Monitoring-Oriented Programming

Seyed Mohammad Mehdi Ahmadpanah
smahmadpanah@aut.ac.ir

Supervisor
Dr. Mehran S. Fallah

Formal Security Lab.
CEIT@AUT

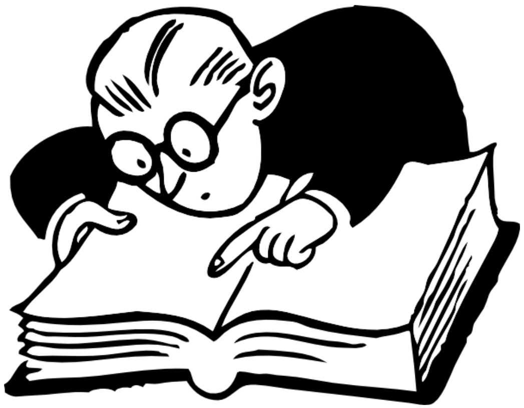
Mar. 5, 2017

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

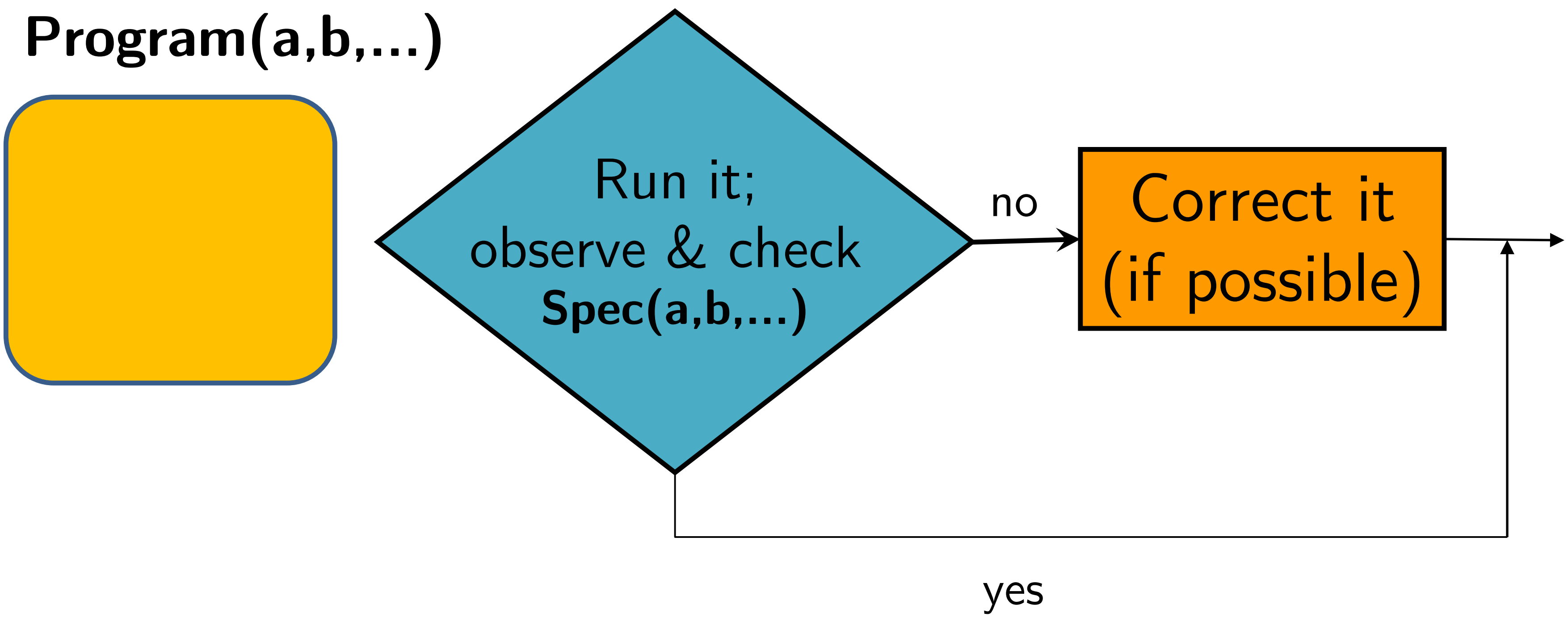
- Aims at achieving benefits of both testing and formal verification, avoiding their pitfalls
- Question: what do we really want … ?

    A. To prove a program correct?

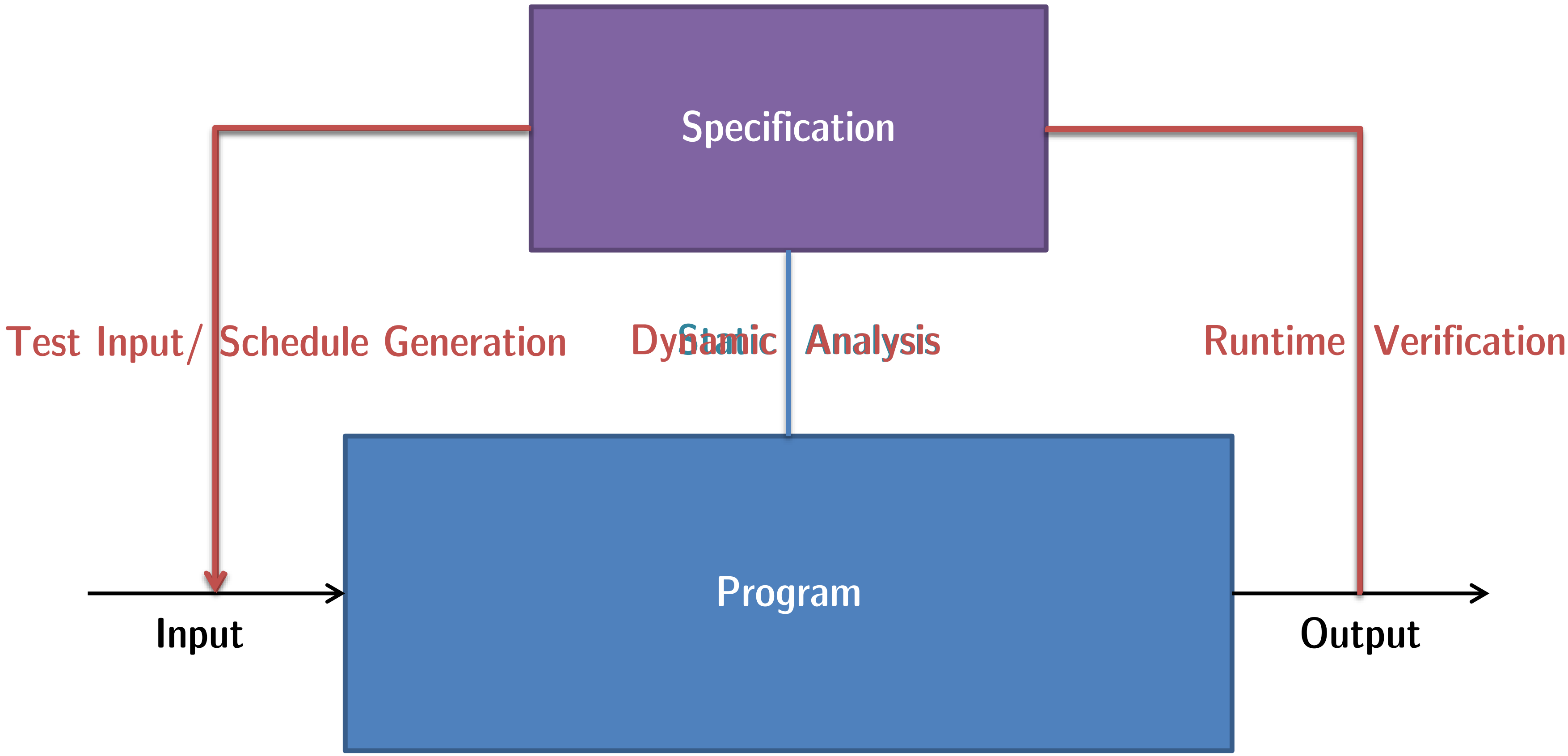
    B. To achieve correct execution?

    – 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

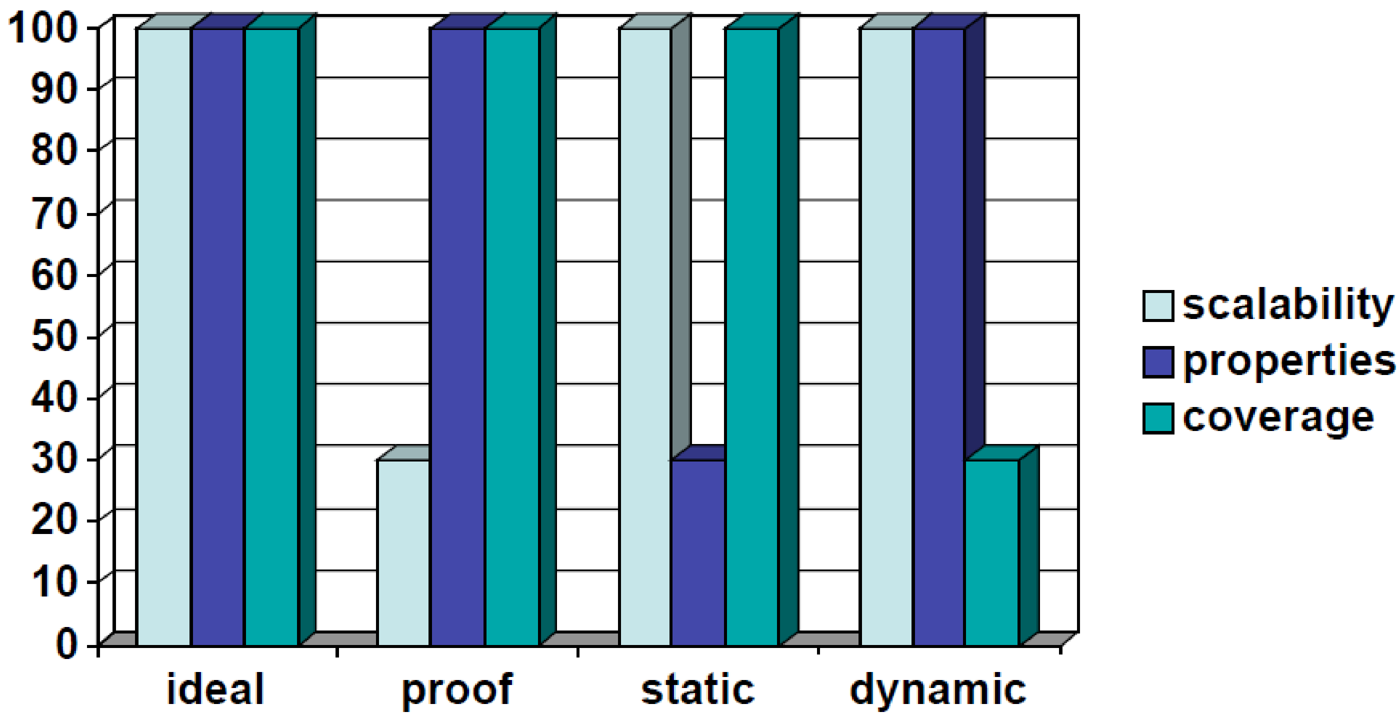


Specification and Programming



Comparison of Techniques

- Giving up on coverage to write better specifications and scale



What is Trace?

