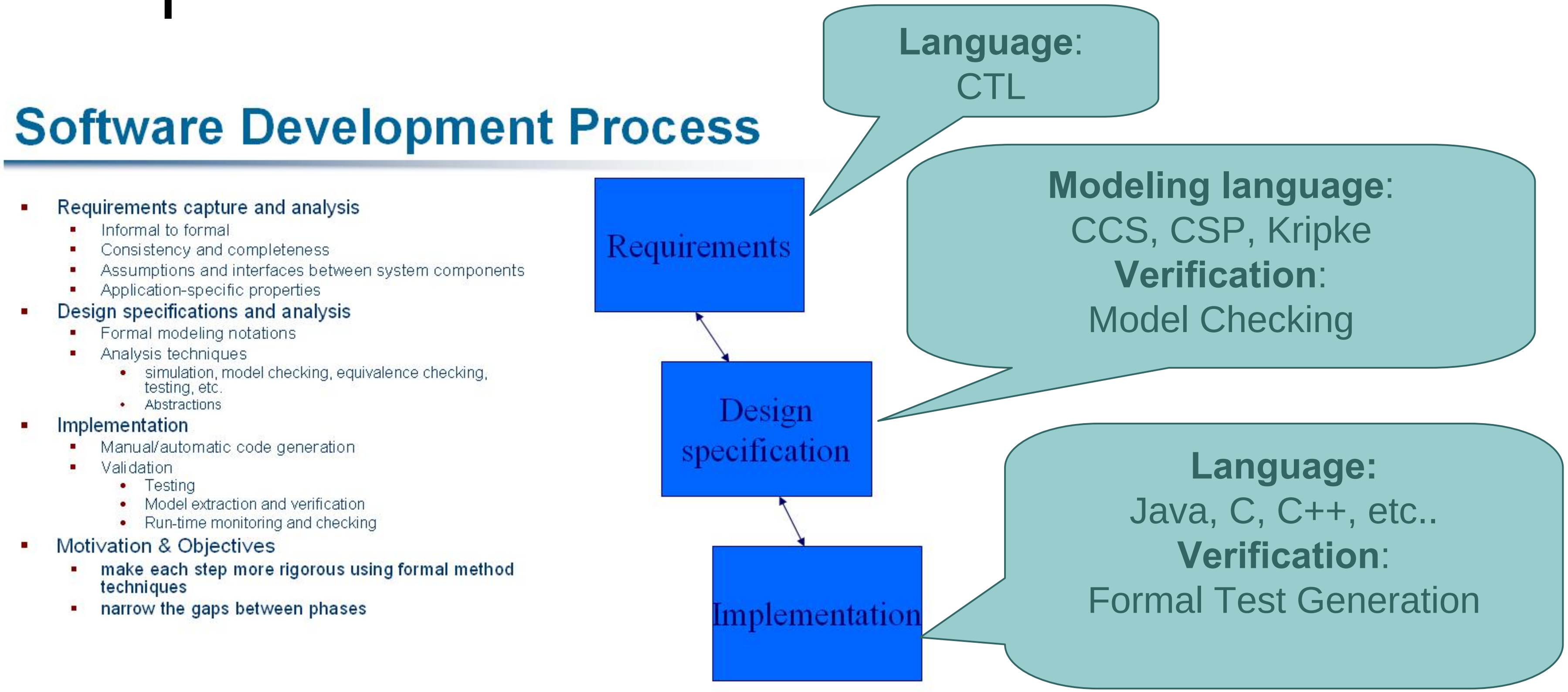
Runtime Verification (RV)

Usa Sammapun
University of Pennsylvania
September 29, 2004

Introduction



Software Development Process

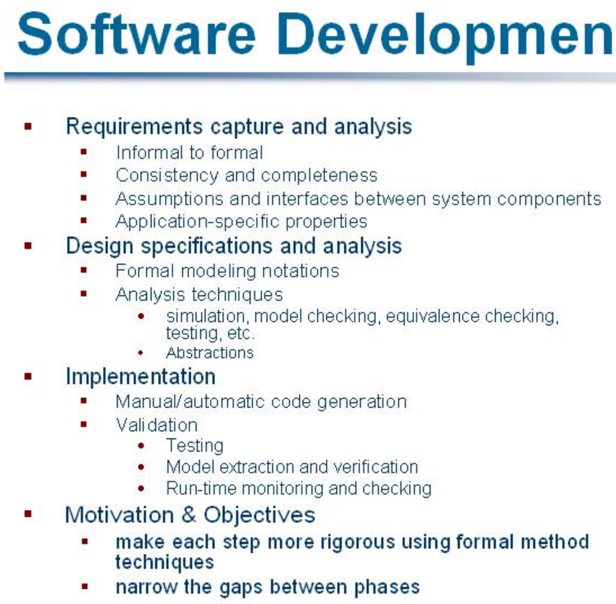
- Requirements capture and analysis
  - Informal to formal
  - Consistency and completeness
  - Assumptions and interfaces between system components
  - Application-specific properties
- Design specifications and analysis
  - Formal modeling notations
  - Analysis techniques
    - simulation, model checking, equivalence checking, testing, etc.
    - Abstractions
- Implementation
  - Manual/automatic code generation
  - Validation
    - Testing
    - Model extraction and verification
    - Run-time monitoring and checking
- Motivation & Objectives
  - **make each step more rigorous using formal method techniques**
  - **narrow the gaps between phases**

Language:
CTL

Modeling language:
CCS, CSP, Kripke
Verification:
Model Checking

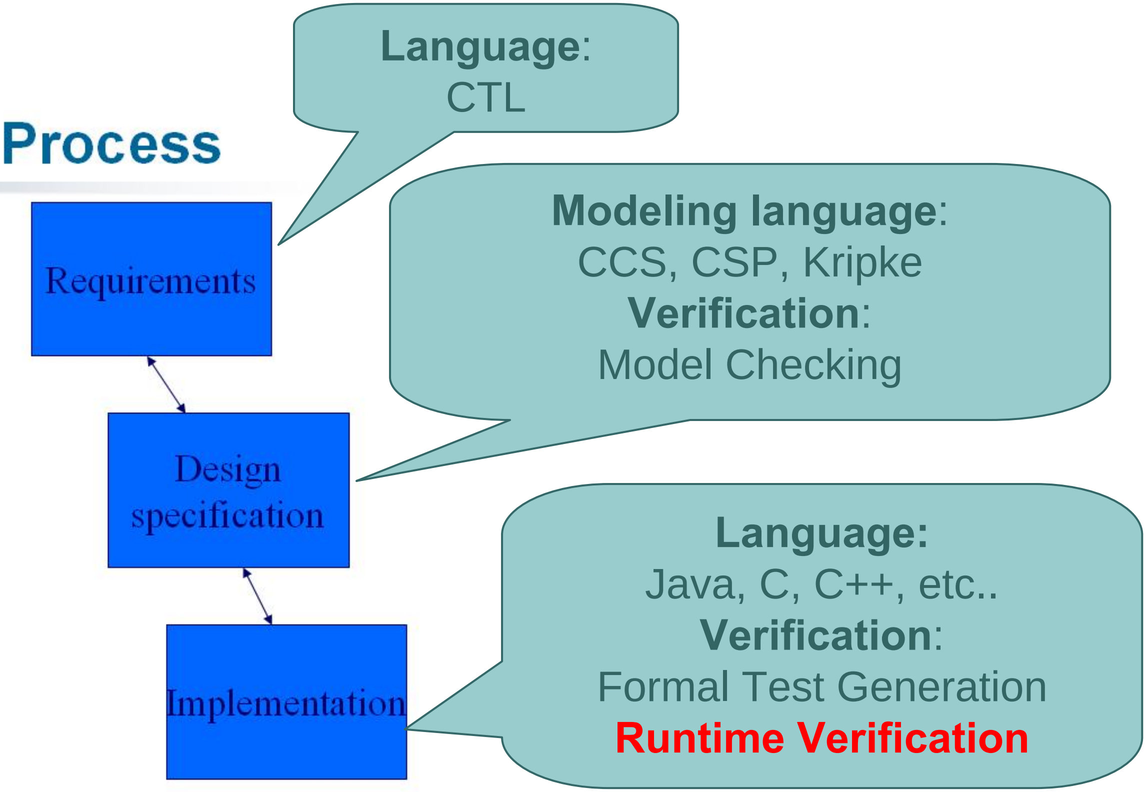
Language:
Java, C, C++, etc..
Verification:
Formal Test Generation

