jPredictor
A Predictive Runtime Analysis Tool for Java

Traian Florin Șerbănuță
(joint work with Feng Chen and Grigore Roșu)
University of Illinois at Urbana-Champaign

Challenge

- Concurrent programs are hard to analyze
  - Model checking: number of interleavings is prohibitively large
  - Testing: interleavings depend on environment
- What can we do about it?

Combine dynamic and static methods:
Find bad behaviors near correct executions.

Our Solution

- Sliced Causality
  - General purpose technique to predict (bad) behaviors from correct runs
  - Sound: predictions are right
- jPredictor
  - Tool implementing Sliced Causality
  - Better than tools specialized for detecting dataraces or atomicity violations

Predicting Concurrency Errors

Property: "authenticate before access"

Main Thread:

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

Task Thread:

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

Observed execution: … s_1 s_2 s_3 s_4 …

Predicting Concurrency Errors

Property: "authenticate before access"

```
Main Thread:

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

```
Task Thread:

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

Observed execution: ... s_1 s_2 s_3 s_4 ...

- Buggy: s_4 can be executed before s_1
- Low possibility to hit the error in testing

Predicting Concurrency Errors

Property: "authenticate before access"

Main Thread:

s_1: resource.authenticate();

s_2: flag.value = true;

...

Task Thread:

...

...

s_3: **if** (! flag.value)

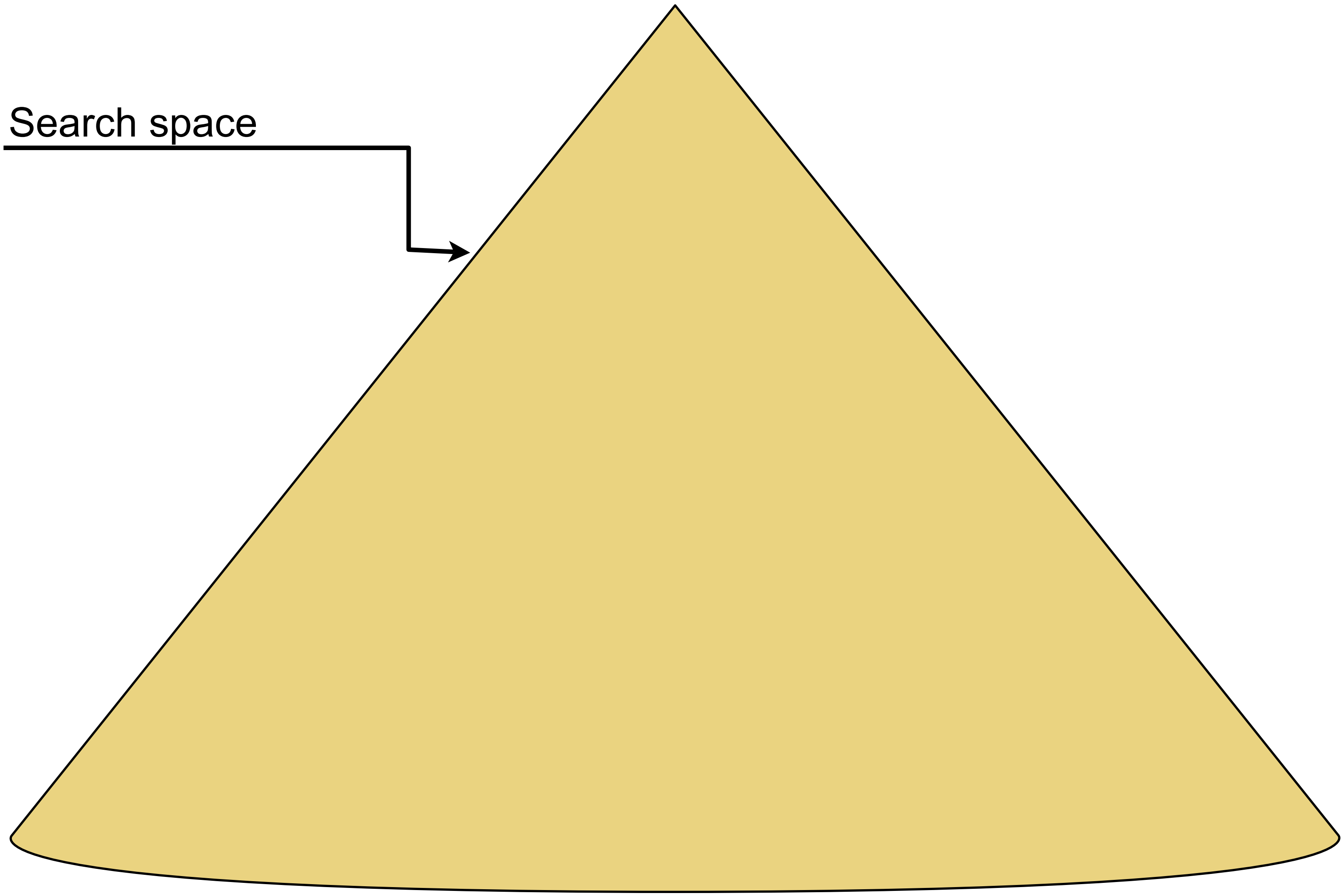
Thread.yield() ;

s_4: resource.access();

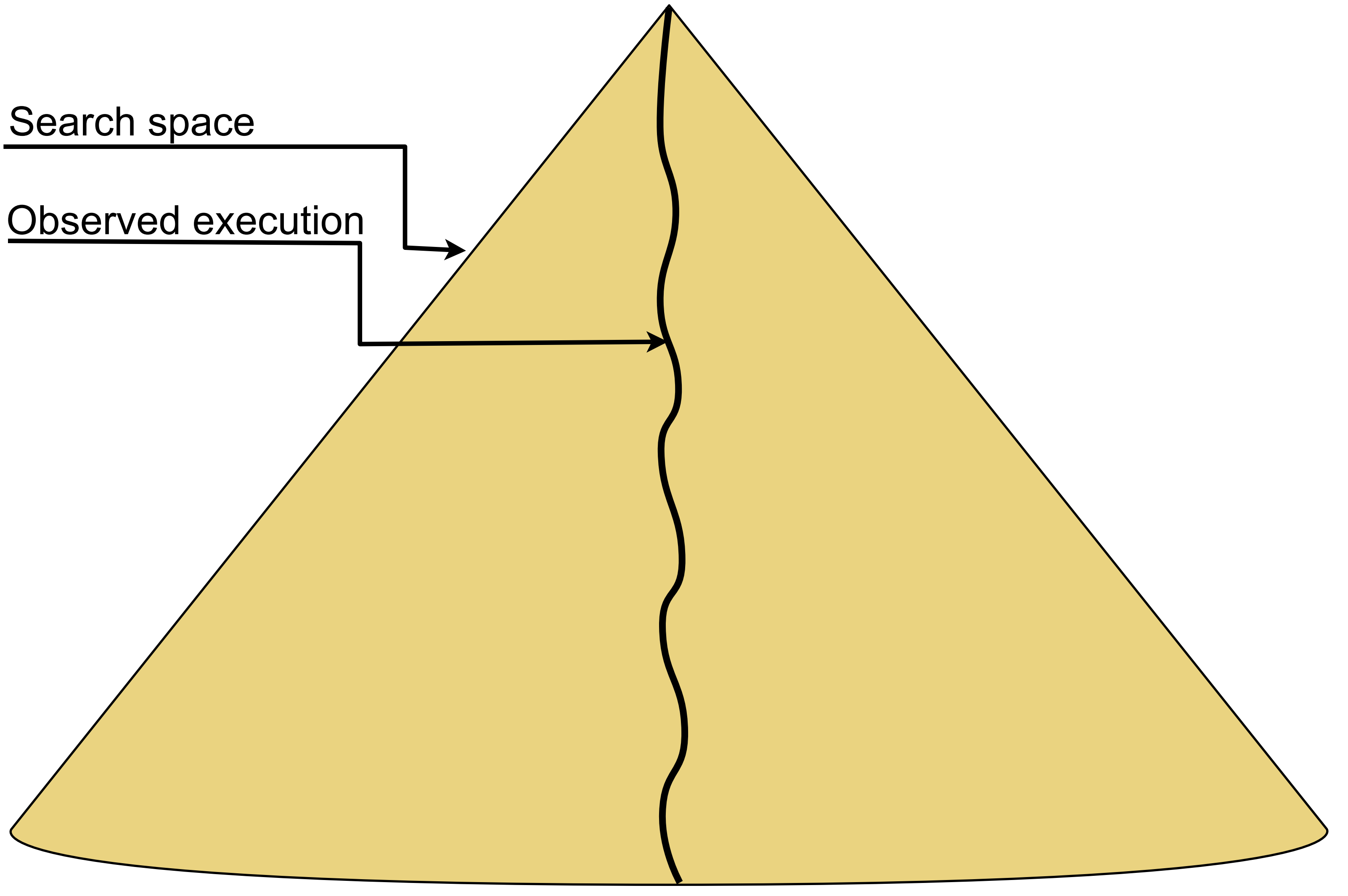
Can we predict the error even when the above execution is observed?
Yes! But not in a traditional way.

