(2021 Spring) Defense

Constraint-based Program Analysis for Concurrent Software

Presenter: Chungha Sung

Committee members

Nenad Medvidović | Sandeep Gupta | Chao Wang

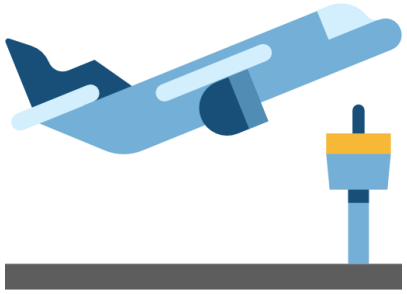
2021. 03. 12.

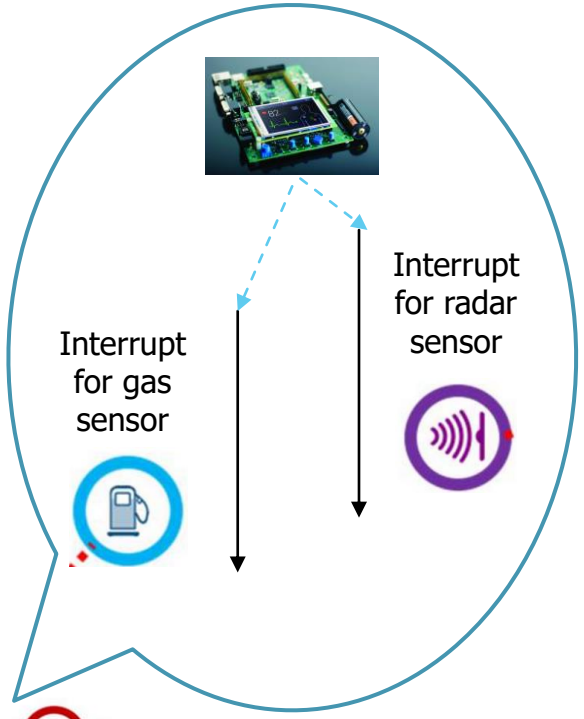
Concurrency

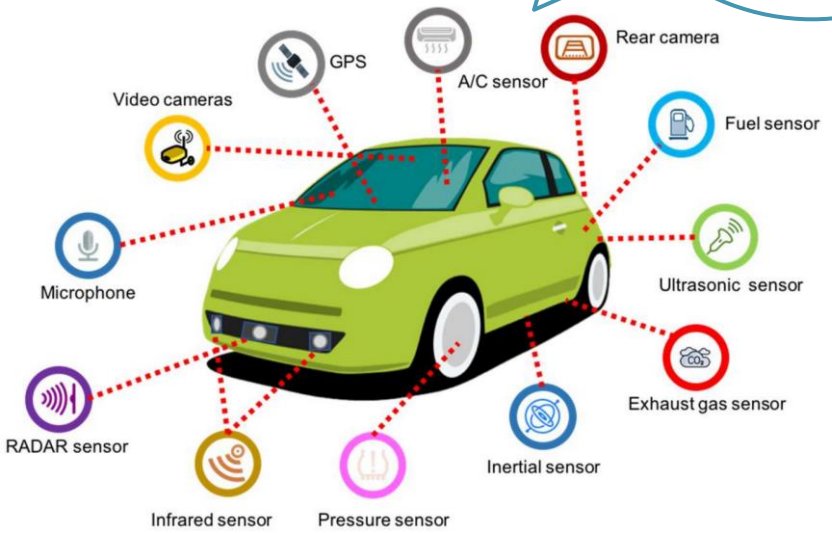
- Concurrency is the ability to execute different parts or units of a program in an interleaved manner

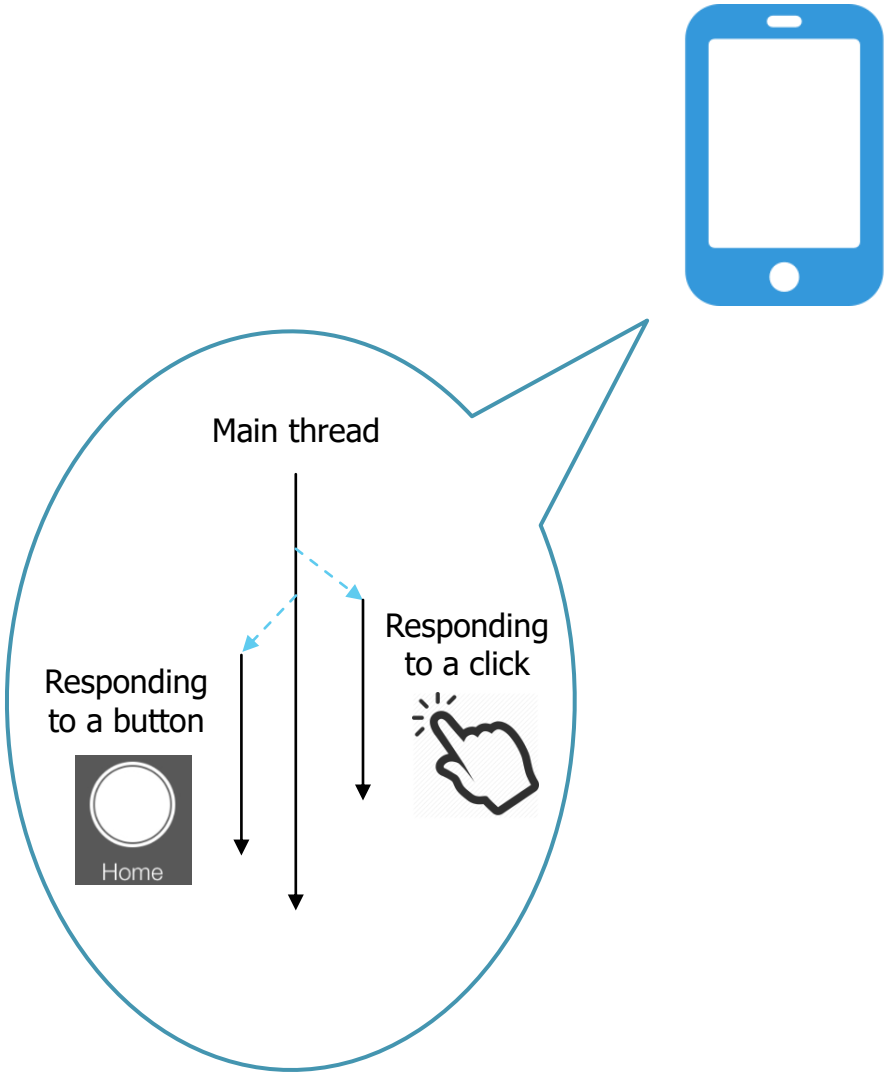
Example: many users deposit/withdraw money through bank systems concurrently.

Concurrent Software

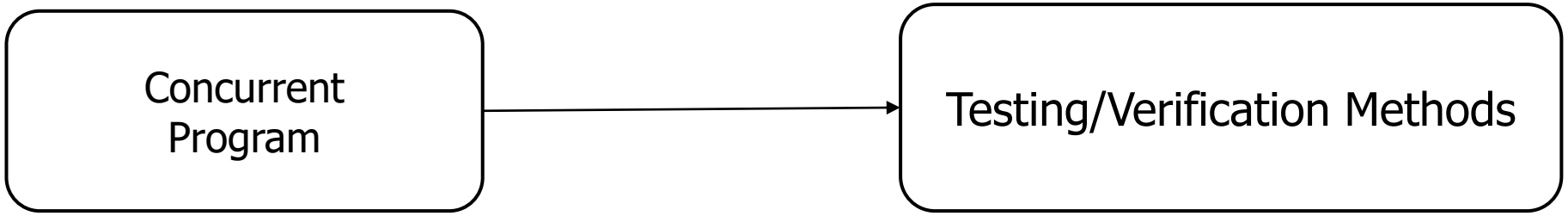



Interrupt for gas sensor
Interrupt for radar sensor


GPS
A/C sensor
Rear camera
Video cameras
Fuel sensor
Microphone
Ultrasonic sensor
Exhaust gas sensor
RADAR sensor
Inertial sensor
Infrared sensor
Pressure sensor


Main thread
Responding to a button
Home
Responding to a click

Testing/Verification of concurrent software



Enumerating interleavings

Provide actual interleavings in bug detection (soundness), but hard to explore all possible interleavings (not completeness)
Ex) Test case generation

Merging interleavings

It covers all possible interleavings (completeness), but it is hard to provide accurate bug detection (not soundness)
Ex) Abstract interpretation

Enumerating interleavings - example

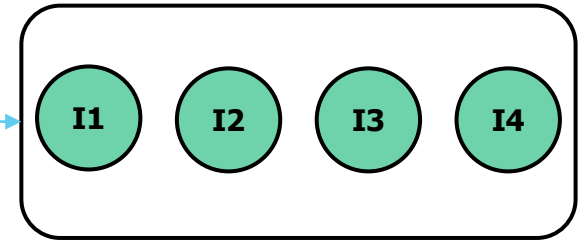
```
I1() {
  x = 1;
}
```

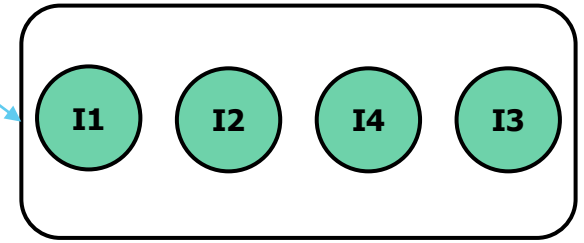
```
I2() {
  y = x;
}
```

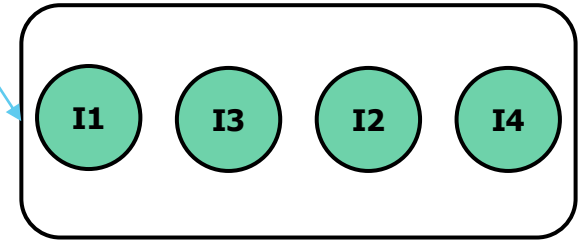
```
I3() {
  a = x;
}
```

```
I4() {
  b = y;
}
```

Possible event sequences: 4!



