The RV System Tutorial

Patrick Meredith and Grigore Rosu
joint work with
Feng Chen, Dongyun Jin, Dennis Griffith, Michael Ilseman
Runtime Verification, Inc.
University of Illinois

Wednesday, November 10, 2010
The RV System

- Combines Runtime Monitoring and Predictive Analysis

**RV-Monitor**
- Runtime Monitoring
- Prediction Logging Aspect
- Prediction Monitor Library
- Parametric Slicing

**RV-Predict**
- Program Under Test
- Instrumentation
- Logging
- Causal Slicing
- Race Detection
- Atomicity Violation Detection
- Generic Property Detection

**System Overview**

![System Overview Diagram]

New Version of JavaMOP

New Version of jPredictor
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Why Monitoring

• Monitoring is **well-adopted** in many engineering disciplines
  – Fuses, watchdogs, fire-alarms, etc.

• Monitoring **adds redundancy**
  – Increases reliability, robustness and confidence in correct behavior, reduces risk

• Provably correct **systems can fail**, too
  – Unexpected environment, wrong/strong assumptions, hardware or OS errors, etc.
Applications of Monitoring

• **Debugging**
  – Development
  – Error messages

• **Testing**
  – Development
  – Error messages

• **Security/Reliability/Robustness/…**
  – Production
  – Recovery mechanisms

• **Programming Paradigm**
  – Production
  – General actions
Runtime Monitoring Systems
(a few of them)

- ≤ 2001
  - MAC (UPenn), PAX (NASA), TimeRover (commercial)
- 2002–2004
  - HAWK/Eagle (NASA), MOP (UIUC), POTA (UTA)
- ≥ 2005:
  - PQL (Stanford)
  - Tracematches (Oxford)
  - PTQL (Berkeley/Stanford/Novell)
  - Pal (UPenn)
  - RuleR (Manchester)
  - … many others
FailFast Iterator

• Following code throws exception in Java (FailFast):

```java
Vector v = new Vector();
Iterator i = v.iterator();
v.add(new Integer(2));
while (i.hasNext()) ... 
```

• **FailFast**: if the underlying vector is changed when an iterator is used for enumerating elements, the iterator fails.

• However ...
MOP Example: Safe Enumeration

- No exception raised if one uses `Enumeration` instead of `Iterator`
  - Java language decision, showing that

```java
SafeEnum(Vector v, Enumeration+ e) {
    event create after(Vector v) returning(Enumeration e): ...
    event updatesource after(Vector v): ...
    event next before(Enumeration e): ...

    ere : create next* updatesource+ next
    @match { System.out.println("Failed Enumeration!"); } 
}
```

> 250 AspectJ LOC generated …
Complexity of SafeEnum

• Tricky to check SafeEnum manually
  – Two counters needed, one in the vector (the current timestamp) and the other in the enumeration (the vector’s timestamp when creating the enumeration).
  – Accesses to vector / enumerator can be scattered all over the code, both in program and in libraries
  – Accesses to vector’s counter must be synchronized
  – Every implementation of the Enumeration interface should repeat the above work
Example of Unsafe Enumeration

- Bug found in jHotDraw

Main Thread:

Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:

... v.remove(0);
...

- The underlying vector should not be changed when one of its enumerations is being used!
Example of Unsafe Enumeration

• Bug found in jHotDraw

Main Thread:

```java
Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...
```

Task Thread:

```java
...
...
v.remove(0);
...
```

May cause unexpected behaviors, e.g., a NoSuchElementException or some errors later.

• The underlying vector should not be changed when one of its enumerations is being used!
RV–Monitor
(Based on Monitoring Oriented Programming – MOP)

http://fsl.cs.uiuc.edu/mop

A generic runtime verification framework
– proposed in 2003 –
RV’03, ICFEM’04, RV’05, CAV’05,
TACAS’05, CAV’06,
CAV’07, OOPSLA’07, ASE’08, RTSS’08,
AOSD’08, TACAS’09, …
What RV-Monitor Supports

• **Observe** a run of a system
  – Requires instrumentation
  – Can be offline or online

• **Check** it against desired properties
  – Specified using patterns or in a logical formalism

• **React/Report** (if needed)
  – Error messages
  – Recovery mechanisms
  – General code
RV–Monitor Model

Program Execution
RV-Monitor Model

Program Execution

Observation/Abstraction

Abstract Trace

Verification

Monitors

M₁, M₂, M₃, ...