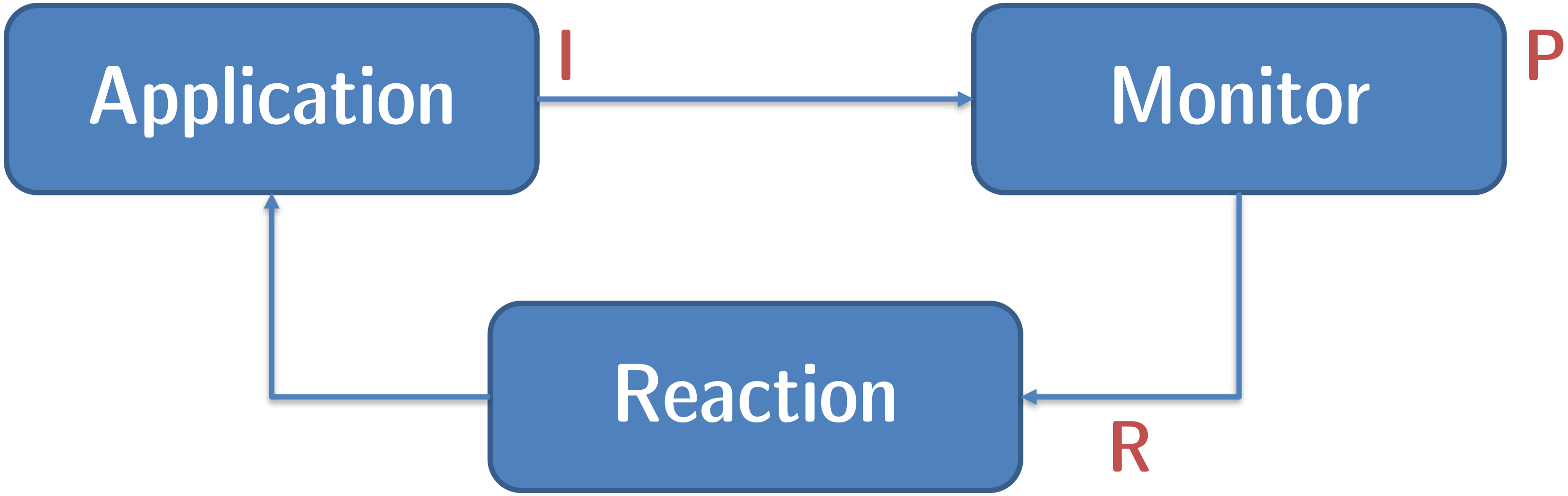
- A formal view of an execution is to consider it as a sequence σ of program states:

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

- Past in known vs. Future is unknown

The Cycle

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

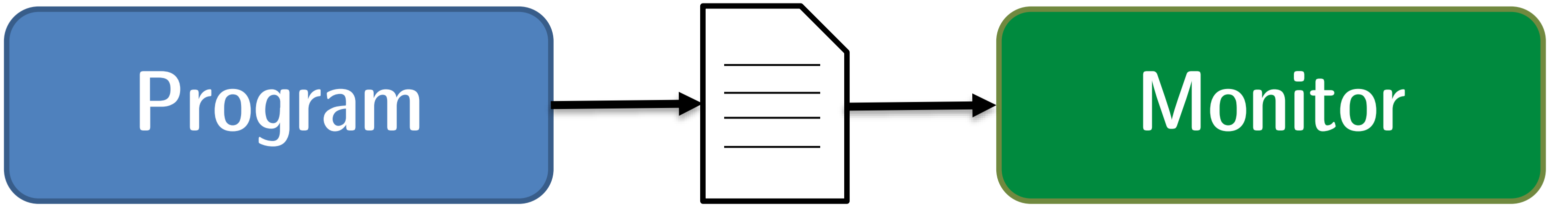


Property Languages

- 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
  - Inline
  - Outline
    - Monitoring code is embedded into the application
    - Monitor runs in parallel with application
      - Synchronous (Application waits for response)
      - Asynch

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

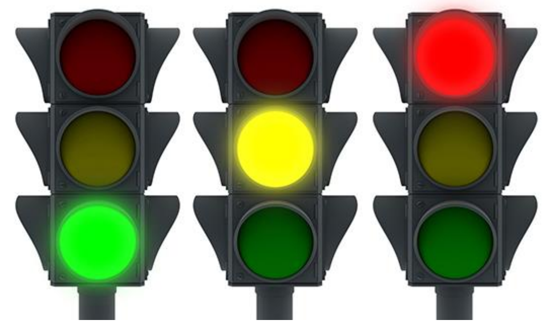
- Violation
  - checking that the systems conf… a property, and reporting when the pr… is “violated”.
- Validation
  - Stating prop… negative form: *what we do not want to* …, Reporting when the bad property get… ated”.
  - … it is a good property and we just want to log whenever something good happens.

Most systems can only do one of the two forms!

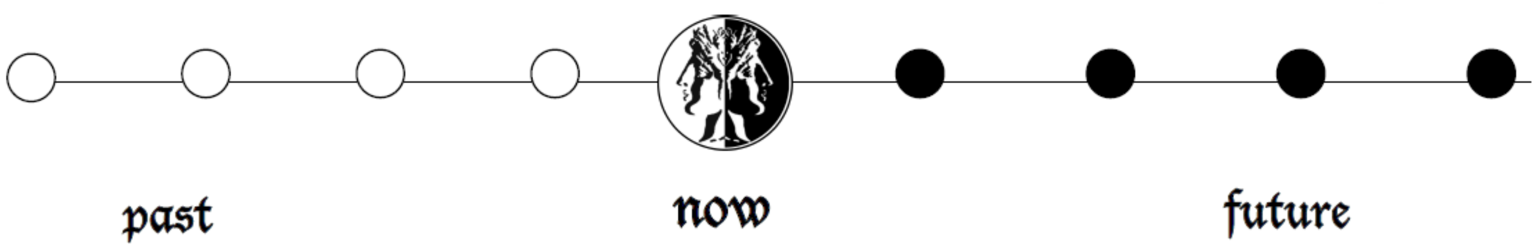
Example - Violation

- Property:

  (green yellow red)*



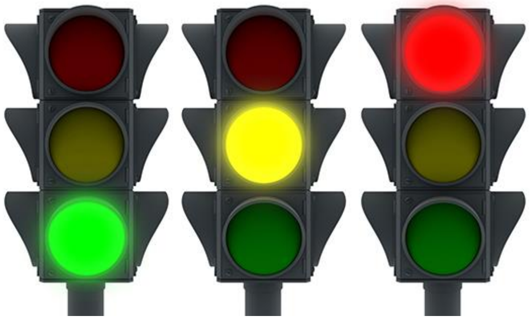
green red

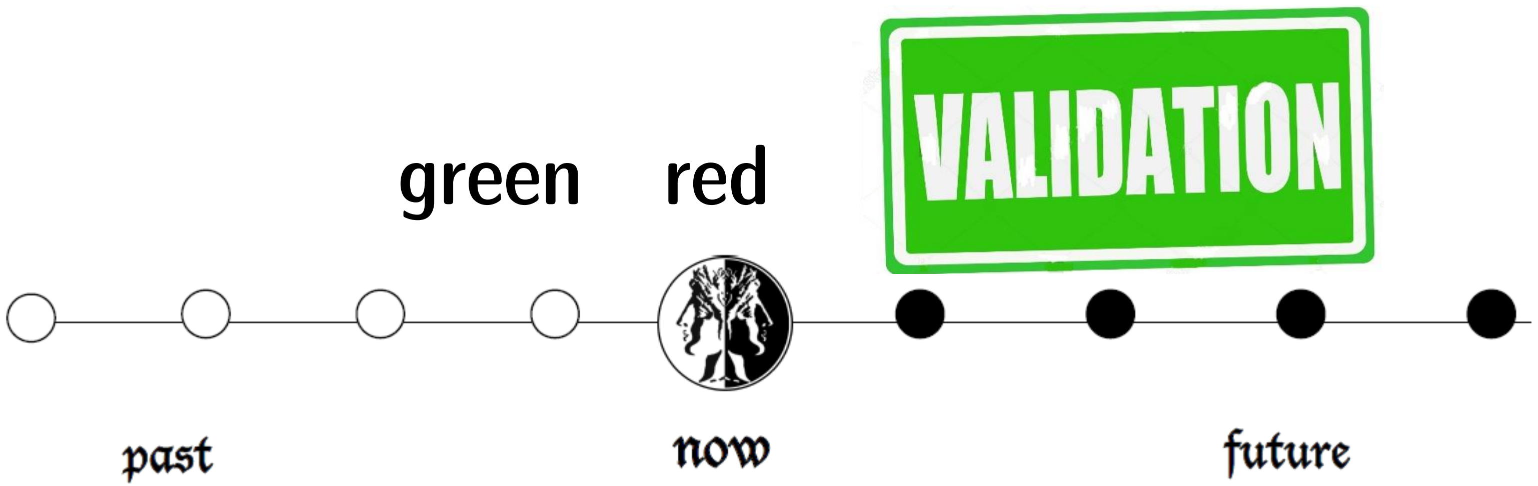


Example - Validation

- Property:

  green red



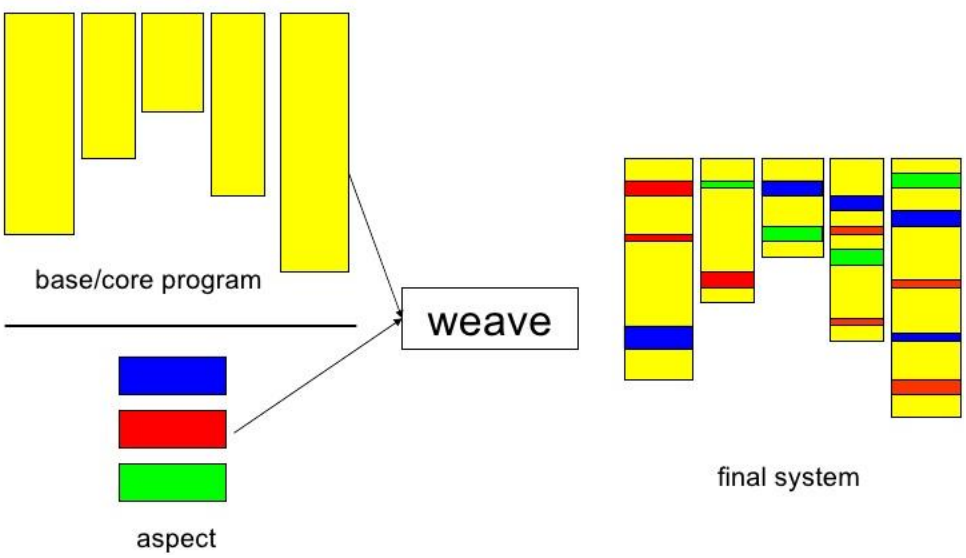


Challenges

- 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

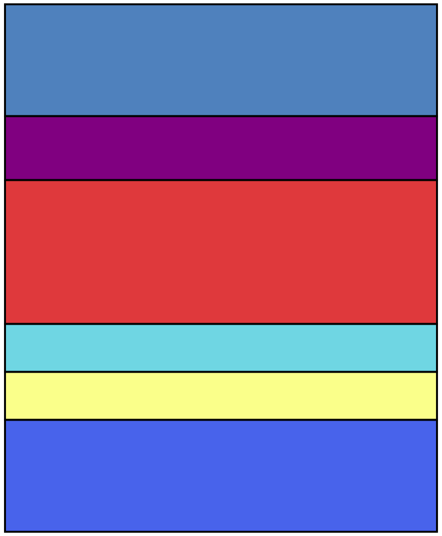


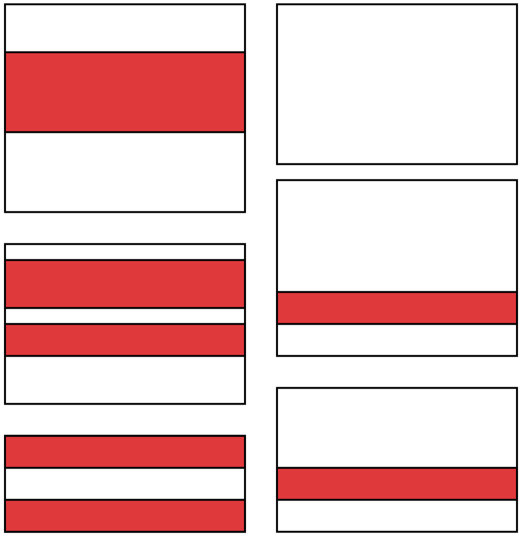
Aspect-Oriented Programming (cont.)