- Buggy: s_4 can be executed before s_1
- Low possibility to hit the error in testing

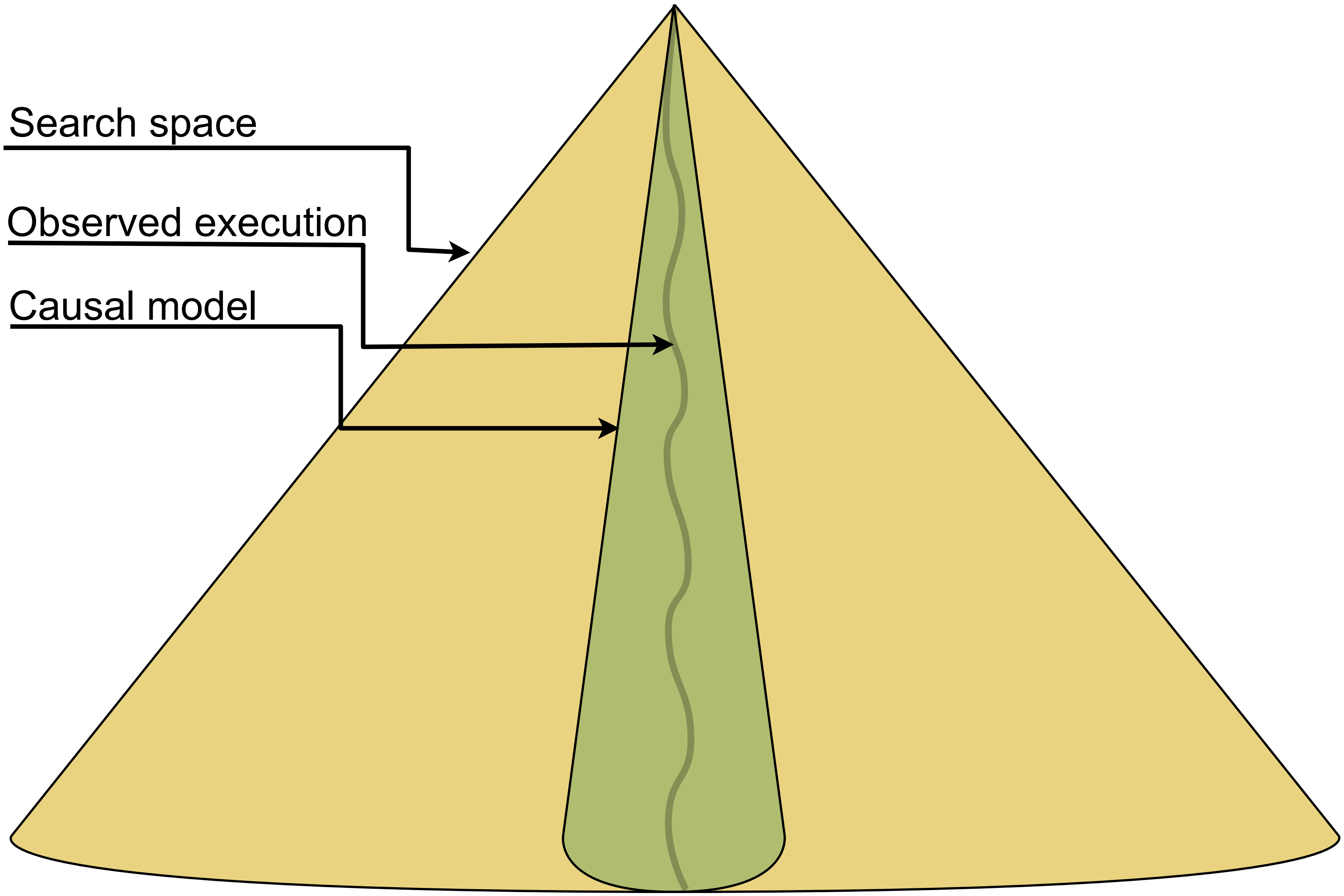
Predictive Runtime Analysis



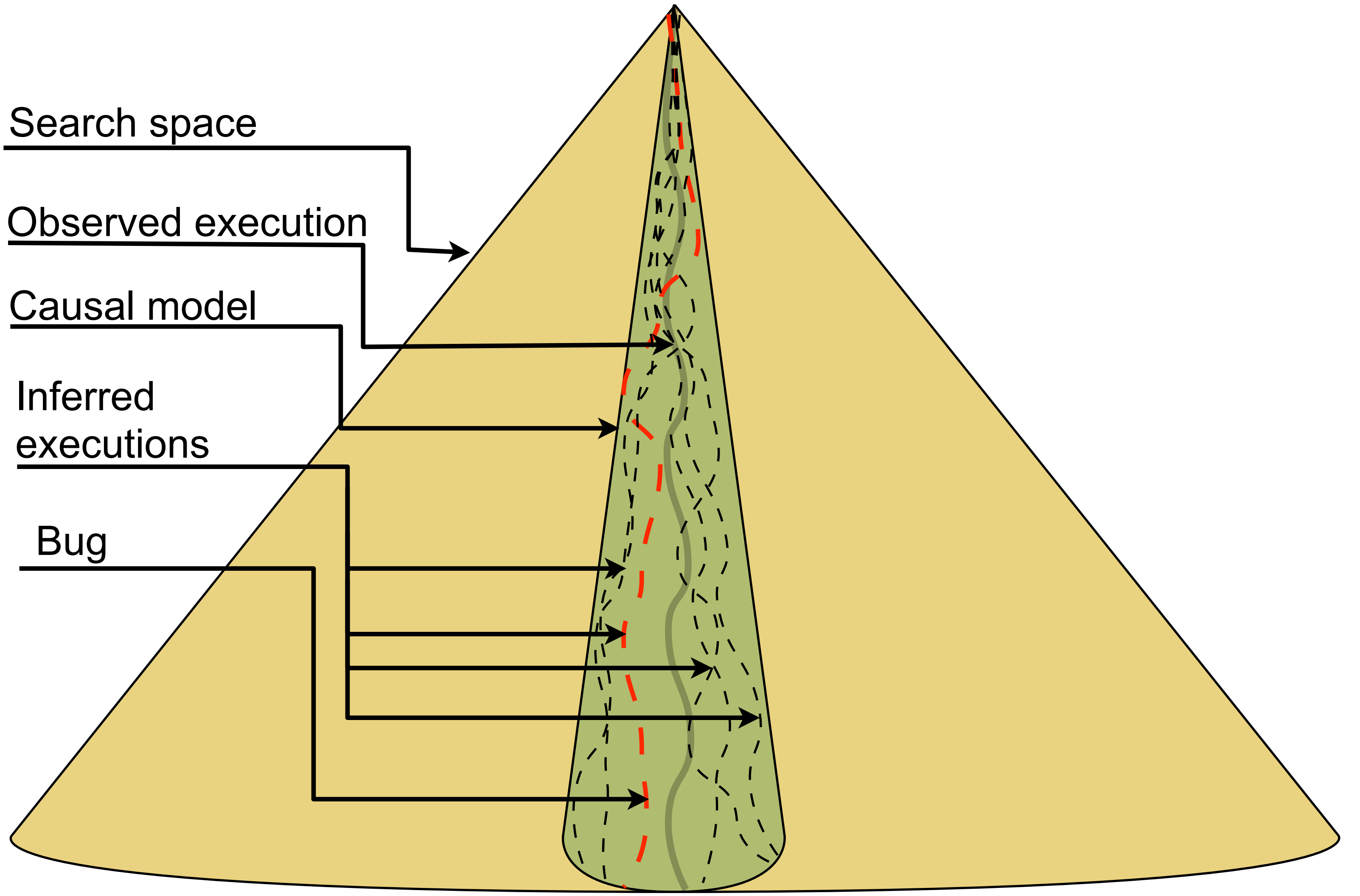
Predictive Runtime Analysis



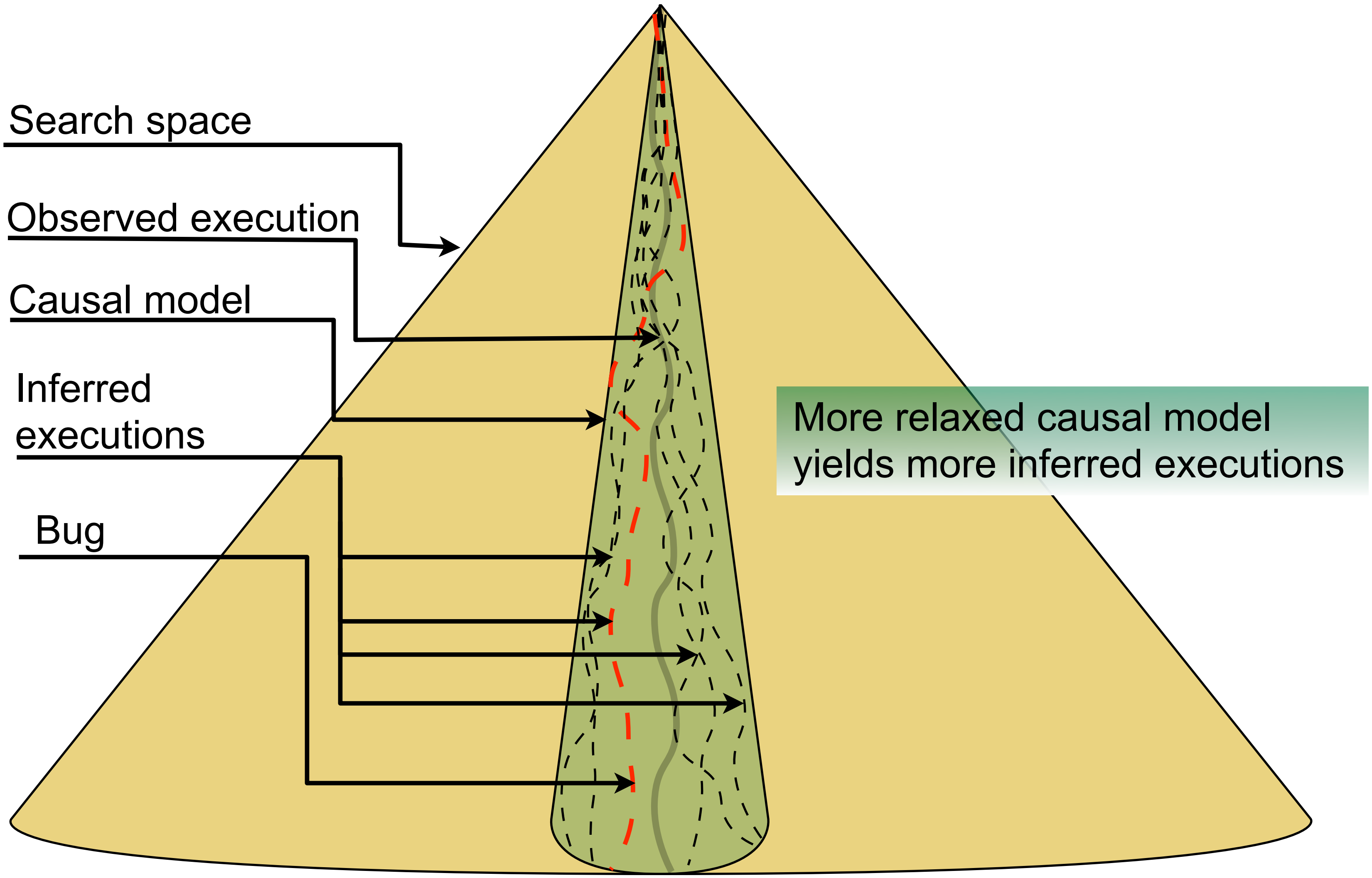
Predictive Runtime Analysis



Predictive Runtime Analysis



Predictive Runtime Analysis

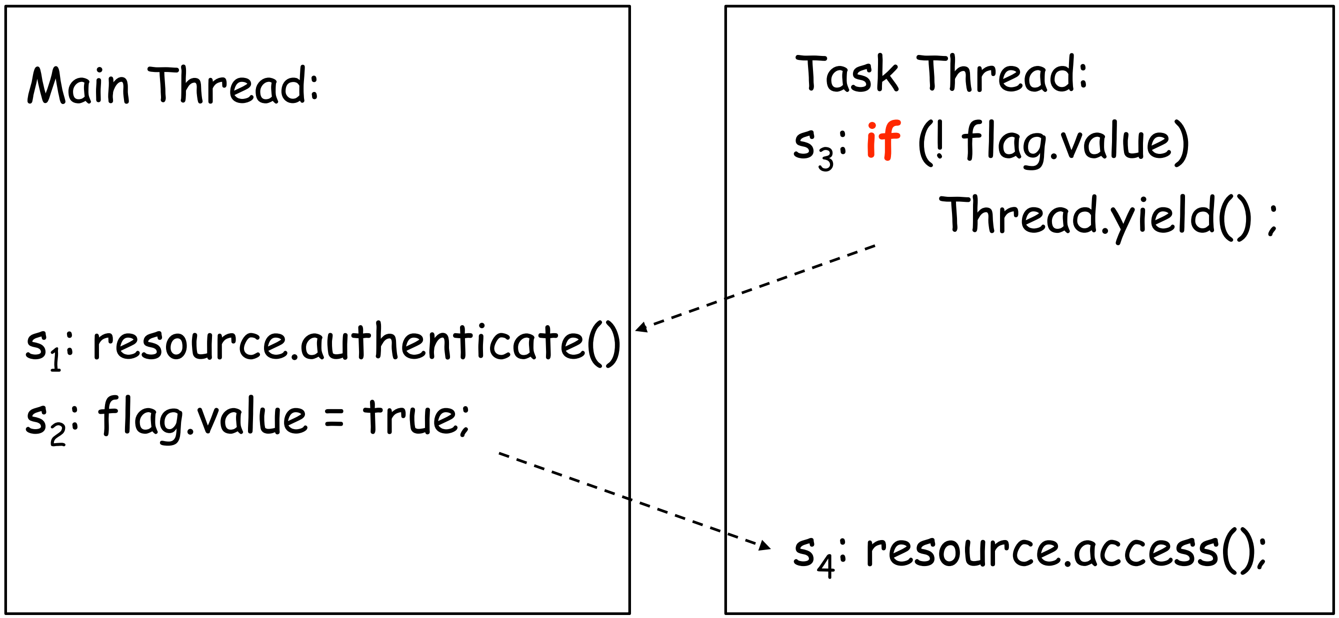


Traditional Predictive Runtime Analysis: Happens-Before

- Originally for distributed systems [Lamport-78]
  - Applied to shared-memory systems by many authors
- Causal model = non-permutable pairs of events