Wednesday, November 10, 2010
RV-Monitor Model

Program Execution

Abstract Trace

Observation/Abstraction

Verification

Monitors

M1 M2 M3 ...

Action

Wednesday, November 10, 2010
Monitors verify abstract traces against desired properties; can be dynamically created or destroyed.
Overview

• Monitoring
• RV-Monitor Demo
• RV-Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV-Predict Demo
• RV-Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Overview

• Monitoring
• RV–Monitor Demo
• **RV–Monitor Techniques and Implementation**
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• **RV–Predict Techniques and Implementation**
  – Sliced Causality
  – Pipeline
  – Race Prediction

Wednesday, November 10, 2010
Monitor Synthesis

Program Execution

Abstract Trace

Observation/Abstraction

Verification

Monitors

M_1

M_2

M_3

...
Monitor Synthesis

Program Execution

Observation/Abstraction

Abstract Trace

Verification

Monitors

How do we generate efficient monitors?
MOP: Extensible Logic Framework

• Generic in specification formalisms
• Logic plugin: plugable monitor synthesis components for different logics
• Already provides plugins for many logics
  – FSM (Finite State Machine), ERE (extended regular expressions), PTLTL (Past-time LTL), FTLTL (Future-time LTL), ATL (Allen temporal logic), JML (Java modeling language), PtCaRet (Past-time Call/Return), CFG (Context-free grammars), …

• The desired property can be arbitrarily complex: Raw specifications
FSM Plugin
Finite State Machine

• Easy to use, yet powerful
• Many approaches/users encode important properties directly in finite state machines, e.g., Typestates
• Monitoring FSM
  – Direct translation from an FSM specification to a monitor
ERE Plugin
Extended Regular Expressions

• Regular expressions
  – Widely used in programming, easy to master for ordinary programmers
  – Existing monitor synthesis algorithm

• Extended regular expressions
  – Extend regular exps with complement (negation)
  – Specify properties non-elementarily more compactly
  – More complicated to monitor
LTL Plugins
Linear Temporal Logic

• MOP includes both a past-time plugin (PTLTL) and an over-all LTL plugin for LTL
• PTLTL uses a dynamic programming algorithm, low resources, suitable for hardware
• LTL uses a translation through alternating automata. Semantics of past is different than PTLTL
Most systems support finite state monitors
- Regular languages
- Linear temporal logics

These cannot monitor structured properties:

\[
S \rightarrow S \text{ begin } S \text{ end}
\]

\[
| S \text{ acquire } S \text{ release}
\]

\[
| \epsilon
\]
Most systems support finite state monitors

- Regular languages
- Linear temporal logics

These cannot monitor structured properties:

\[
S \rightarrow S \text{ begin } S \text{ end} \\
| \quad S \text{ acquire } S \text{ release} \\
| \quad \epsilon
\]

\begin{align*}
\text{begin} & \quad \text{acquire} & \text{acquire} & \text{release} & \text{release} & \text{end}
\end{align*}
GLR Parsing Yields CFG Monitors

• Reads input **Left to right, produces Right-most derivation; table driven**
• Bottom–up parsing
  – keeps stack with the current, and previous states
• Efficient
• Handles all context–free grammars, even those with ambiguity
• Makes it a good candidate for CFG monitor synthesis!
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Monitors can be dynamically created or destroyed

Parametric monitoring
Parametric Properties

Needed, but hard to monitor efficiently

```java
SafeEnum(Vector v, Enumeration+ e) {
    event create after(Vector v) returning(Enumeration e): ...
    event updatesource after(Vector v) : ...
    event next before(Enumeration e) : ...

    ere : create next* updatesource+ next
    @match { System.out.println("Failed Enumeration!"); } }
```
Safe Enumeration as Parametric Property

Usage pattern (using regular expressions) of three events

- \texttt{updatesource(v)} : change vector \texttt{v}
- \texttt{create(v,e)} : create enumeration \texttt{e} from vector \texttt{v}
- \texttt{next(e)} : use enumeration \texttt{e}

Violation state
Monitoring Safe Enum

Main Thread:
Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:
...
...
v.remove(0);
...