code tangling:
one module
many concerns

example:
logging

code scattering:
one concern
many modules



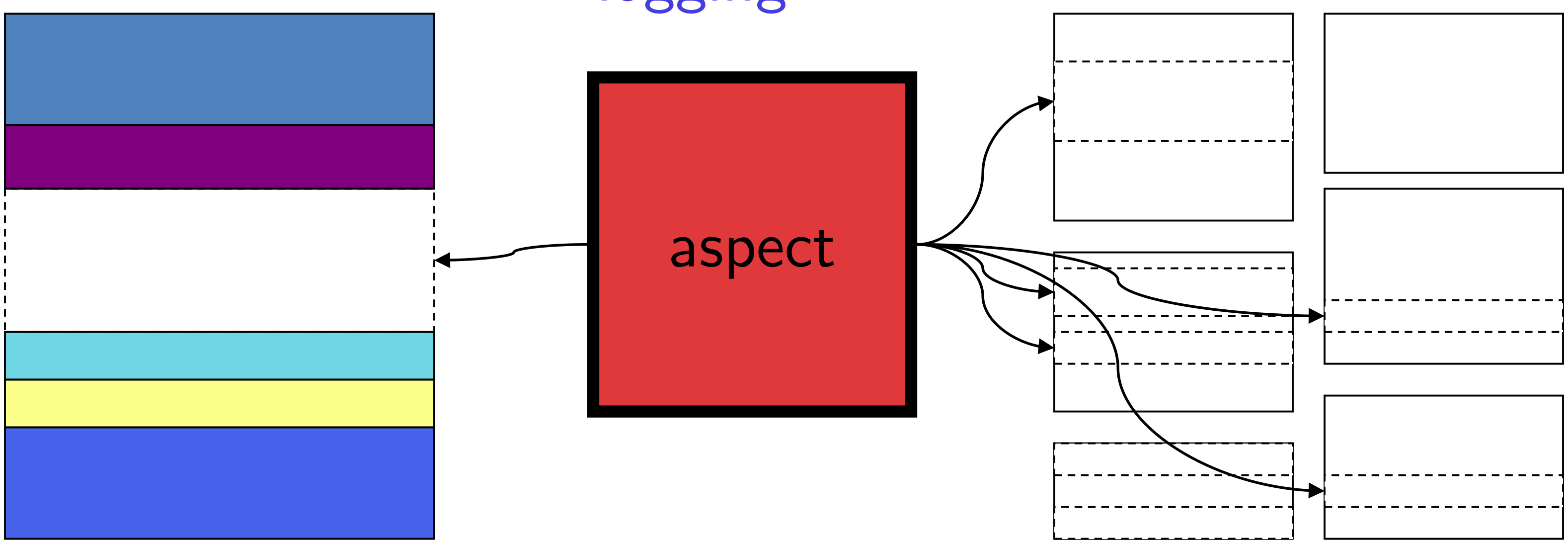


Aspect-Oriented Programming (cont.)

code tangling:
one module
many concerns

example:
logging

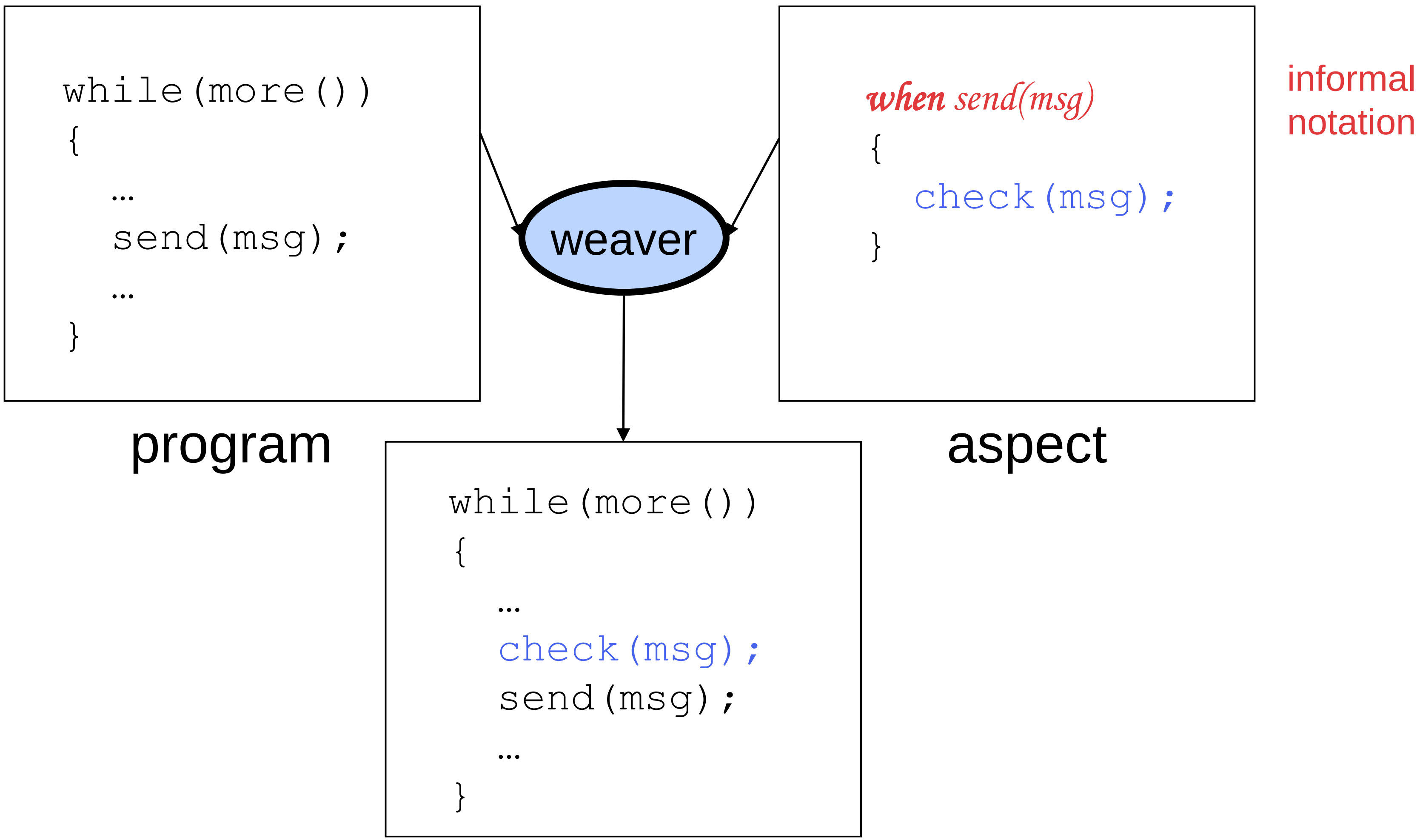
code scattering:
one concern
many modules



AOP Concepts

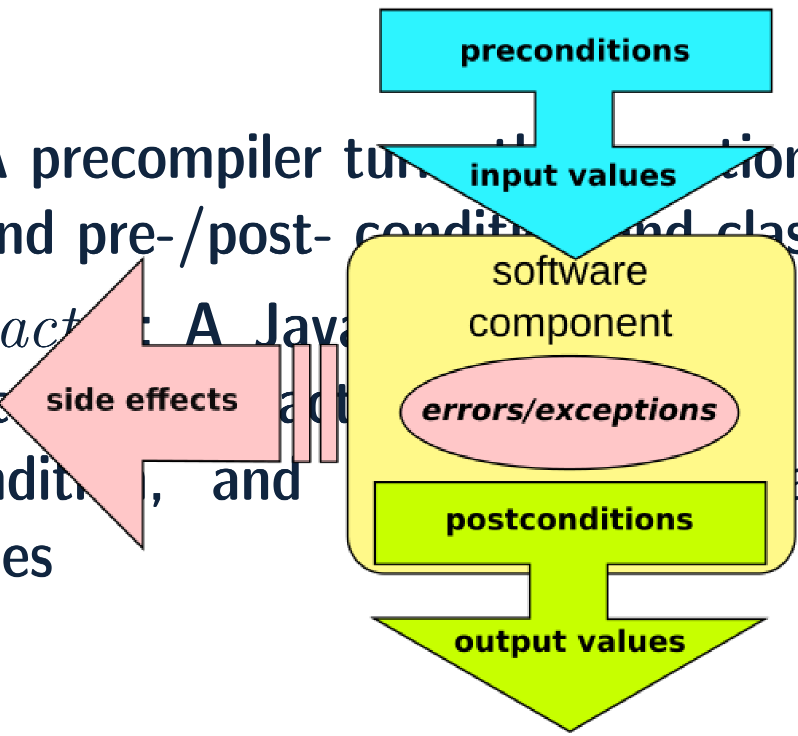
- An aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches).