  - = {intra-thread total orders} U {causal dependencies}
  - Causal dependency: if two events access same location and one writes it, then their execution order matters
- Inferred executions = extending the causal model

Happens-Before Works ... If Lucky

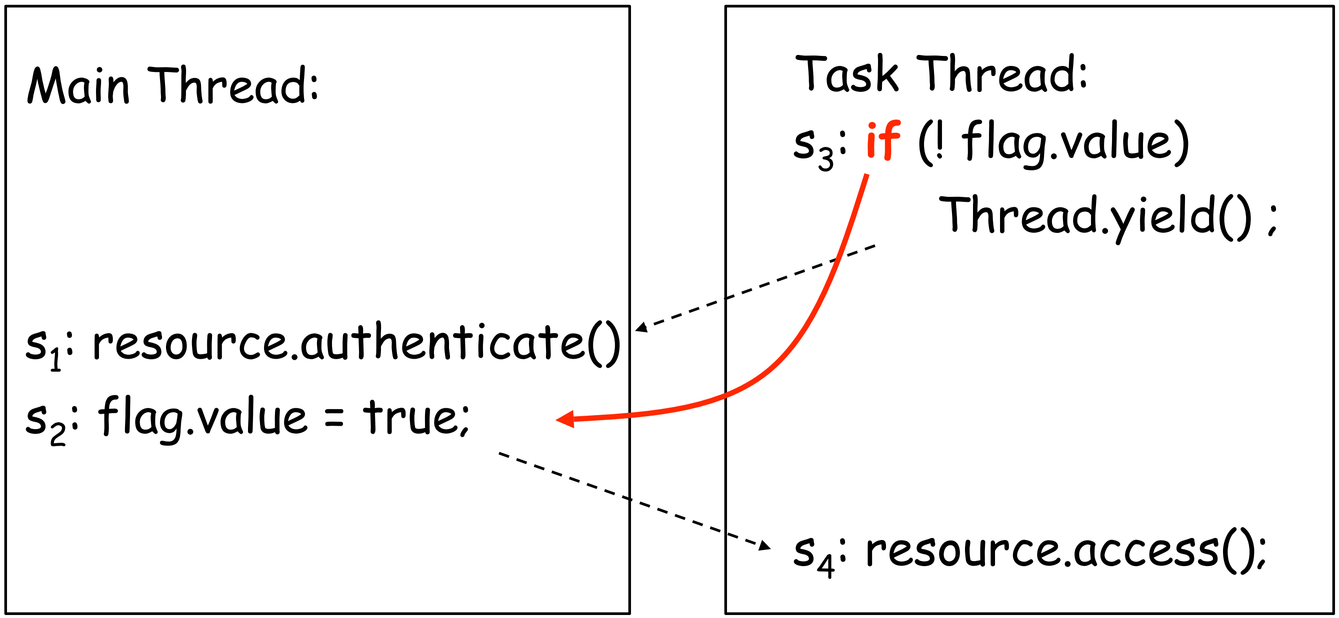
Property: “authenticate before access”



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

Happens-Before Works … If Lucky

Property: “authenticate before access”



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

Causal dependency: $s_3 < s_2$

Bad execution inferred: $s_3\ s_4\ s_1\ s_2$. Bug detected!

