RV: A Runtime Verification Framework for Monitoring, Prediction and Mining

Patrick Meredith
Grigore Rosu

University of Illinois at Urbana-Champaign (UIUC)
Runtime Verification, Inc.

(joint work with Dongyun Jin and Choonghwan Lee at UIUC)
Assurances for Self-Adaptive Systems

• Self-adaptive systems come with new challenges
  – Increased complexity
  – Reasoning in the presence of uncertainty
  – System analysis pushed to a whole new dimension
    • Concerned not only with functional correctness of the system, but also with how system changes/adapts
  – ...

• This talk focuses on a very limited related aspect
  – How to specify a subset of essentially dynamic properties and how to efficiently runtime verify and validate them
Runtime Verification (RV)

- **Observe** the system while it executes, possibly after appropriate instrumentation
- **Check** the observed behavior against desirable explicit or implicit properties
- **React** to detected violations
  - *Report error*, if used before deployment
  - *Recover* (when possible), during deployment
RV Workshop/Conference

• Runtime verification had a hard time initially
  – Not testing, not verification; what it is?
• Started in 2001 as workshop, about 20 people
  – ENTCS proceedings
  – Selected papers in journal (FMSD)
• Turned into a conference in 2010
  – LNCS proceedings
  – FMSD special issue
  – 80 participants
• RV’11 had ~80 submissions
Current Limitations of Runtime Verification

- **Runtime and memory overhead**: system observation and monitoring may add significant runtime and memory overhead.
- **Limited coverage**: if done naively, runtime verification guarantees lack of bugs only in the observed path, but not in other paths.
- **Specifications**: the difficulty of producing property specifications is underestimated. Developers do not want to write them.
Overview

• The RV system
  – Highlights and goals
• Parametric properties
  – What they are, semantics and slicing
• Underlying technologies of RV
  – Monitoring
  – Prediction
  – Mining
• Conclusion
The RV System

• Developed by Runtime Verification, Inc., a startup company in Urbana, in collaboration with the Formal Systems Lab (FSL) at UIUC

• Aims at overcoming the current limitations of runtime verification and, implicitly, at becoming the best runtime verification system

• Builds upon technologies developed during the last 7 years within the FSL@UIUC
How RV System Overcomes Current Limitations of Runtime Verification

• **Low runtime and memory overhead**: efficient instrumentation; delay monitor creation; garbage collect monitors

• **Increase coverage**: predictive runtime analysis used to analyze causal models instead of flat execution traces; causal models comprise many traces

• **Mine specifications**: running and observing unit tests, it learns (1) the most likely interacting events, and (2) the most likely properties they obey
RV System Overview
RV Builds Upon UIUC Technologies

• Efficient monitoring of parametric properties
  – JavaMOP
    • RV’03-10, TACAS’05, OOPSLA’07, ASE’08, ASE’09, PLDI’11

• Predictive runtime analysis
  – jPredictor
    • FSE’03, TACAS’04, FMOODS’05, CAV’07, SAS’07, ICSE’08

• Mining of parametric properties
  – jMiner
    • ICSE’11
RV at Work: Monitoring I

• Consider the following (wrong) Java program:

```java
import java.util.*;
public class SafeEnum_1 {
    public static void main(String[] args){
        Vector<Integer> v = new Vector<Integer>();
        v.add(1); v.add(2); v.add(4); v.add(8);
        Enumeration e = v.elements();
        int sum = 0;
        if(e.hasMoreElements()){
            sum += (Integer)e.nextElement();
        }
        while(e.hasMoreElements()){
            sum += (Integer)e.nextElement();
        }
        v.clear();
        System.out.println("sum: " + sum);
    }
}
```

• It modifies vector while enumerating it
RV at Work: Monitoring II

• Write parametric specifications under ./mop:

```bash
bash-3.2$ more mop/SafeEnum.mop
package mop;
import java.io.*;
import java.util.*;

SafeEnum(Vector v, Enumeration e) {
  event create after(Vector v) returning(Enumeration e) :
    call(Enumeration Vector+.elements())
    && target(v) {
  event update source after(Vector v) :
    (call(* Vector+.remove*(..))
     || call(* Vector+.add*(..))
     && target(v){})
  event next before(Enumeration e) :
    call(* Enumeration+.nextElement())
    && target(e) {
    ere : create next* update source+ next
    @match {
      System.out.println("improper enumeration usage at " + __LOC);
      __RESET;
    }
  }
```
RV at Work: Monitoring III