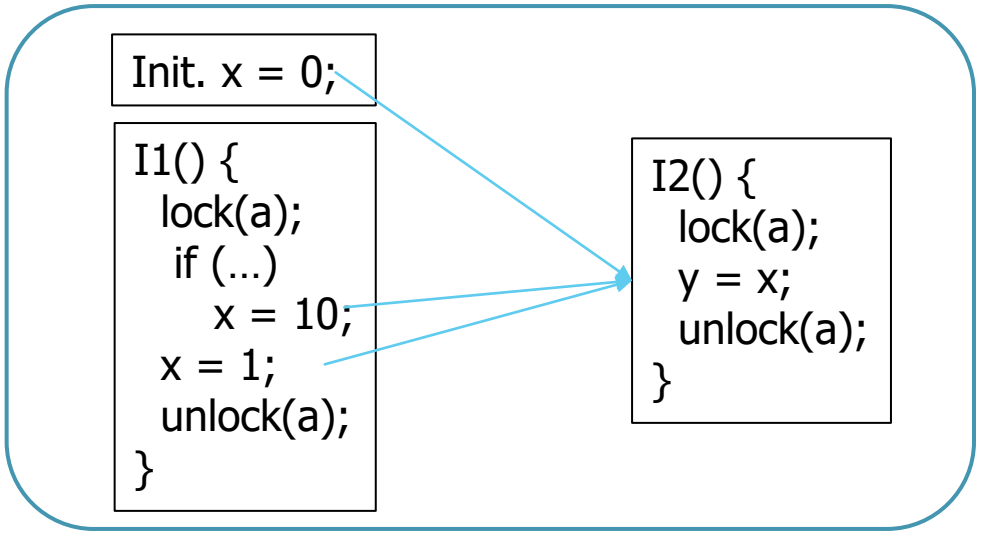


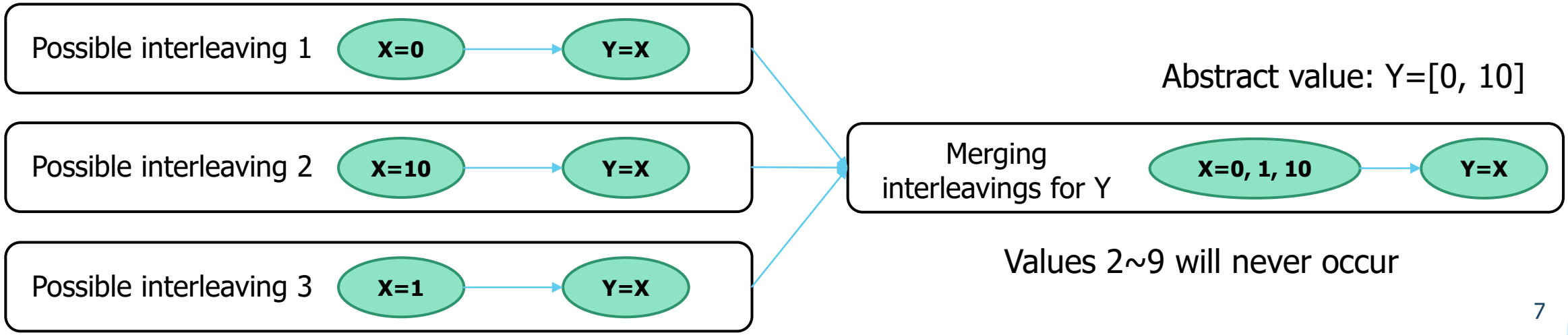


● ● ●

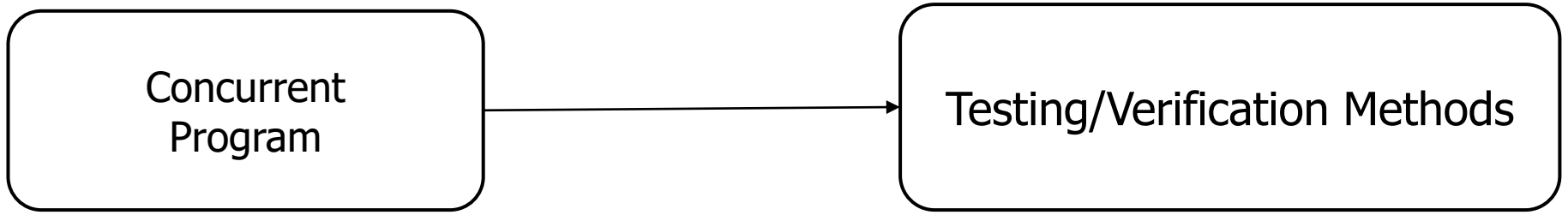
With N events: N!
state explosion

Merging interleavings - example





Testing/Verification of concurrent software



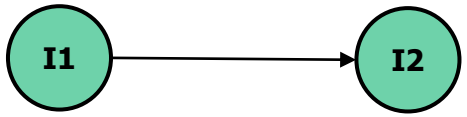
Enumerating interleavings → Sound, not complete

Merging interleavings → Complete, not sound

Can we make them better?
- Can we provide better interleaving exploration for enumeration?
- Can we provide more accurate program states for merging?

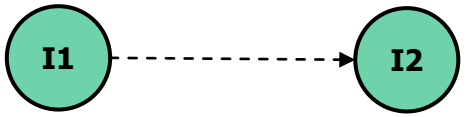
Analysis of interference between concurrent events

Feasible interference



- Executing concurrent event I1 affects a program state of running event I2
(Example: a value written by I1 is read by I2)

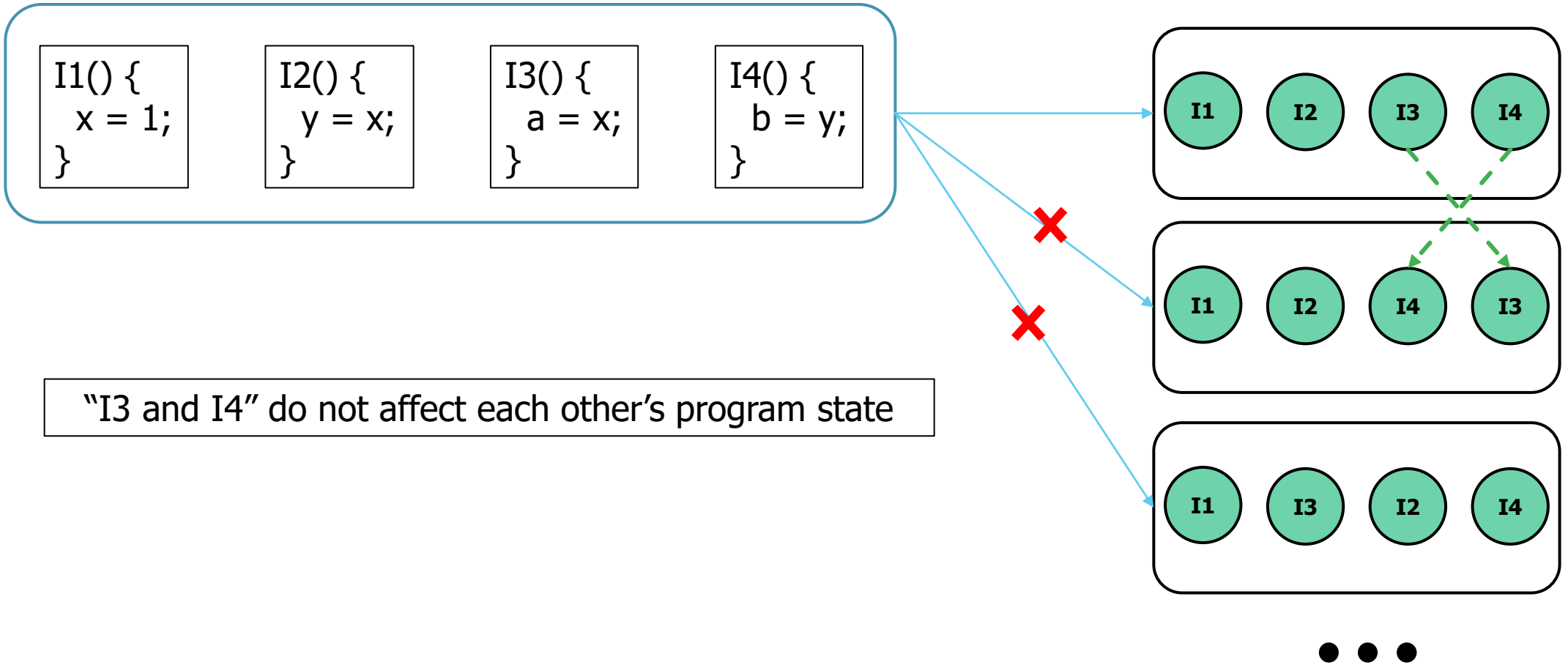
Infeasible interference



- Executing concurrent event I1 does not affect a program state of running event I2
(Example: any values written by I1 are not read/written by I2)

Feasible/infeasible interference can be used to improve testing and verification of concurrent software