Simplified View of AOP



Design By Contract

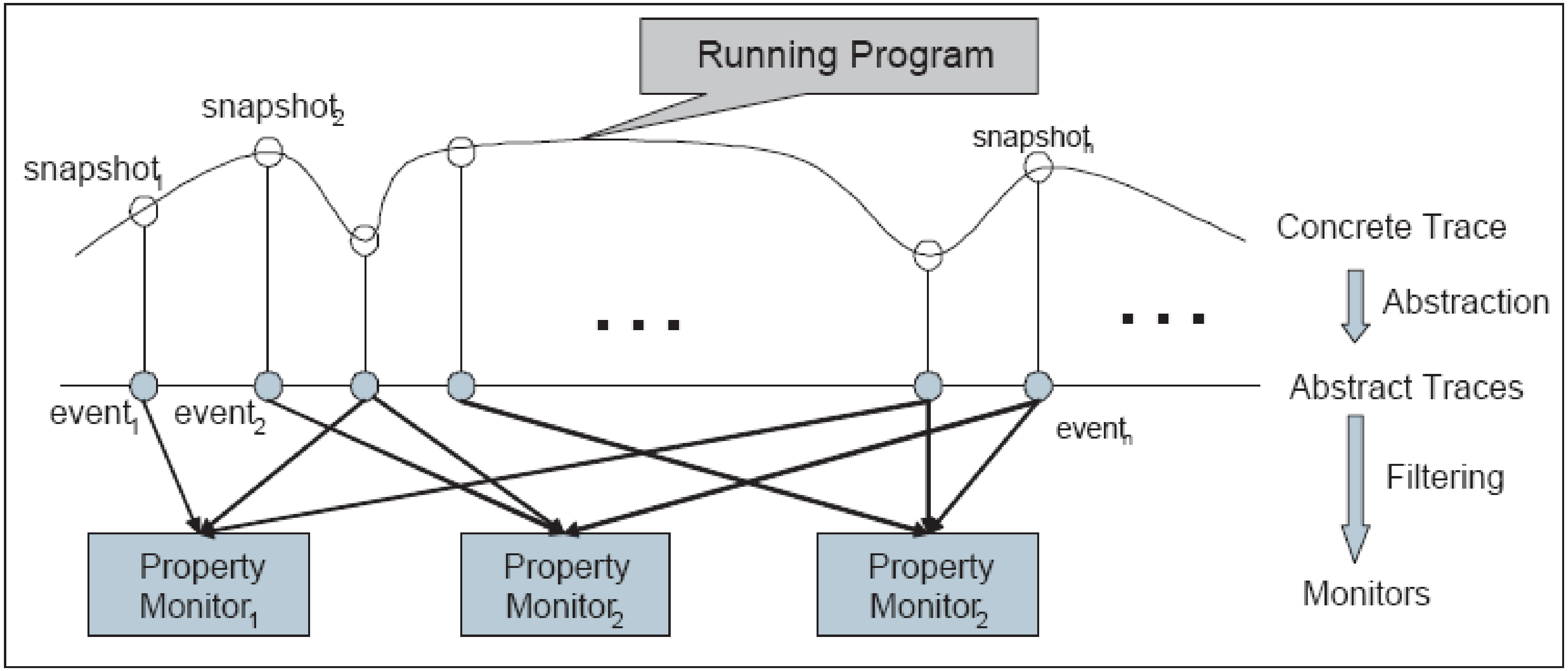
- Add semantic information to a program by specifying assertions regarding the program's runtime state, and then checking the specification at runtime
  - *Jass*: A precompiler tu[illegible]tion comments into Java code, and pre-/post- [illegible] class invariants
  - *jContract*[illegible]: A Java [illegible]lows programmers to associat[illegible] of precondition, postcondit[illegible], and [illegible]any Java classes or interfaces



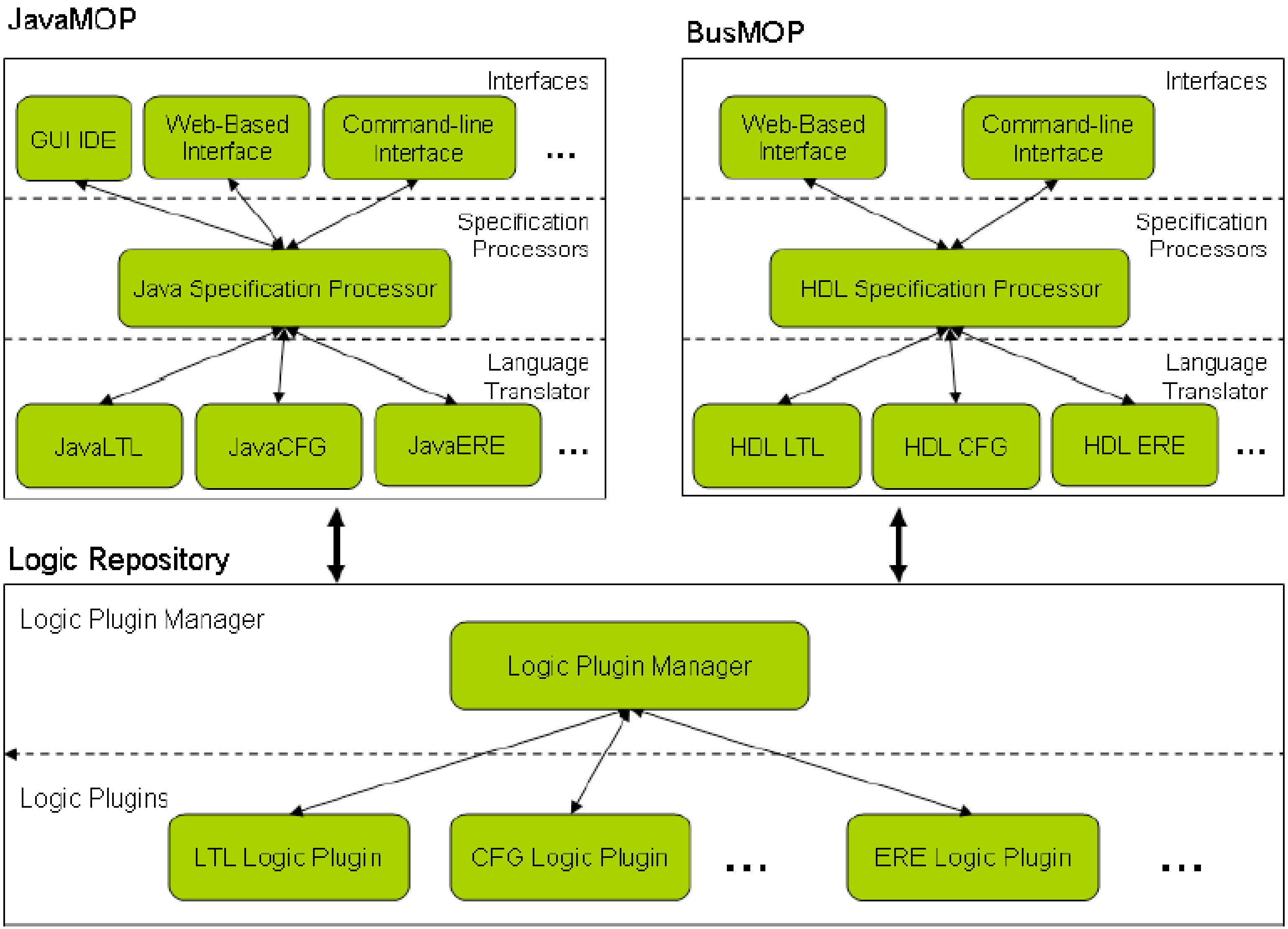
What is Monitoring-Oriented Programming?

- 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

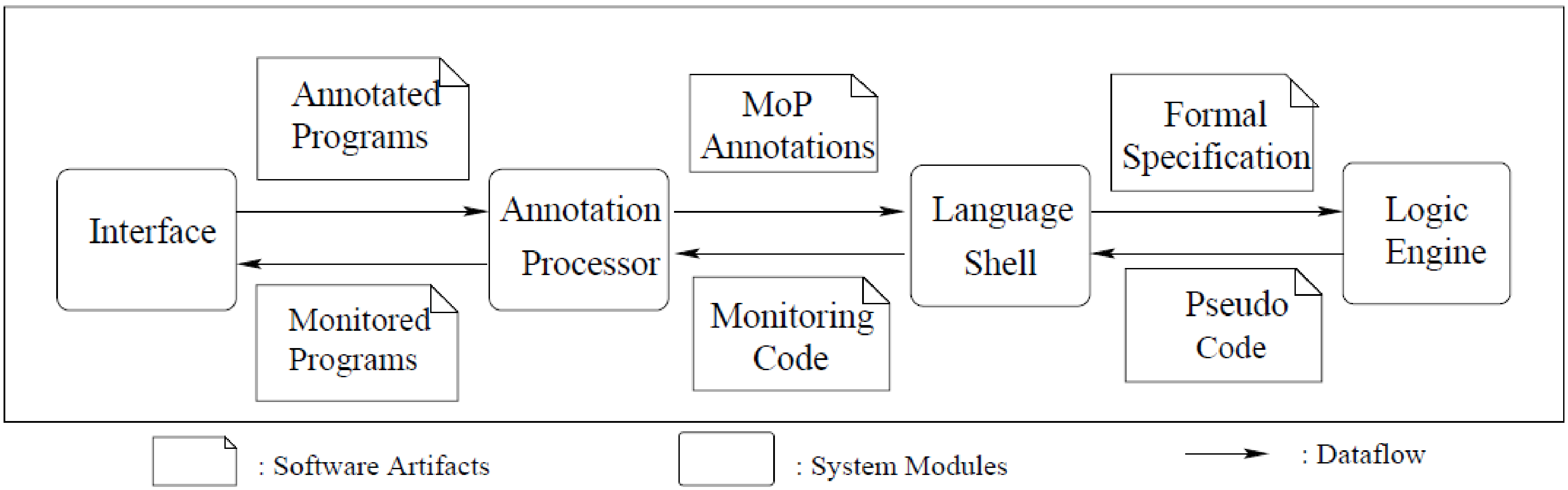
MOP Approach to Monitoring



MOP Architecture



Program Transformation Flow in MOP



MOP

One can understand MOP from at least **three** perspectives:

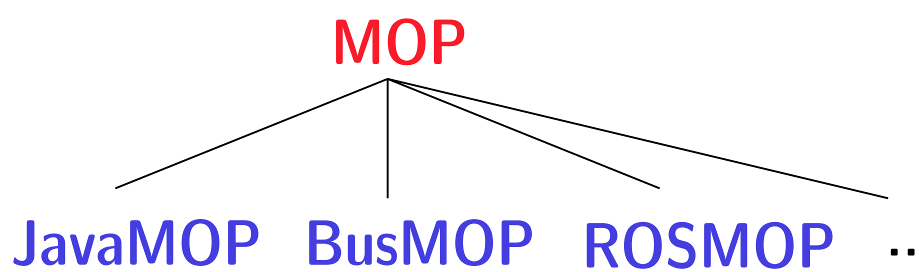
1. Improving reliability of a system by monitoring its requirements against its implementation at runtime. By generating and integrating the monitors automatically rather than manually
2. An extension of programming languages with logics
3. A lightweight formal method
   – by not letting it go wrong at runtime

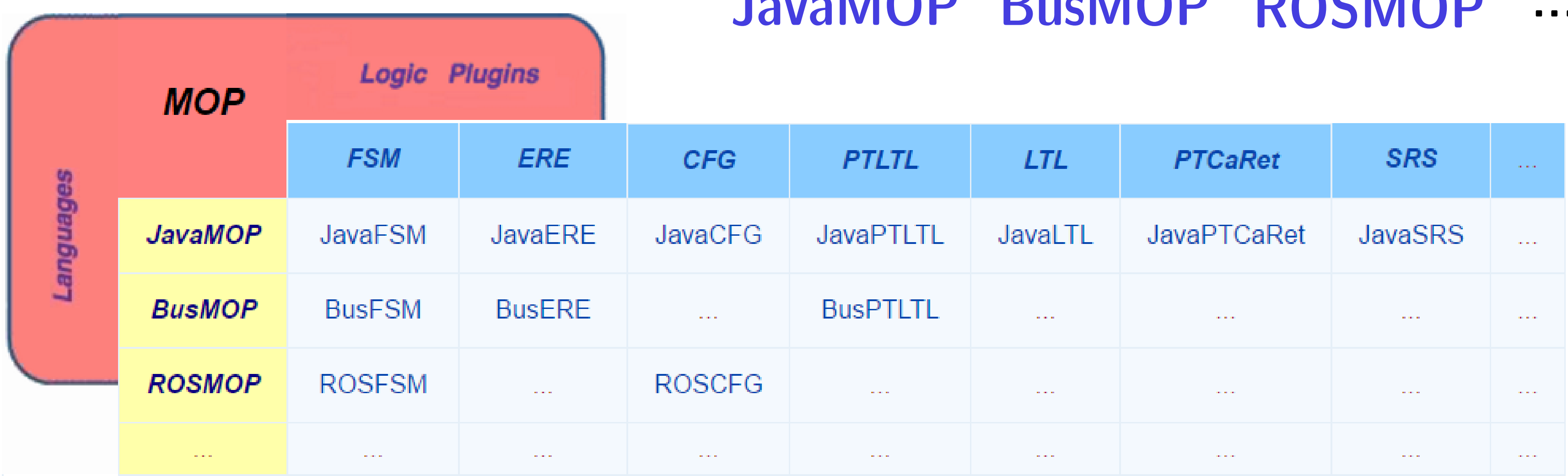
MOP (cont.)