• Then call RV on the Java program to monitor:

```
bash-3.2$ rv-monitor SafeEnum_1
-Processing ./mop/SafeEnum.mop
  SafeEnumMonitorAspect.aj is generated

SafeEnum_1.java
Executing your program:
improper enumeration usage at SafeEnum_1.java:23
  sum: 26
Done
```

• RV compiles the program (with javac), generates monitors as aspects, then weaves them within the program binary (with ajc), then runs the resulting programs
RV at Work: Prediction I

• Consider the following (wrong) Java program:

```java
package simple;
public class Simple extends Thread {
    static public int i = 1;
    public static void main(String[] args) {
        (new Simple()).start();
        (new Simple()).start();
    }
    public void run() {
        i++;
        System.out.println(i);
    }
}
```

• Program has a race on the i and a race on output; these races will most likely not appear during testing; RV can predict them from one “successful” execution
RV at Work: Prediction II

- By default, RV prediction searches for races:

```
bash-3.2$ javac simple/Simple.java
bash-3.2$ rv-predict simple.Simple
Instrumenting...
...Done
Executing the instrumented code...
  2
  3
...Done
Running jpredictor...
Determining race candidates
The following are our race candidates:
  | java.io.PrintStream (instance #657291792) | simple.Simple.i
Predicting for race candidate: java.io.PrintStream (instance #657291792)

/--- Race found on java.io.PrintStream (instance #657291792) ---\
  | Write at simple.Simple:10
  \-----------------------------------------
Predicting for race candidate: simple.Simple.i

/--- Race found on simple.Simple.i ---\
  | Read  at simple.Simple:10
  | Write at simple.Simple:9
  \-----------------------------------------

...Done
bash-3.2$
```
RV at Work: Prediction III

- To predict violations of parametric properties, one needs to place them under ./mop, same like when monitoring them
- A program may have races but those may not lead to violations of specifications
  - Though not a good idea to allow races anyway
- Also, a program may run “successfully”, have no races predicted in it, but still violate behavioral specifications
RV at Work: Mining

• The RV miner is less obvious to use
• It works in two stages:
  – First, it observes executions of (small and tricky) unit tests to learn groups of parameters and corresponding events that interact
  – Then it observes (large and likely correct) programs making use of the learned events and
    • It slices the observed (large) parametric traces into (typically lots of small) non-parametric traces
    • It learns non-parametric properties that best explain the computed slices
    • Then adds the learned parameters to them
Parametric Properties

• At the core of RV: all requirements are specified as parametric properties
• These are properties over parametric events
• Comprise a potentially unbounded number of property instances, one for each parameter
• The main challenge is how to efficiently and correctly keep track of all property instances
Examples of Parametric Properties

- Typestates are particular, one-parameter properties
- For example, “hasnext() must be called and hold before each next() is called”, can be defined as follows:

\[ \Lambda_i \]
Examples of Parametric Properties II

• Same "hasnext() must be called and hold before each next() is called" typestate property, but specified using different parametric variants:
  – As a parametric regular expression:
    \[ \Lambda i . \left( \text{hasnexttrue}\langle i \rangle^+ \text{next}\langle i \rangle | \text{hasnextfalse}\langle i \rangle^* \right)^* \]
  – As a parametric linear temporal logic formula:
    \[ \Lambda i . \Box (\text{next}\langle i \rangle \implies \Diamond \text{hasnexttrue}\langle i \rangle) \]
Examples of Parametric Properties III

- Collections are not allowed to change while accessed through iterators:

\[ \Lambda c, i . \ createlter\langle c \ i \rangle \ next\langle i \rangle^* \ updateColl\langle c \rangle^+ \ next\langle i \rangle \]