Introduction



Software Development Process

- Requirements capture and analysis
  - Informal to formal
  - Consistency and completeness
  - Assumptions and interfaces between system components
  - Application-specific properties
- Design specifications and analysis
  - Formal modeling notations
  - Analysis techniques
    - simulation, model checking, equivalence checking, testing, etc.
    - Abstractions
- Implementation
  - Manual/automatic code generation
  - Validation
    - Testing
    - Model extraction and verification
    - Run-time monitoring and checking
- Motivation & Objectives
  - **make each step more rigorous using formal method techniques**
  - **narrow the gaps between phases**



Language:
CTL

Modeling language:
CCS, CSP, Kripke
Verification:
Model Checking

Language:
Java, C, C++, etc..
Verification:
Formal Test Generation
Runtime Verification

Motivation

- Limitation of current verification techniques
  - Model checking
  - Testing

Model Checking

- **Pro**
  - Formal
  - Complete – Provides guarantees
- **Con**
  - Doesn’t scale well
  - Checks design, not implementation

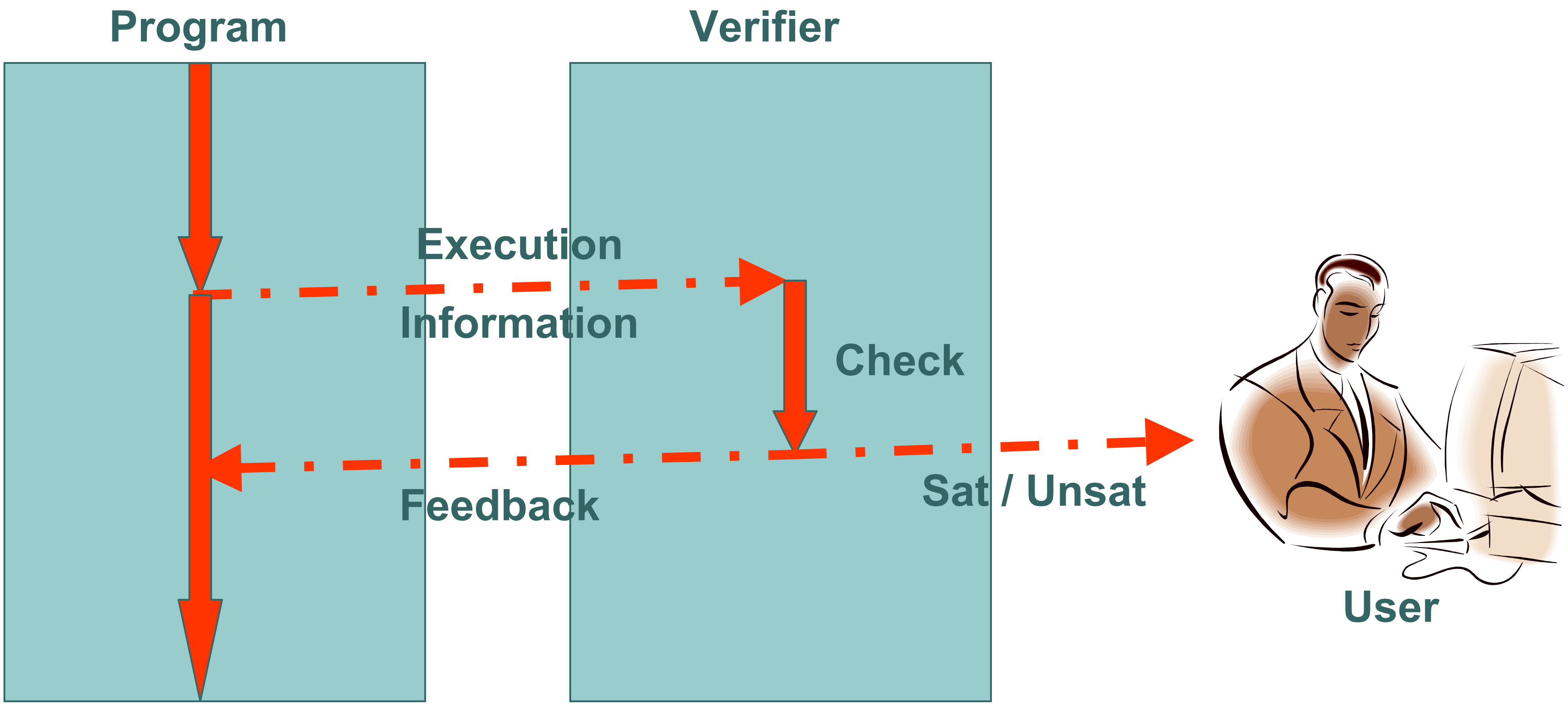
Testing

- **Pro**
  - Scales well
  - Tests an implementation directly
- **Con**
  - Informal
  - Incomplete – Doesn’t provide guarantees.

How does RV verify?

- **1.** **Specify** formal requirements
- **2.** **Extract** information from current executing program
- **3.** **Check** the execution against formal requirements

Runtime Verification



Program

Verifier

Execution

Information

Check

Feedback

Sat / Unsat

User

Runtime Verification

- Formal
- Done at implementation
- Not complete
  - Guarantee for current execution

JPaX: Java PathExplorer

Klaus Havelund
Grigore Rosu
(NASA)