Chances of observing this execution are very low

Happens-Before: Limitations

Property: "authenticate before access"

Main Thread:

s_1: resource.authenticate()

s_2: flag.value = true;

Task Thread:

s_3: if (! flag.value)

Thread.yield() ;

s_4: resource.access();

Observed execution: s_1 s_2 s_3 s_4

Causal dependency: $s_2 < s_3$. No bug found …

Too constrained: access will be performed regardless of the flag

Our Technique: Sliced Causality

- Relax the Happens-Before causal model
  - Yields no false alarms: formally proved in [chen-rosu-07]
- How? Focus on the property!
- Use static information about the program
- Remove events and causalities irrelevant to property
  - Smaller and more relaxed causal model
  - (exponentially) more inferred executions
  - better predictive capability

Static Information: Control Scope

[chen-rosu-06]

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

```
s1: if (flag) {
s2:     ...
    } else {
s3:     ...
    }
s4: ...
```

```
s0: i=0;
s1: while (i<3) {
s2:     ...
s3:     i++
    }
s4: ...
```

```
s1: while (!flag) {
s2:     ...
    }
s3: ...
```

- Extends to other control statements
  - break/continue, return, exceptions

Static Information: Control Scope

[chen-rosu-06]

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

```
s1: if (flag) {
s2:     ...
    } else {
s3:     ...
    }
s4: ...
```

```
s0: i=0;
s1: while (i<3) {
s2:     ...
s3:     i++
    }
s4: ...
```

```
s1: while (!flag) {
s2:     ...
    }
s3: ...
```

- Extends to other control statements
  - break/continue, return, exceptions

Static Information: Control Scope

[chen-rosu-06]

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

```
s1: if (flag) {
s2:     ...
    } else {
s3:     ...
    }
s4: ...
```

```
s0: i=0;
s1: while (i<3) {
s2:     ...
s3:     i++
    }
s4: ...
```

```
s1: while (!flag) {
s2:     ...
    }
s3: ...
```

- Extends to other control statements
  - break/continue, return, exceptions

Static Information: Control Scope

[chen-rosu-06]

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

```
s1: if (flag) {
s2:     ...
    } else {
s3:     ...
    }
s4: ...
```

```
s0: i=0;
s1: while (i<3) {
s2:     ...
s3:     i++
    }
s4: ...
```

```
s1: while (!flag) {
s2:     ...
    }
s3: ...
```

- Extends to other control statements
  - break/continue, return, exceptions

Sliced Causality Works!

Property: “authenticate before access”

Main Thread:

s_1: resource.authenticate()

s_2: flag.value = true;

Task Thread:

s_3: if (! flag.value)

Thread.yield() ;

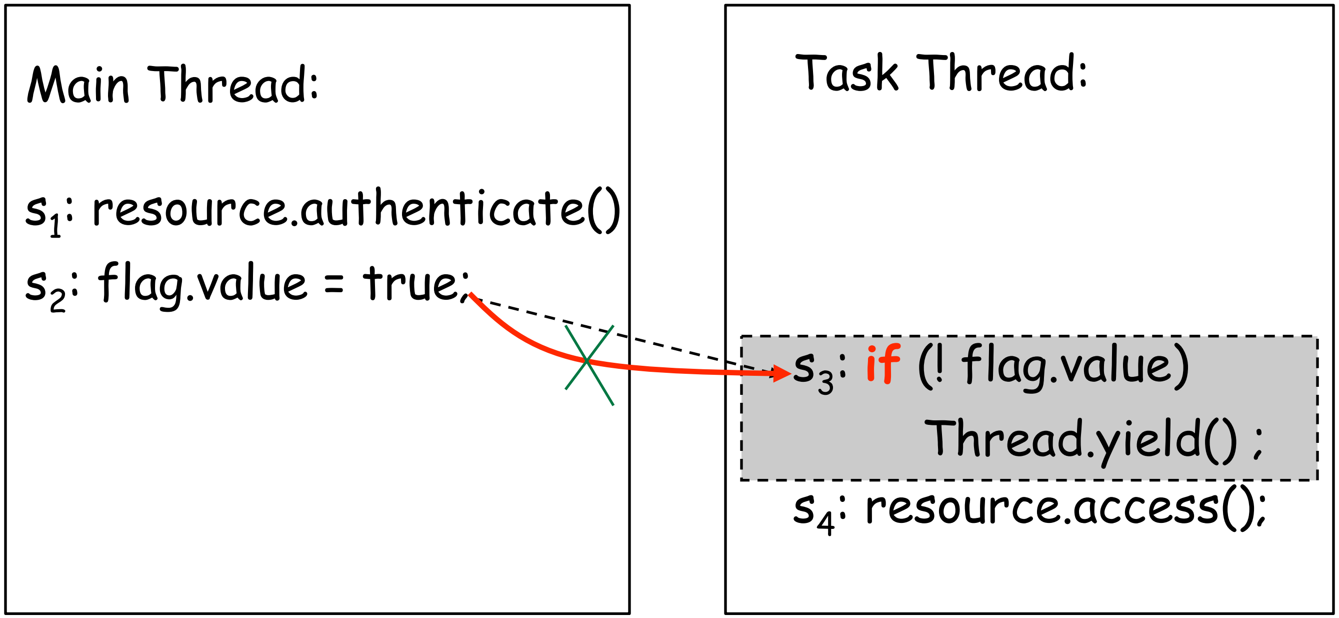
s_4: resource.access();

Observed execution: s_1 s_2 s_3 s_4

Only s_1 and s_4 directly relevant to the property

Sliced Causality Works!

Property: “authenticate before access”



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

Only s_1 and s_4 directly relevant to the property

Execution of s_4 not dependent of s_3; ignore the causal dependency $s_2 < s_3$

Sliced causality: $s_1 <> s_4$; $s_4\ s_1$ is a potential execution. Bug detected!

No False Alarms ☺

Property: "authenticate before access"

Main Thread:

s_1: resource.authenticate()

s_2: flag.value = true;

Task Thread:

s_3: **while** (! flag.value)

Thread.yield();

s_4: resource.access();

Observed execution: s_1 s_2 s_3 s_4

No False Alarms ☺

Property: "authenticate before access"

Main Thread:

s_1: resource.authenticate()

s_2: flag.value = true;

Task Thread:

s_3: **while** (! flag.value)

Thread.yield();

s_4: resource.access();

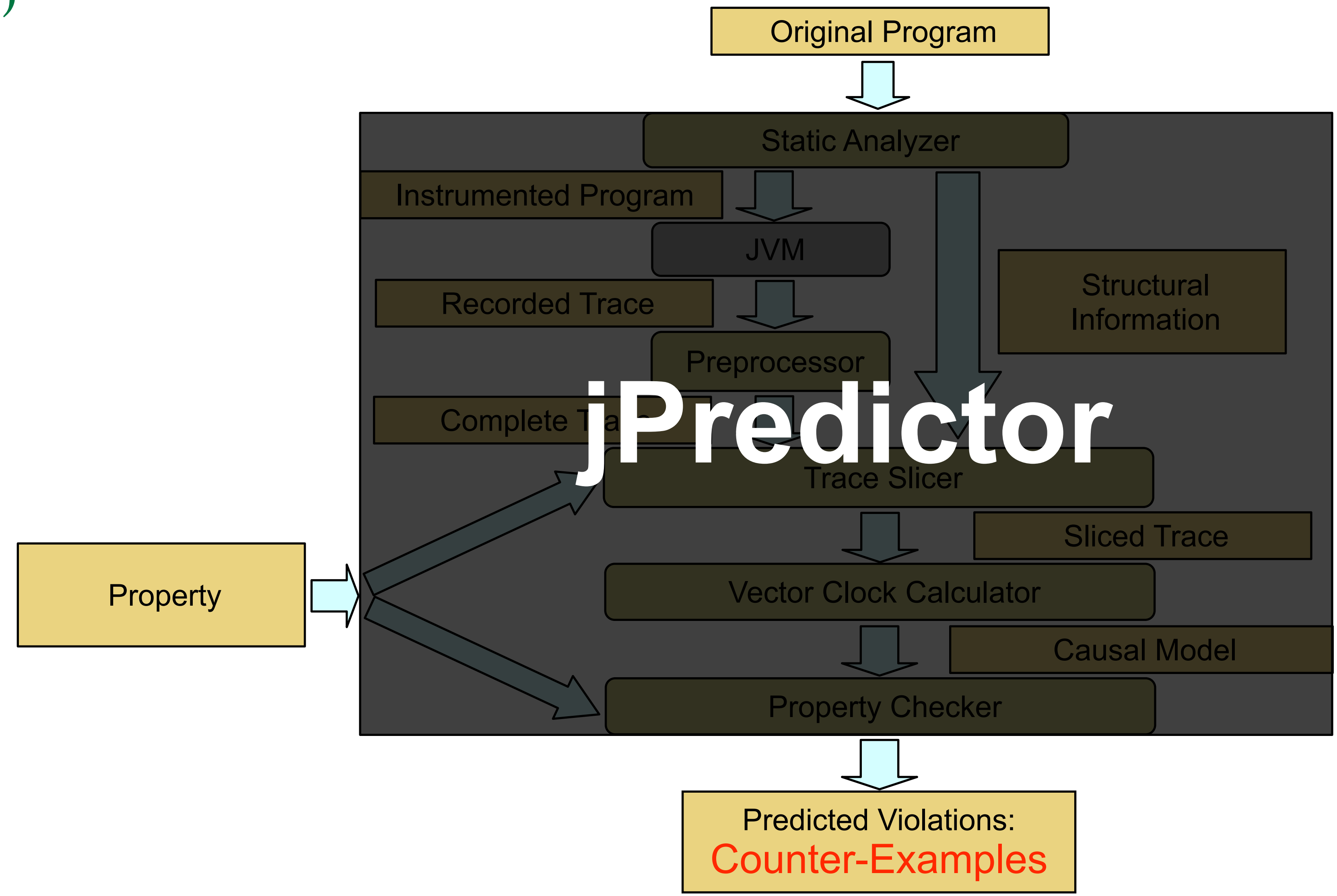
Observed execution: $s_1\ s_2\ s_3\ s_4$