- 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 generic in both specification formalisms (logics) and programming languages





MOP	FSM	ERE	CFG	PTLTL	LTL	PTCaRet	SRS	...
JavaMOP	JavaFSM	JavaERE	JavaCFG	JavaPTLTL	JavaLTL	JavaPTCaRet	JavaSRS	...
BusMOP	BusFSM	BusERE	...	BusPTLTL	...	...	...	...
ROSMOP	ROSFSM	...	ROSCFG	...	...	...	...	...
...	...	...	...	...	...	...	...	...

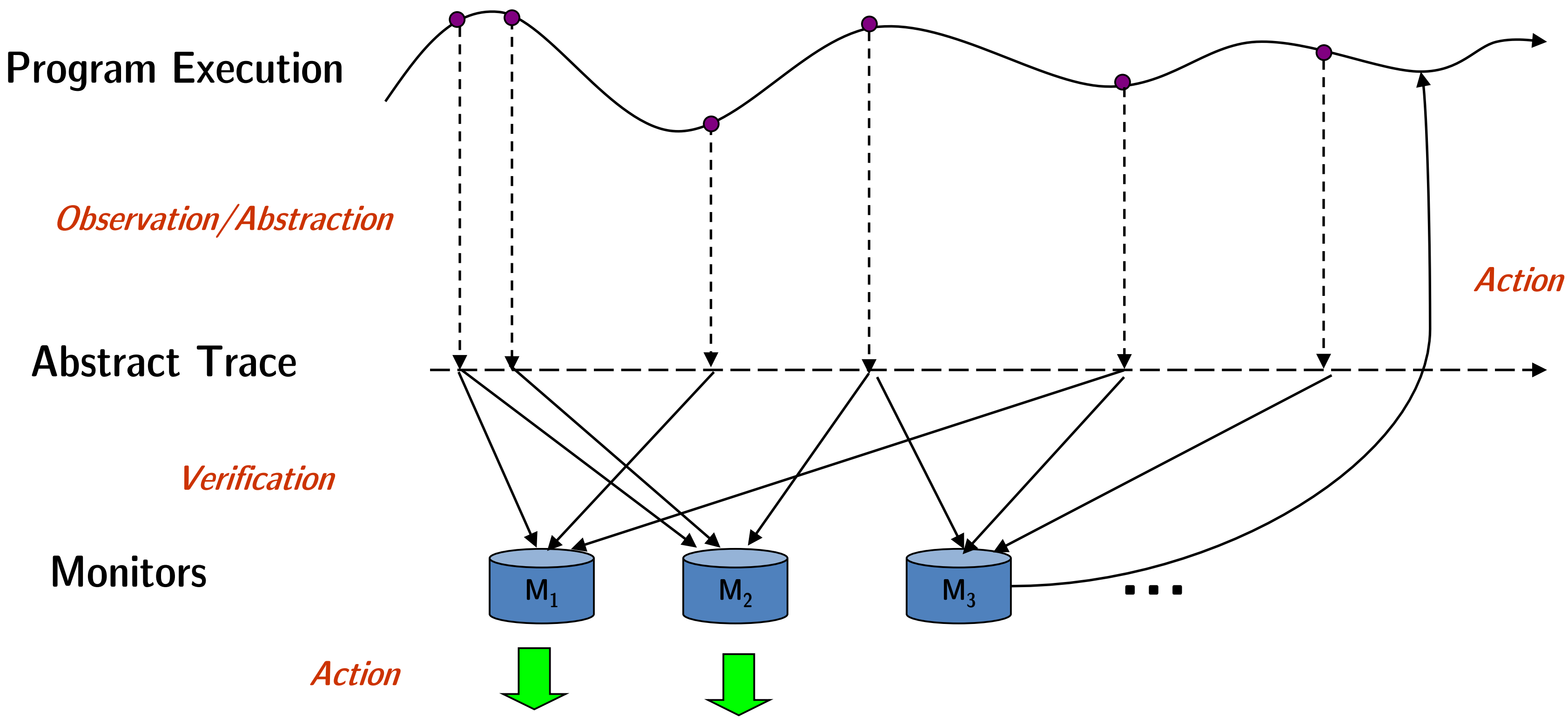
Examples of Runtime Verification Systems

Approach	Language	Logic	Scope	Mode	Handler
Hawk	Java	Eagle	global	inline	violation
J-Lo	Java	ParamLTL	global	inline	violation
Jass	Java	Assertions	global	inline	violation
JavaMaC	Java	PastLTL	class	outline	violation
jContractor	Java	Contracts	global	inline	violation
JML	Java	Contracts	global	inline	violation
JPaX	Java	LTL	class	offline	violation
P2V	C/C++	PSL	global	inline	violation/validation
PQL	Java	PQL	global	inline	validation
PTQL	Java	SQL	global	outline	validation
Spec#	C#	Contracts	global	inline/offline	violation
RuleR	Java	RuleR	global	inline	violation
Temporal Rover	*Several*	MiTL	class	inline	violation
Tracematches	Java	Reg. Ex.	global	inline	validation

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

MOP Monitoring Model



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

$\langle Specification \rangle ::= \texttt{/*@}\ \langle Header \rangle\ \langle Body \rangle\ \langle Handlers \rangle\ \texttt{@*/}$

$\langle Header \rangle ::= \langle Attribute \rangle^{*}[\texttt{scope =}\langle Scope \rangle][\texttt{logic =}\langle Logic \rangle]$

$\langle Attribute \rangle ::= \texttt{static} \mid \texttt{outline} \mid \texttt{offline} \mid \texttt{centralized}$

$\langle Scope \rangle ::= \texttt{global} \mid \texttt{class} \mid \texttt{interface} \mid \texttt{method}$

$\langle Name \rangle ::= \langle Identifier \rangle$

$\langle Logic \rangle ::= \langle Identifier \rangle$

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

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

$\langle Handlers \rangle ::= [\langle ViolationHandler \rangle]\ [\langle ValidationHandler \rangle]$

$\langle ViolationHandler \rangle ::= \texttt{violation handler}\ \{\ \langle Code \rangle\ \}$

$\langle ValidationHandler \rangle ::= \texttt{validation handler}\ \{\ \langle Code \rangle\ \}$

MOP Example: Safe Enumeration

```
/*@
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); }
@*/
```

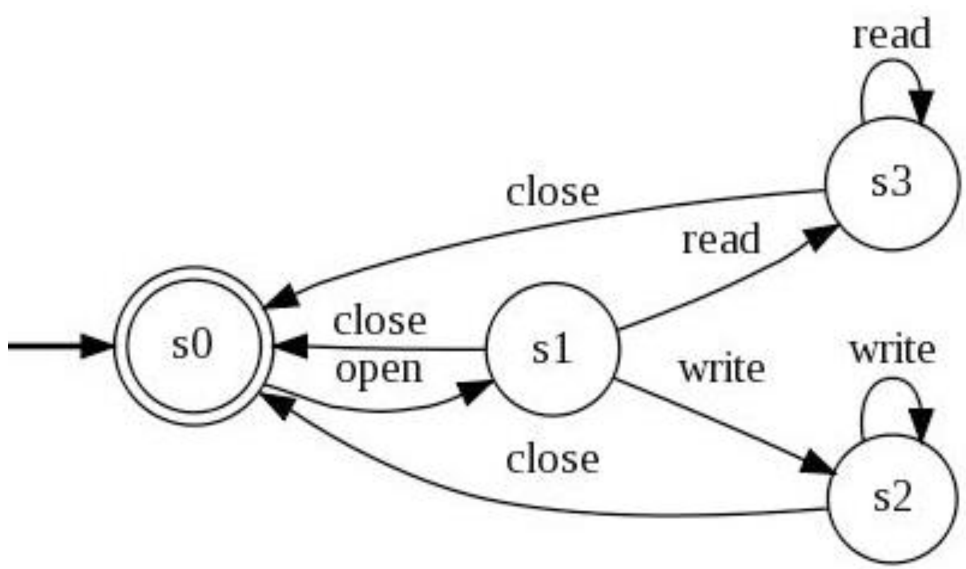
FSM Plugin
(Finite State Machine)

- 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

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
]
```

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 expressions with complement (negation)
  - Specify properties non-elementarily more compactly
  - More complicated to monitor

Syntax for ERE

$$E ::= \emptyset \mid \epsilon \mid A \mid E\;E \mid E^* \mid E+E \mid E\&E \mid \neg E$$

extended with negation

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

aab	{aab}
(ab)*	$\{\epsilon$,ab,abab,...}
$(a+b)^*$ & $\neg(ab)^*$	words of randomly interleaved a's and b's but not only cleanly alternating (ababab...) {a, aa, abba, bbbb,...}

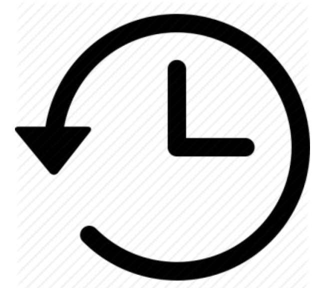
Limitations of Regular Expressions for Specification

- 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

$$F ::= true \mid false \mid A \mid \neg F \mid F \; op \; F$$

$$\odot F \mid \diamond F \mid \boxdot F \mid F \; \mathcal{S}_s \; F \mid F \; \mathcal{S}_w \; F$$

previous eventually always since

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

start end F butnot F'

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

$$\boxdot(\uparrow (dialing) \rightarrow \neg \odot(busyTone \vee connected))$$

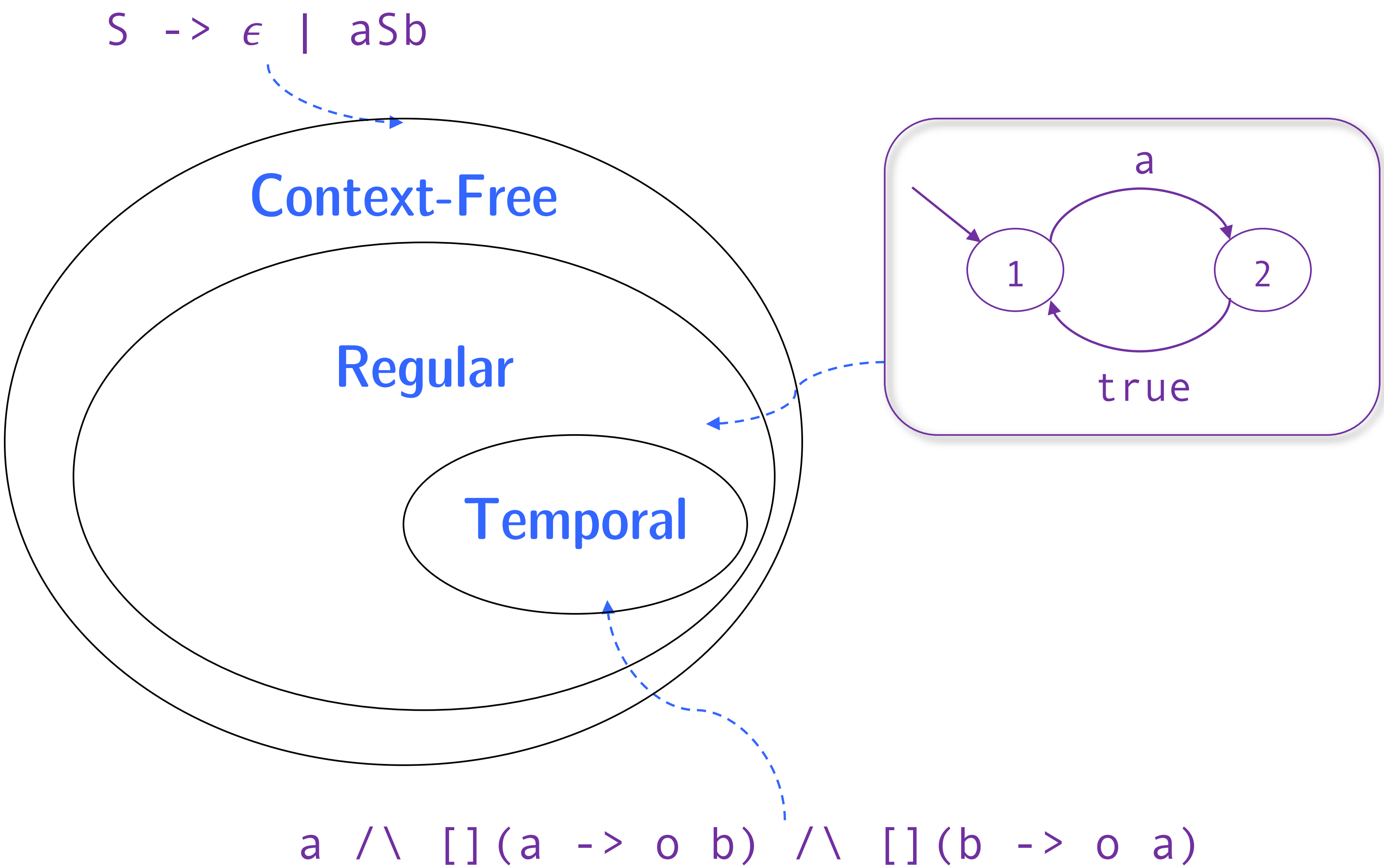
```
... (Java code A) ...
/*@ FTLTL
```

