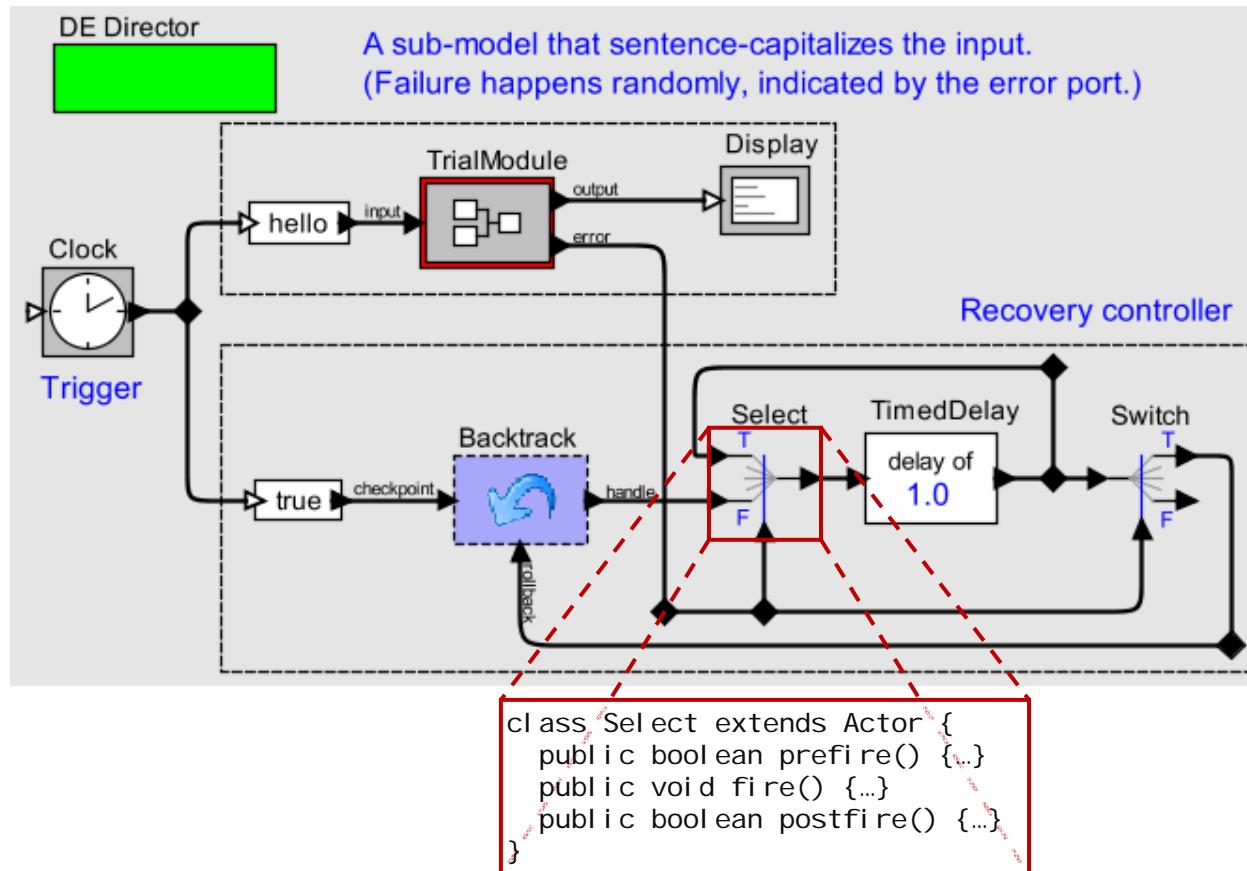


Outline

- Background: heterogeneous model simulation with Ptolemy II
- Problem: state recovery for backtracking
- Approach: automatic state recovery with incremental checkpointing
- Application: optimistic Time Warp simulation
- Conclusion