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

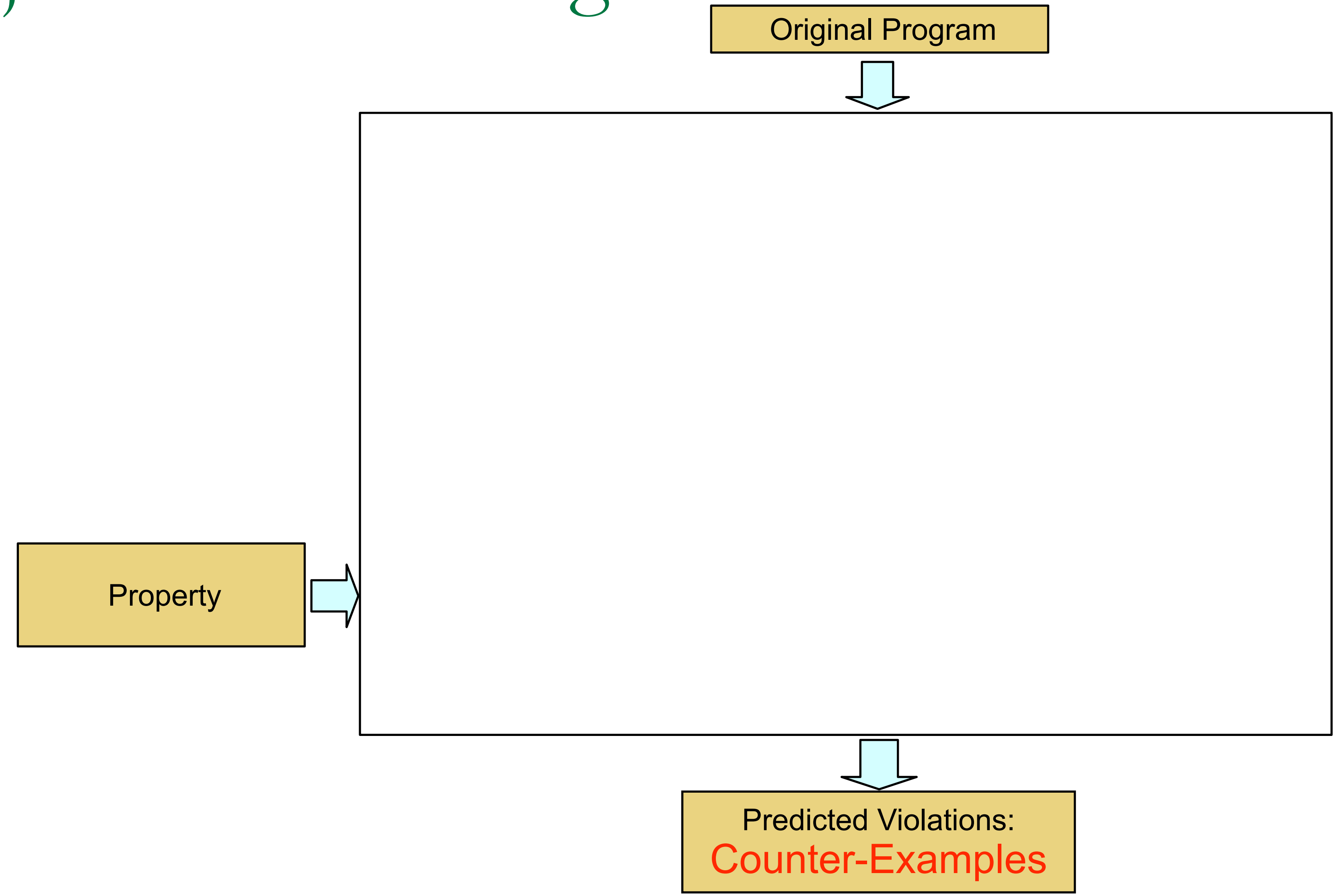
causal dependency $s_2 < s_3$ matters

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

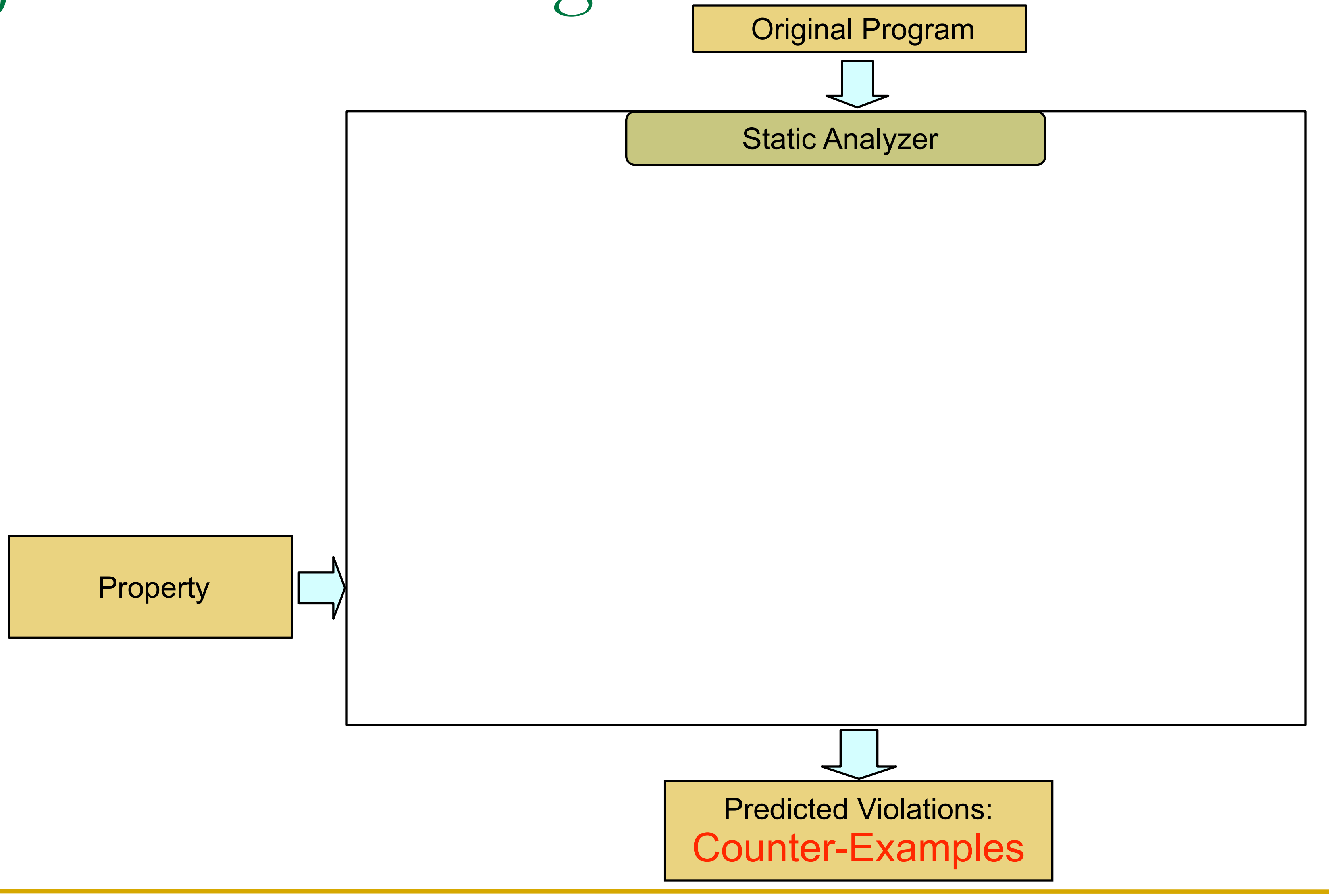
jPredictor: Black-Box View



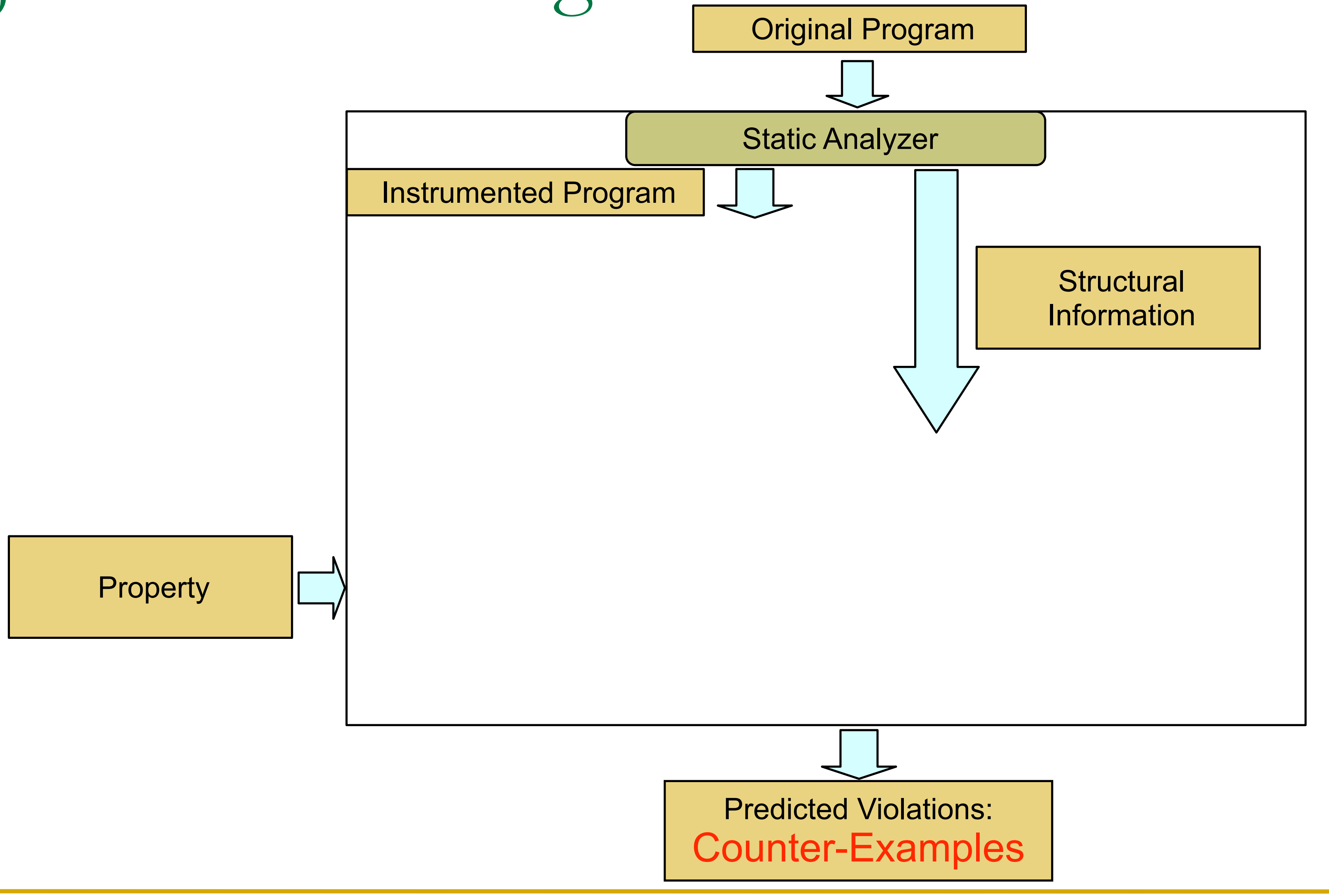
jPredictor: Filling the box



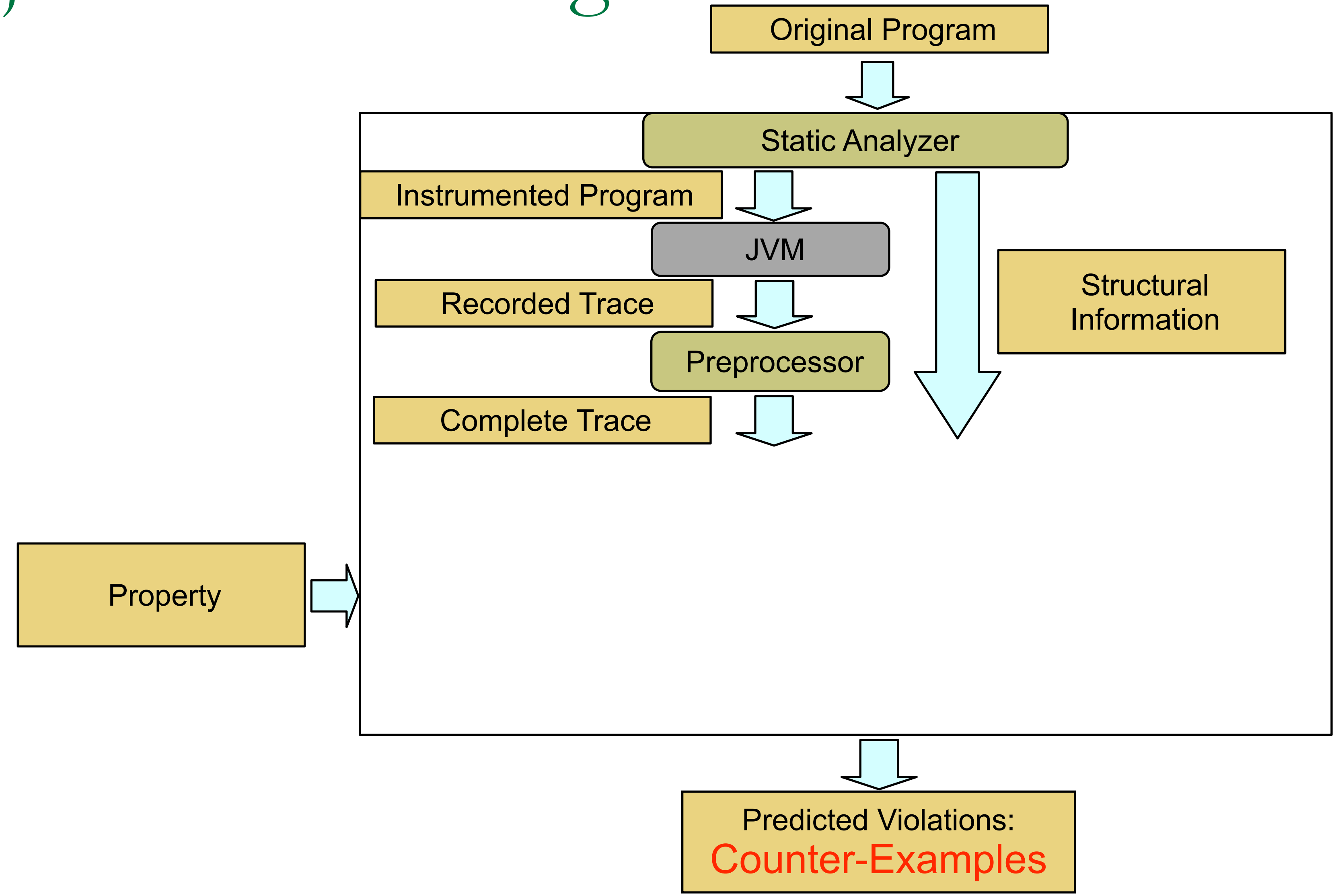
