Outline

• Introduction
• JavaMOP
• Practical Parametric Monitoring
  o Efficient Parametric Monitoring
  o Scalable Parametric Monitoring
  o Expressive Parametric Monitoring
• Evaluation
• Multi-threaded Unit Testing
• Conclusion
Reliable Software is Important
Runtime Monitoring

- Feasible technique for ensuring software reliability
- Observe run of a program

- Analyze execution trace against desired properties
- React/report using handlers (if needed)
Parametric Monitoring

- Runtime monitoring with parameterized events

![Diagram showing the process of executing and observing a program, with events being analyzed to produce an execution trace.](image)
Applications of Parametric Monitoring

- Development
  - Debugging
  - Testing

- Deployment
  - Security
  - Reliability
  - Runtime Verification

- There are many academic tools
- However, there are not many real world applications
  - There are many challenges to overcome
Challenges

1) Efficiency

- Overall, Reasonable Overhead [ASE’09], [PLDI’11]
  TM-07 – 15/44 cases generated ≥10% overhead
  MOP-07 – 9/66 cases generated ≥10% overhead

- Excessive Overheads in Corner Cases
  TM-07: >1300% for bloat
  MOP-07: >400% for bloat

- Static analysis is limited
  - Fomalism-Dependent
  - Only reduce the number of points to monitor

1) Efficiency

2) Scalability
   - All parametric monitoring systems focus on monitoring a single property
   - MOP-07 and TM-07 cannot monitor large numbers of properties simultaneously
   - In Real Usages → Multiple Properties
Challenges

1) Efficiency

2) Scalability

3) Expressiveness
   - For performance reasons, many parametric systems choose either:
     - Hardwired Logical Formalism (e.g. TM-07)
     - Limitations on Parameters (e.g. MOP-07)
My PhD Thesis

• Practical Parametric Monitoring Techniques
  o Efficient Parametric Monitoring Techniques (Chap. 3)
  o Scalable Parametric Monitoring Techniques (Chap. 4)
  o Expressive Parametric Monitoring Techniques (Chap. 5)
  ➞ Integrated into JavaMOP

• IMUnit: Improved Multi-threaded Unit Testing (Chap. 6)
  o Framework for multi-threaded unit testing
  o Uses JavaMOP for monitoring/enforcing thread scheduling
## Why JavaMOP?

<table>
<thead>
<tr>
<th>Approach</th>
<th>Language</th>
<th>Logic</th>
<th>Scope</th>
<th>Mode</th>
<th>Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hawk</td>
<td>Java</td>
<td>Eagle</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>J-Lo</td>
<td>Java</td>
<td>ParamLTL</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>Jass</td>
<td>Java</td>
<td>assertions</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>JavaMaC</td>
<td>Java</td>
<td>PastLTL</td>
<td>class</td>
<td>outline</td>
<td>violation</td>
</tr>
<tr>
<td>jContractor</td>
<td>Java</td>
<td>contracts</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>JML</td>
<td>Java</td>
<td>contracts</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>JPaX</td>
<td>Java</td>
<td>LTL</td>
<td>class</td>
<td>offline</td>
<td>violation</td>
</tr>
<tr>
<td>P2V</td>
<td>C, C++</td>
<td>PSL</td>
<td>global</td>
<td>inline</td>
<td>validation/violation</td>
</tr>
<tr>
<td>PQL</td>
<td>Java</td>
<td>PQL</td>
<td>global</td>
<td>inline</td>
<td>validation</td>
</tr>
<tr>
<td>PTQL</td>
<td>Java</td>
<td>SQL</td>
<td>global</td>
<td>outline</td>
<td>validation</td>
</tr>
<tr>
<td>Spec#</td>
<td>C#</td>
<td>contracts</td>
<td>global</td>
<td>inline/offline</td>
<td>violation</td>
</tr>
<tr>
<td>RuleR</td>
<td>Java</td>
<td>RuleR</td>
<td>global</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>Temporal Rover</td>
<td>Several</td>
<td>MiTL</td>
<td>class</td>
<td>inline</td>
<td>violation</td>
</tr>
<tr>
<td>Tracematches</td>
<td>Java</td>
<td>Reg. Exp</td>
<td>global</td>
<td>inline</td>
<td>validation</td>
</tr>
<tr>
<td>JavaMOP</td>
<td>Java</td>
<td>Several</td>
<td>global/class</td>
<td>inline</td>
<td>several</td>
</tr>
</tbody>
</table>
Outline and Contributions