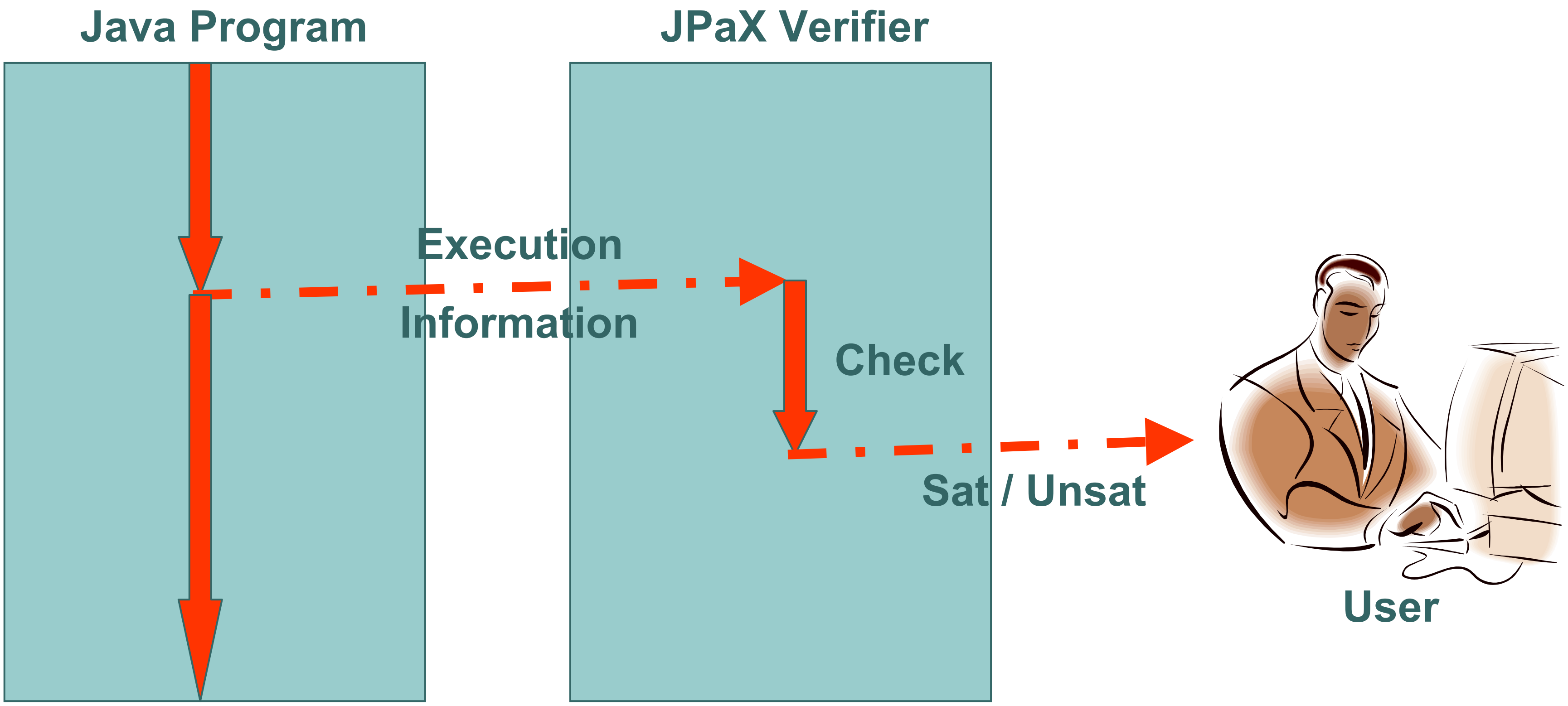
[HR01, HR04]

JPaX

- Checks the execution of **Java** program
  - During program testing to gain info about execution
  - During operation to survey safety critical systems

- Extracts interesting events from an executing program
- Checks those events
  - Logic based monitoring
  - Error pattern analysis
    - Deadlock
    - Data race

JPaX



Java Program

JPaX Verifier

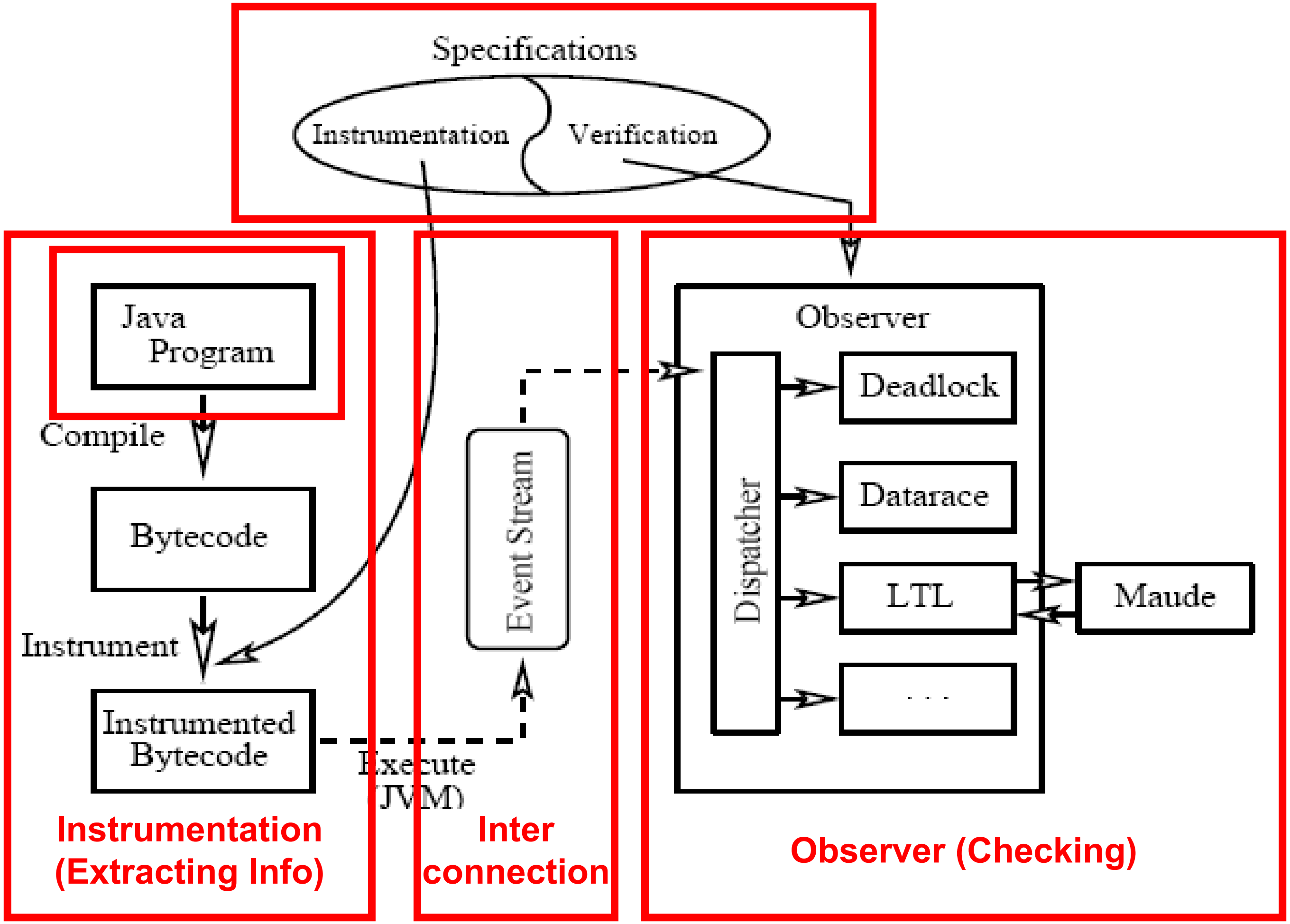
Execution Information

Check

Sat / Unsat

User

JPaX Verifier



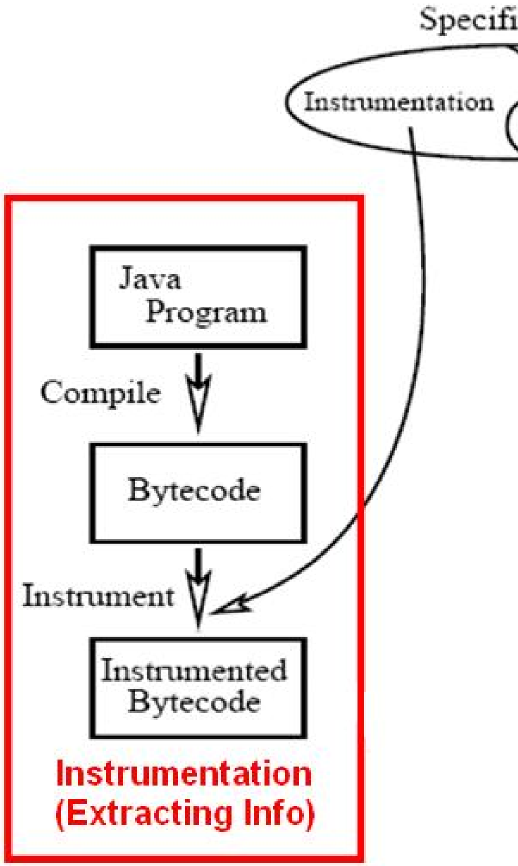
Instrumentation (Extracting Info)