Ptolemy II Background

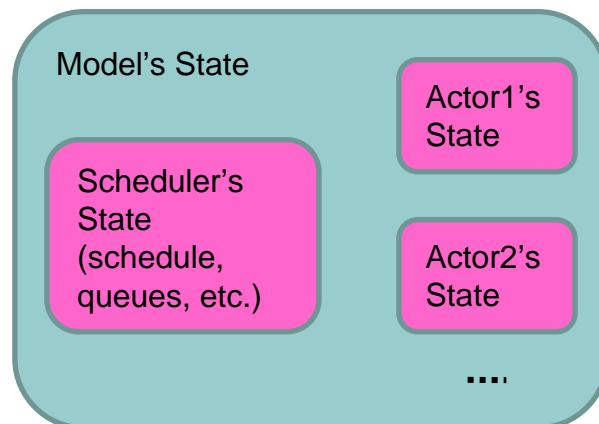


- Hierarchical composition of actors.
- Ports connected with relations.
- The scheduler determines the semantics.
- Users may define actors in Java.



Model State

- Situations requiring state recovery:
e.g., optimistic Time Warp simulation (Jefferson 1985).
- A model's run-time state:

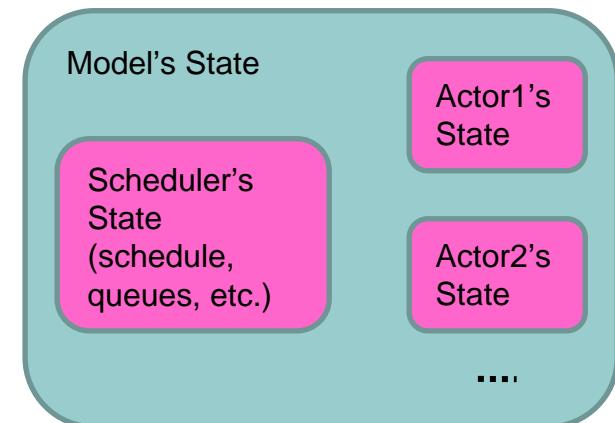


```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        sum = sum + value;  
        output.send(0, sum);  
    }  
}
```



Problems of State Recovery (1)

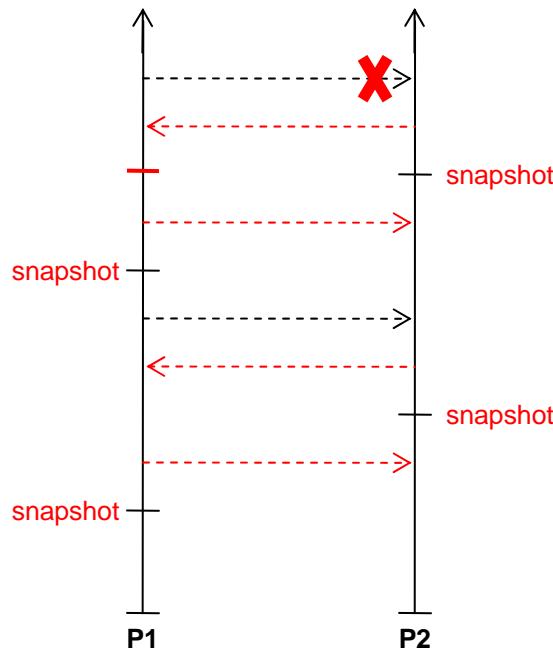
- We could recover the scheduler's state with a careful design of the simulation environment.
- But how to recover actors' states?
 - Provide programmers with a library of state saving/recovery functions.
Complicated user interaction.
 - Make assumptions on the states, and provide automatic recovery.
Not flexible.





Problems of State Recovery (2)

- The extra cost usually prevents us from creating as many checkpoints as we want.
- Domino effect:





Our Approach to State Recovery

- Correctness can be proved easily.
- Incremental.
- Small constant cost for checkpoint creation.
 - Allow to create checkpoint whenever it may be needed.
- The program is slowed down gracefully.
 - Real-time property can be proved.



Java Analysis and Transformation

1. Use an analyzer to identify the states in the Java code.

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        sum = sum + value;  
        output.send(0, sum);  
    }  
}
```

state → sum
update → sum = sum + value

2. Use a transformer to modify the program.

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        $ASSIGN(sum(sum + value));  
        output.send(0, sum);  
    }  
}
```



Simple Assignment Transformation

- For each Java class:

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        sum = sum + value;  
        output.send(0, sum);  
    }  
}
```



```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        $ASSIGN$sum(sum + value);  
        output.send(0, sum);  
    }  
    int $ASSIGN$sum(int v) {  
        ... // save current sum  
        return sum = v;  
    }  
}
```

- \$ASSIGN\$sum first saves sum's current value, and then performs the Java assignment.



Simulating Assignment with Function

- Assignments are expressions.

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        output.send(0,  
                   sum = sum + value);  
    }  
}
```



```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        int value = input.get();  
        output.send(0,  
                   $ASSIGN$sum(sum + value));  
    }  
    int $ASSIGN$sum(int v) {  
        ... // record sum  
        return sum = v;  
    }  
}
```

- Function call precisely simulates assignment.
- Inlining as a compiler optimization.