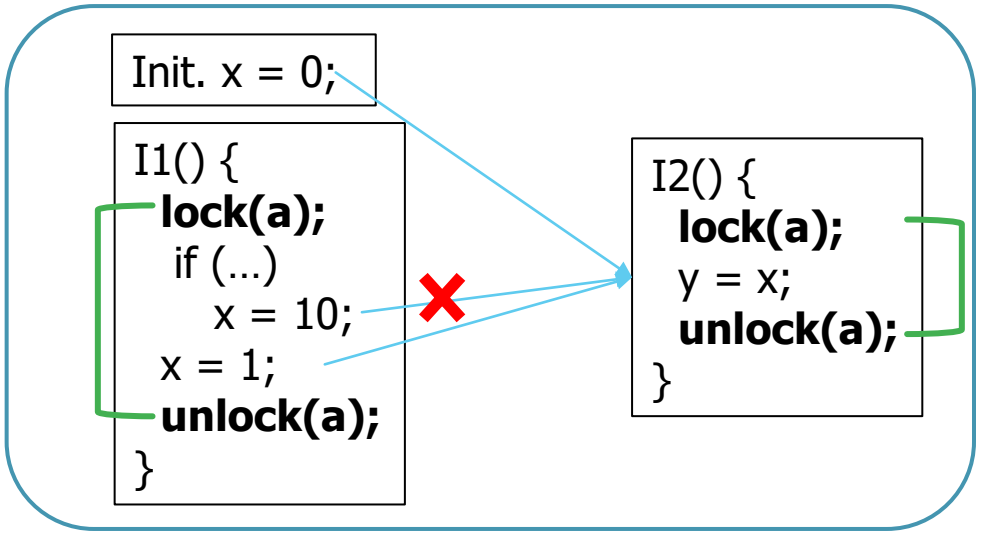
Enumerating interleavings with the analysis - example

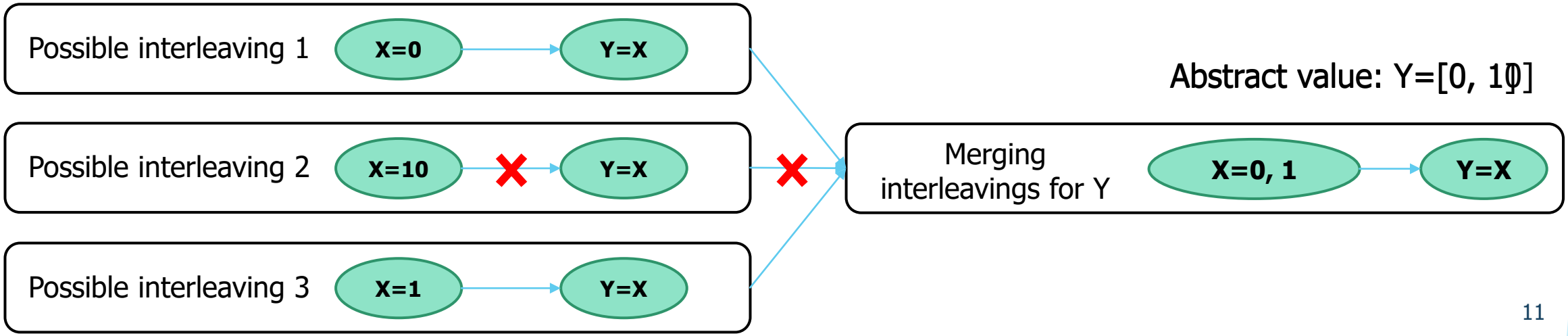


"I3 and I4" do not affect each other's program state

Merging interleavings with the analysis - example

"x=10 -> y = x" never happens due to the lock pairs



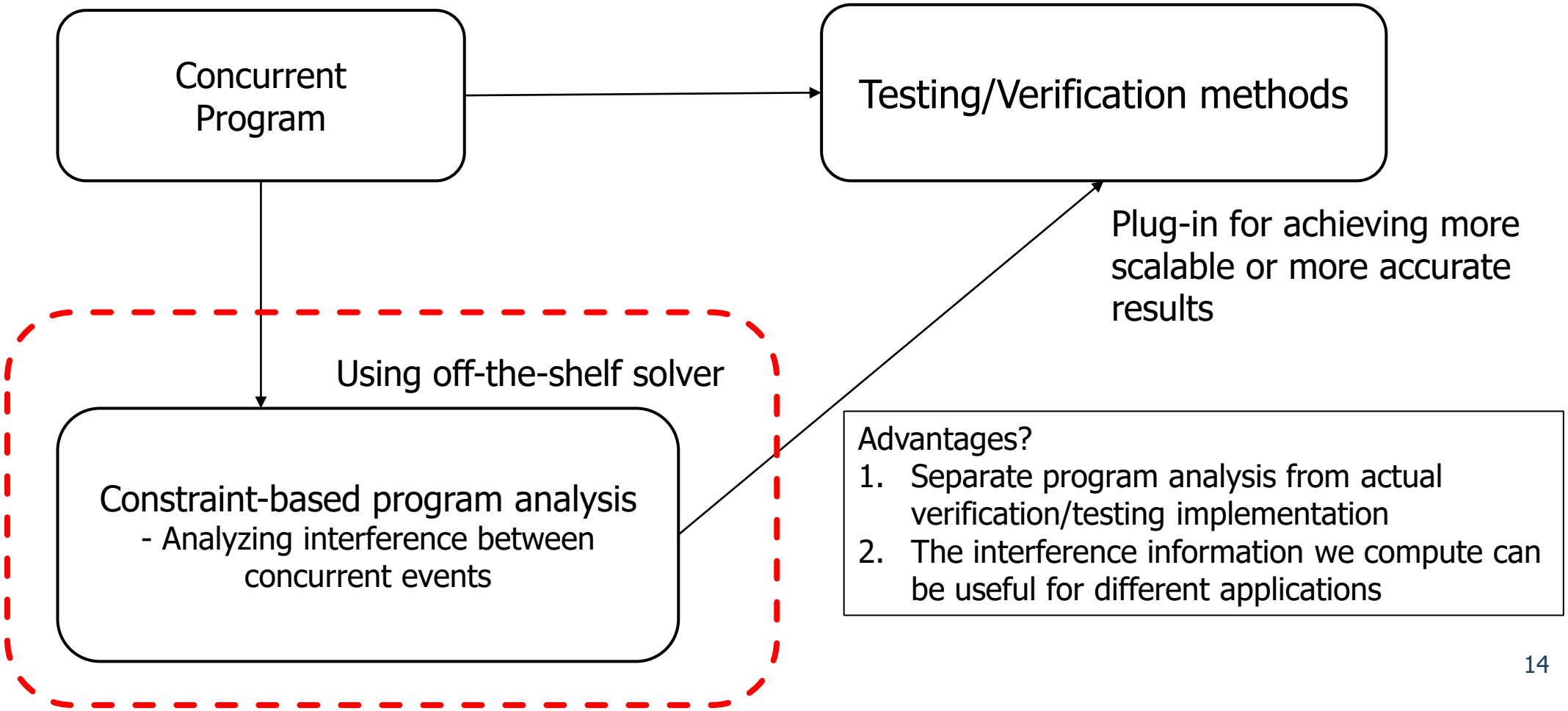


Dissertation outline

- Problem
  - It is hard to achieve completeness or soundness for testing and verification of concurrent software due to "State explosion" in concurrent software
- Hypothesis
  - **Analysis of interference between concurrent events** improves testing/verification techniques for concurrent software
- Solution
  - We **customize constraint-based program analysis** to analyze interference between concurrent events
- Demonstration
  - We validate the hypothesis with three important testing/verification applications

- **Background & Motivation**
- **Proposed method**
  - **Customized constraint-based analysis for concurrent software**
- **Three applications**
  - More accurate assertion checking for interrupt-driven software
    [Sung et al., ASE 2017]
  - More scalable semantic diffing of multi-threaded programs
    [Sung et al., ASE 2018]
  - More efficient testing web applications
    [Sung et al., FSE 2016]
- **Conclusion**

Use of constraint-based program analysis

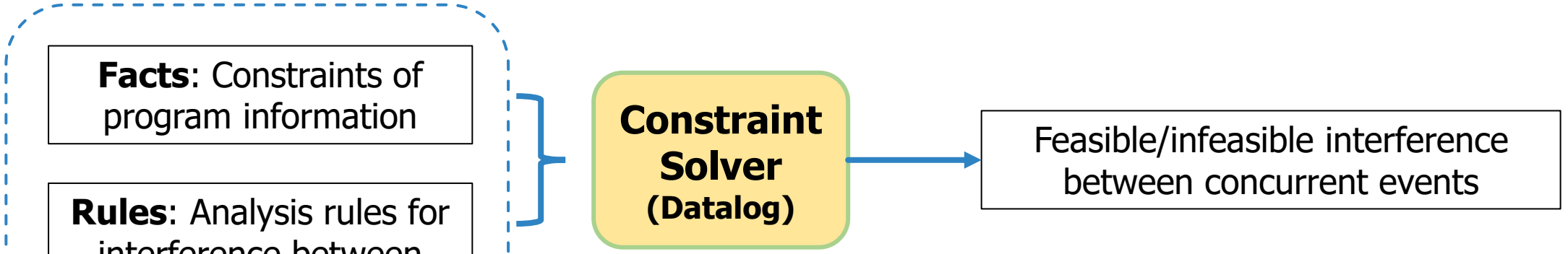


Advantages?

1. Separate program analysis from actual verification/testing implementation
2. The interference information we compute can be useful for different applications

How do we compute the interference?