```
... (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

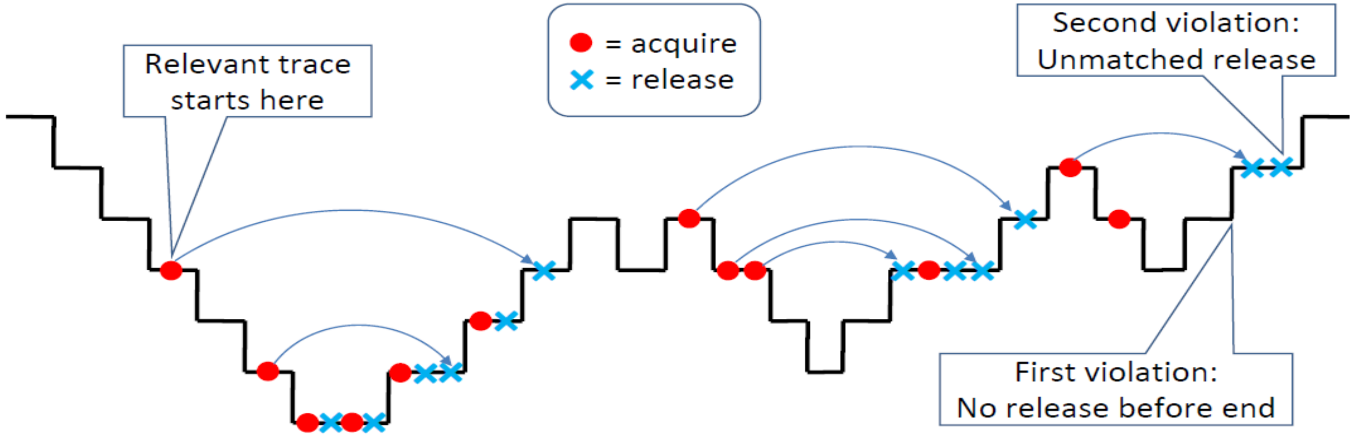
$$\Box(green \rightarrow \neg red\, \mathcal{U}\, yellow)$$

The Language Hierarchy

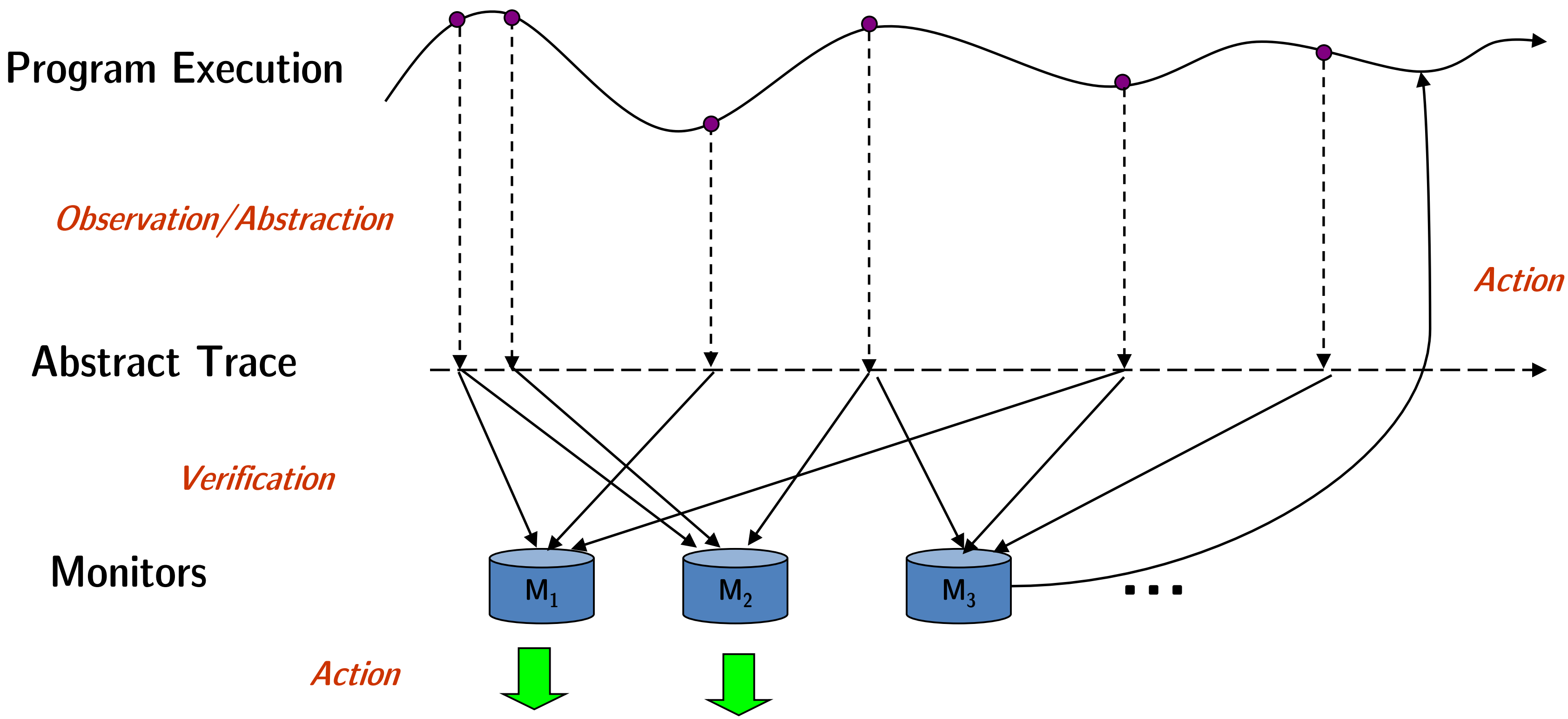


CFG Plugin (Context-Free Grammar)

- Most systems support finite state monitors
  - Regular languages
  - Linear temporal logics
- These cannot monitor *structured* properties:



Recall – MOP Monitoring Model



Monitors can be dynamically created or destroyed – why?

