Monitoring-Oriented Programming

Seyed Mohammad Mehdi Ahmadpanah
smahmadpanah@aut.ac.ir
ceit.aut.ac.ir/~ahmadpanah

Supervisor
Dr. Mehran S. Fallah

Formal Security Lab.
CEIT@AUT

Mar. 12, 2017
Outline

• Introduction and Preliminaries
  – Runtime Verification and Monitor
• Related Work
  – Aspect-Oriented Programming and Design By Contract
• Monitoring-Oriented Programming
  – Concepts
  – Logic Plugins
  – Parametric Monitoring
  – JavaMOP and Extensions
  – In Relation to Enforceable Security Policies
  – Examples and Demo
• Conclusion and Future Work
What does “Monitor” mean?

• (noun) a device used for observing, checking, or keeping a continuous record of something - Oxford
• (noun) someone who gives a warning so that a mistake can be avoided – Concise
• (verb) observe and check the progress or quality of something over a period of time; keep under systematic review - Oxford
• (verb) keep an eye on - Concise
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.
Runtime Verification and Monitoring

• Aims at achieving benefits of both testing and formal verification, avoiding their pitfalls

• Question: what do we really want ... ?
  A. To prove a program correct?
  B. To achieve correct execution?
  – Often “A ⇒ B”, but isn’t the price too high?
  – Focusing on B, one sometimes also gets A

• Instead of proving systems correct, observe, check and control their execution
General Idea of Runtime Verification

Program(a,b,...)

Run it; observe & check Spec(a,b,...)

Correct it (if possible)

yes

no
Specification and Programming

Specifcation

Static Analysis

Test Input/ Schedule Generation

Dynamic Analysis

Runtime Verification

Program

Input

Output
Field still has many Names

- Runtime Verification
- Runtime Monitoring
- Runtime Checking
- Runtime Result Checking
- Runtime Reflection
- Design By Contract
- Runtime Analysis
- Dynamic Analysis
- Trace Analysis
- Monitoring-Oriented Programming
Comparison of Techniques

- Giving up on coverage to write better specifications and scale
What is Trace?

• A formal view of an execution is to consider it as a sequence $\sigma$ of program states:

$$\sigma = s_1 \ s_2 \ s_3 \ ... \ s_n$$

• Past in known vs. Future is unknown
The Cycle

- Instrumentation Language (I)
- Property Specification Language (P)
- Reaction Language (R)
Property Languages

- Programming Languages
  - Program (built-in algorithms focused on specific problem)
    - Data Race Detection
    - Atomicity Violation
    - Deadlock Detection
- Formal Languages
  - Design By Contract (pre/post condition)
  - State Machines
  - Regular Expressions
  - Grammars e.g. Context-Free
  - Temporal Logic (past time, future time)
  - Process Algebra (CSP/CCS)
  - Full Fledged Formal Specification Languages e.g. Z
  - Graphical Languages e.g. UML
Monitoring Integration

• Offline
  – Analyzing log file / trace dump
Monitoring Integration (Cont.)

- **Online**
  - **Online**
    - **Online**
      - **Online**
        - **Online**
          - **Online**

- **Monitorings run in parallel with application**
  - **Synchronous** (Application waits for response)
  - **Asynchronous** (Buffered communication)
Monitoring Integration (Cont.)

- **Offline**
  - Analyzing log file / trace dump

- **Online**
  - Outline
    - Monitor runs in parallel with application
      - Synchronous (Application waits for response)
      - Asynchronous (Buffered communication)
  - Inline
    - Monitoring code is embedded into the application
Violation vs. Validation

• **Violation**
  
  – checking that the systems conforms to a property, and reporting when the property is “violated”.

• **Validation**
  
  – Stating property in negative form: *what we do not want to happen*, Reporting when the bad property gets “validated”.

  – Or, it is a good property and we just want to log whenever something good happens.
Example - Violation

- Property:
  \[(\text{green yellow red})^*\]
Example - Validation

- Property:
  green red

![Diagram showing validation process with traffic lights and validation sign]

- Past: green
- Now: red
- Future: red
Challenges

• Code Instrumentation
• Definition of Specification Languages
• Creation of Efficient Monitors from Specification
• Minimize Impact on Monitored System
• Integrate Static and Dynamic Analysis
• Controlling the Application in case of Violation/Validation
Aspect-Oriented Programming

- Aims to increase modularity by allowing the separation of cross-cutting concerns.
  - Example: logging
    - Crosscut all logged classes and methods
Aspect-Oriented Programming (cont.)

code tangling:
one module
many concerns

example: logging

code scattering:
one concern
many modules
Aspect-Oriented Programming (cont.)

code tangling:
one module
many concerns

code scattering:
one concern
many modules

code example: logging
AOP Concepts

• An aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches).
AOP Concepts (cont.)