Inter connection

Observer (Checking)

Instrumentation Module: How JPaX extracts info

- Given
  - Java bytecode
  - Instrumentation specification
- To extract
  - Examine java bytecode
  - Insert some code at places specified instrumentation specification
    - Logic based / error pattern analysis
  - Send this info to the observer



Insert Code: Logic Based

```
class C {
  int x;
  main() {
     x = -1;
     x = -2;
     x = 1;
     x = -3;
} }
```

+

```
instrumentation:
  monitor C.x;
  proposition A is C.x > 0
```

=

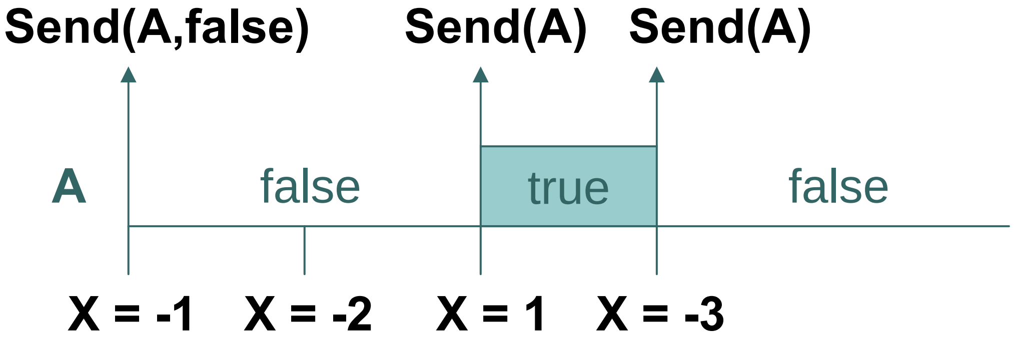
```
class C {
  int x;
  main() {
     x = -1;
     send(x,-1);
     x = -2;
     send(x,-2);
     x = 1;
     send(x,1);
     x = -3;
     send(x,-3);
}  }
```

Sent to observer:

[(x,-1), (x,-2), (x,1), (x,-3)]

Not all info is needed

```
instrumentation:
  monitor C.x;
  proposition A is C.x > 0
```



```
class C {
  int x;
  main() {
      x = -1;
      eval(x,-1);
      x = -2;
      eval(x,-2);
      x = 1;
      eval(x,1);
      x = -3;
      eval(x,-3);
}  }
```

Sent to observer:

[(A,false), A, A]

Not all info is needed

- What **eval(x,value)** does
  - Look at all propositions **P** corresponding to variable **x**
  - Evaluate the value of **P** (true, false)
    - Using value of x
  - If **P** has no value,
    - Send event **(P, P_val)** to observer
  - Else
    - If **P** changes value,
    - Send **(P)** to observer

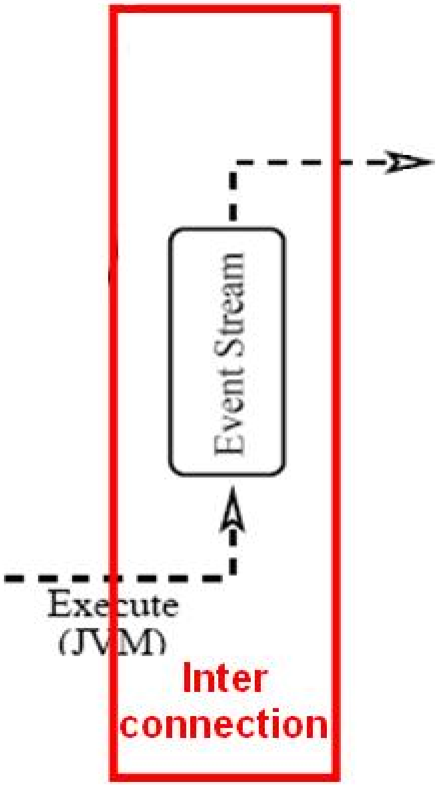
Insert Code: Error Pattern

- Instead of sending propositions to the observer
- Sends events
  - Acquiring locks (deadlock, data race)
  - Releasing locks (deadlock, data race)
  - Accessing variables (data race)

Interconnection Module

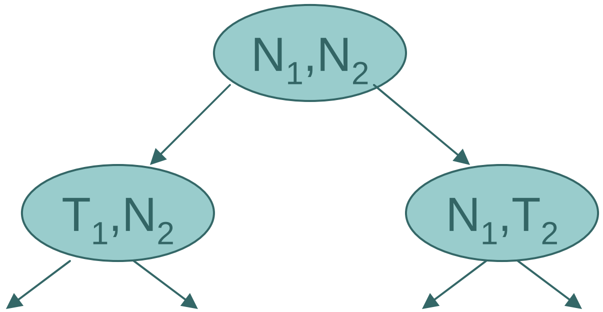
- Send extracted info
  - From the <u>java program</u> to the <u>observer</u>
  - Via socket, shared memory, file

- Extraced Info
  - Event stream

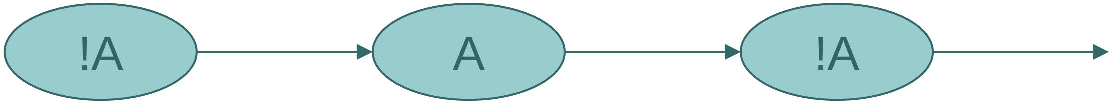


Event Stream

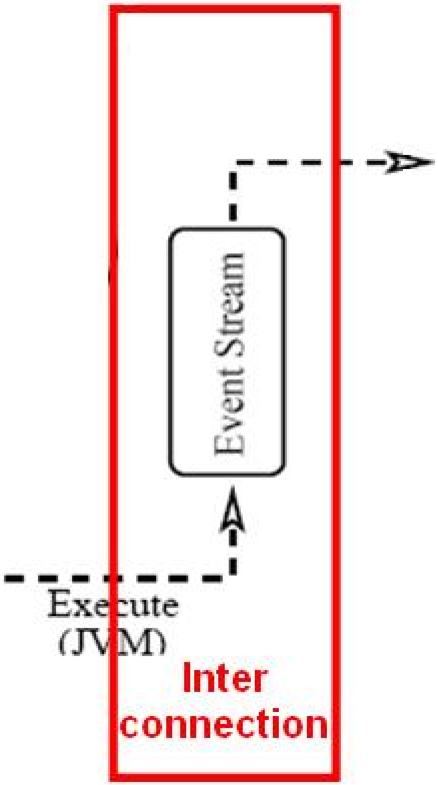
- Similar to Kripke structure
- Kripke



- Event Stream (Trace)