Operators that Update Operands

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        value = input.get();  
        output.send(0, sum += value);  
    }  
}
```

\$update\$sum handles
all the Java operators,
including +=, -=, ++, --,
etc.

```
class Accumulator extends Actor {  
    private int sum;  
    Port input, output;  
    ...  
    void fire() {  
        value = input.get();  
        output.send(0, $UPDATE$sum(0, value));  
    }  
  
    /* Simulate Java operators.  
     * int $UPDATE$sum(int type, int v) {  
     *     ... // record sum  
     *     switch (type) {  
     *         case 0: return sum += v;  
     *         case 1: return sum -= v;  
     *         case 2: return sum ++; // v ignored  
     *         case 3: return sum --;  
     *         case 4: return ++ sum;  
     *         case 5: return -- sum;  
     *         ...  
     *     default: return error();  
     *     }  
     * }
```



Array Operations

```
class StringStack {  
    private String[] array;  
    private int index;  
    ...  
    void push(String s) {  
        array[index++] = s;  
    }  
    String pop() {  
        return array[index--];  
    }  
    void clean() {  
        index = 0;  
        array = new String[MAX];  
    }  
    void copy(StringStack stack) {  
        stack.array =  
            (String[]) array.clone();  
        stack.index = index;  
    }  
}
```

```
class StringStack {  
    private String[] array;  
    private int index;  
    ...  
    void push(String s) {  
        $ASSIGN$array($UPDATE$index(2, 0), s);  
    }  
    String pop() {  
        return array[$UPDATE$index(3, 0)];  
    }  
    void clean() {  
        $ASSIGN$index(0);  
        $ASSIGN$array(new String[MAX]);  
    }  
    void copy(StringStack stack) {  
        stack.$ASSIGN$array(  
            (String[]) array.clone());  
        stack.$ASSIGN$index(index);  
    }  
    ...  
}
```





Checkpoint Control

- To create a checkpoint:

```
long handle = createCheckpoint();
```

- To recover state:

```
rollback(handle);
```

- To discard a checkpoint:

```
discard(handle);
```

Application to threaded systems, e.g., Ptolemy II:

- Hierarchical composition of (sub-)models.
- An actor does not directly reference other actors.

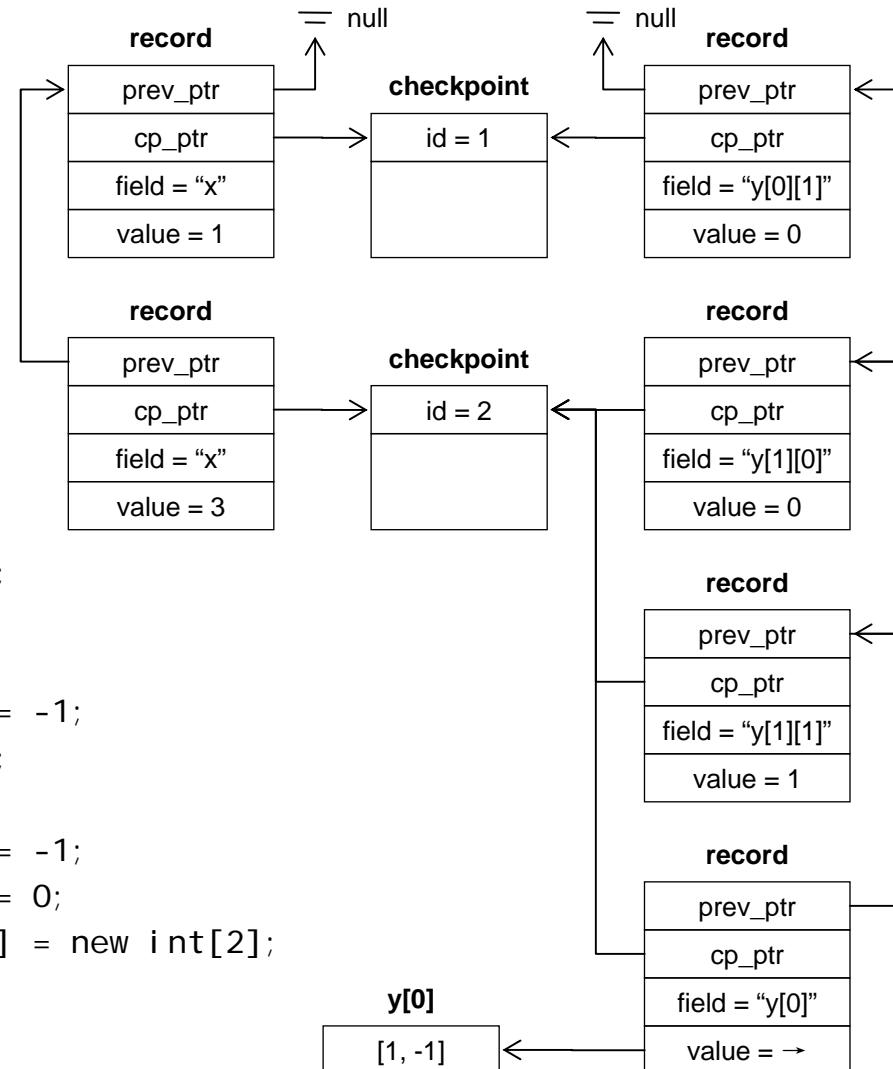


Data Structure

$$x = 1, \quad y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```

int handle1 = createCheckpoint();
$ASSIGN$x(2);           // x = 2;
$ASSIGN$x(3);           // x = 3;
$ASSIGN$y(0, 1, -1);    // y[0][1] = -1;
int handle2 = createCheckpoint();
$UPDATE$x(2, 0);         // x++;
$ASSIGN$y(1, 0, -1);    // y[1][0] = -1;
$ASSIGN$y(1, 1, 0);     // y[1][1] = 0;
$ASSIGN$y(0, new int[2]); // y[0] = new int[2];