- To avoid clutter, we drop the understood event parameters, i.e., we write the above as

\[ \Lambda c, i . \ createlter \ next^* \ updateColl^+ \ next \]
Examples of Parametric Properties IV

• Authenticate each key before use:

\[ \Lambda k . \Box (\text{use} \rightarrow \Diamond \text{authenticate}) \]

• Each function should release each lock as many times as it acquires it (this is a parametric context-free property):

\[ \Lambda l . \begin{array}{|c|} \hline \text{S} \rightarrow \text{S} \text{ begin } \text{S} \text{ end} \mid \text{S acquire } \text{S release} \mid \epsilon \end{array} \]
Parametric Trace Slicing I

• RV is all about monitoring, predicting violations of, and mining parametric properties

• All the above work with parametric traces
  – Traces formed with parametric events

• Therefore, all face the problem of trace slicing
  – How to identify the sub-traces of events corresponding to each parameter instance?
For given parameters \((v, e)\)

**Challenge:** How to do it efficiently?
RV Monitoring

• Sends each trace slice to one monitor instance
• Problem: The number of slices can be huge (potentially unbounded)
• Challenge: Manage monitors efficiently
  – Index them for efficient retrieval
  – Garbage collect them when become unnecessary
Monitor Indexing

- Indexing trees using weak references
  - One tree per parameter combination in events
  - Monitors are collected when no events available
More Garbage Collection

- Indexing trees alone not good enough
- Pathological examples where unnecessary monitors stay alive forever
- Solution: analyze the property statically; then collect monitors when they have no future
RV Monitoring Performance Evaluation