Parametric Monitoring

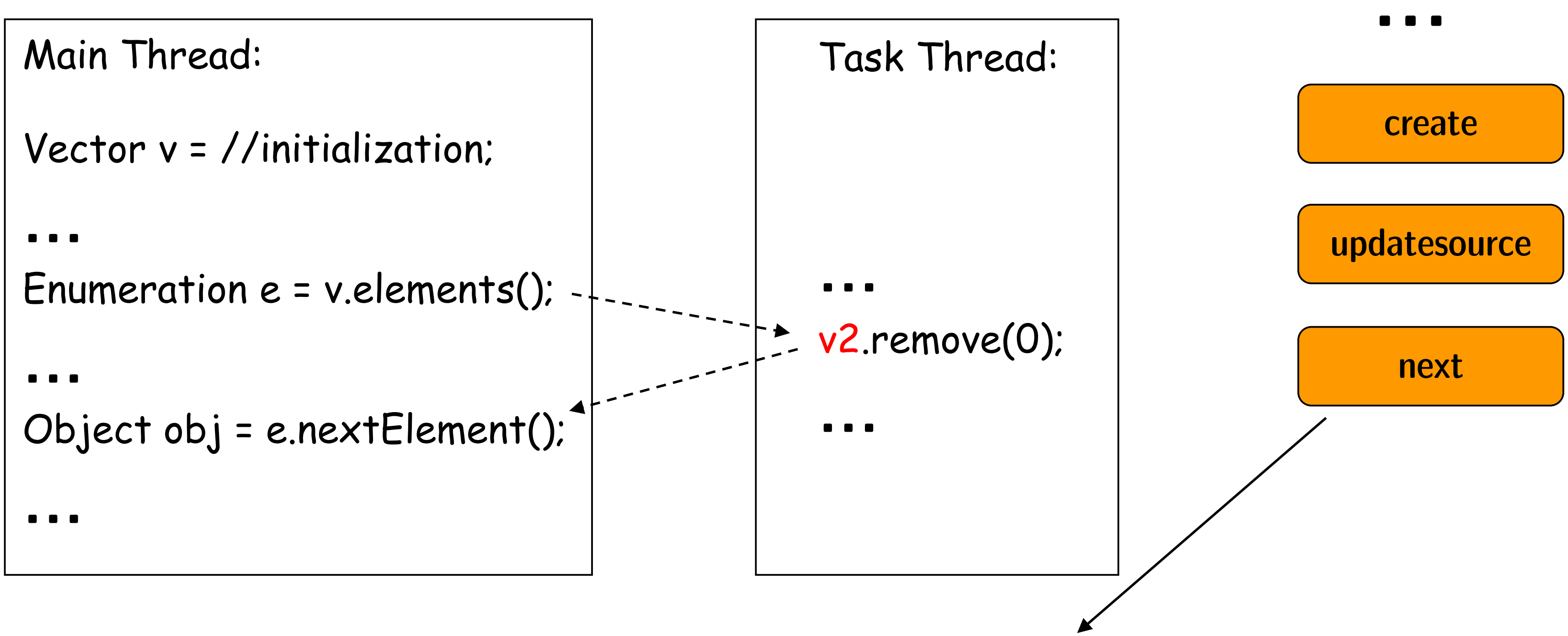
Parametric Properties

- Imperatively needed, but hard to monitor efficiently

Parameters

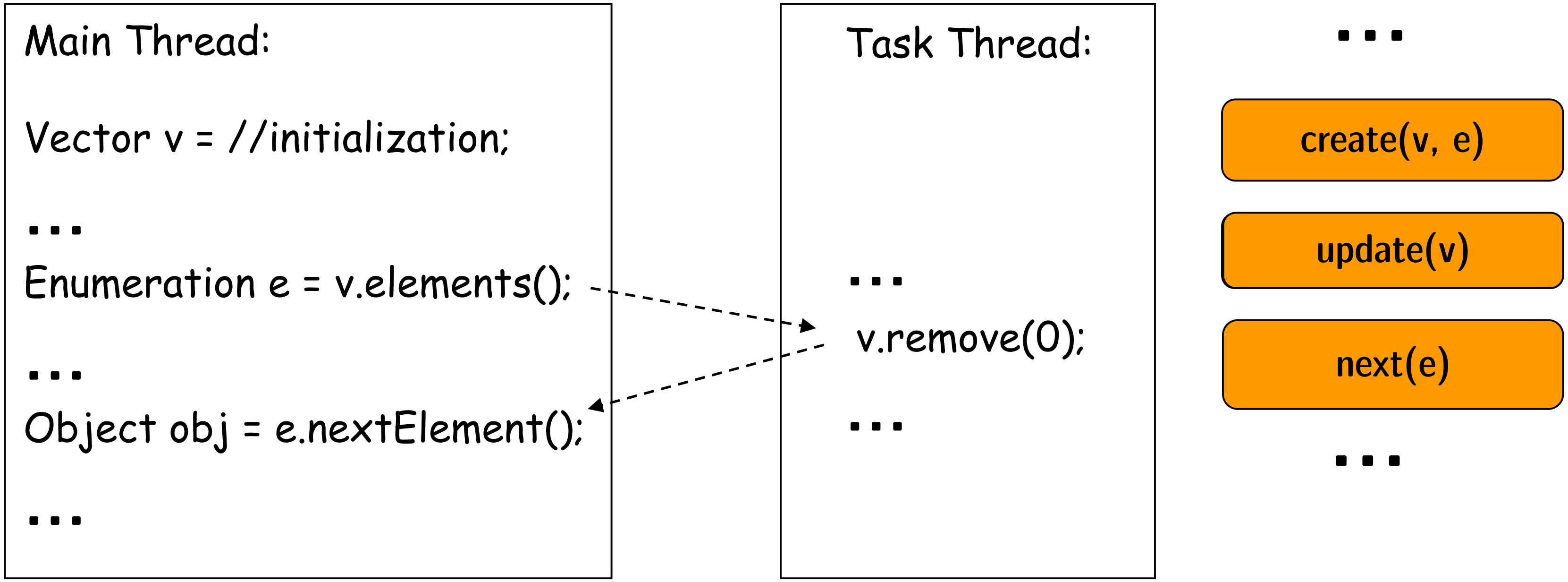
```
SafeEnum(Vector v, Enumeration+ e) {
  event create after(Vector v) returning(Enumeration e): ...
  event updatesource after(Vector v) : ...
  event next before(Enumeration e) : ...

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

Lack of Parameters Leads to False Alarms



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

Adding Parameters to Events



- *Parametric traces*: traces containing events with parameters

Checking Parametric Traces

parametric trace

updatesource(v1)

create (v1,e1)

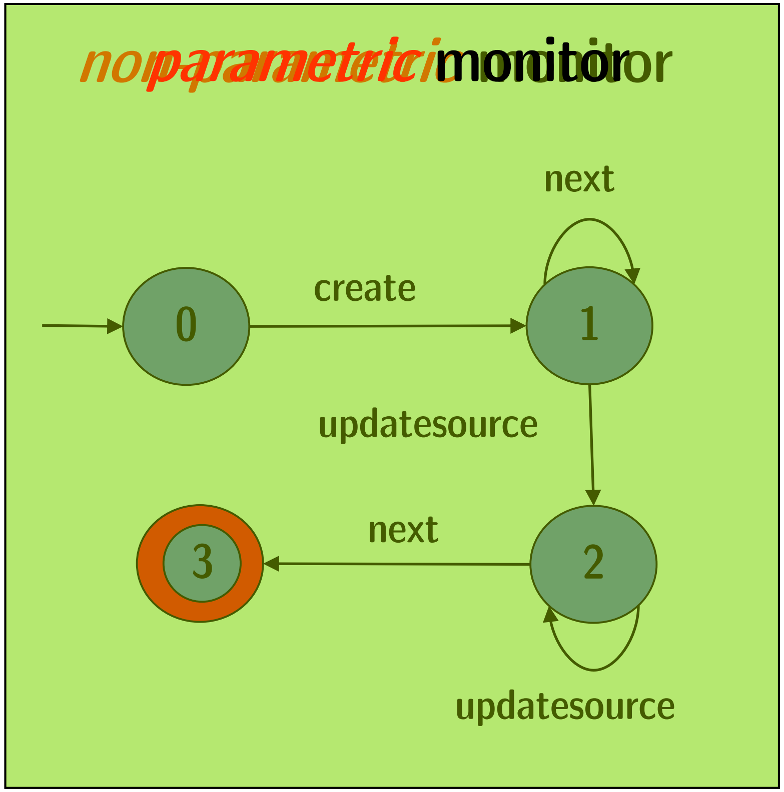
updatesource(v2)

next(e1)

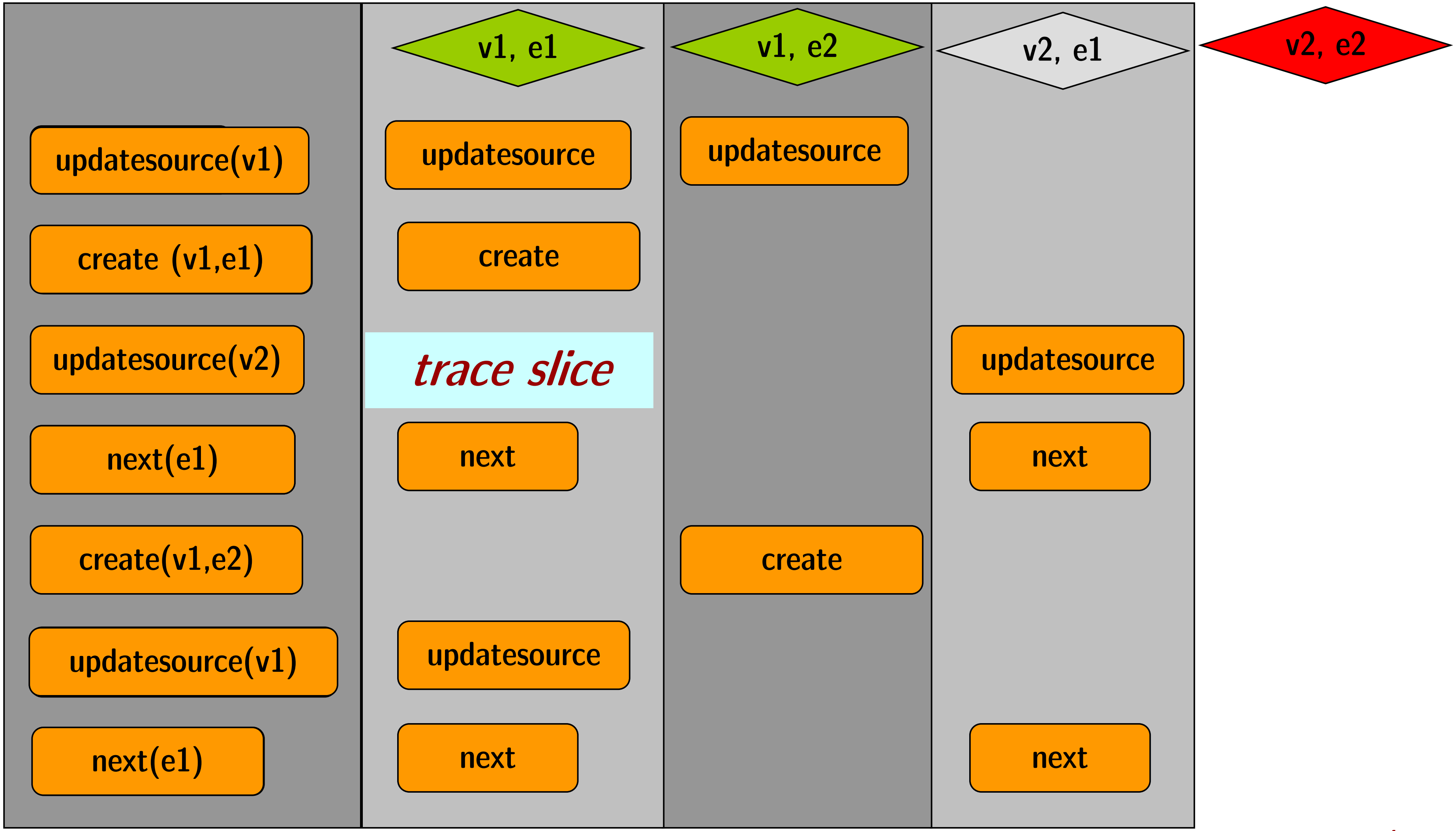
create(v1,e2)

updatesource(v1)

next(e1)



Parametric Trace Slicing

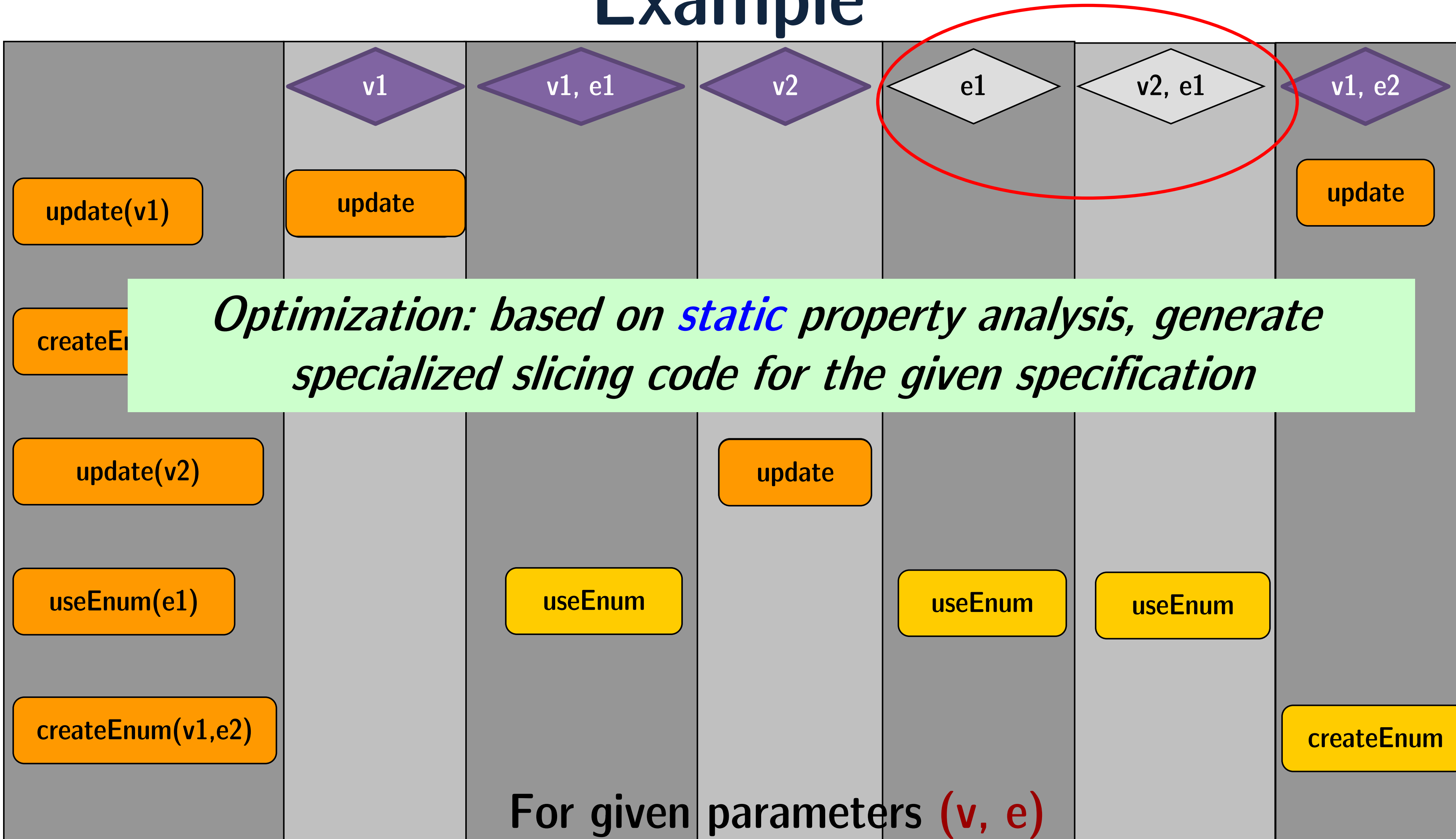


For given parameters (v, e)

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



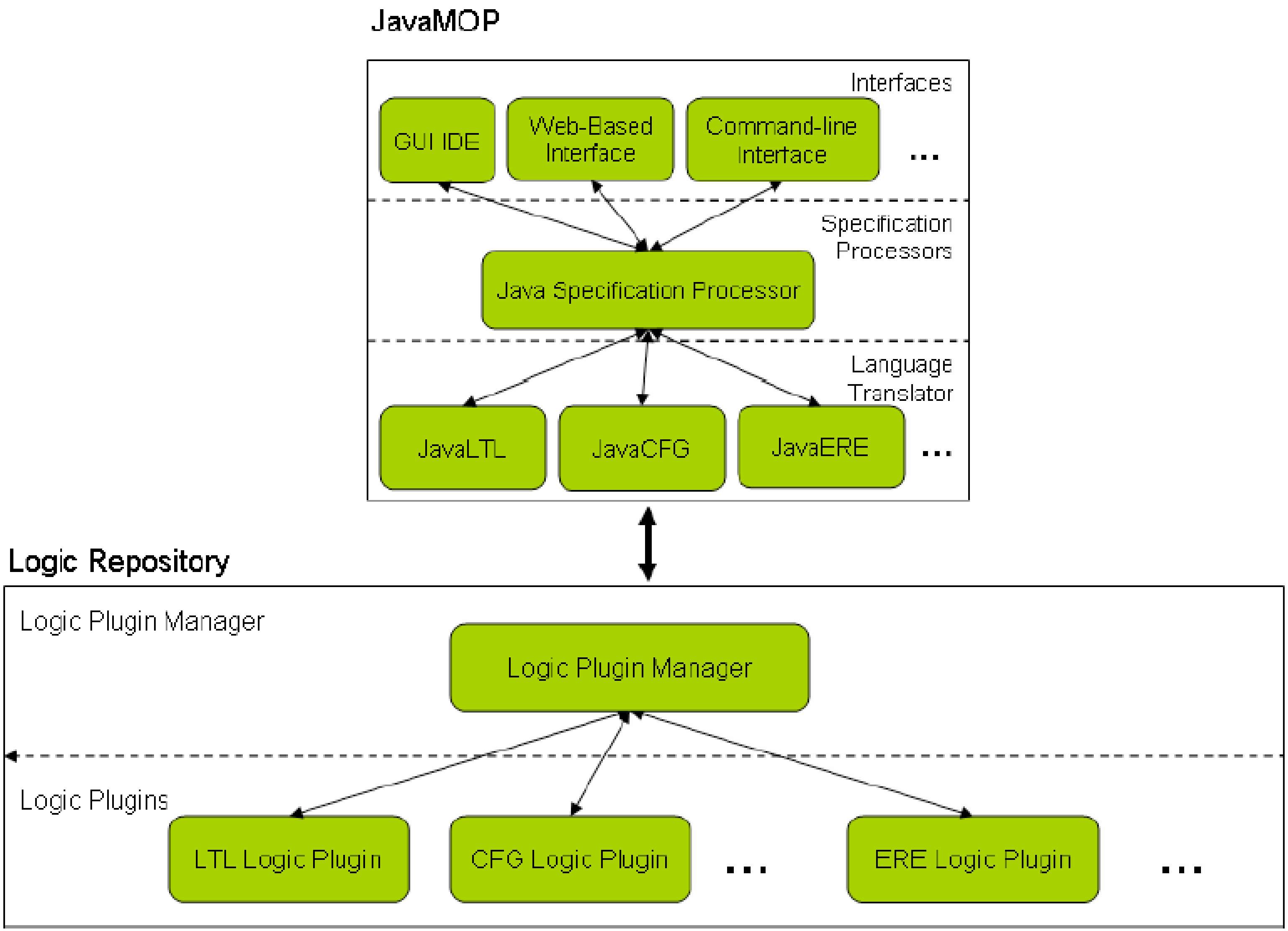
Optimization: based on static property analysis, generate specialized slicing code for the given specification

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

JavaMOP Architecture



Evolution-Aware MOP

- Extend MOP to support *multiple software versions*
- The key idea:

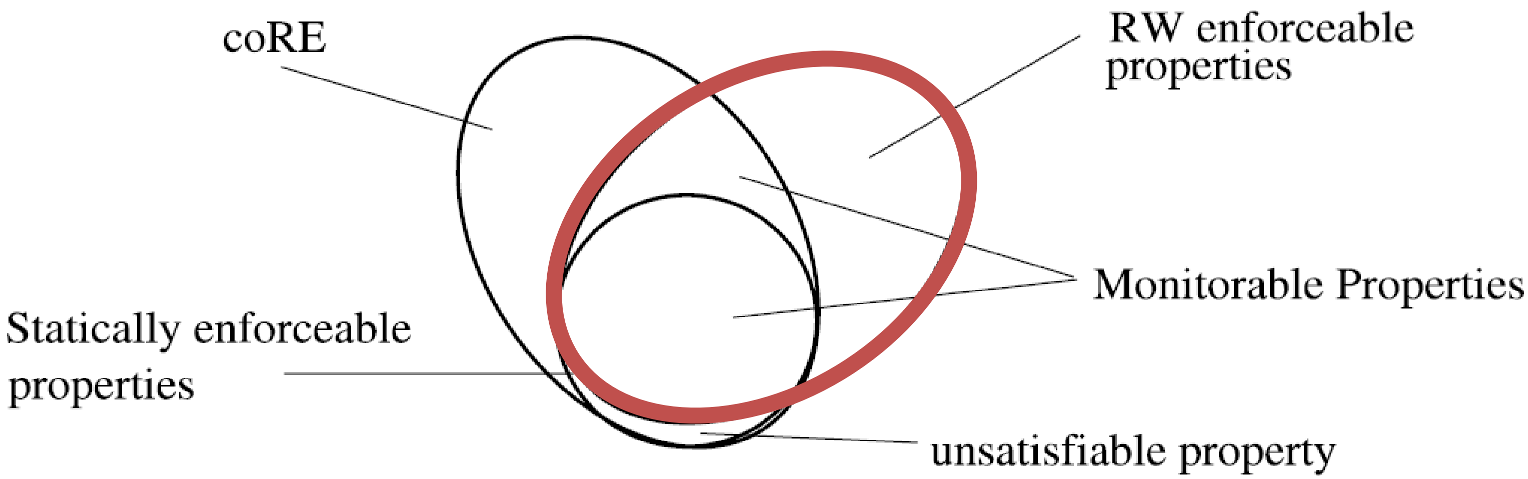
  To monitor only the parts of code that changed between versions

  - 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. Chineese Wall in JavaMOP using CFG
  - Should not be used for low-level security policies

JavaMOP and Security Policies (cont.)



- **JavaMOP with AspectJ is able to rewrite the target program**
  - A $\boldsymbol{Program\ Rewriter}$
    - Can enforce $\boldsymbol{RW\text{-}enforceable\ policies}$
      - Including $\boldsymbol{EM\text{-}Policies}$ and $\boldsymbol{Satisfiable\ static\ policies}$

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)

[1] **F. Chen** and G. Roșu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," in Electronic Notes in Theoretical Computer Science, 2003, vol. 89, no. 2, pp. 113–132.

[2] **F. Chen**, D. Jin, P. Meredith, and G. Roșu, "Monitoring Oriented Programming - A Project Overview," in Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09), 2009, pp. 72–77.