rollback(handle1);
  
```





Stateful Java Classes

Hashtable, List, Random, Set, Stack, etc.

1. Get the source (Hashtable, Map, ...) from Sun (alternatively, GCJ):

```
package java.util;
class Hashtable implements Map {
    ...
}
```

2. Apply the same transformation:

java.util.Hashtable → ptolemy.backtrack.java.util.Hashtable

3. Fix references:

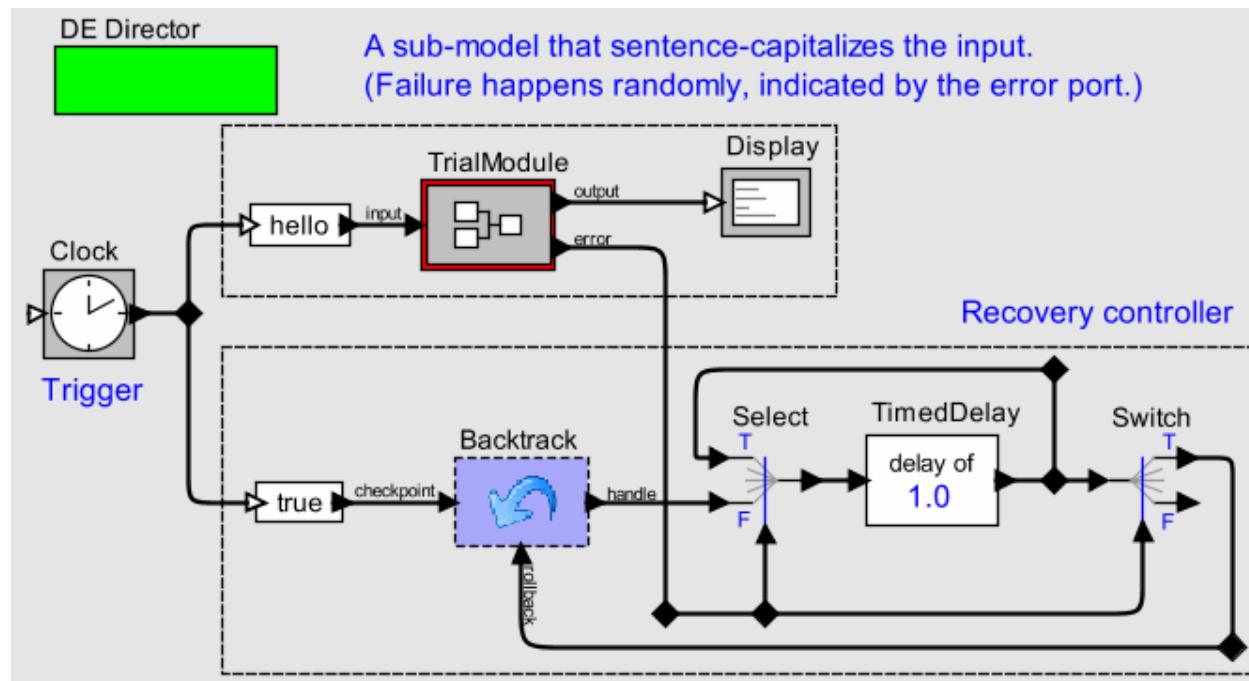
```
import java.util.Hashtable;
class A extends Actor {
    private Hashtable table;
    ...
}
```



```
import ptolemy.backtrack.java.util.Hashtable;
class A extends Actor {
    private Hashtable table;
    ...
}
```