>2x faster than JavaMOP, >10x faster than Tracematches

(A) Runtime overhead in %
(B) Memory overhead in MB

<table>
<thead>
<tr>
<th></th>
<th>ORIG (sec)</th>
<th>HASNEXT</th>
<th>UNSAFEITER</th>
<th>UNSAFEMAPITER</th>
<th>UNSAFESYNCCOLL</th>
<th>UNSAFESYNCMAP</th>
<th>ALL</th>
<th>RV</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>3.6</td>
<td>2.19</td>
<td>19.19</td>
<td>∞</td>
<td>1359</td>
<td>774</td>
<td>212</td>
<td>1942</td>
</tr>
<tr>
<td>jython</td>
<td>8.9</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>avora</td>
<td>13.6</td>
<td>45</td>
<td>637</td>
<td>∞</td>
<td>75</td>
<td>144</td>
<td>80</td>
<td>54</td>
</tr>
<tr>
<td>batik</td>
<td>3.5</td>
<td>3</td>
<td>355</td>
<td>∞</td>
<td>208</td>
<td>9</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>eclipse</td>
<td>79.0</td>
<td>-2</td>
<td>0</td>
<td>5</td>
<td>-4</td>
<td>2</td>
<td>1</td>
<td>∞</td>
</tr>
<tr>
<td>h2</td>
<td>2.0</td>
<td>200</td>
<td>350</td>
<td>∞</td>
<td>868</td>
<td>21</td>
<td>6</td>
<td>83</td>
</tr>
<tr>
<td>lindice</td>
<td>18.7</td>
<td>89</td>
<td>128</td>
<td>∞</td>
<td>1350</td>
<td>21</td>
<td>6</td>
<td>83</td>
</tr>
<tr>
<td>lusearch</td>
<td>2.9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>pmd</td>
<td>25.3</td>
<td>-1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sunflow</td>
<td>32.7</td>
<td>47</td>
<td>7</td>
<td>9</td>
<td>13</td>
<td>6</td>
<td>5</td>
<td>17</td>
</tr>
<tr>
<td>tomcat</td>
<td>13.8</td>
<td>8</td>
<td>37</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>tradebeans</td>
<td>45.5</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>-1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>tradesoap</td>
<td>xalan</td>
<td>94.4</td>
<td>1</td>
<td>27</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>20.3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ORIG (MB)</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
<th>TM</th>
<th>MOP</th>
<th>RV</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>4.9</td>
<td>56.8</td>
<td>19.3</td>
<td>13.2</td>
<td>7.7</td>
<td>146.8</td>
<td>79.0</td>
<td>173.4</td>
<td>56.1</td>
<td>6.8</td>
<td>127.9</td>
<td>48.3</td>
<td>6.9</td>
<td>155.4</td>
<td>12.7</td>
<td>340.9</td>
<td>7.3</td>
<td>150.9</td>
<td>13.2</td>
</tr>
<tr>
<td>jython</td>
<td>5.3</td>
<td>5.7</td>
<td>4.6</td>
<td>4.8</td>
<td>4.9</td>
<td>4.6</td>
<td>4.8</td>
<td>6.0</td>
<td>19.5</td>
<td>4.7</td>
<td>5.3</td>
<td>4.5</td>
<td>4.4</td>
<td>5.9</td>
<td>4.8</td>
<td>5.1</td>
<td>4.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>avora</td>
<td>4.7</td>
<td>4.7</td>
<td>12.4</td>
<td>9.1</td>
<td>4.4</td>
<td>136.2</td>
<td>15.8</td>
<td>∞</td>
<td>14.7</td>
<td>8.5</td>
<td>4.3</td>
<td>28.0</td>
<td>12.6</td>
<td>4.4</td>
<td>13.0</td>
<td>4.9</td>
<td>22.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>batik</td>
<td>79.1</td>
<td>79.2</td>
<td>78.7</td>
<td>79.3</td>
<td>75.2</td>
<td>93.6</td>
<td>86.6</td>
<td>∞</td>
<td>91.2</td>
<td>79.6</td>
<td>78.2</td>
<td>93.2</td>
<td>85.1</td>
<td>79.9</td>
<td>86.9</td>
<td>76.7</td>
<td>104.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>95.9</td>
<td>100.8</td>
<td>107.6</td>
<td>97.1</td>
<td>98.3</td>
<td>100.0</td>
<td>110.3</td>
<td>106.9</td>
<td>93.8</td>
<td>101.1</td>
<td>100.4</td>
<td>109.2</td>
<td>90.1</td>
<td>∞</td>
<td>98.6</td>
<td>98.7</td>
<td>98.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h2</td>
<td>20.7</td>
<td>97.4</td>
<td>47.1</td>
<td>52.5</td>
<td>24.3</td>
<td>24.2</td>
<td>29.4</td>
<td>∞</td>
<td>69.2</td>
<td>28.1</td>
<td>∞</td>
<td>54.8</td>
<td>24.8</td>
<td>∞</td>
<td>55.9</td>
<td>25.2</td>
<td>47.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lindice</td>
<td>265.0</td>
<td>267.8</td>
<td>598.5</td>
<td>565.2</td>
<td>267.2</td>
<td>266.2</td>
<td>262.4</td>
<td>312.4</td>
<td>268.3</td>
<td>268.2</td>
<td>271.4</td>
<td>690.3</td>
<td>265.5</td>
<td>271.0</td>
<td>718.3</td>
<td>270.0</td>
<td>283.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lusearch</td>
<td>6.8</td>
<td>6.5</td>
<td>5.5</td>
<td>5.6</td>
<td>6.3</td>
<td>6.9</td>
<td>6.8</td>
<td>7.4</td>
<td>8.2</td>
<td>6.9</td>
<td>7.4</td>
<td>7.4</td>
<td>7.1</td>
<td>7.4</td>
<td>11.0</td>
<td>11.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>4.6</td>
<td>4.7</td>
<td>4.4</td>
<td>4.8</td>
<td>4.6</td>
<td>4.8</td>
<td>4.2</td>
<td>4.0</td>
<td>4.3</td>
<td>4.8</td>
<td>4.5</td>
<td>4.5</td>
<td>4.6</td>
<td>4.6</td>
<td>4.7</td>
<td>4.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>18.0</td>
<td>17.2</td>
<td>146.3</td>
<td>86.4</td>
<td>121.7</td>
<td>93.6</td>
<td>217.3</td>
<td>20.3</td>
<td>238.4</td>
<td>84.6</td>
<td>∞</td>
<td>117.1</td>
<td>32.9</td>
<td>420.0</td>
<td>45.7</td>
<td>45.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcat</td>
<td>11.6</td>
<td>12.5</td>
<td>11.0</td>
<td>11.4</td>
<td>11.9</td>
<td>11.4</td>
<td>11.0</td>
<td>11.3</td>
<td>11.4</td>
<td>11.0</td>
<td>11.4</td>
<td>11.4</td>
<td>11.3</td>
<td>11.4</td>
<td>11.8</td>
<td>11.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tradebeans</td>
<td>63.2</td>
<td>63.7</td>
<td>63.9</td>
<td>64.1</td>
<td>63.3</td>
<td>62.5</td>
<td>62.7</td>
<td>63.2</td>
<td>62.8</td>
<td>62.0</td>
<td>64.0</td>
<td>62.8</td>
<td>64.0</td>
<td>64.0</td>
<td>64.0</td>
<td>64.0</td>
<td>62.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tradesoap</td>
<td>xalan</td>
<td>64.1</td>
<td>63.4</td>
<td>63.1</td>
<td>64.4</td>
<td>64.1</td>
<td>63.5</td>
<td>62.0</td>
<td>67.0</td>
<td>65.0</td>
<td>65.9</td>
<td>65.5</td>
<td>64.5</td>
<td>65.6</td>
<td>64.5</td>
<td>64.5</td>
<td>64.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>5.0</td>
<td>4.8</td>
<td>4.9</td>
<td>5.1</td>
<td>4.9</td>
<td>4.9</td>
<td>5.0</td>
<td>4.9</td>
<td>4.9</td>
<td>5.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RV Prediction