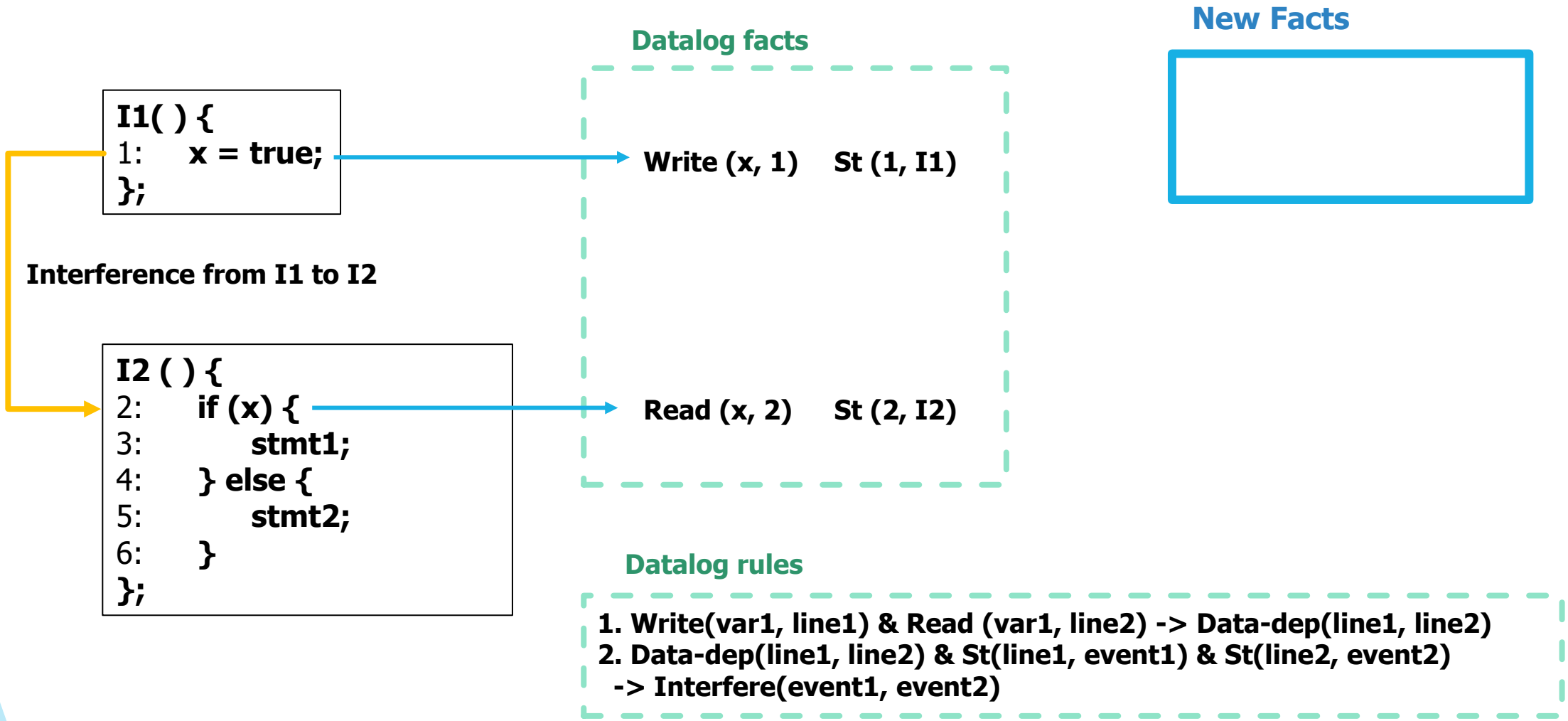
- Constraint-based program analysis
  - [Whaley & Lam, PLDI 2004], [Livshits & Lam, USENIX 2005], [Naik et al. PLDI 2006]



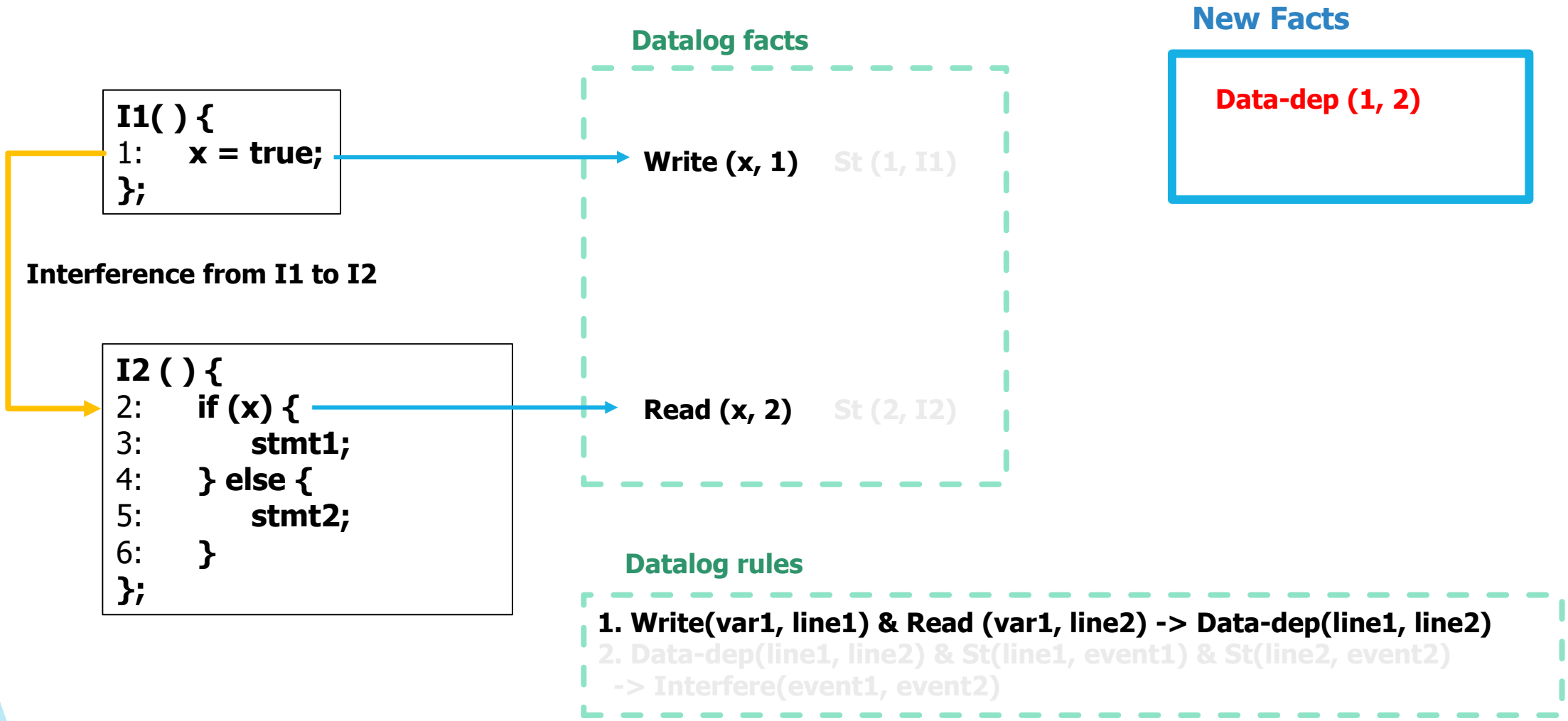
We customize our rules and facts

Datalog?
- Declarative language for deductive database
- Polynomial-time solvable
- Fixed-point computation -> well-matched with program analysis algorithms
- Program analysis can be implemented with shorter lines of code

Computing interference with Datalog - example



Computing interference with Datalog - example



Computing interference with Datalog - example

```
I1( ) {
1:    x = true;
};
```

Interference from I1 to I2

```
I2 ( ) {
2:    if (x) {
3:        stmt1;
4:    } else {
5:        stmt2;
6:    }
};
```

Datalog facts

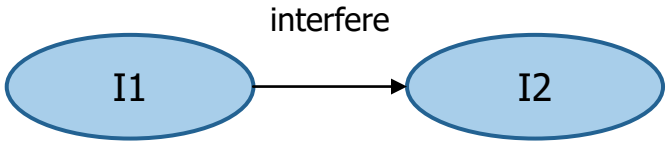
Write (x, 1) **St (1, I1)**

Read (x, 2) **St (2, I2)**

New Facts

Data-dep (1, 2)

Interfere (I1, I2)



We make specialized rules to analyze feasible/infeasible interference

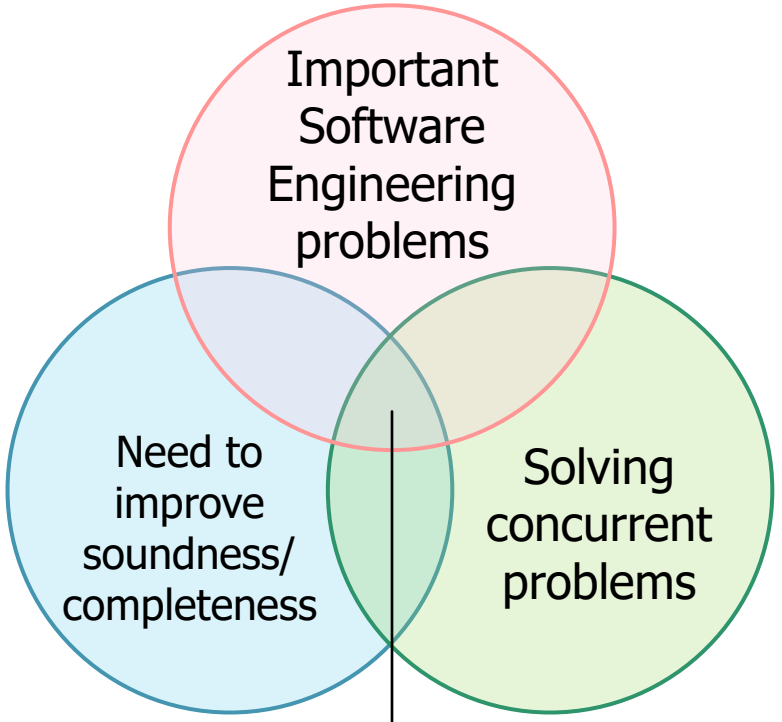
Datalog rules

1. Write(var1, line1) & Read (var1, line2) -> Data-dep(line1, line2)

2. Data-dep(line1, line2) & St(line1, event1) & St(line2, event2)
-> Interfere(event1, event2)

- **Background & Motivation**
- **Proposed method**
  - Customized constraint-based analysis for concurrent software
- **Three applications**
  - More accurate assertion checking for interrupt-driven software
    [Sung et al., ASE 2017]
  - More scalable semantic diffing of multi-threaded programs
    [Sung et al., ASE 2018]
  - More efficient testing web applications
    [Sung et al., FSE 2016]
- **Conclusion**