Use Case: Recoverable Model Execution



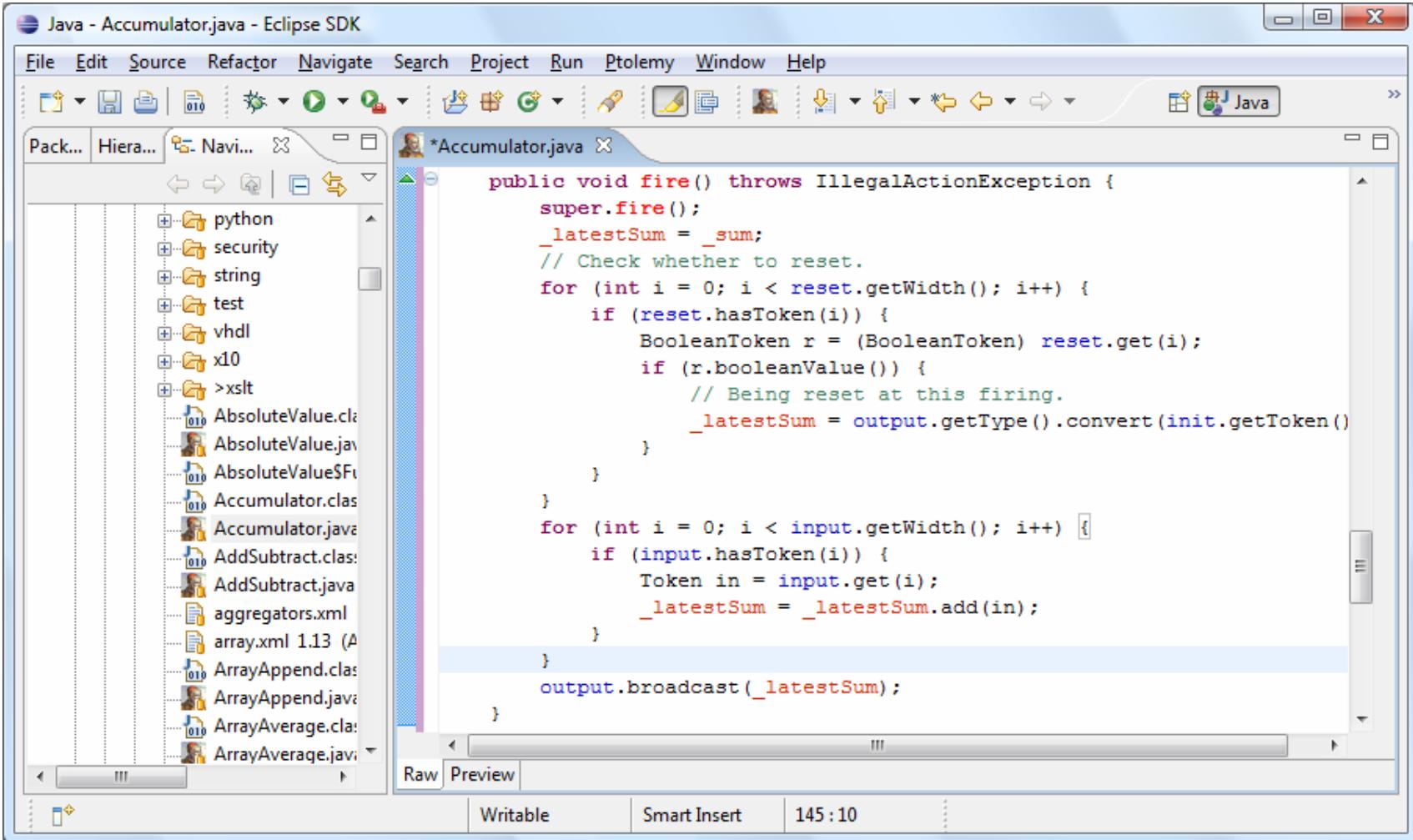
A DE model that autonomously recovers from error