[3] **F. Chen** and G. Roșu, "Java-MOP: A Monitoring Oriented Programming Environment for Java," in Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 546–550.

[4] **F. Chen**, M. D'Amorim, and G. Roșu, "A Formal Monitoring-Based Framework for Software Development and Analysis," in Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004. Proceedings, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 357–372.

[5] **F. Chen** and G. Roșu, "Mop: an efficient and generic runtime verification framework," ACM SIGPLAN Not., no. 448501, pp. 569–588, 2007.

References (cont.)

[6] P. O. N. Meredith, D. Jin, D. Griffith, **F. Chen**, and G. Roșu, "An overview of the MOP runtime verification framework," Int. J. Softw. Tools Technol. Transf., vol. 14, no. 3, pp. 249–289, 2012.

[7] O. Legunsen, D. Marinov, and G. Roșu, "Evolution-Aware Monitoring-Oriented Programming," in Proceedings - International Conference on Software Engineering, 2015, vol. 2, pp. 615–618.

[8] D. Jin, P. O. N. Meredith, C. Lee, and G. Roșu, "JavaMOP: Efficient parametric runtime monitoring framework," in Proceedings - International Conference on Software Engineering, 2012, pp. 1427–1430.

[9] S. Hussein, P. Meredith, and G. Roșu, "Security-policy monitoring and enforcement with JavaMOP," PLAS '12 Proc. 7th Work. Program. Lang. Anal. Secur., pp. 1–11, 2012.

[10] P. O. N. Meredith, D. Jin, **F. Chen**, and G. Roșu, "Efficient monitoring of parametric context-free patterns," in Automated Software Engineering, 2010, vol. 17, no. 2, pp. 149–180.

[11] **F. Chen**, M. d'Amorim, and G. Roșu, "Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP," Electron. Notes Theor. Comput. Sci., vol. 144, no. 4 SPEC. ISS., pp. 3–20, 2006.

A Tragedy – Feng Chen

- Due to a sudden vascular accident an[illegible] complications from an undetecte[illegible] clot, Feng Chen passed away on August [illegible]09
- Ph.D. : Defended in Ju[illegible]
- After his graduati[illegible]g had accepted a tenure-track positio[illegible]a State University
- He has [illegible]apers AFTER his death!
- T[illegible]from Rosu, Meseguer, Pnueli, and ...
- I[illegible]emoriam

Live the moment NOW! ☺

Any Questions?