Applications that validate the hypothesis

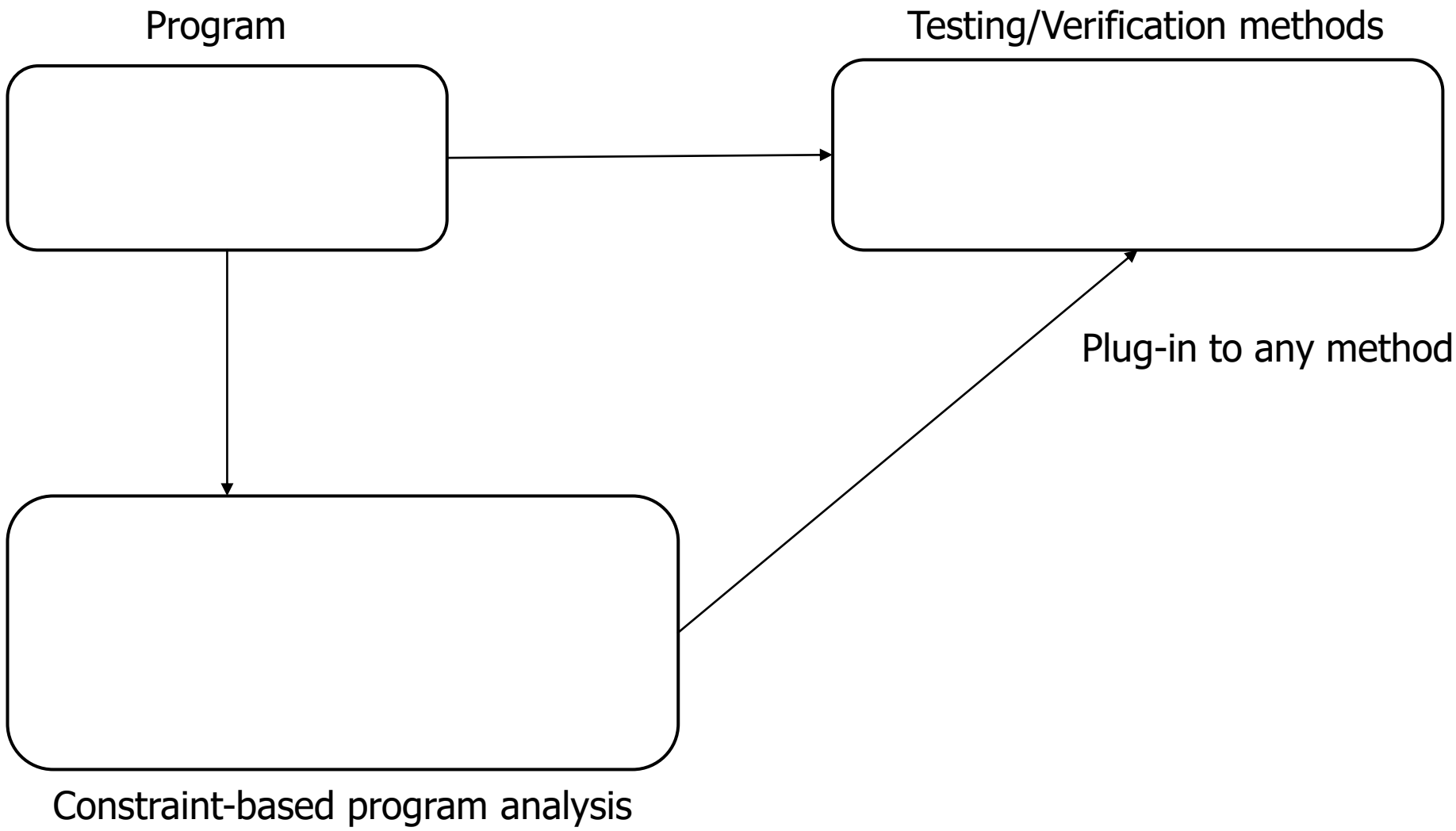


- Verification of interrupt-driven programs
- Semantic diffing of concurrent programs for program changes
- Testing of web applications

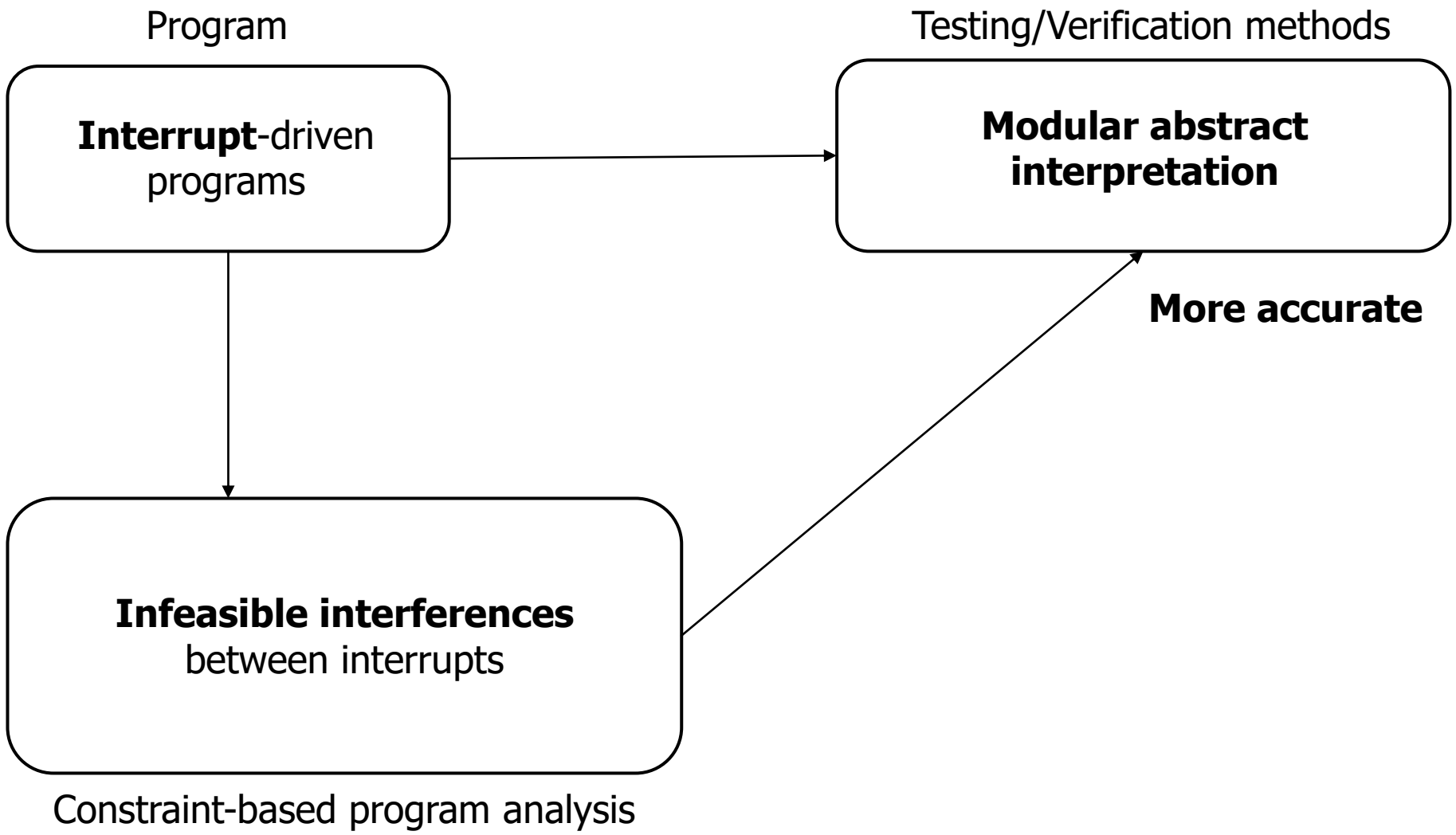
Desired applications with the framework

- Verification of interrupt-driven software
  - Problem: merging interleavings -> not accurate enough
  - Analysis: **Infeasible interferences** between interrupts
  - Result: More accurate verification
- Semantic diffing of concurrent programs
  - Problem: enumerating interleavings -> not scalable enough
  - Analysis: **Infeasible interferences** between threads
  - Result: More scalable semantic diffing for program changes
- Testing of web applications
  - Problem: enumerating interleavings -> not efficient enough
  - Analysis: **Feasible interferences** between events
  - Result: More efficient testing web applications