0 -> create -> 1 -> next -> updatesource -> 2 -> next -> 3

Wednesday, November 10, 2010
Monitoring Safe Enum

**Main Thread:**

Vector v = //initialization;

...  
Enumeration e = v.elements();  
...  
Object obj = e.nextElement();  
...

**Task Thread:**

...  
v.remove(0);  
...
Monitoring Safe Enum

Main Thread:
Vector v = //initialization;
   ...
Enumeration e = v.elements();
   ...
Object obj = e.nextElement();
   ...

Task Thread:
   ...
   v.remove(0);
   ...

create

updateSource

next

create

updateSource

next

updateSource

next

updateSource
Monitoring Safe Enum

Main Thread:

Vector v = //initialization;

Enumeration e = v.elements();

Object obj = e.nextElement();

Task Thread:

... v.remove(0);

...
Monitoring Safe Enum

Main Thread:
Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:
...
...
v.remove(0);
...

create

0 create

1 updatesource

2 next

3

next
updatesource
create

Wednesday, November 10, 2010
Monitoring Safe Enum

Main Thread:
Vector v = //initialization;
…
Enumeration e = v.elements();
…
Object obj = e.nextElement();
…

Task Thread:
…
v.remove(0);
…

create
updatesource

Wednesday, November 10, 2010
Monitoring Safe Enum

Main Thread:

Vector v = //initialization;

Enumeration e = v.elements();

Object obj = e.nextElement();

Task Thread:

... v.remove(0);

...
Monitoring Safe Enum

Main Thread:

Vector v = //initialization;

 Enumeration e = v.elements();

Object obj = e.nextElement();

... create

Task Thread:

... updatesource

v.remove(0);

... next
Monitoring Safe Enum

Main Thread:

Vector v = //initialization;

Enumeration e = v.elements();

Object obj = e.nextElement();

Task Thread:

... 

v.remove(0);

...
Lack of Parameters Leads to False Alarms

Main Thread:

Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:

...

v.remove(0);
...


Wednesday, November 10, 2010
Lack of Parameters Leads to False Alarms

Main Thread:

Vector v = //initialization;

... 

Enumeration e = v.elements();

... 

Object obj = e.nextElement();

... 

Task Thread:

... 

v2.remove(0); 

...
Lack of Parameters Leads to False Alarms

**Main Thread:**

Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

**Task Thread:**

... 
... 
v2.remove(0);
... 

...
Lack of Parameters Leads to False Alarms

Main Thread:

Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:

... 
v2.remove(0);
...

Appear to be a violation but it is not; false alarm!
Adding Parameters to Events

Main Thread:

Vector v = //initialization;

...  
Enumeration e = v.elements();

...  
Object obj = e.nextElement();

...  

Task Thread:

...  
v2.remove(0);

...
Adding Parameters to Events

Main Thread:
Vector v = //initialization;

...  
Enumeration e = v.elements();  
...  
Object obj = e.nextElement();  
...

Task Thread:

create(v, e)
update(v2)
next(e)

...
Adding Parameters to Events

Main Thread:
Vector v = //initialization;
...
Enumeration e = v.elements();
...
Object obj = e.nextElement();
...

Task Thread:
create(v, e)
update(v)
next(e)
...

Wednesday, November 10, 2010
Adding Parameters to Events

**Main Thread:**

Vector v = //initialization;

...  
Enumeration e = v.elements();

...  
Object obj = e.nextElement();

...  

**Task Thread:**

...  
v.remove(0);

...  