• Predicts errors in concurrent programs
  – Model checking: number of interleavings is prohibitively large
  – Testing: interleavings depend on environment

• Combine dynamic and static methods to find bad behaviors near correct executions
Prediction Example

Property: “authenticate before access”

Main Thread:

\[ s_1: \text{resource.authenticate}(); \]
\[ s_2: \text{flag.value} = \text{true}; \]
... 

Task Thread:

... 
... 
\[ s_3: \text{while} (! \text{flag.value}) \]
\[ \text{Thread.yield}(); \]
\[ s_4: \text{resource.access}(); \]
... 

Observed execution: \(... s_1 \ s_2 \ s_3 \ s_4 ...\)
Prediction Example

Property: “authenticate before access”

Main Thread:

\[ s_1: \text{resource.authenticate()}; \]
\[ s_2: \text{flag.value = true}; \]
\[ \cdots \]

Task Thread:

\[ \cdots \]
\[ \cdots \]
\[ s_3: \text{if (! flag.value)} \]
\[ \quad \text{Thread.yield()}; \]
\[ s_4: \text{resource.access()}; \]
\[ \cdots \]

Observed execution: \( \cdots s_1 s_2 s_3 s_4 \cdots \)

- Buggy \( S_4 \) can be executed before \( S_1 \)
- Low likelyhood to hit error in testing
Predictive Runtime Analysis

Search space
Predictive Runtime Analysis

Search space

Observed execution
Predictive Runtime Analysis
Predictive Runtime Analysis
Happens-Before Works ... If Lucky

*Property*: “authenticate before access”

**Main Thread:**

\[ s_1: \text{resource.authenticate}() \]
\[ s_2: \text{flag.value} = \text{true}; \]

**Task Thread:**

\[ s_3: \text{if (!flag.value)} \]
\[ \quad \text{Thread.yield}(); \]
\[ s_4: \text{resource.access}(); \]

Observed execution: \[ s_3 \ s_1 \ s_2 \ s_4 \]
Property: "authenticate before access"

Main Thread:

\[ s_1: \text{resource.authenticate}() \]
\[ s_2: \text{flag.value} = \text{true}; \]

Task Thread:

\[ s_3: \text{if (\neg flag.value)} \]
\[ \text{Thread.yield}(); \]
\[ s_4: \text{resource.access}(); \]

Observed execution: \( s_3 \ s_1 \ s_2 \ s_4 \)

Causal dependency: \( s_3 \prec s_2 \)

Bad execution inferred: \( s_3 \ s_4 \ s_1 \ s_2 \). Bug detected!