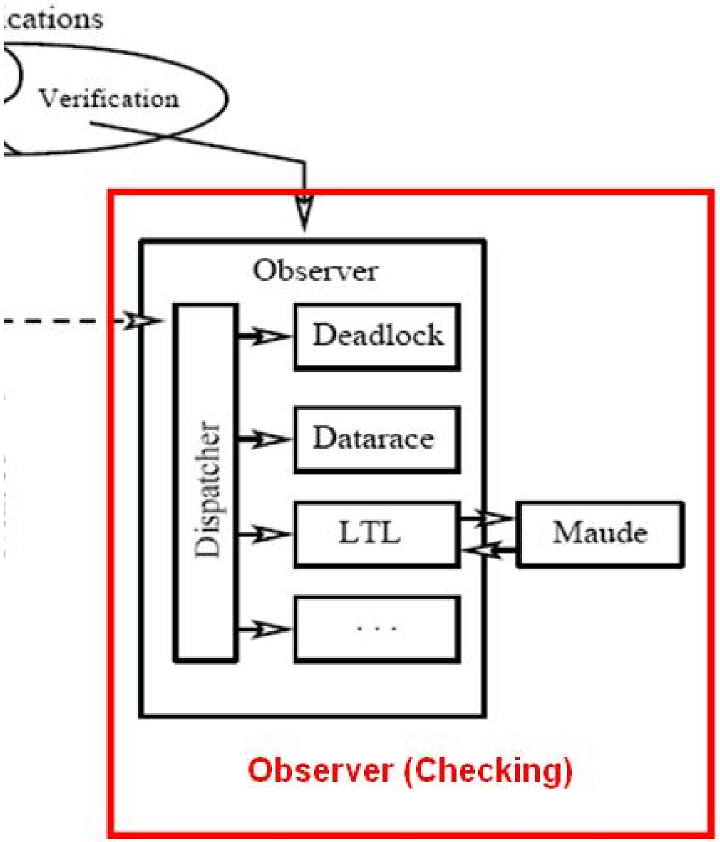


[(A,false), A, A]



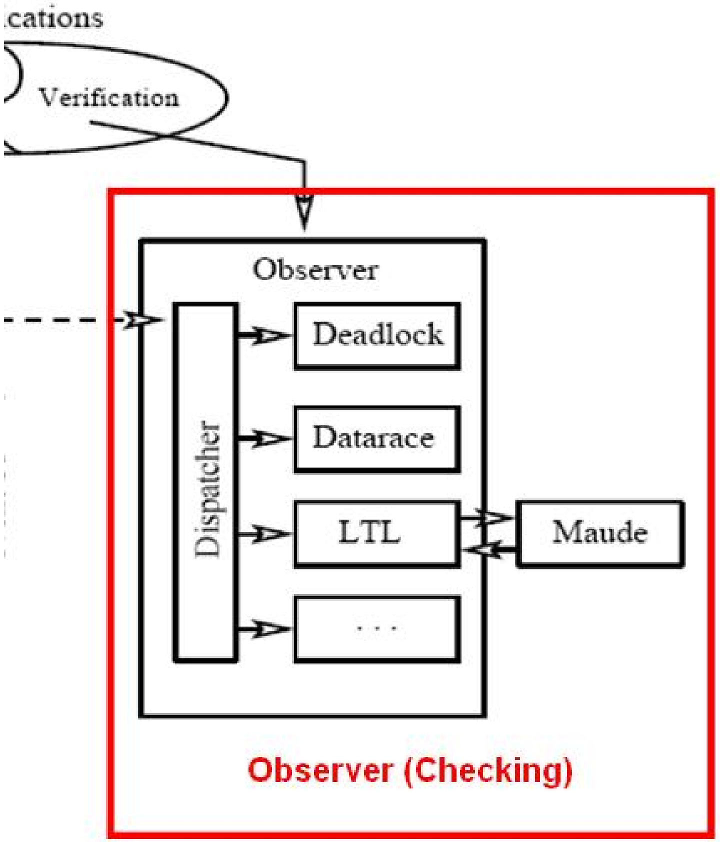
Observer Module

- Runs in parallel with the Java program
- Monitors and analyzes
- 2 Components
  - Logic based monitoring
  - Error Pattern Analysis
    - Deadlock
    - Data race



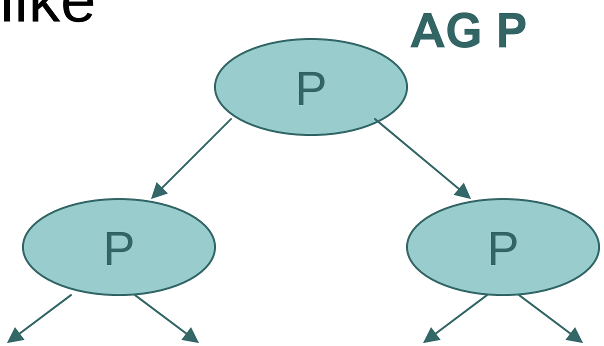
1. Logic Based Monitoring

- Given
  - Trace (Event stream)
  - Specification in some logic
- To check
  - Check if properties in specification hold in the trace



Logic Based Monitoring

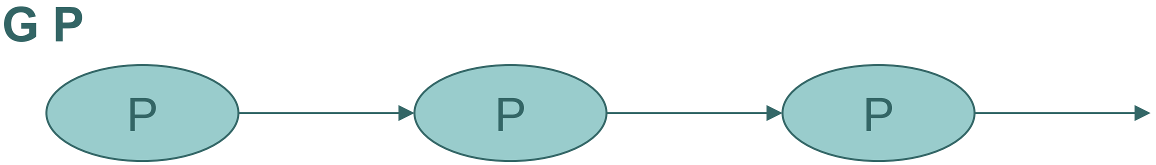
- Logic
  - CTL – Model checking
    - A**G**, E**G**, A**F**, E**F**, A**X**, E**X**, A**U**, E**U**
    - Tree like



- Not appropriate for event stream
  - Only has one path

LTL

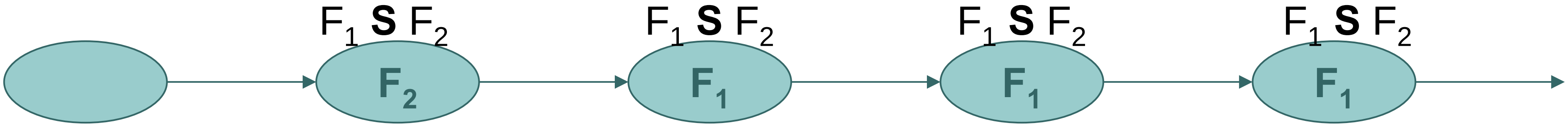
- LTL – Linear Temporal Logic
  - G, F, X, U (□, ◇, ○, U)
  - Linear