• **JavaMOP**
  - I refactored and rewrote JavaMOP (about 80%)
  - **Efficient**
    - Resulted in MOP-11
    - See the next slide for results
  - **Scalable**
    - Resulted in MOP-12
    - See the next slide for results
  - **Expressive**
    - More logical formalisms – PTCaRet, CFG
    - No limitation on parameters

• **IMUnit: Improved Multi-threaded Unit Testing**
Outline and Contributions

- Monitoring 5 **extreme** specifications

<table>
<thead>
<tr>
<th></th>
<th>MOP-07</th>
<th>MOP-11</th>
<th>MOP-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Runtime Overhead</td>
<td>54%</td>
<td>21%</td>
<td>17%</td>
</tr>
<tr>
<td>Maximum Runtime Overhead</td>
<td>858%</td>
<td>251%</td>
<td>269%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage</td>
<td>1.00</td>
<td>0.35</td>
<td>0.29</td>
</tr>
</tbody>
</table>

- Monitoring 137 specifications, on average

<table>
<thead>
<tr>
<th></th>
<th>MOP-07</th>
<th>MOP-11</th>
<th>MOP-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Overhead (Sum of Individual)</td>
<td>&gt;500%</td>
<td>243%</td>
<td>180%</td>
</tr>
<tr>
<td>Runtime Overhead (Simultaneous)</td>
<td>N/A</td>
<td>262%</td>
<td>123%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Sum)</td>
<td>&gt;3.00</td>
<td>1.00</td>
<td>0.72</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Simultaneous)</td>
<td>N/A</td>
<td>1.34</td>
<td>0.69</td>
</tr>
</tbody>
</table>
My Research

• JavaMOP
  o Collaborators: Feng Chen, Patrick Meredith, Choonghwan Lee, Dennis Griffith, Soha Hussein, Grigore Rosu
  o ICSE ‘12 Demo, PLDI ‘11, J.STTT ‘11, J. of ASE ‘10, ASE ‘09, ICICIS ‘09, ASE ‘08

• IMUnit
  o Collaborators: Vilas Jagannath, Milos Gligoric, Qingzhou Luo, Darko Marinov, Grigore Rosu
  o FSE ‘11, IWMSE ‘10(ICSE Workshop)
JavaMOP Overview

- Parametric Properties
  - 137 specs

- JavaMOP
  - 39 secs

- AspectJ Code for Monitoring

- Program
  - DaCapo Benchmarks

- Program with Monitor
  - 26 mins

- AspectJ Compiler

- All timings are measured on a Core 2 Duo E8500 (3.16GHz) machine with JavaMOP 2012 version, ajc and the DaCapo benchmark suite.
Parametric Properties

- Properties Referring to Object Instances
- The Following Property Describes a Bad Behavior between Each Vector $v$ and Enumeration $e$:

  $$\text{update}(v) \quad \text{next}(e) \quad \text{update}(v)$$

  $$\text{create}(v, e) \quad \text{update}(v) \quad \text{next}(e)$$

- Generalize Typestates
  - Typestates are Parametric Properties with One Parameter
import java.util.*;

SafeEnum(Vector v, Enumeration e) {
    event create after(Vector v) returning(Enumeration e) :
        call(Enumeration Vector+.elements()) && target(v) {
    }

    event update after(Vector v) :
        (call(* Vector+.remove*(..))
        || call(* Vector+.add*(..))
        || call(* Vector+.clear(..))
        || call(* Vector+.insertElementAt(..))
        || call(* Vector+.set*(..))
        || call(* Vector+.retainAll(..))) && target(v) {
    }

    event next before(Enumeration e) :
        call(* Enumeration+.nextElement()) && target(e){}

    ere : update* create next* update+ next
    @match {
        System.out.println("improper Concurrent Modification found!");
    }
}
Parametric Monitoring in JavaMOP

• Keep One Monitor for Each Parameter Instance
  o A parameter instance binds parameters to objects
  o E.g., $(v \mapsto v_2, \ e \mapsto e_3)$

• Each monitor knows nothing of parameters; operates exclusively on only one trace slice
Parametric Monitoring in JavaMOP