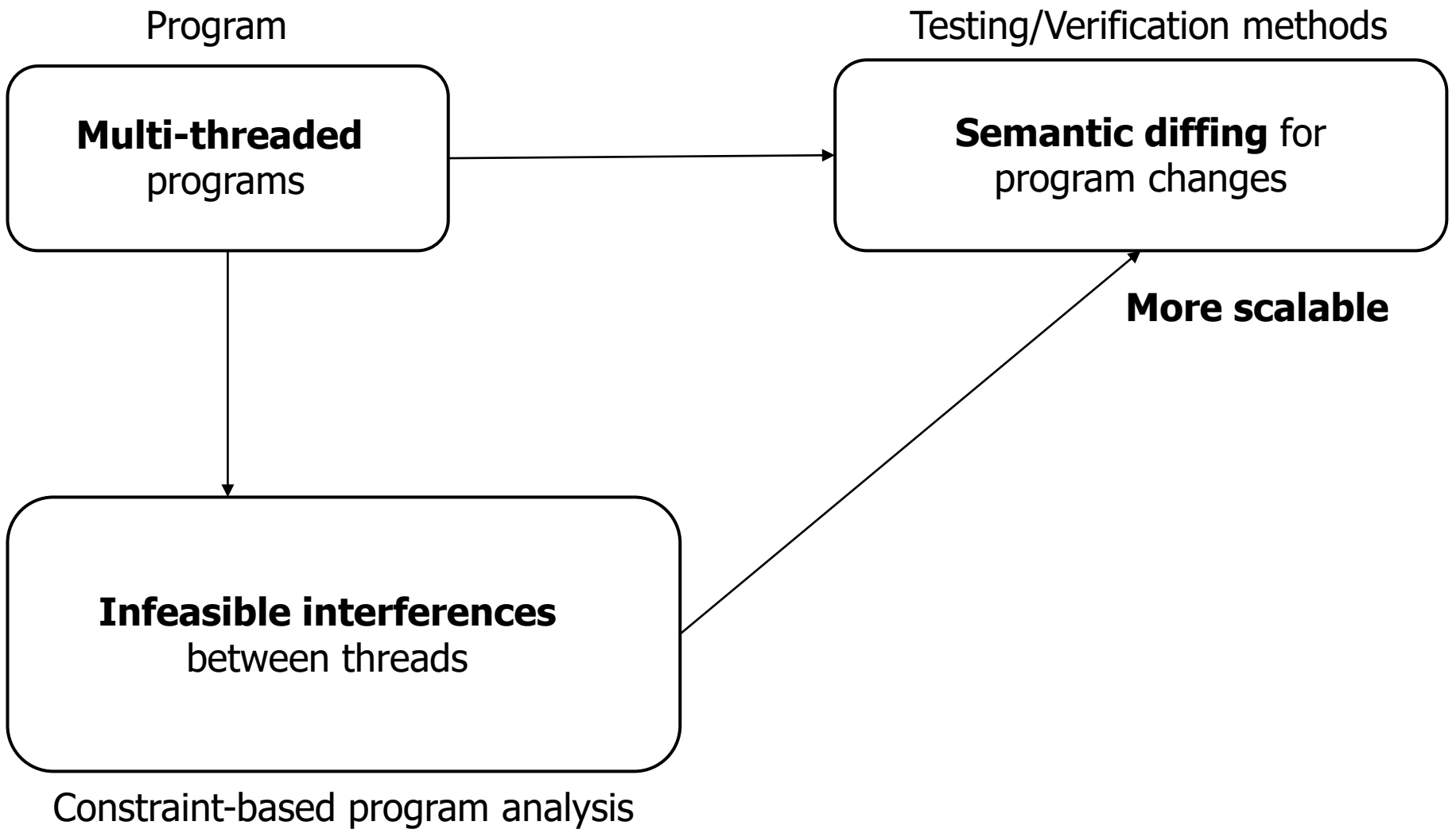
Overview


Program
Testing/Verification methods
Plug-in to any method
Constraint-based program analysis

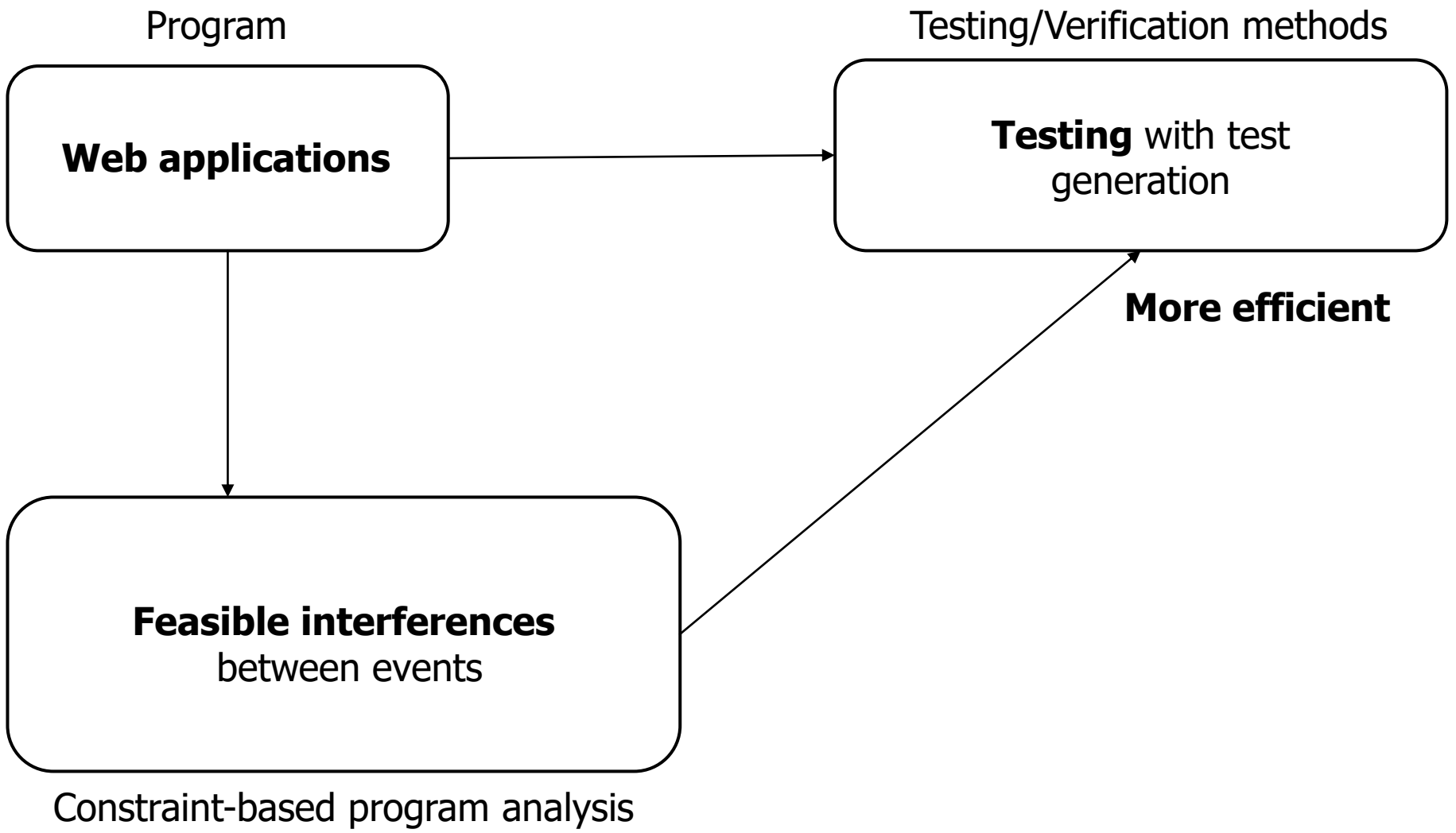
Application (1)



Application (2)

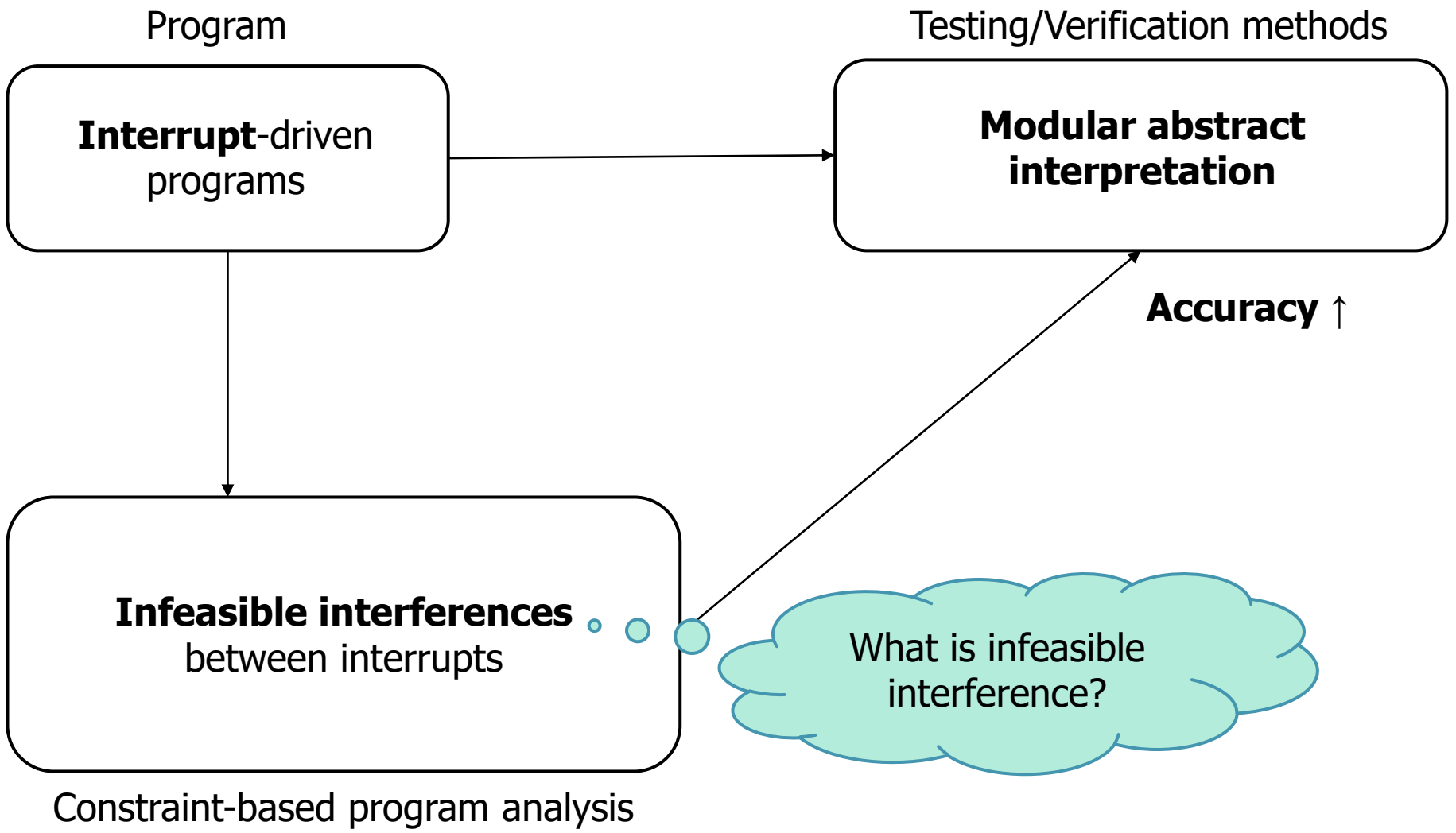


Application (3)