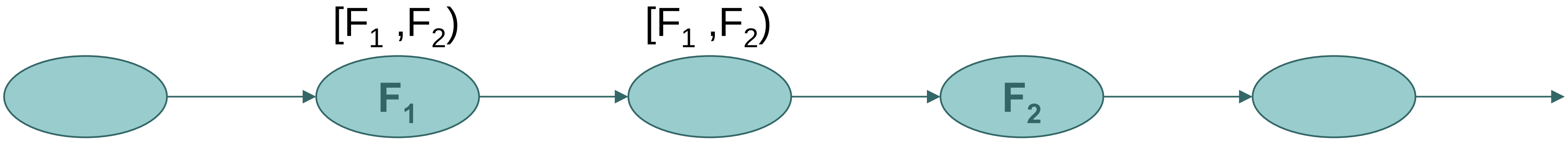
```
instrumentation:
  monitor C.x;
  proposition A is C.x > 0
verification:
  formula F1 is <> A
```

Past Time LTL (ptLTL)

- More natural for RV
  - ⊙ F – previous F (as oppose to next)
  - ⊡ F – always F in the past
  - ◈ F – eventually F in the past
  - F_1 S F_2 – F_1 since F_2



  - $[F_1, F_2)$ – interval F_1, F_2



JPaX Checking

- Given
  - Trace
  - LTL or ptLTL
- Check
  - Use “Maude engine” to check
  - Or dynamic programming
- Result
  - True or false
  - We want it to always return true
    - The requirement is satisfied
    - Nothing bad has happened

Maude

- Rewriting engine
  - Treat LTL or ptLTL as an equation
  - “Rewrite” or “consume” this LTL/ptLTL equation and produce a new equation
    - A new equation – a new state
    - Normal form (true or false)
    - Just another LTL or ptLTL

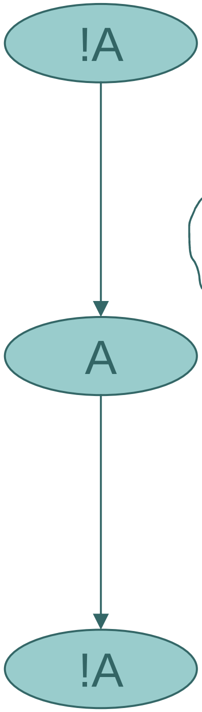
Rewrite LTL

```
** propositional logic **
eq true /\ X = X
eq false /\ X = false
eq true \/ X = true
eq false \/ X = X
eq X /\ (Y \/ Z) = (X /\ Y) \/ (X /\ Z)

Eq (X /\ Y){As} = X{As} /\ Y{As}
Eq (X \/ Y){As} = X{As} \/ Y{As}

** LTL **
eq ([] X){As} = ([] X) /\ X{As}
eq (<> X){As} = (<> X) \/ X{As}
eq (o X){As} = X
eq (X U Y){As} = Y{As} \/ ( X{As} /\ (X U Y) )
```

X{As} = assignment of a boolean value to a variable X

Example

```
instrumentation:
  monitor C.x;
  proposition A is C.x > 0
verification:
  formula F1 is <> A
```



```
Here A is false
eq (<> A){As} = (<> A) \/ A{As}
              = (<> A) \/ false
              = (<> A)
```

```
Here A is true
eq (<> A){As} = (<> A) \/ A{As}
              = (<> A) \/ true
              = true
```

```
Here A is false
eq true
```

Dynamic Programming

- ptLTL
- For each formula P
  - Divide P into subformulae
  - Keep the value of each proposition and subformulae from the previous state (pre[ ])
  - Calculate the value of each subformulae for current state (now[ ]) by using pre[ ] and now[ ]

Dynamic Programming

- Propositional logic
  - now[x \/ y] = now[x] \/ now[y]
  - now[x /\ y] = now[x] /\ now[y]
  - now[!x] = ! now[x]
- ptLTL
  - now[(.) x] = pre[x]
  - now[[.] x] = pre[[.] x] /\ now[x]
  - now[<.>x] = pre [<.> x] \/ now[x]
  - now[x S y] = now[y] \/ now[[(.) y, !x)]
  - now[[x,y)] = (pre[[x,y)] \/ now[x]) /\ !now[y]

Example: x ∧ [y, z)

```
bit pre[0..4]
bit now[0..4]
INPUT: trace t = $e_1e_2e_3...e_n$;

Subformulae:
0: x /\ [y, z)
1: x
2: [y, z)
3: y
4: z

Init:
pre[4] = z(state);
pre[3] = y(state);
pre[2] = pre[3] and not pre[4];
pre[1] = x(state);
pre[0] = pre[1] and pre[2];
```

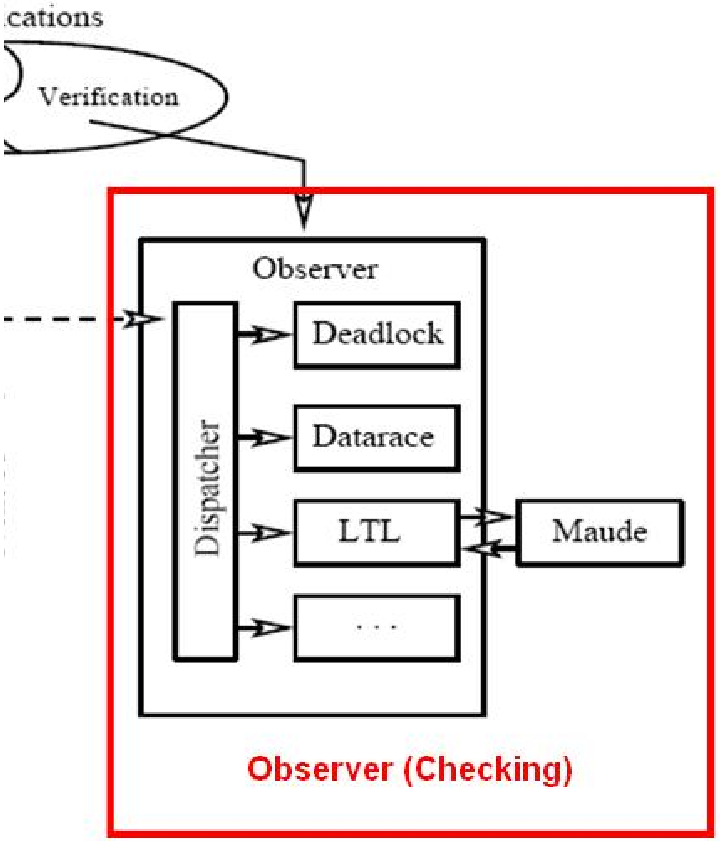
```
for i = 2 to n do {
 state = update(state,$e_i$);

 now[4] = z(state);
 now[3] = y(state);
 now[2] = (pre[2] or now[3])
          and not now[4];
 now[1] = x(state);
 now[0] = now[1] and now[2];

 if now[0] = 0 then
  output('property violate');

 pre = now;
}
```

Running Time

- At one point
    - O (m)
        - m = size of formula
- Overall
    - O (n m)
        - n = number of events
        - m = size of formula

2. Error Pattern Analysis

- Use well-known algorithm to detect
  - Data race
  - Deadlock



Data Race – Cause

- Cause
  - Two or more concurrent threads
  - Access a shared variable
    - At least one access is write
  - No explicit critical section mechanism