Chances of observing this execution are very low
Happens-Before Limitations

**Property:** “authenticate before access”

Main Thread:

- $s_1$: resource.authenticate()
- $s_2$: flag.value = true;

Task Thread:

- $s_3$: if (!flag.value)
  
  Thread.yield();
- $s_4$: resource.access()

Observed execution: $s_1 \ s_2 \ s_3 \ s_4$

Causal dependency: $s_2 \prec s_3$. No bug found ...

Too constrained: access will be performed regardless of the flag
RV Uses Sliced Causality

• Relaxes the Happens-Before causal model
• How? Focus on the **property**
• Use static information about the program
• Remove irrelevant events and causalities
  – Smaller and more relaxed causal model
  – (Exponentially) more inferred executions
  – Better predictive capability
Static Information: Control Scope

• $S_2$ is in the control scope of $S_1$ if its execution depends on a choice at $S_1$

```
s_1: if (flag) {
  s_2: ...
  } else {
  s_3: ...
  }
  s_4: ...
```

```
s_0: i=0;
  s_1: while (i<3) {
    s_2: ...
    s_3: i++
  }
  s_4: ...
```

```
s_1: while (!flag) {
  s_2: ...
  }
  s_3: ...
```

• Extends to other control statements
  – break/continue, return, exceptions
Sliced Causality Works!

Property: “authenticate before access”

Main Thread:

\[ s_1: \text{resource.authenticate}() \]
\[ s_2: \text{flag.value} = \text{true}; \]

Task Thread:

\[ s_3: \text{if} \ (! \text{flag.value}) \]
\[ \text{Thread.yield}(); \]
\[ s_4: \text{resource.access}(); \]

Observed execution: \[ s_1 \ s_2 \ s_3 \ s_4 \]

Only \( s_1 \) and \( s_4 \) directly relevant to the property
Slice Causality Works!

**Property:** “authenticate before access”

- **Main Thread:**
  - $s_1$: resource.authenticate()
  - $s_2$: flag.value = true;
- **Task Thread:**
  - $s_3$: if (! flag.value)
    - Thread.yield();
  - $s_4$: resource.access();

**Observed execution:** $s_1\ s_2\ s_3\ s_4$

Only $s_1$ and $s_4$ directly relevant to the property.

Execution of $s_4$ not dependent of $s_3$; ignore the causal dependency $s_2 < s_3$.

**Sliced causality:** $s_1 \leftrightarrow s_4$; $s_4\ s_1$ is a potential execution. **Bug detected!**
No False Alarms 😊

**Property:** “authenticate before access”

**Main Thread:**

\[ s_1: \text{resource.authenticate}() \]

\[ s_2: \text{flag.value} = \text{true}; \]

**Task Thread:**

\[ s_3: \text{while} (! \text{flag.value}) \]

\[ \text{Thread.yield}(); \]

\[ s_4: \text{resource.access}(); \]

**Observed execution:** \( s_1 \ s_2 \ s_3 \ s_4 \)
No False Alarms 😊

**Property:** “authenticate before access”

**Main Thread:**

\[ s_1: \text{resource.authenticate}() \]
\[ s_2: \text{flag.value} = \text{true}; \]

**Task Thread:**

\[ s_3: \textbf{while} (! \text{flag.value}) \]
\[ \text{Thread.yield}(); \]
\[ s_4: \text{resource.access}(); \]

Observed execution: \[ s_1 \ s_2 \ s_3 \ s_4 \]