- **Aspect**: a modularization of a concern that cuts across multiple classes
  - Example of crosscutting concern: Transaction management
- **Join point**: a point during the execution of a program
  - The execution of a method or the handling of an exception
- **Advice**: action taken by an aspect at a particular join point
  - Including "around," "before" and "after"
- **Pointcut**: a predicate that matches join points.
  - Advice is associated with a pointcut expression and runs at any join point matched by the pointcut
- **Weaving**: linking aspects with other application types or objects to create an advised object
  - This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime
Simplified View of AOP

```
while(more())
{
  ...
  send(msg);
  ...
}
```

```
program

while(more())
{
  ...
  check(msg);
  send(msg);
  ...
}
```

```
when send(msg)
{
  check(msg);
}
```

Weaver
Design By Contract

• Add semantic information to a program by specifying **assertions** regarding the program’s runtime state, and then checking the specification at runtime

  – *Jass*: A precompiler turns the assertion comments into Java code, and pre-/post- conditions, and class invariants

  – *jContractor*: A Java library allows programmers to associate contracts, consisting of precondition, postcondition, and invariant, with any Java classes or interfaces
What is Monitoring-Oriented Programming?

• Framework for **reliable** software development
  – Monitoring is basic design discipline
  – Recovery allowed and encouraged
  – Provides to programmers and hides under the hood a large body of formal methods knowledge/techniques
  – **Generic** for different languages and application domains
    • Language- and Logic-independent
MOP Approach to Monitoring

Diagram showing the relationship between running a program, taking snapshots, and monitoring properties. The diagram illustrates how concrete traces are abstracted and filtered to identify monitors.
MOP Architecture

JavaMOP

GUI IDE
Web-Based Interface
Command-line Interface
Java Specification Processor
JavaLTL
JavaCFG
JavaERE

Interfaces
Specification Processors
Language Translator

BusMOP

Web-Based Interface
Command-line Interface
HDL Specification Processor
HDL LTL
HDL CFG
HDL ERE

Interfaces
Specification Processors
Language Translator

Logic Repository

Logic Plugin Manager

Logic Plugins
LTL Logic Plugin
CFG Logic Plugin
ERE Logic Plugin

Logic Plugin Manager
Program Transformation Flow in MOP

- Interface
- Annotated Programs
- Annotation Processor
- MoP Annotations
- Language Shell
- Formal Specification
- Logic Engine
- Monitored Programs
- Monitoring Code
- Pseudo Code

: Software Artifacts
: System Modules
: Dataflow
One can understand MOP from at least three perspectives:

1. Improving reliability of a system by monitoring its requirements against its implementation at runtime. By generating and integrating the monitors automatically rather than manually.

2. An extension of programming languages with logics.

3. A lightweight formal method
   - by not letting it go wrong at runtime.
MOP (cont.)

- Same idea as design by contract: specifications are written as comments in code. Monitors are generated from specs
- Philosophy: no silver-bullet logic for specs
- MOP logic plugins (a subset):
  - ERE (Extended Regular Expressions)
  - CFG (Context-Free Grammars)
  - PtLTL (Past-time LTL) and FtLTL (Future-time LTL)
  - JML (fragment of Java Modeling Language)
  - ATL (Allen Temporal Logic)
  - Jass (The CSP Process algebra)
- Generic wrt. parameters
  - Provide a plugin for a propositional logic, and MOP does the rest wrt. data parameterization
  - Makes designing a new logic extremely easy compared to other frameworks
Instances of MOP

MOP is generic in both specification formalisms (logics) and programming languages.

<table>
<thead>
<tr>
<th>Languages</th>
<th>FSM</th>
<th>ERE</th>
<th>CFG</th>
<th>PTLTL</th>
<th>LTL</th>
<th>PTCaRet</th>
<th>SRS</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaMOP</td>
<td>JavaFSM</td>
<td>JavaERE</td>
<td>JavaCFG</td>
<td>JavaPTLTL</td>
<td>JavaLTL</td>
<td>JavaPTCaret</td>
<td>JavaSRS</td>
<td>...</td>
</tr>
<tr>
<td>BusMOP</td>
<td>BusFSM</td>
<td>BusERE</td>
<td>...</td>
<td>BusPTLTL</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ROSMOP</td>
<td>ROSFSM</td>
<td>...</td>
<td>ROSCFG</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# Examples of Runtime Verification Systems

<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>violation/validation</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. Ex.</td>
<td>global</td>
<td>inline</td>
<td>validation</td>
</tr>
</tbody>
</table>
How does MOP work?