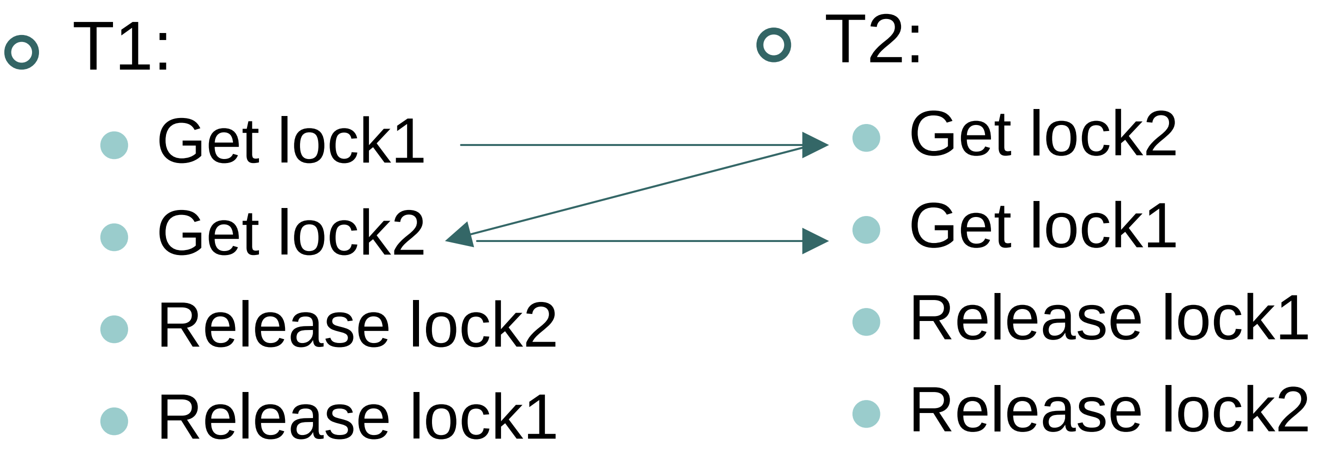
```
Init x = 0;
T1:               x=0  T1 reads      x=0  T1 reads
 x = x+1;         x=1  T1 writes     x=0  T2 reads
T2:               x=1  T2 reads      x=1  T1 writes
 x = x+10;        x=11 T2 writes     x=10 T2 writes
```

Data Race – Check

- Events
  - Acquiring, releasing locks
  - Shared variable accessing
- Checks – make sure that
  - The lock is held by any thread whenever it accesses the variable

Deadlock – Cause

- Order of acquiring and releasing locks



- T1:
  - Get lock1
  - Get lock2
  - Release lock2
  - Release lock1
- T2:
  - Get lock2
  - Get lock1
  - Release lock1
  - Release lock2

Deadlock – Check

- Events
  - Acquiring, releasing locks
- Checks
  - Thread map – keep track of locks owned by each thread
  - Lock graph – edge record locking orders
    - Introduce from a lock to another lock each time when a thread that already owns the first lock acquires the other
  - If lock graph is cyclic, deadlock potential

That’s it for JPaX

- Specification Logic
  - LTL
- Information Extraction
  - Instrument bytecode
  - Events
    - Propositions
    - Get/Release locks, variable access
- Check
  - Rewriting engine
  - Dynamic programming
  - Error pattern analysis

Other RV tools

- Different Logic
  - LTL
  - Timed LTL [TR04]
    - $F_1 \, U^{<t} \, F_2$
  - (Extended) Regular Expression [CR03]
  - Interval logic
  - Automata [LBW03]

Extracting Information

- **From bytecode**
  - Instrument bytecode [HR01, HR04, KKL+04]
    - Code, specification in different files
    - Normal compiler
- **From sourcecode**
  - Instrument source code
    - Code, specification in different files
    - Normal compiler
  - Specification embedded in source code [LBW03, Dru03]
    - Special compiler translates specification into some code
- **Use debugger** [BM02]
    - Does not modify program code
    - Configure the debugger to generate events at desirable points in the code

Checker

- Rewriting engine
  - Maude
- Dynamic programming
- Translate LTL to Automata [CR03]
  - States – states in a trace
  - Transitions – inputs are events
  - Accepting states – satisfied

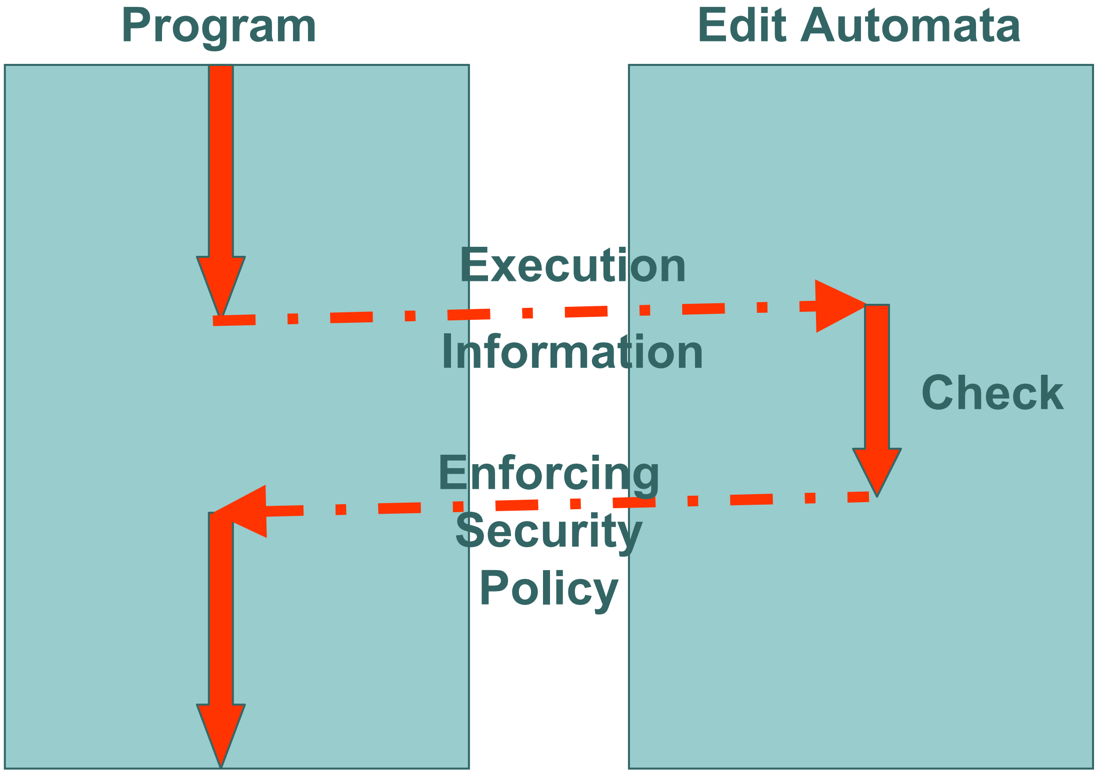
Same technique, different purpose

- Security
  - Check security policy
    - Edit automata
    - Model-carrying code
    - Intrusion detection

Edit Automata [LBW03]

- How can we run untrusted code on our machine?
  - Use monitor, called ‘*edit automata*’
    - Analogous to the JPaX observer
  - ‘*edit automata*’ monitors and enforces security policies
    - Analogous to the JPaX specification

Edit Automata



Program

Edit Automata

Execution Information

Check

Enforcing Security Policy

User

Edit Automata

- Specification Logic (Security policy)
  - Automata
- Information Extraction
  - Embedded in source code
  - Events
    - Actions (Method calls)
- Check
  - Automata