Execution of \( s_4 \) depends on \( \text{flag.value} \) being \textit{true} at \( s_3 \)

causal dependency \( s_2 \prec s_3 \) matters

Sliced causality: \( s_1 \prec s_2 \prec s_3 \prec s_4 \), no false alarm!
## RV Prediction Performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>jPredictor</th>
<th>rv-predict</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Real Time</td>
<td>Disk Usage</td>
</tr>
<tr>
<td>account</td>
<td>-</td>
<td>0:02.57</td>
<td>236K</td>
</tr>
<tr>
<td>elevator</td>
<td>-</td>
<td>5:55.29</td>
<td>63M</td>
</tr>
<tr>
<td>tsp</td>
<td>map4 2</td>
<td>4:24.44</td>
<td>16M</td>
</tr>
<tr>
<td>tsp</td>
<td>map5 2</td>
<td>8:12.31</td>
<td>17M</td>
</tr>
<tr>
<td>tsp</td>
<td>map10 2</td>
<td>&gt; 3 hours</td>
<td>&gt; 230M</td>
</tr>
<tr>
<td>huge</td>
<td></td>
<td>crash</td>
<td>crash</td>
</tr>
<tr>
<td>medium</td>
<td></td>
<td>crash</td>
<td>crash</td>
</tr>
<tr>
<td>small</td>
<td></td>
<td>crash</td>
<td>crash</td>
</tr>
<tr>
<td>mixedlockshuge</td>
<td>-</td>
<td>&gt; 2 hours</td>
<td>&gt; 250M</td>
</tr>
<tr>
<td>mixedlocksbig</td>
<td>-</td>
<td>4:39.08</td>
<td>25M</td>
</tr>
<tr>
<td>mixedlocksmedium</td>
<td>-</td>
<td>0:08.92</td>
<td>2.7M</td>
</tr>
<tr>
<td>mixedlockssmall</td>
<td>-</td>
<td>0:05.46</td>
<td>1.5M</td>
</tr>
</tbody>
</table>
RV Mining

• RV monitoring slices parametric traces and sends each slice to a monitor instance

• RV miner is dual to RV monitor:
  – It uses (almost) the same trace slicer, but instead of sending the slices to monitors, uses them as input to a regular property learner (PFSA)

• This way, we were able to mine most of the parametric specifications that we previously used for monitoring
RV Mining Example I

Two properties in one:
- the collection-iterator
- the hasNext typestate
RV Mining Example

• Server socket parametric specification:
RV Mining Preliminary Evaluation I

- Used several packages that come with unit tests

<table>
<thead>
<tr>
<th>Target package</th>
<th>Event Specifications</th>
<th>Param. Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># files</td>
<td># events</td>
</tr>
<tr>
<td>java.util</td>
<td>370</td>
<td>65,854,349</td>
</tr>
<tr>
<td>java.io</td>
<td>382</td>
<td>28,835,588</td>
</tr>
<tr>
<td>java.lang</td>
<td>372</td>
<td>41,784,568</td>
</tr>
<tr>
<td>java.net</td>
<td>221</td>
<td>9,429,744</td>
</tr>
</tbody>
</table>

(a) Traces used in the experiments

<table>
<thead>
<tr>
<th>Target package</th>
<th># event specifications</th>
<th># parametric specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>145</td>
<td>66</td>
</tr>
<tr>
<td>java.lang</td>
<td>82</td>
<td>48</td>
</tr>
<tr>
<td>java.net</td>
<td>90</td>
<td>36</td>
</tr>
<tr>
<td>java.util</td>
<td>181</td>
<td>80</td>
</tr>
</tbody>
</table>

(b) Inferred events and mined specifications
RV Mining Preliminary Evaluation II

- Still relatively slow, though the mined specifications are very useful and general-purpose.
- Slicing is the most expensive.

<table>
<thead>
<tr>
<th>Target package</th>
<th>Event Specifications</th>
<th>Param. Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io</td>
<td>24</td>
<td>115</td>
</tr>
<tr>
<td>java.lang</td>
<td>38</td>
<td>112</td>
</tr>
<tr>
<td>java.net</td>
<td>59</td>
<td>14</td>
</tr>
<tr>
<td>java.util</td>
<td>59</td>
<td>133</td>
</tr>
</tbody>
</table>

(c) Times (minutes; Windows, 3GHz, 1GB)
Conclusion

• The RV system combines monitoring, prediction and specification mining

• Uses at its core the notion of parametric specification, together with algorithms for parametric trace slicing used across any of the monitoring, prediction and mining capabilities

• RV shows that the three apparently different technologies, namely monitoring, prediction and mining, in fact belong together