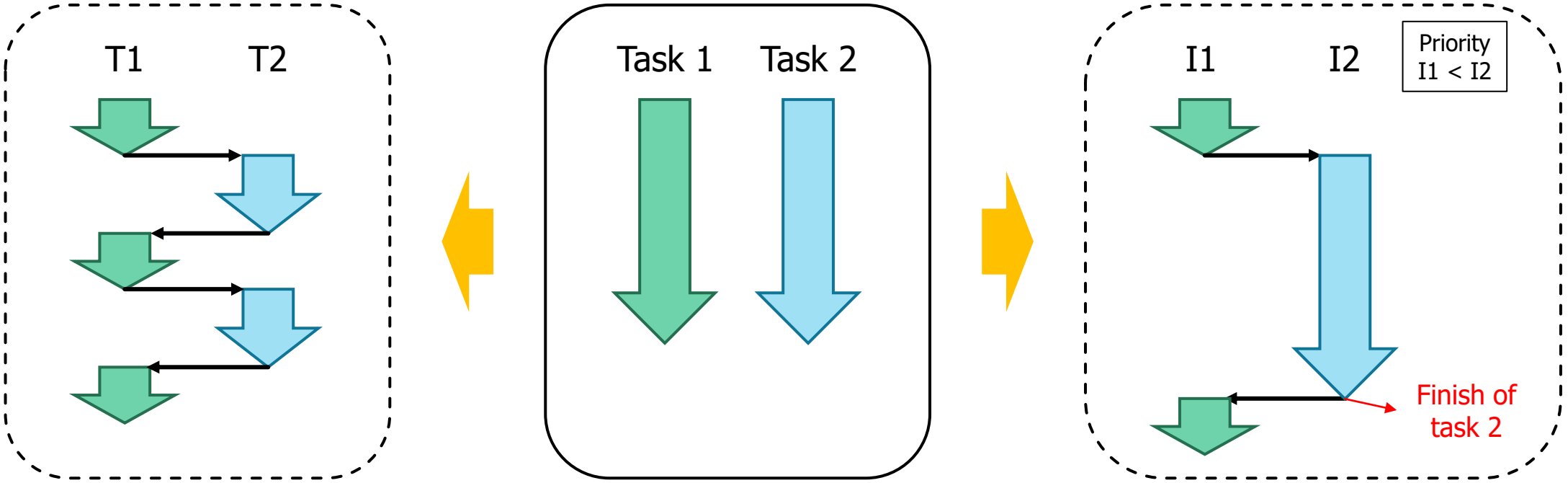


- **Background & Motivation**
- **Proposed method**
  - Customized constraint-based analysis for concurrent software
- **Three applications**
  - **More accurate assertion checking for interrupt-driven software [Sung et al., ASE 2017]**
  - More scalable semantic diffing of multi-threaded programs [Sung et al., ASE 2018]
  - More efficient testing web applications [Sung et al., FSE 2016]
- **Conclusion**

Application (1) - revisit



Understanding the basic interrupt behavior



Multi-threaded programs
(Any preemption between threads can happen)

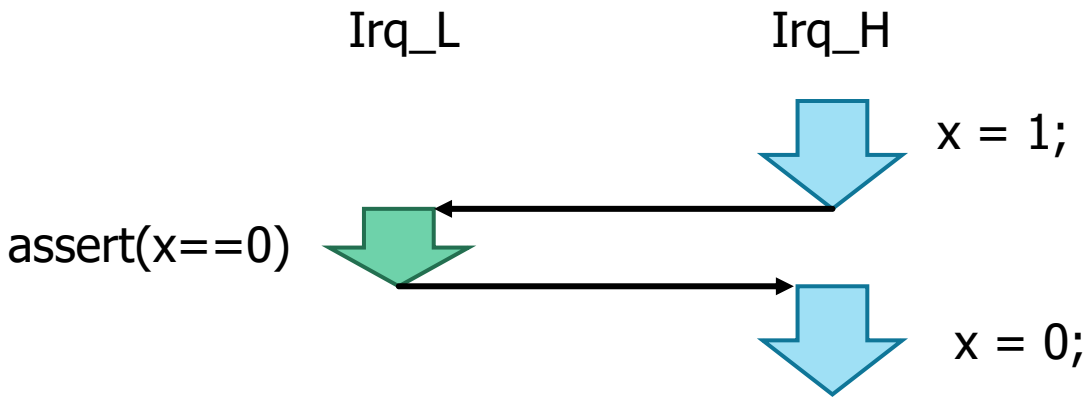
Concurrent Tasks

Interrupts
(Lower priority interrupt cannot preempt higher priority interrupt)

Infeasible interference – example

```
Irq_L() {                    Irq_H() {
   assert(x == 0);              if(...)
};                                x = 1;
                                 x = 0;
                             };
```

Multi-threaded scheduling
- The read-from is feasible



Infeasible interference – example

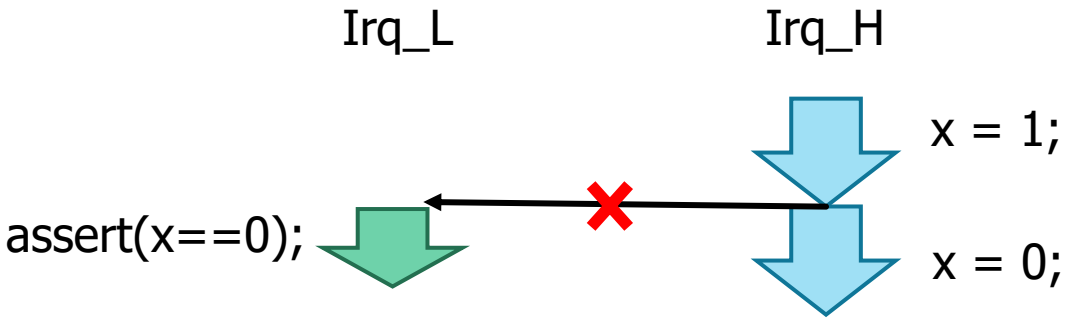
Priority: L < H

```
Irq_L() {
  assert(x == 0);
};
```

```
Irq_H() {
  if(...)
    x = 1;
  x = 0;
};
```