• **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
Monitors verify abstract traces against desired properties; can be dynamically created or destroyed
MOP: Extensible Logic Framework

• Can we generate monitors *automatically* from specifications?
  – Generic in specification formalisms

• **Logic Plugin**: monitor synthesis components for different logics as plugins

• **Current Plugins**
  – FSM, ERE, PTLTL, FTLTL, ATL, JML, PtCaRet, CFG,…

• Also, **Raw** specifications are allowed
MOP Syntax

\(\langle\text{Specification}\rangle ::= \text{ /* @} \langle\text{Header}\rangle \langle\text{Body}\rangle \langle\text{Handlers}\rangle \text{ @} */}\)

\(\langle\text{Header}\rangle ::= \langle\text{Attribute}\rangle^*[\text{scope} = \langle\text{Scope}\rangle][\text{logic} = \langle\text{Logic}\rangle]\)

\(\langle\text{Attribute}\rangle ::= \text{static} | \text{outline} | \text{offline} | \text{centralized}\)

\(\langle\text{Scope}\rangle ::= \text{global} | \text{class} | \text{interface} | \text{method}\)

\(\langle\text{Name}\rangle ::= \langle\text{Identifier}\rangle\)

\(\langle\text{Logic}\rangle ::= \langle\text{Identifier}\rangle\)

\(\langle\text{Body}\rangle ::= [\langle\text{Name}\rangle][(\langle\text{Parameters}\rangle)]\{\langle\text{LogicSpecificContent}\rangle\}\)

\(\langle\text{Parameters}\rangle ::= (\langle\text{Type}\rangle \langle\text{Identifier}\rangle)^+\)

\(\langle\text{Handlers}\rangle ::= [\langle\text{ViolationHandler}\rangle] [\langle\text{ValidationHandler}\rangle]\)

\(\langle\text{ViolationHandler}\rangle ::= \text{violation handler} \{ \langle\text{Code}\rangle \}\)

\(\langle\text{ValidationHandler}\rangle ::= \text{validation handler} \{ \langle\text{Code}\rangle \}\)
/*@ 
scope = global 
logic = ERE 
SafeEnum (Vector v, Enumeration+ e) { 
  [String location = "";]
  event create<v,e>: end(call(Enumeration+.new(v,..))) with (e);
  event updatesource<v>: end(call(* v.add*(..))) \/
      end(call(* v.remove*(..))) \/ ...
    {location = @LOC;}
  event next<e>: begin(call(* e.nextElement()));
  formula : create next* updatesource+ next
} 
validation handler { System.out.println("Vector updated at " 
      + @MONITOR.location); }
@*/
FSM Plugin (Finite State Machine)

• Easy to use, yet powerful
• Many approaches/users encode important properties directly in finite state machines
• Monitoring FSM
  – Direct translation from an FSM specification to a monitor

Regular Expression (RE) ➔ Deterministic Finite State Automaton (DFA) ➔ Monitor (M)
FSM Plugin - Example

File Access Property

(open (read* + write*) close)*

fsm:
!s0[
  open -> s1
]
s1[
  read -> s3
  write -> s2
  close -> s0
]
s2[
  write -> s2
  close -> s0
]
s3[
  read -> s3
  close -> s0
]
ERF 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 expressions with complement (negation)
  - Specify properties non-elementarily more compactly
  - More complicated to monitor
Syntax for ERE

\[ E ::= \emptyset | \epsilon | A | E\ E | E^* | E+E | E&E | \neg E \]

Example - \( A = \{a,b,c\} \):

<table>
<thead>
<tr>
<th>expression</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>aab</td>
<td>{aab}</td>
</tr>
<tr>
<td>((ab)^*)</td>
<td>{\epsilon, ab, abab, ...}</td>
</tr>
<tr>
<td>((a+b)^* &amp; \neg(ab)^*)</td>
<td>words of randomly interleaved a’s and b’s but not only cleanly alternating (ababab...) {a, aa, abba, bbbb, ...}</td>
</tr>
</tbody>
</table>
Semantics for ERE

\[
\begin{align*}
L(\emptyset) & = \{\} \\
L(\epsilon) & = \{\epsilon\} \\
L(\alpha) & = \{\alpha\} \\
L(E_1 E_2) & = \{\omega_1 \omega_2 \mid \omega_1 \in L(E_1) \land \omega_2 \in L(E_2)\} \\
L(E^*) & = \{\omega_1 \omega_2 \ldots \omega_i \ldots \omega_n \mid \omega_i \in L(E)\} \\
L(E_1 + E_2) & = L(E_1) \cup L(E_2) \\
L(E_1 \& E_2) & = L(E_1) \cap L(E_2) \\
L(\neg E) & = A^* \setminus L(E)
\end{align*}
\]
Limitations of Regular Expressions for Specification