**Parametric traces:** traces containing events with parameters; Abundant in practice, especially in object-oriented programs
Checking Parametric Traces
Checking Parametric Traces

parametric trace

updatesource(v1)
create(v1,e1)
updatesource(v2)
next(e1)
create(v1,e2)
updatesource(v1)
next(e1)
Checking Parametric Traces

**Parametric trace**
- updatesource(v1)
- create(v1,e1)
- updatesource(v2)
- next(e1)
- create(v1,e2)
- updatesource(v1)
- next(e1)

**Non-parametric monitor**

```

next

create

updatesource

next

updatesource
```
Checking Parametric Traces

**parametric** trace:
- updatesource(v1)
- create(v1,e1)
- updatesource(v2)
- next(e1)
- create(v1,e2)
- updatesource(v1)
- next(e1)

**non-parametric** monitor:
- 0 [create]
- 1 [updatesource]
- 2 [next]
- 3 [next, updatesource]
Checking Parametric Traces

**parametric trace**
- `updatesource(v1)`
- `create(v1,e1)`
- `updatesource(v2)`
- `next(e1)`
- `create(v1,e2)`
- `updatesource(v1)`
- `next(e1)`

**parametric monitor**

- State 0: `create`
- State 1: `updatesource`
- State 2: `next`
- State 3: `updatesource`
Parametric Monitors

• Other approaches: Monolithic (centralized) monitors
  – Tracematches [Oxford], Program Query Language (PQL) [Stanford], Eagle [NASA], etc.
  – Bound to specific formalisms/checking mechanisms
  – Limited expressiveness, specific to application domains

• Our solution: decentralized monitors
  – Formalism–independent, works with any formalism
    • More expressive, adaptive to different domains
  – Facilitates optimization (separation of concerns)
    • Evaluation shows better performance
Parametric Trace Slicing

For given parameters \((v, e)\)
Parametric Trace Slicing

For given parameters \((v, e)\)
Parametric Trace Slicing

<table>
<thead>
<tr>
<th>updatesource(v1)</th>
<th>create(v1,e1)</th>
<th>updatesource(v2)</th>
<th>next(e1)</th>
<th>create(v1,e2)</th>
<th>updatesource(v1)</th>
<th>next(e1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1, e1</td>
<td>v1, e2</td>
<td>v2, e1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For given parameters \((v, e)\)

Wednesday, November 10, 2010
Parametric Trace Slicing

For given parameters \((v, e)\)
Parametric Trace Slicing

For given parameters \((v, e)\)
Parametric Trace Slicing

For given parameters \((v, e)\)
Naive monitoring of Parametric Traces

- Every parametric trace contains multiple non-parametric trace slices, each corresponding to a particular parameter binding.
Naive monitoring of Parametric Traces

- Every parametric trace contains multiple non-parametric trace slices, each corresponding to a particular parameter binding.
For given parameters \((v, e)\)
Parametric Trace Slicing – Challenges

For given parameters \((v, e)\)

How to do it efficiently?
Parametric Trace Slicing – Challenges

update(v1)
createEnum(v1,e1)
update(v2)
useEnum(e1)
createEnum(v1,e2)
update(v1)
useEnum(e1)
update
createEnum
useEnum
update
createEnum
useEnum
update
createEnum
useEnum

How to do it efficiently?
What if the trace is not complete?

For given parameters \((v, e)\)
Online Parametric Trace Slicing

• Online: process events as receiving them and do not look back for the previous events

• Efficient
  – Scan the trace once
  – Events discarded immediately after being processed

• What information should be kept for the unknown future?
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Slicing Example

For given parameters \((v, e)\)
Slicing Example

For given parameters \((v, e)\)

Wednesday, November 10, 2010
**Slicing Example**

<table>
<thead>
<tr>
<th>update(v1)</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>createEnum(v1,e1)</td>
<td></td>
</tr>
</tbody>
</table>

For given parameters \((v, e)\)

Wednesday, November 10, 2010
## Slicing Example

### For given parameters \((v, e)\)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1)</td>
<td>update</td>
</tr>
<tr>
<td>(v_1, e_1)</td>
<td>update</td>
</tr>
</tbody>
</table>

- **update**
- **createEnum**
### Slicing Example

<table>
<thead>
<tr>
<th>update(v1)</th>
<th>update(v1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>createEnum(v1,e1)</td>
<td>createEnum(v1,e1)</td>
</tr>
</tbody>
</table>