Application to Time Warp Simulation

- On receiving an event: create a checkpoint and record the handle.
- Process the event.
- When inconsistency happens:
 - Detect the event that caused this problem.
 - Recover the state *right before* the event was processed.
 - Re-process the events since that time in a *better* order.

Development Environment

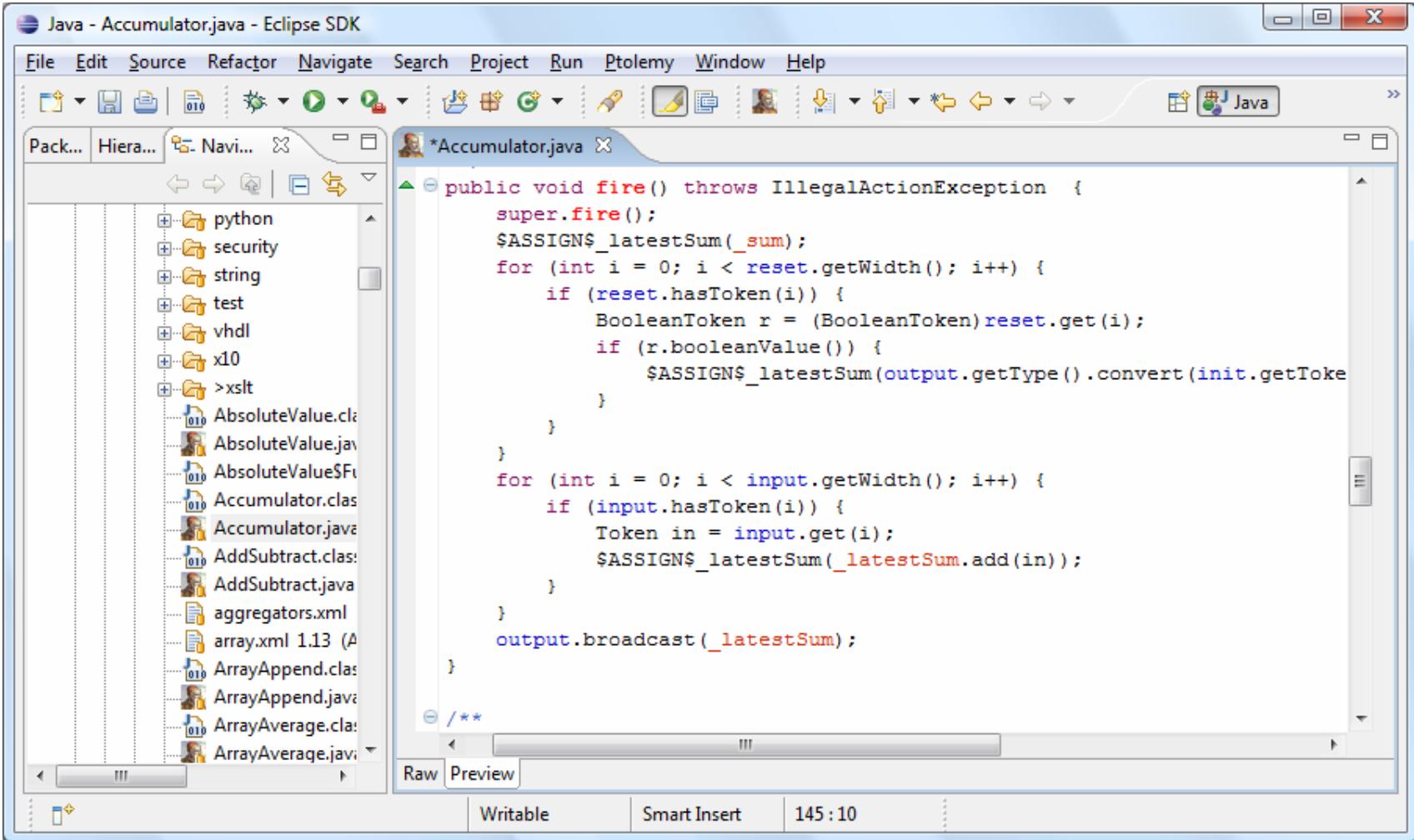


The screenshot shows the Eclipse IDE interface with the title "Java - Accumulator.java - Eclipse SDK". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Ptolemy, Window, and Help. The toolbar has various icons for file operations like Open, Save, and Run. The left sidebar shows a project tree with packages like python, security, string, test, vhdl, x10, >xslt, and subfolders containing files such as AbsoluteValue.class, AbsoluteValue.java, AbsoluteValue\$Fu, Accumulator.class, Accumulator.java, AddSubtract.class, AddSubtract.java, aggregators.xml, array.xml 1.13 (A), ArrayAppend.class, ArrayAppend.java, ArrayAverage.class, and ArrayAverage.java. The main editor window displays the Java code for the Accumulator class:

```
public void fire() throws IllegalActionException {
    super.fire();
    _latestSum = _sum;
    // Check whether to reset.
    for (int i = 0; i < reset.getWidth(); i++) {
        if (reset.hasToken(i)) {
            BooleanToken r = (BooleanToken) reset.get(i);
            if (r.booleanValue()) {
                // Being reset at this firing.
                _latestSum = output.getType().convert(init.getToken());
            }
        }
    }
    for (int i = 0; i < input.getWidth(); i++) {
        if (input.hasToken(i)) {
            Token in = input.get(i);
            _latestSum = _latestSum.add(in);
        }
    }
    output.broadcast(_latestSum);
}
```

The status bar at the bottom indicates "Writable", "Smart Insert", and the current line number "145:10".

Development Environment



The screenshot shows the Eclipse IDE interface with the title "Java - Accumulator.java - Eclipse SDK". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Ptolemy, Window, and Help. The toolbar has various icons for file operations like Open, Save, and Run. The left sidebar shows a package explorer with several projects and files listed under "Accumulator" and "x10". The main editor window displays the Java code for "Accumulator.java". The code implements a method "fire()" that iterates through tokens in "reset" and "input" to calculate a sum and broadcast it. The code uses annotations like \$ASSIGN\$ and \$latestSum. The bottom status bar shows "Writable", "Smart Insert", and the current line number "145:10".

```
public void fire() throws IllegalActionException {
    super.fire();
    $ASSIGN$_latestSum(_sum);
    for (int i = 0; i < reset.getWidth(); i++) {
        if (reset.hasToken(i)) {
            BooleanToken r = (BooleanToken)reset.get(i);
            if (r.booleanValue()) {
                $ASSIGN$_latestSum(output.getType().convert(init.getToke
            }
        }
        for (int i = 0; i < input.getWidth(); i++) {
            if (input.hasToken(i)) {
                Token in = input.get(i);
                $ASSIGN$_latestSum(_latestSum.add(in));
            }
        }
        output.broadcast(_latestSum);
    }
}
```



Conclusion

- Easily proved correct using operational semantics of the Java language. (Refer to the paper.)
- High-performance backtracking mechanism.
 - Checkpoint creation incurs a small constant cost.
 - Rollback and discard operations are linear in the updates.
- Precisely roll back to any execution point (assuming enough memory).
- Applicable to real-time systems by slowing them down gracefully.
 - Proportional slowdown because of the constant extra cost per update.