- Keep One Monitor for Each Parameter Instance
  - A parameter instance binds parameters to objects
  - E.g., $(v \leftarrow v_2, \ e \leftarrow e_3)$

- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

<table>
<thead>
<tr>
<th>Trace: create($v_1$, $e_1$) create($v_1$, $e_2$) next($e_1$) update($v_1$) next($e_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_1, e_1)$ Trace Slice: create next update</td>
</tr>
<tr>
<td>$(v_1, e_2)$ Trace Slice: create update next</td>
</tr>
</tbody>
</table>
Parametric Monitoring in JavaMOP

- Keep One Monitor for Each Parameter Instance
  - A parameter instance binds parameters to objects
  - E.g., \((v \leftrightarrow v_2, \ e \leftrightarrow e_3)\)
- Each monitor knows nothing of parameters; operates exclusively on only one trace slice

<table>
<thead>
<tr>
<th>Trace: create((v_1, e_1)) create((v_1, e_2)) next((e_1)) update((v_1)) next((e_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((v_1, e_1)) Trace Slice: create next update</td>
</tr>
<tr>
<td>((v_1, e_2)) Trace Slice: create update next</td>
</tr>
</tbody>
</table>

Challenge: Large numbers of monitors
Indexing Tree

- Multi-level Hash-Mapping from parameter instance to monitor(s)
- Constant time performance
- Self-Manage the bucket size
- Self-Clean up broken mappings
- WeakReference
  - Reference which does not block garbage collection of the referent object
Efficient Parametric Monitoring

• Summary
  o Enable Set Optimization
  o Indexing Cache
  o Monitor Garbage Collection
  o Efficient Support for Stack Properties

• Result Overview
  o Using the DaCapo benchmark suite against five extreme properties

<table>
<thead>
<tr>
<th></th>
<th>MOP-07</th>
<th>MOP-11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Runtime Overhead</td>
<td>54%</td>
<td>21%</td>
</tr>
<tr>
<td>Maximum Runtime Overhead</td>
<td>858%</td>
<td>251%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage</td>
<td>1.00</td>
<td>0.35</td>
</tr>
</tbody>
</table>
Techniques for Efficiency

• Life of a Monitor

Birth

Monitoring

Death
Techniques for Efficiency

- Life of a Monitor
  - Birth
  - Monitoring
  - Death

- Enable Set Optimization
  - Use static knowledge about the property to ignore parameter instances that can never reach the target states
Property Static Analysis (Enable Set)

- Property analysis tells us what parameter instances are useful.
- Our analysis is formalism independent; we only show it for FSM monitors (see paper for CFG monitors).

<table>
<thead>
<tr>
<th>The Current Event</th>
<th>Parameter Instances that must pre-exist</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>∅ or {v}</td>
</tr>
<tr>
<td>update</td>
<td>∅ or {v, e}</td>
</tr>
<tr>
<td>next</td>
<td>{v, e}</td>
</tr>
</tbody>
</table>

update(\textit{v})  \quad next(\textit{e})  \quad update(\textit{v})

create(\textit{v}, \textit{e})  \quad update(\textit{v})  \quad next(\textit{e})
Techniques for Efficiency

• Life of a Monitor

Birth

Monitoring

Death

• Indexing Caching
  • Caches the last retrieved monitor(s)
  • Utilize temporal locality (Cache hit rate: >90%)
Indexing Cache

- 5 consecutive access to the same monitor

Without Cache – 10 hashings

With Cache – 2 hashings, 5 cache refs
Techniques for Efficiency

- Life of a Monitor
  - Birth
  - Monitoring
  - Death

- Monitor Garbage Collection
  - Collect monitors that become unnecessary, earlier
Monitor Garbage Collection

- \((v_1, e_1) \implies \text{Obviously, the monitor dies}\)
- \((v_1, e_1) \implies ?\)

- The Monitor Will Never Reach to the Target State
  \(\implies\) Collect the monitor

- Apply the Enable Set Analysis in the Reversed Way
Scalable Parametric Monitoring

• Goal
  o Monitoring a large number of specifications **efficiently**

• Observation
  o There are likely to be multiple specifications for the same class
  o They share parameters and possibly events as well