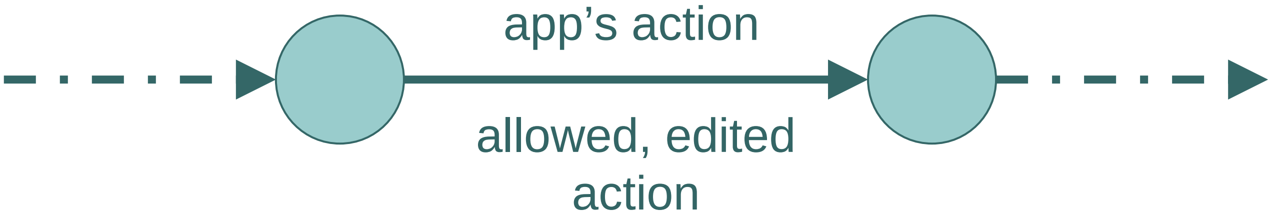
Enforcing Policy

- When it recognizes a dangerous operation, it may
  - halt the application
  - suppress (skip) the operation but allow the application to continue
  - insert (perform) some computation on behalf of the application

This slide is taken from David Walker

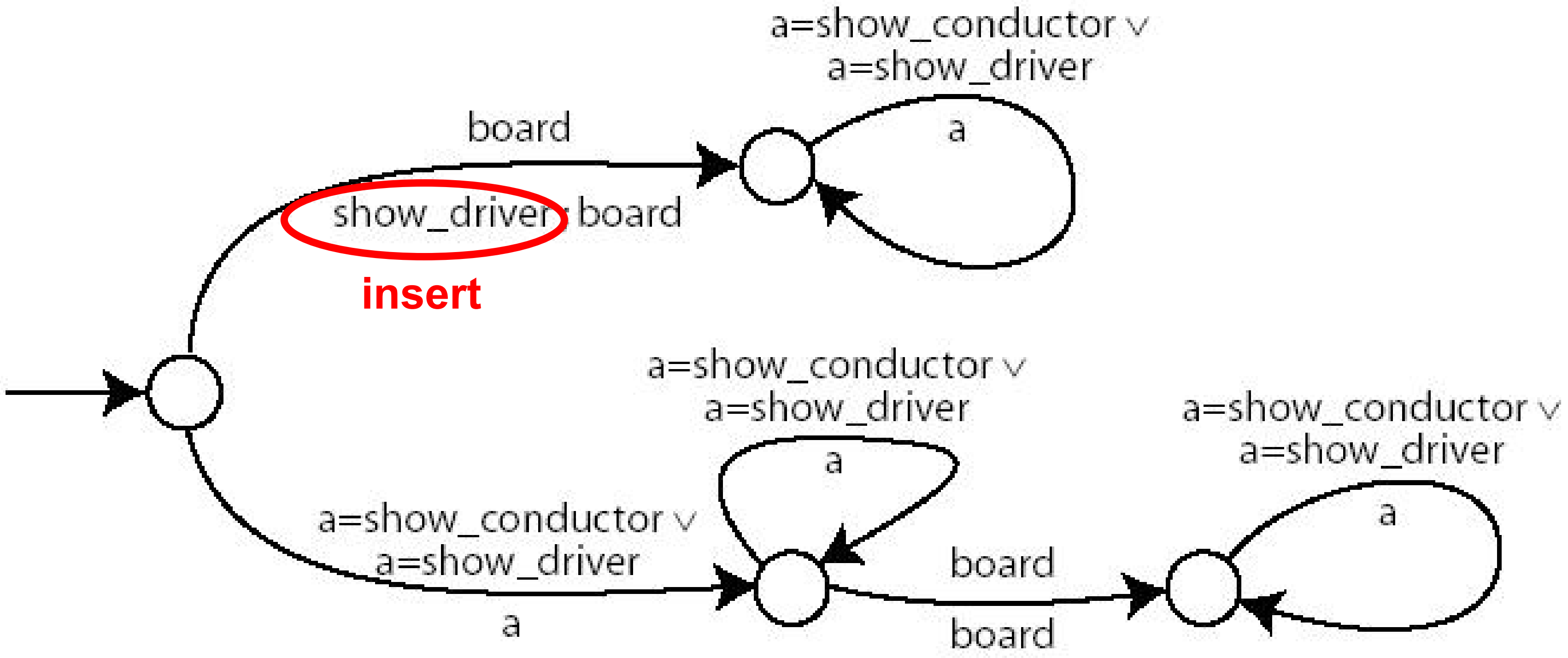
Security Policy

- Automata



Example

- **Program:** Cable car
- **Policy:** Showing ticket policy



Model-Carrying Code [SVB+03]

- How can we run untrusted code on our machine?
  - Untrusted code comes with a model of its security-relevant behavior
  - Users have their own security policies

Two checking

- Does untrusted program’s model respect user’s security policy?
  - Use model checking to check
    - Security policy is a specification
- Does model capture program behavior?
  - Use runtime checking
    - Model is a specification (Automata)
    - Events are system calls

Intrusion Detection

- 2 Approaches for ID
  - Anomaly-based
    - Behavior deviates from normal behavior is an intrusion
  - Signature-based
    - Define patterns of bad behaviors or attacks
    - Anything fits the patterns is an intrusion

Intrusion Detection using RV [NST04]

- Signature-based
  - Use LTL to define attack pattern
- Use runtime verification
  - Runs in parallel
  - Observes behaviors of programs
  - Check if behaviors match LTL attack pattern
  - If so, raises an alarm

Conclusion

- Lightweight Verification alternative to model checking and testing
  - Formal
  - Done at Implementation
  - Security

- Development
  - Multithreaded [SRA03]
  - Distributed [SVAR04]
  - Probabilistic

References - RV

- **[HR04]** Klaus Havelund and Grigore Rosu. An Overview of the Runtime Verification Tool Java PathExplorer. *Journal of Formal Methods in System Design*, 24(2):189-215, 2004.
- **[HR01]** Klaus Havelund and Grigore Rosu. Java PathExplorer - A runtime verification tool. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, (ISAIRAS'01)*, Montreal, Canada, Jun 2001.
- **[TR04]** Prasanna Thati and Grigore Rosu. Monitoring Algorithms for Metric Temporal Logic Specifications. In *Proceedings of the 4nd International Workshop on Run-time Verification*, Apr 2004.
- **[CR03]** Feng Chen and Grigore Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
- **[BM02]** Mark Brörkens and Michael Möller. Dynamic Event Generation for Runtime Checking using the JDI. In Klaus Havelund and Grigore Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier, 2002.

References - RV

- **[KKL+04]** Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-MaC: a Run-time Assurance Approach for Java Programs. *Formal Methods in Systems Design*, 24(2):129-155, Mar 2004.
- **[Dru03]** Doron Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proceedings of the 2003 Computer Aided Verification Conference (CAV)*, volume 2725, pages 114-118. Springer-Verlag, Jul 2003.
- **[SRA03]** Koushik Sen, Grigore Rosu, and Gul Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, (FSE/ESEC 03)*, pages 337-346, Helsinki, Finland, Sep 2003.
- **[SVAR04]** Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE 04)*, 2004.

References - Security

- **[LBW03]** Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. Technical report, Princeton University, Computer Science Department, May 2003.
- **[SVB+03]** R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications . In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 214-228, Bolton Landing, New York, Oct 2003.
- **[NST04]** Prasad Naldurg, Koushik Sen, Prasanna Thati. A Temporal Logic Based Approach to Intrusion Detection. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004),* 2004.

The End