For given parameters \((v, e)\)
Slicing Example

For given parameters (v, e)
### Slicing Example

<table>
<thead>
<tr>
<th>v1</th>
<th>v1, e1</th>
<th>v2</th>
<th>e1</th>
</tr>
</thead>
<tbody>
<tr>
<td>update(v1)</td>
<td>update</td>
<td>update</td>
<td>update</td>
</tr>
<tr>
<td>createEnum(v1, e1)</td>
<td>createEnum</td>
<td></td>
<td>useEnum</td>
</tr>
<tr>
<td>update(v2)</td>
<td></td>
<td></td>
<td>useEnum</td>
</tr>
</tbody>
</table>

For given parameters \((v, e)\)
Slicing Example

For given parameters \((v, e)\)
Slicing Example

For given parameters \((v, e)\)

Wednesday, November 10, 2010
Slicing Example

For given parameters \((v, e)\)
Slicing Example

For given parameters \((v, e)\)

- update\((v1)\)
- createEnum\((v1, e1)\)
- update\((v2)\)
- createEnum\((v1, e2)\)
- useEnum\((e1)\)

Wednesday, November 10, 2010
Slicing Example

For given parameters \((v, e)\)
Slicing Example

Optimization: based on *static* property analysis, generate specialized slicing code for the given specification

For given parameters \((v, e)\)
Optimization: based on static property analysis, generate specialized slicing code for the given specification.
# RV–Monitor Performance

## Comparison of Tracematches (TM), JavaMOP (MOP), and RV: Average percent runtime overhead

<table>
<thead>
<tr>
<th>HasNext</th>
<th>UnsafeIter</th>
<th>Unsafe-MapIter</th>
<th>Unsafe-SyncColl</th>
<th>Unsafe-SyncMap</th>
<th>All Prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td>MOP</td>
<td>RV</td>
<td>TM</td>
<td>MOP</td>
<td>RV</td>
</tr>
<tr>
<td>antlr</td>
<td>1</td>
<td>4</td>
<td>-2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>bloat</td>
<td>2119</td>
<td>448</td>
<td>116</td>
<td>19194</td>
<td>569</td>
</tr>
<tr>
<td>chart</td>
<td>1</td>
<td>0</td>
<td>-2</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>eclipse</td>
<td>1</td>
<td>-4</td>
<td>-2</td>
<td>1</td>
<td>-5</td>
</tr>
<tr>
<td>fop</td>
<td>2</td>
<td>4</td>
<td>-2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>hsqldb</td>
<td>15</td>
<td>0</td>
<td>-3</td>
<td>13</td>
<td>-1</td>
</tr>
<tr>
<td>jython</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>luindex</td>
<td>-7</td>
<td>1</td>
<td>-1</td>
<td>4</td>
<td>-2</td>
</tr>
<tr>
<td>lusearch</td>
<td>3</td>
<td>-1</td>
<td>-2</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>pmd</td>
<td>70</td>
<td>26</td>
<td>-1</td>
<td>207</td>
<td>12</td>
</tr>
<tr>
<td>xalan</td>
<td>5</td>
<td>1</td>
<td>-1</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

Comparison of Tracematches (TM), JavaMOP (MOP), and RV: Average percent runtime overhead.
Overview

• Monitoring
• RV-Monitor Demo
• RV-Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV-Predict Demo
• RV-Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Why Prediction

• Concurrent programs are hard to analyze
  – 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
Our Solution

• Sliced Causality
  – General purpose technique to predict (bad) behaviors from correct runs
  – **Sound**: No false alarms

• RV–Predict
  – Tool implementing Sliced Causality
  – Allows for prediction of any property for which an algorithm exists
  – **Better** than tools specialized simply for data race or atomicity violations
Prediction Example

Property: “authenticate before access”

Main Thread:

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

Task Thread:

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

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

Property: “authenticate before access”

Main Thread:

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

... 

Task Thread:

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

... 