⇒ Let’s share resources between specs
Scalable Parametric Monitoring

• Summary
  o Global WeakReference Table
  o Cache for Global WeakReference Table
  o Combining Indexing Tree
  o Eliminating HashEntry
  o Specification Activator

• Result Overview – Average Runtime Overhead
  o Using the DaCapo benchmark suite against 137 specifications from the Java API

<table>
<thead>
<tr>
<th></th>
<th>MOP-07</th>
<th>MOP-11</th>
<th>MOP-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Overhead (Sum of Individual)</td>
<td>N/A</td>
<td>243%</td>
<td>180%</td>
</tr>
<tr>
<td>Runtime Overhead (Simultaneous)</td>
<td>N/A</td>
<td>262%</td>
<td>123%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Sum)</td>
<td>N/A</td>
<td>1.00</td>
<td>0.72</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Simultaneous)</td>
<td>N/A</td>
<td>1.34</td>
<td>0.69</td>
</tr>
</tbody>
</table>
Global WeakReference Table

• WeakReference
  o Reference that does not block garbage collection of referent object
  o Indexing trees heavily use it
  o But, no sharing between different specs
  o **Expensive**: JVM involves its behavior

• Global WeakReference Table
  o Create/Query WeakReferences
  o Hashing result is saved in WeakReference

• Cache for Global WeakReference Table
  o Benefits all specs sharing the same parameter
Combining Indexing Tree

- Smaller number of indexing trees
  ➞ Smaller number of broken mappings to clean up

![Diagram of combining indexing trees]
Specification Activator

• What if there is no Vector usage, but Enumeration usages?
  o No need to monitor the property about Vector
  o Generate unnecessary runtime/memory overhead

• Solution: Specification Activator
  o Creation events activate the specification
  o Deactivated specifications cost no overhead
Expressive Parametric Monitoring – Logics

• Previously Supported Logical Formalisms:
  o ERE – Extended Regular Expression
  o FSM – Finite State Machine
  o PTLTL – Past Time Linear Temporal Logic
  o FTLTL – Future Time Linear Temporal Logic

• Newly Introduced Logical Formalisms:
  o PTCaRet – Past Time Linear Temporal Logic with Calls and Returns
  o CFG – Context Free Grammars
Expressive Parametric Monitoring – Parameters

• Previously, the first event must initiate all parameters
  o For Performance Reasons
  o Handling other cases is non-trivial

• Generic Parametric Monitoring Algorithm
  o No such limitation
  o Copies monitor(s) upon events with partial parameters
  o Problem: a huge number of monitor instances
  o Solution: the enable set optimization
Experimental Settings

• DaCapo Benchmark Suite
  o Collection of open-source, real world applications

• Monitoring Systems
  o TM-07 – Tracematches
  o MOP-07 – JavaMOP before the dissertation
  o MOP-11 – JavaMOP implementing efficiency techniques
  o MOP-12 – JavaMOP implementing scalability techniques

• Specifications
  o Five extreme specifications from various papers
  o 137 specifications from the Java API Doc of three main packages (io, lang, util)
Efficiency

- Monitoring 5 extreme specifications
- Average for 18 benchmarks in DaCapo

<table>
<thead>
<tr>
<th></th>
<th>TM-07</th>
<th>MOP-07</th>
<th>MOP-11</th>
<th>MOP-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Runtime Overhead</td>
<td>309%</td>
<td>54%</td>
<td>21%</td>
<td>17%</td>
</tr>
<tr>
<td>Maximum Runtime Overhead</td>
<td>11258%</td>
<td>858%</td>
<td>251%</td>
<td>269%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage</td>
<td>0.12</td>
<td>1.00</td>
<td>0.35</td>
<td>0.29</td>
</tr>
</tbody>
</table>
Runtime Overhead – bloat

- HasNext
- UnsafeIter
- UnsafeMapIter
- UnsafeSyncColl
- UnsafeSyncMap

Graph showing runtime overhead with categories from TM'07, MOP'07, MOP'11, MOP'12.
Memory Overhead – bloat

![Graph showing memory overhead for different classes and years: TM'07, MOP'07, MOP'11, MOP'12. The class names include HasNext, UnsafeIter, UnsafeMapIter, UnsafeSyncColl, UnsafeSyncMap. The x-axis represents the classes, and the y-axis represents memory overhead.]
Memory Overhead – avrora
Runtime Overhead – pmd

The chart above illustrates the runtime overhead of various methods and classes within the pmd library, specifically focusing on HasNext, UnsafeIter, UnsafeMapIter, UnsafeSyncColl, and UnsafeSyncMap.