• Convenient for *brief* properties

• Less convenient on very state-full problems, where all good or bad behaviors must be formulated

• Can only express regular properties, cannot count an *apriori* unknown number of times, e.g. lock-release property
LTL Plugin
(Linear Temporal Logic)

• MOP includes both a past-time plugin (PTLTL) and a future-time plugin (FTLTL) for LTL

• PTLTL uses a dynamic programming algorithm, low resources, suitable for hardware

• FTLTL uses a transformed/optimized Buchi automata construction, but still may generate large monitors that cannot be stored
Syntax for LTL

- **PastLTL**

\[
F ::= \text{true} \mid \text{false} \mid A \mid \neg F \mid F \ op \ F
\]

\[
\neg F \mid \diamond F \mid \square F \mid F \ S_s \ F \mid F \ S_w \ F
\]

previous \hspace{1cm} eventually \hspace{1cm} always \hspace{1cm} since

\[
\uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w
\]

start \hspace{1cm} end \hspace{1cm} F \ butnot \ F'

Example: one cannot dial when the phone is busy or connected

\[
\square(\uparrow (\text{dialing}) \rightarrow \neg \diamond(\text{busyTone} \lor \text{connected}))
\]
... (Java code A) ...

/*@ FTLTL

... (Java code A) ...
switch(FTLTL_1_state) {
case 1:
    FTLTL_1_state = (tlc.state.getColor() == 3) ? 1 :
        (tlc.state.getColor() == 2) ? (tlc.state.getColor() == 1) ? -2 : 2 : 1; break;
case 2:
    FTLTL_1_state = (tlc.state.getColor() == 3) ? 1 :
        (tlc.state.getColor() == 1) ? -2 : 2; break ;
}
if (FTLTL_1_state == -2) { ...(Violation Handler)... }
// Validation Handler is empty
... (Java code B) ...

Example: after green yellow comes

□(green → ¬red ∪ yellow)
The Language Hierarchy

\[ S \rightarrow \epsilon \mid aSb \]

Context-Free

Regular

Temporal

\[ a \land \left[ (a \rightarrow \epsilon b) \lor \left[ (b \rightarrow \epsilon a) \right] \right] \]
CFG Plugin (Context-Free Grammar)

- Most systems support **finite** state monitors
  - **Regular** languages
  - **Linear** temporal logics

- These cannot monitor **structured** properties:
LR(1) 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
• 1 in LR(1) denotes a look-ahead of one token for Reductions

• Makes it a good candidate for CFG monitor synthesis!
Recall – MOP Monitoring Model

Program Execution

Abstract Trace

Observation/Abstraction

Verification

Monitors

Monitors can be dynamically created or destroyed – why?