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

• Buggy \( S_4 \) can be executed before \( S_1 \)
• Low possibility to hit error in testing
Prediction Example

Property: “authenticate before access”

Main Thread:

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

Task Thread:

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

Can we predict the error even when the above execution is observed?
Yes! But not in the traditional way

• Buggy \( S_4 \) can be executed before \( S_1 \)
• Low possibility to hit error in testing
Special Case: Data Races

• Our techniques work for any behavioral property

• One of the simplest properties is race detection
  – Two accesses to a shared variable can take place concurrently
  – At least one of the accesses is a write
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Predictive Runtime Analysis

Search space
Predictive Runtime Analysis

- Search space
- Observed execution
Predictive Runtime Analysis

- Search space
- Observed execution
- Causal model
Predictive Runtime Analysis

Search space

Observed execution

Causal model

Inferred executions

Bug
Predictive Runtime Analysis

More relaxed causal model yields more inferred executions
Traditional Predictive Runtime Analysis: Happens-Before

• Originally for distributed systems [Lamport–78]
  – Applied to shared memory systems by several authors

• Causal model = non-permutable pairs of events
  – \( \{ \text{intra-thread total orders} \} \cup \{ \text{causal dependencies} \} \)
  – Causal dependency: if two events access the same location and one writes it, then their execution order matters

• Inferred executions = extending the causal model
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:

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

Observed execution: \( s_3 \ s_1 \ s_2 \ s_4 \)
**Happens–Before Works... If Lucky**

*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: $s_3 \; s_1 \; s_2 \; s_4$

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: \( s_2 < s_3 \). No bug found …

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

• Relaxes the Happens-Before causal model
  – Formally proved in [chen–rosu–07]
• How? Focus on the property
• Use static information about the program
• Remove events and causalities irrelevant to the property
  – Smaller and more relaxed causal model
  – (Exponentially) more inferred executions
  – Better predictive capability
Sliced Causality

• Start with those events relevant to the property

• Add events on which they are control dependent (transitively, intrathread)

• Add events on which they are data dependent (transitively, interthread)
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
Static Information: Control Scope

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

\[
\begin{align*}
  s_1 & : \text{if (flag) } \{ \\
    s_2 & : \ldots \\
    \} \text{ else } \{ \\
    s_3 & : \ldots \\
    \} \\
  s_4 & : \ldots \\
\end{align*}
\]

• Extends to other control statements – break/continue, return, exceptions

\[
\begin{align*}
  s_0 & : i=0; \\
  s_1 & : \text{while (i<3) } \{ \\
    s_2 & : \ldots \\
    s_3 & : i++ \\
    \} \\
  s_4 & : \ldots \\
  s_1 & : \text{while (!flag) } \{ \\
    s_2 & : \ldots \\
    \} \\
  s_3 & : \ldots \\
\end{align*}
\]
Static Information: Control Scope

• S₂ is in the control scope of S₁ if its execution depends on a choice at S₁

• Extends to other control statements – break/continue, return, exceptions
Static Information: Control Scope

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

• Extends to other control statements
  – break/continue, return, exceptions

```
s_0: i=0;
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: ...
```
Slice Causality Works!

Property: “authenticate before access”

Main Thread:

s₁: resource.authenticate()
s₂: flag.value = true;

Task Thread:

s₃: if (! flag.value)
    Thread.yield();
    s₄: resource.access();

Observed execution: s₁ s₂ s₃ s₄

Only s₁ and s₄ directly relevant to the property
Slice Causality Works!

Property: “authenticate before access”

Main Thread:

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

Task Thread:

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

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

Only \( s_1 \) and \( s_4 \) directly relevant to the property

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:

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

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

Property: “authenticate before access”

Main Thread:

$s_1$: resource.authenticate()

$s_2$: flag.value = true;

Task Thread:

$s_3$: while (! flag.value)

Thread.yield();

$s_4$: resource.access();

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

Execution of $s_4$ depends on flag.value being true at $s_3$

causal dependency $s_2 < s_3$ matters

Sliced causality: $s_1 < s_2 < s_3 < s_4$, no false alarm!
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
RV-Predict Pipeline