jPredictor: Filling the box



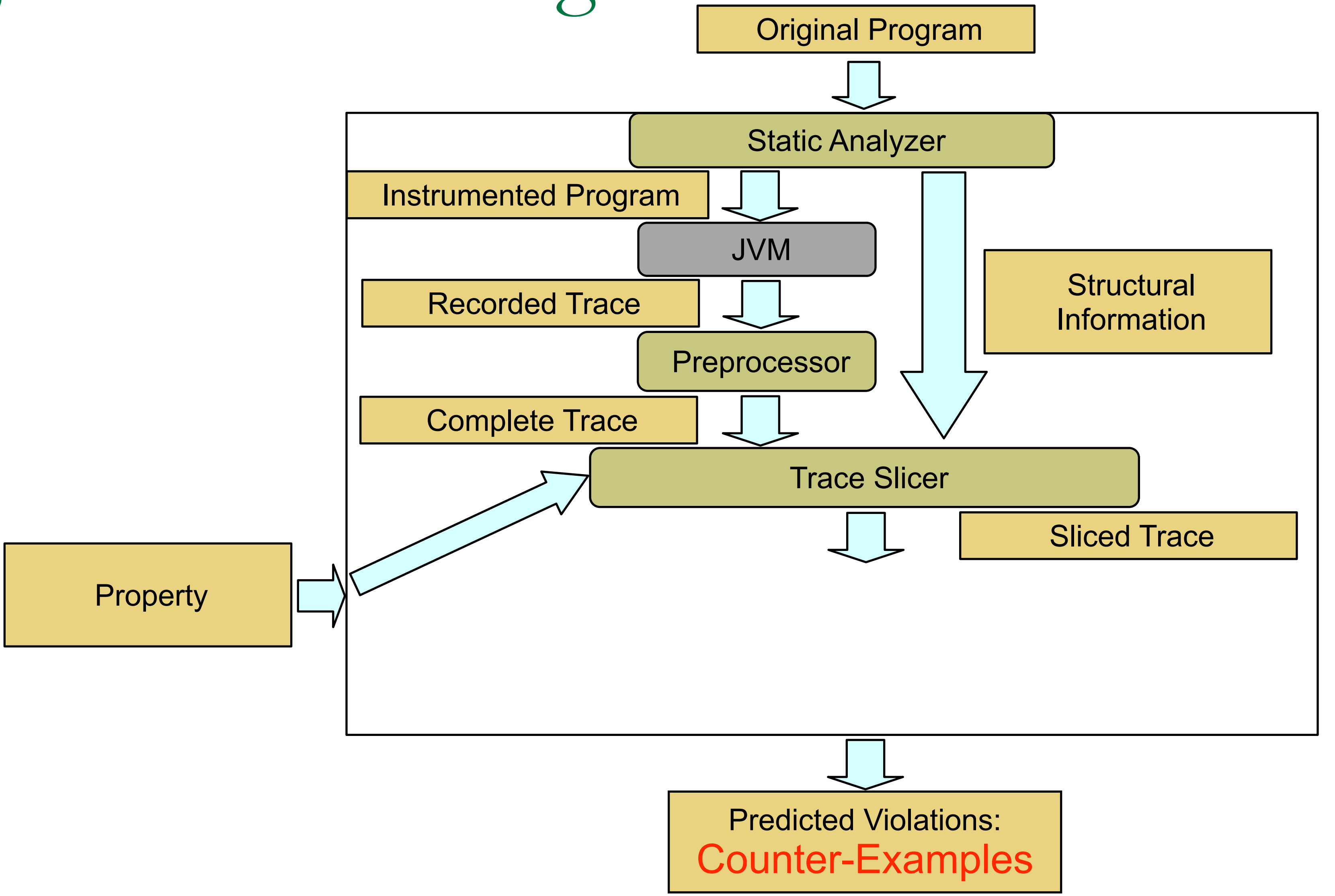
jPredictor: Filling the box



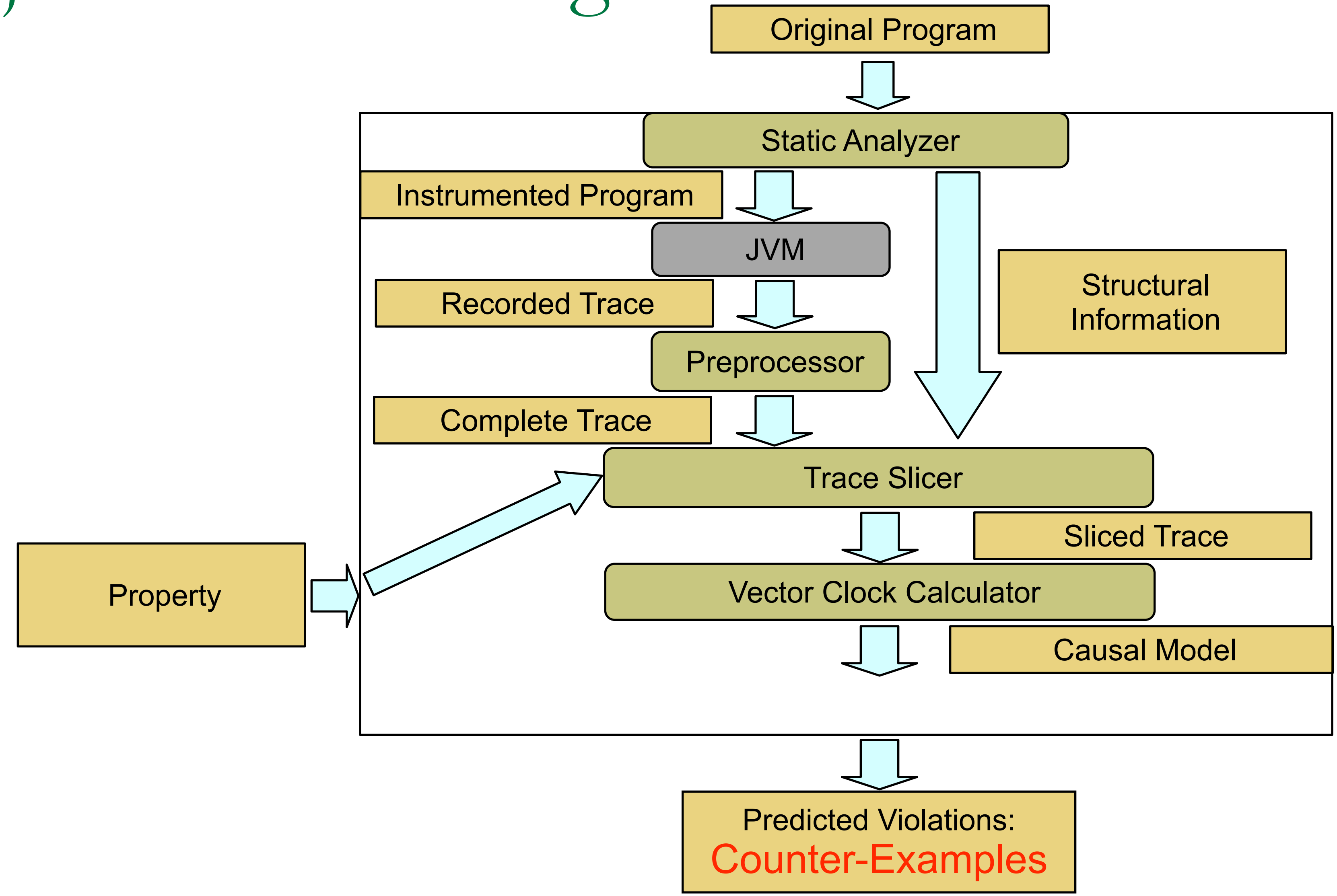
jPredictor: Filling the box



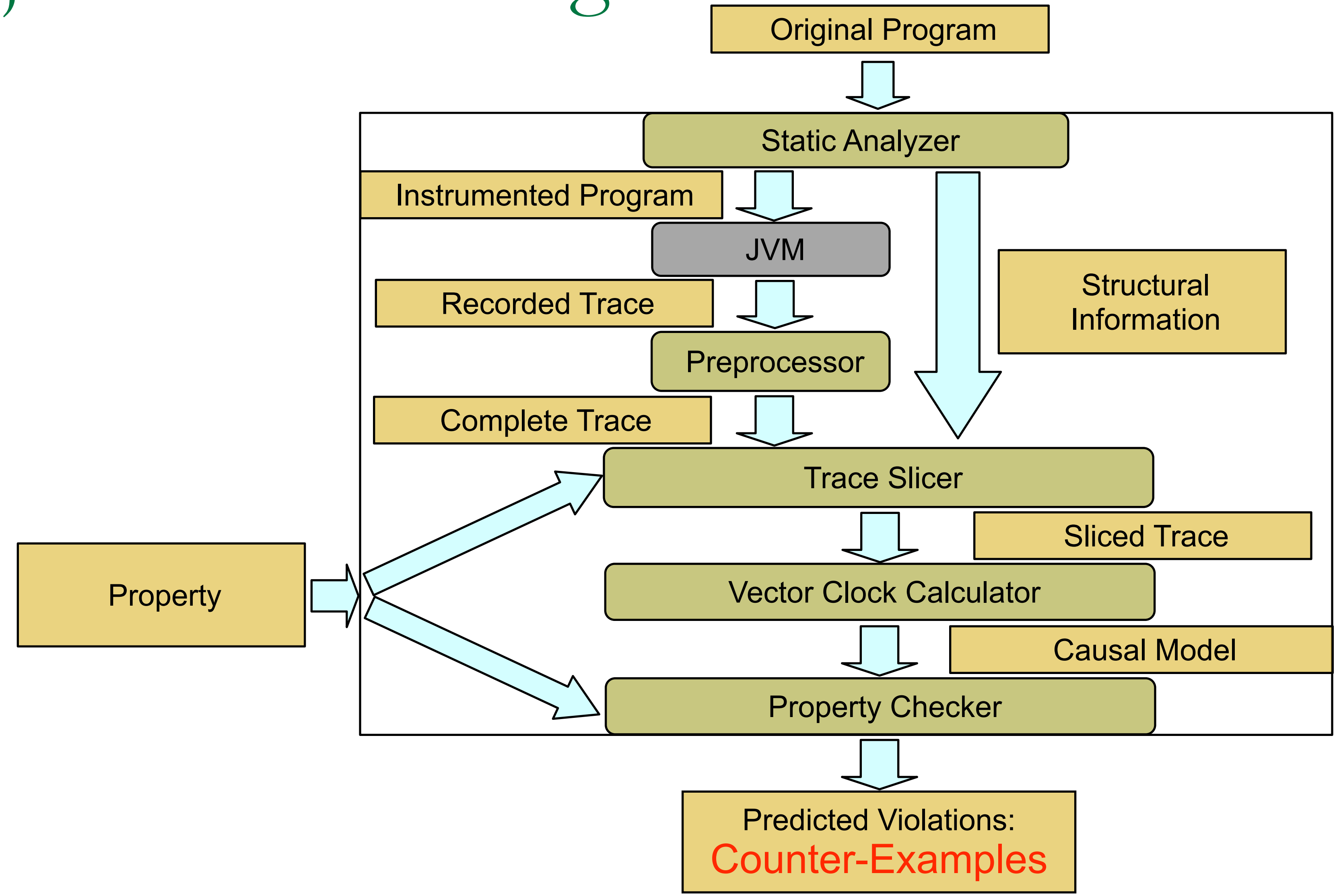
jPredictor: Filling the box



jPredictor: Filling the box



jPredictor: Filling the box



Evaluation

- Evaluated on many real life applications
  - Benchmark from previous work
    - Banking, tsp, hedc…
  - Open source programs
    - Apache FTP server, Apache Common library, Tomcat…
  - Industry programs
    - Java Collection Library, IBM WebCrawler…
- Focused on dataraces and atomicity violations
  - Sliced causality is general purpose
    - Replacement for Happened-Before Causality