Parametric Monitoring
Parametric Properties

• Imperatively 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 updatesource* next
    @match { System.out.println("Failed Enumeration!"); } }
```
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:
...
update(v)
v.remove(0);
...

- Parametric traces: traces containing events with parameters
Checking Parametric Traces

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

**no parametric monitor**

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

Diagram of a system with parametric operations and a non-parametric monitor.
Parametric Trace Slicing

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

• Naïve Monitoring
  – Every parametric trace contains multiple non-parametric trace slices, each corresponding to a particular parameter binding
    • NOT Efficient

• Online Parametric Trace Slicing
  – 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?
### Online Parametric Trace Slicing - Example

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

<table>
<thead>
<tr>
<th>v1</th>
<th>v1, e1</th>
<th>v2</th>
<th>v2, e1</th>
</tr>
</thead>
<tbody>
<tr>
<td>update(v1)</td>
<td></td>
<td>update</td>
<td></td>
</tr>
<tr>
<td>createEnum</td>
<td></td>
<td></td>
<td>v1, e2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update(v2)</td>
<td></td>
<td>useEnum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>useEnum(e1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>createEnum(v1,e2)</td>
<td></td>
<td>useEnum</td>
<td></td>
</tr>
</tbody>
</table>

For given parameters *(v, e)*
Parametric Monitors

• Monolithic (Centralized) Monitors
  – Bound to specific formalisms/checking mechanisms
  – Limited Expressiveness, specific to application domains

• Decentralized Monitors
  – Formalism-independent
  – More expressive
  – Adaptive to different domains
  – Optimization and better performance
Monitors Indexing

• How can we retrieve all needed monitor instances efficiently?
  – Centralized and Naïve implementation: very inefficient
Monitors Indexing (cont.)

- Solution: Decentralized Indexing
JavaMOP

• Layered architecture for **extensibility**
• Supports most logics provided by the MOP framework e.g. **FSM, ERE, PTLTL, FTLTL, PTCaRet, and CFG**
• **Efficient** support for generic universal parameters
  – Supports both *centralized* and *decentralized* indexing for better flexibility in practice

Overhead <10% in most cases; close to hand-optimized
More expressivity and less overhead in comparison with other tools
Evolution-Aware MOP

• Extend MOP to support *multiple software versions*

• The key idea:
  To monitor only the parts of code that changed between versions
  
  – inspired by *Regression Test Selection (RST)*
    • Improving *efficiency* and *usability*
  
  – *Regression Property Selection (RPS)* and *Regression Monitor Selection (RMS)*
JavaMOP and Security Policies

- **Inlined Reference Monitor (IRM) vs. Runtime Verification**
  - Security specification vs. System specification

- The usage of JavaMOP as an IRM system to specify and enforce security policies
  - Highly expressive and More efficient
    - e.g. Chinese Wall in JavaMOP using CFG
  - Should **not** be used for low-level security policies
JavaMOP and Security Policies (cont.)

- It monitors a single execution trace as it occurs
  - Cannot monitor nonproperties (e.g. Information Flow Policies)
  - Can enforce properties
    - Liveness properties cannot be expressed by any monitor
      - By bounding to a limit, they became safety policies
  - JavaMOP with AspectJ is able to rewrite the target program
    - A Program Rewriter
      - Can enforce RW-enforceable policies
        - Including EM-Policies and Satisfiable static policies
More Examples – SafeEnum in ERE

SafeEnum(Vector v, Enumeration e) {
    event create after(Vector v) {
        returning(Enumeration e) :
            call(Enumeration Vector+.elements())
            && target(v) {}}
    event updatesource 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 : create next* updatesource+ next
@match {
    System.out.println("improper enumeration usage");
    ___RESET;
}
}
More Examples – SafeLock in CFG

```java
SafeLock(Lock l, Thread t) {
    event acquire before (Lock l, Thread t):
        call(* Lock.acquire()) && target(l)
        && thread(t) {}
    event release before (Lock l, Thread t):
        call(* Lock.release()) && target(l)
        && thread(t) {}
    event begin before (Thread t):
        execution(* *(.)) && thread(t)
        && !within(Lock+) {}
    event end after (Thread t):
        execution(* *(.)) && thread(t)
        && !within(Lock+) {}
    cfg : S -> S begin S end |
        S acquire S release | epsilon
    @fail {
        System.out.println("improper lock usage");
    }
}
```
More Examples – HasNext in LTL

```java
HasNext(Iterator i) {
    event hasnexttrue after(Iterator i)
    returning(boolean b) :
        call(* Iterator.hasNext())
        && target(i) && condition(b) {}
    event hasnextfalse after(Iterator i)
    returning(boolean b) :
        call(* Iterator.hasNext())
        && target(i) && condition(!b) {}
    event next before(Iterator i) :
        call(* Iterator.next()) && target(i) {}

    ltl: [](next => (* hasnexttrue)
    @violation {
        System.out.println("improper iterator usage");
    }
}
```
Demo Video of the Tool
Conclusion

• MOP a *generic* yet *efficient* runtime verification framework
  – Extensible logic framework: FSM, ERE, PTLTL, FTLTL, LTL, CFG, PTCaRet, ...
  – Adaptable for different programming languages
    • JavaMOP, BusMOP
Future Work

• There is room for richer/better RV systems
  – More suitable logics for specifications
  – More programming languages/platforms
    • System level monitoring
• JavaMOP: using RV as a crosscutting configurable feature of runtime execution environments for "configurable Java".
• Combining RV with specification mining
• Combining RV and static program verification
References

- Monitoring-Oriented Programming (MOP) official website: http://fsl.cs.uiuc.edu/mop
- Some slides from http://www.runtime-verification.org/course/ (9 lectures)


References (cont.)


A Tragedy – Feng Chen

- Due to a sudden vascular accident and complications from an undetected blood clot, Feng Chen passed away on **August 8, 2009**
- Ph.D.: Defended in **July 2009**
- After his graduation, Feng had accepted a tenure-track position at Iowa State University
- He has three papers AFTER his death!
- Tributes from Rosu, Meseguer, Pnueli, and …
- **In Memoriam**
Any Questions?