Original Program

Static Analyzer

Instrumented Program

JVM

Recorded Trace

Preprocessor

Complete Trace

Structural Information

Property

Predicted Violations: Counter-Examples
RV-Predict Pipeline

Original Program

Static Analyzer

Instrumented Program

JVM

Recorded Trace

Preprocessor

Complete Trace

Trace Slicer

Sliced Trace

Property

Predicted Violations: Counter-Examples
RV-Predict Pipeline

Original Program

Static Analyzer

Instrumented Program

JVM

Recorded Trace

Preprocessor

Complete Trace

Trace Slicer

Vector Clock Calculator

Sliced Trace

Causal Model

Property

Predicted Violations: Counter-Examples
RV–Predict Pipeline

Original Program

Static Analyzer

Instrumented Program

JVM

Recorded Trace

Preprocessor

Complete Trace

Trace Slicer

Property

Sliced Trace

Vector Clock Calculator

Causal Model

Property Checker

Predicted Violations: Counter-Examples

Counter-Examples
Overview

• Monitoring
• RV–Monitor Demo
• RV–Monitor Techniques and Implementation
  – Monitor Synthesis
  – Parametric Monitoring
  – Optimizations

• Prediction
• RV–Predict Demo
• RV–Predict Techniques and Implementation
  – Sliced Causality
  – Pipeline
  – Race Prediction
Data Race Prediction

• Consider all pairs of accesses in the trace
  – We actually do something smarter
• Check if either access is a write
  – We are not worried about read–read races
• Check if they have incomparable VCs
  – Incomparable VCs means accesses could be reordered
• If they have different lock sets then race found
## RV-Predict Performance

### jPredictor vs. RV-Predict

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Real Time</th>
<th>Disk Usage</th>
<th>Real Time</th>
<th>Disk Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>-</td>
<td>0:02.57</td>
<td>236K</td>
<td>0:06.07</td>
<td>364K</td>
</tr>
<tr>
<td>elevator</td>
<td>-</td>
<td>5:55.29</td>
<td>63M</td>
<td>1:20.31</td>
<td>864K</td>
</tr>
<tr>
<td>tsp</td>
<td>map4 2</td>
<td>4:24.44</td>
<td>16M</td>
<td>1:45.22</td>
<td>744K</td>
</tr>
<tr>
<td>tsp</td>
<td>map5 2</td>
<td>8:12.31</td>
<td>17M</td>
<td>2:45.28</td>
<td>868K</td>
</tr>
<tr>
<td>tsp</td>
<td>map10 2</td>
<td>&gt; 3 hours</td>
<td>&gt; 230M</td>
<td>33:45.32</td>
<td>2.8M</td>
</tr>
<tr>
<td>huge</td>
<td>-</td>
<td>crash</td>
<td>crash</td>
<td>0:42.22</td>
<td>13M</td>
</tr>
<tr>
<td>medium</td>
<td>-</td>
<td>crash</td>
<td>crash</td>
<td>0:06.12</td>
<td>840K</td>
</tr>
<tr>
<td>small</td>
<td>-</td>
<td>crash</td>
<td>crash</td>
<td>0:05.99</td>
<td>292K</td>
</tr>
<tr>
<td>mixedlockshuge</td>
<td>-</td>
<td>&gt; 2 hours</td>
<td>&gt; 250M</td>
<td>0:05.68</td>
<td>2.9M</td>
</tr>
<tr>
<td>mixedlocksbig</td>
<td>-</td>
<td>4:39.08</td>
<td>25M</td>
<td>0:05.68</td>
<td>496K</td>
</tr>
<tr>
<td>mixedlocksmedium</td>
<td>-</td>
<td>0:08.92</td>
<td>2.7M</td>
<td>0:07.25</td>
<td>308K</td>
</tr>
<tr>
<td>mixedlockssmall</td>
<td>-</td>
<td>0:05.46</td>
<td>1.5M</td>
<td>0:05.67</td>
<td>296K</td>
</tr>
</tbody>
</table>
Conclusion

• RV-Monitor is a generic yet efficient monitoring system
  – Extensible logic framework: FSM, ERE, PTLTL, FTLTL, LTL, CFG, PTCaRet, …

• RV-Predict provides very efficient causal predict of generic properties
  – Race detection, atomicity violations, monitoring properties