- **HasNext**: Shows minimal overhead with values close to zero for all years except TM'07.
- **UnsafeIter**: Displays a higher overhead than HasNext, particularly notable in TM'07 and TM'07.
- **UnsafeMapIter**: Exhibits a significant overhead, especially in TM'07 and TM'07.
- **UnsafeSyncColl**: Similar to UnsafeMapIter, with notable overhead in TM'07 and TM'07.
- **UnsafeSyncMap**: Shows moderate overhead, with values increasing in TM'07 and TM'07.

The data suggests that while some methods have consistent overhead, others show variations across different years.
Memory Overhead – pmd

![Graph showing memory overhead for different methods and iterations.]

- **HasNext**
- **UnsafeIter**
- **UnsafeMapIter**
- **UnsafeSyncColl**
- **UnsafeSyncMap**

Legend:
- TM’07
- MOP’07
- MOP’11
- MOP’12

**Introduction**
JavaMOP
Practical Parametric Monitoring
Evaluation
Multi-threaded Unit Testing

**Settings**
Efficiency
Corner Cases
Scalability

Evaluation
Multi-threaded Unit Testing
Scalability

- Monitoring 137 specifications
- Average for 15 benchmarks

<table>
<thead>
<tr>
<th></th>
<th>TM-07</th>
<th>MOP-07</th>
<th>MOP-11</th>
<th>MOP-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Overhead (Sum of Individual)</td>
<td>&gt;3000%</td>
<td>&gt;500%</td>
<td>243%</td>
<td>180%</td>
</tr>
<tr>
<td>Runtime Overhead (Simultaneous)</td>
<td>N/A</td>
<td>N/A</td>
<td>262%</td>
<td>123%</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Sum)</td>
<td></td>
<td>&gt;3.00</td>
<td>1.00</td>
<td>0.72</td>
</tr>
<tr>
<td>Relative Peak Memory Usage (Simultaneous)</td>
<td>N/A</td>
<td>N/A</td>
<td>1.34</td>
<td>0.69</td>
</tr>
</tbody>
</table>
Multi-threaded Unit Testing

- Prev Multi-Threaded Unit Tests are Unreliable and Slow
  - Schedules are implicitly specified by using time delays (Thread.sleep())

- Improved Multi-threaded Unit Testing (IMUnit)
  - Separate schedules and functionality testing code
    - Better modularity, readability, and reliability
    - Easier to specify multiple schedules for a unit test
    - Faster tests

- Executing Unit Tests in IMUnit
  - JavaMOP monitors/enforces thread schedules by observing the order of events
public void testTakeWithAdd() {
...
q = new ArrayBlockingQueue<Integer>(1);
Thread addThread = new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        Thread.sleep(150);
        q.add(2);
      }
    }, "addThread").start();
Thread.sleep(50);
Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());
taken = q.take();
assertTrue(taken == 2 && q.isEmpty());
...}

@Schedule("afterAdd1->beforeTake1, [beforeTake2]->beforeAdd2")
public void testTakeWithAdd() {
...
q = new ArrayBlockingQueue<Integer>(1);
Thread addThread = new Thread(
    new CheckedRunnable() {
      public void realRun() {
        q.add(1);
        @Event("afterAdd1")
        @Event("beforeAdd2")
        q.add(2);
      }
    }, "addThread").start();
@Event("beforeTake1")
Integer taken = q.take();
assertTrue(taken == 1 && q.isEmpty());
@Event("beforeTake2")
taken = q.take();
assertTrue(taken == 2 && q.isEmpty());
...}
Conclusion

• Parametric Monitoring Becomes Practical
  o Efficient
    • Generates less runtime/memory overhead
  o Scalable
    • For the first time, capable of monitoring $\geq 130$ props **efficiently**
  o Expressive
    • Multiple Logical Formalisms
    • No Limitation on Parameters

• Future Work
  o More Applications
  o Structured Specifications
  o Integrating JavaMOP into an AspectJ Compiler
    • Formalism-Independent Static Analysis
    • More Pointcuts – Capture More Program points as Events
Thank You

Questions?