Results: Datarace Detection

- Found almost all previously reported bugs in just one or few runs
- Found new bugs, no false alarms

Benchmarks	Time (seconds)		Races
	Preprocessing	Analysis	
Elevator	10.3	0.15	0
Tsp	30.6	0.23	1
Sor	2.4	0.01	0
Hedc	20.16	0.01	4
StringBuffer	0.38	0.01	0
Vector	0.44	0.01	1
IBM WebCrawler	0.94	0.01	**4**
StaticBucketMap	151.4	0.03	**1**
Pool 1.2	2.02	0.01	**18**
Pool 1.3	1.15	0.01	**1**
Apache FTP Serv.	4.9	0.02	11

red = new bugs

Results: Datarace Detection

- Found almost all previously reported bugs in just one or few runs
- Found new bugs, no false alarms

Benchmarks	Time (seconds)		Races	Hybrid Dynamic [O'Callahan-Choi-03]	Static Analysis [Naik-Aiken-Whaley-06]
	Preprocessing	Analysis			
Elevator	10.3	0.15	0	0 (0)	
Tsp	30.6	0.23	1	1 (2)	1 (12)
Sor	2.4	0.01	0	0 (0)	
Hedc	20.16	0.01	4	4 (0)	4 (13)
StringBuffer	0.38	0.01	0		
Vector	0.44	0.01	1		1 (0)
IBM WebCrawler	0.94	0.01	**4**		
StaticBucketMap	151.4	0.03	**1**		
Pool 1.2	2.02	0.01	**18**		17 (0)
Pool 1.3	1.15	0.01	**1**		
Apache FTP Serv.	4.9	0.02	11		12 (23)

red = new bugs races found(false alarms)

Results: Datarace Detection

- Found almost all previously reported bugs in just one or few runs
- Found new bugs, no false alarms

Benchmarks	Time (seconds)		Races	Hybrid Dynamic [O'Callahan-Choi-03]	Static Analysis [Naik-Aiken-Whaley-06]
	Preprocessing	Analysis			
Elevator	10.3	0.15	0	0 (0)	
Tsp	30.6	0.23	1	1 (2)	1 (12)
Sor	2.4	0.01	0	0 (0)	
Hedc	20.16	0.01	4	4 (0)	4 (13)
StringBuffer	0.38	0.01	0		
Vector	0.44	0.01	1		1 (0)
IBM WebCrawler	0.94	0.01	**4**		
StaticBucketMap	151.4	0.03	**1**		
Pool 1.2	2.02	0.01	**18**		17 (0)
Pool 1.3	1.15	0.01	**1**		
Apache FTP Serv.	4.9	0.02	11		12 (23)

red = new bugs races found(false alarms)

Results: Datarace Detection

- Found almost all previously reported bugs in just one or few runs
- Found new bugs, no false alarms

Benchmarks	Time (seconds)		Races	Hybrid Dynamic [O'Callahan-Choi-03]	Static Analysis [Naik-Aiken-Whaley-06]
	Preprocessing	Analysis			
Elevator	10.3	0.15	0	0 (0)	
Tsp	30.6	0.23	1	1 (2)	1 (12)
Sor	2.4	0.01	0	0 (0)	
Hedc	20.16	0.01	4	4 (0)	4 (13)
StringBuffer	0.38	0.01	0		
Vector	0.44	0.01	1		1 (0)
IBM WebCrawler	0.94	0.01	**4**		
StaticBucketMap	151.4	0.03	**1**		
Pool 1.2	2.02	0.01	**18**		17 (0)
Pool 1.3	1.15	0.01	**1**		
Apache FTP Serv.	4.9	0.02	11		12 (23)

red = new bugs races found(false alarms)

Results: Atomicity Violations

- Check atomicity of synchronized methods
- Based on atomicity conditions from [Vaziri-Tip-Dolby-06]
- Found all known bugs plus new bugs, no false alarms

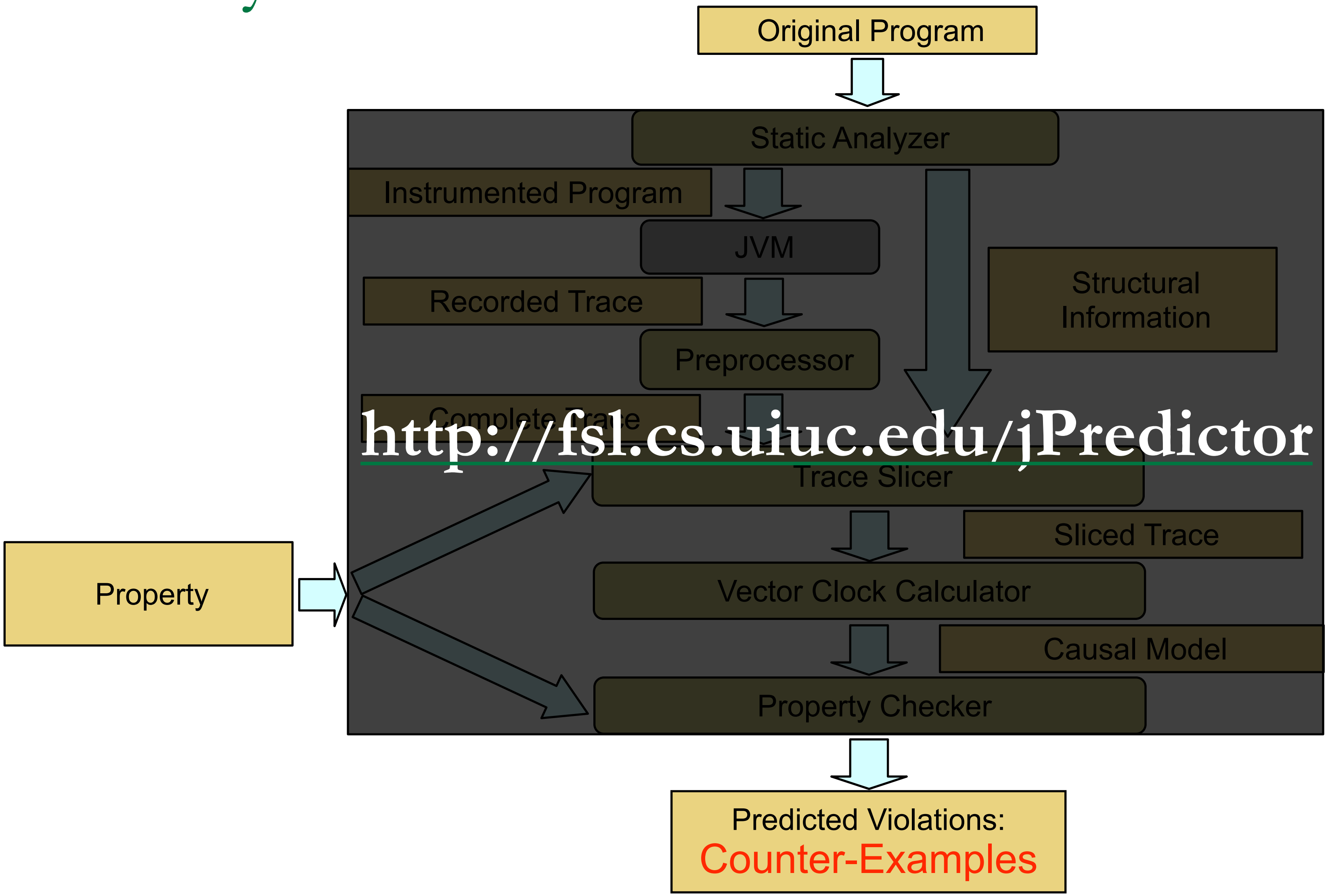
Benchmarks	Analysis Time* (seconds)	Violations	Atomizer [flanagan-freund-04]	Other [wang-stoller-06]
Elevator	4.2	0	0 (2)	0 (2)
Sor	3.6	**1**	0 (0)	0 (0)
Tsp	1.16	0	0 (7)	0 (2)
Hedc	0.22	1	1 (4)	0 (0)
StringBuffer	0.07	**1**		0 (1)
Vector	0.27	4		4 (4)
StaticBucketMap	22.03	1		
Pool 1.2	5.5	10		
Pool 1.3	0.83	0		

* We are reusing the slices generated for race detection

Conclusions and Future Work

- Sliced Causality: sound technique for predicting concurrency errors
- jPredictor: data-races and atomicity violations
  - As good, or better than current specialized tools
- Online jPredictor: predict bugs during runtime
  - Embedding in JVM
- Combine with test-generation techniques

Thank you!



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