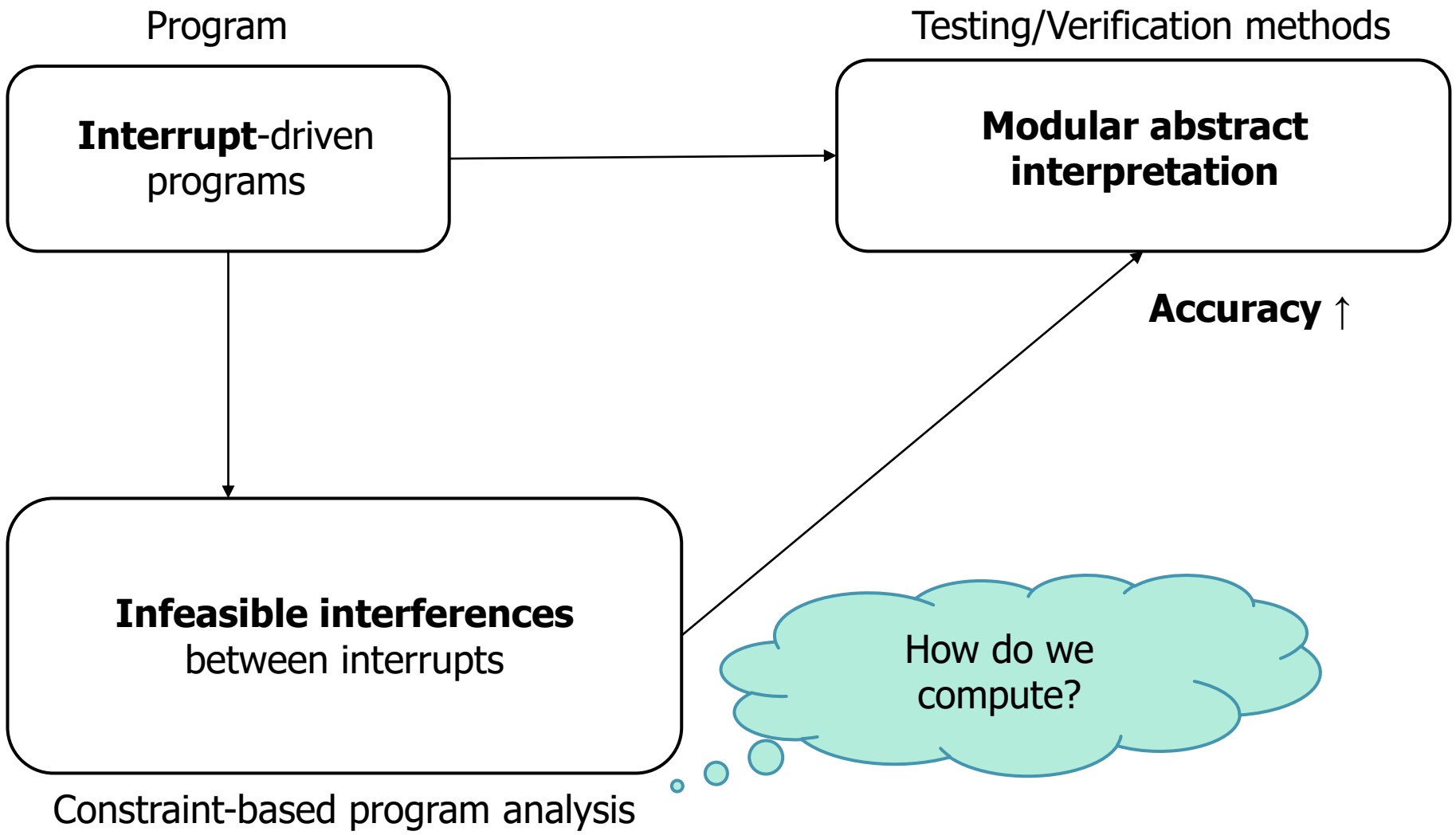
Post-dominate

Interrupt with priority information
- The read-from is **NOT** feasible

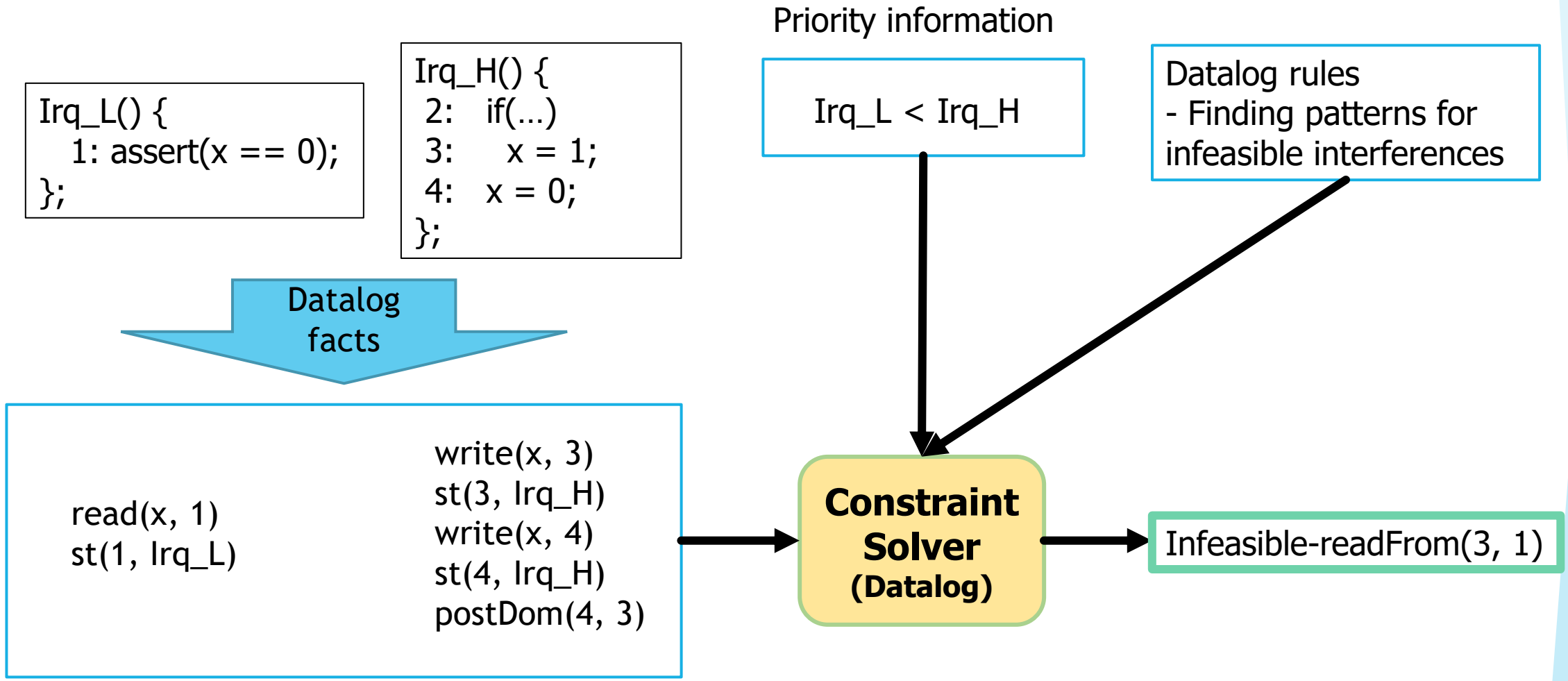


Irq_L cannot preempt Irq_H

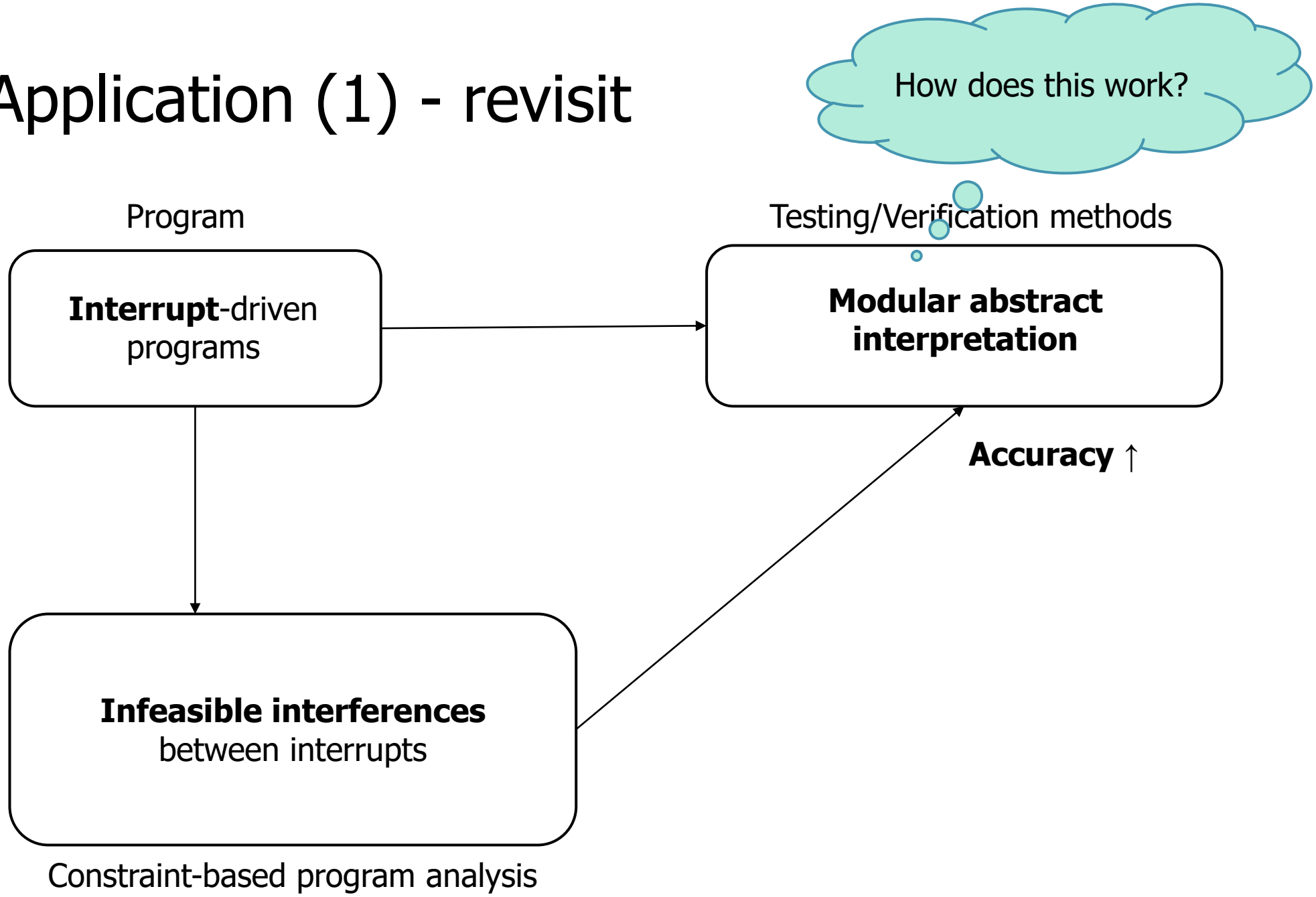
Application (1) - revisit



Example for computing infeasible interference



Application (1) - revisit

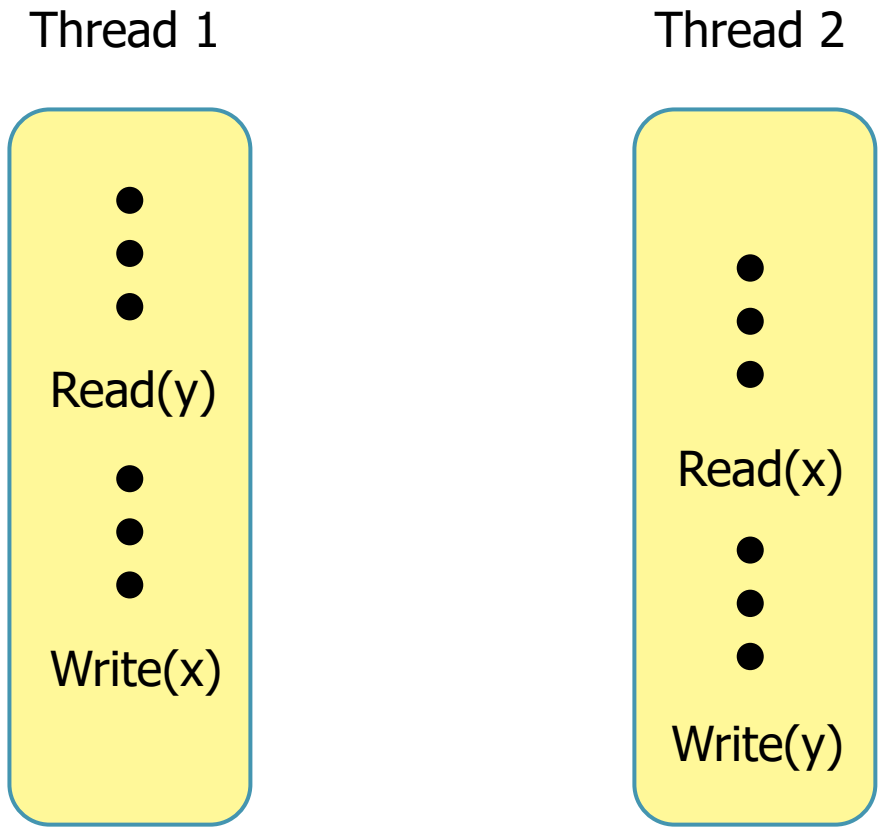


Modular abstract interpretation [Miné, VMCAI 2014]

Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract value to other threads as if there is an interference

Step 3
- If any value is changed, go to step 1
  else terminate

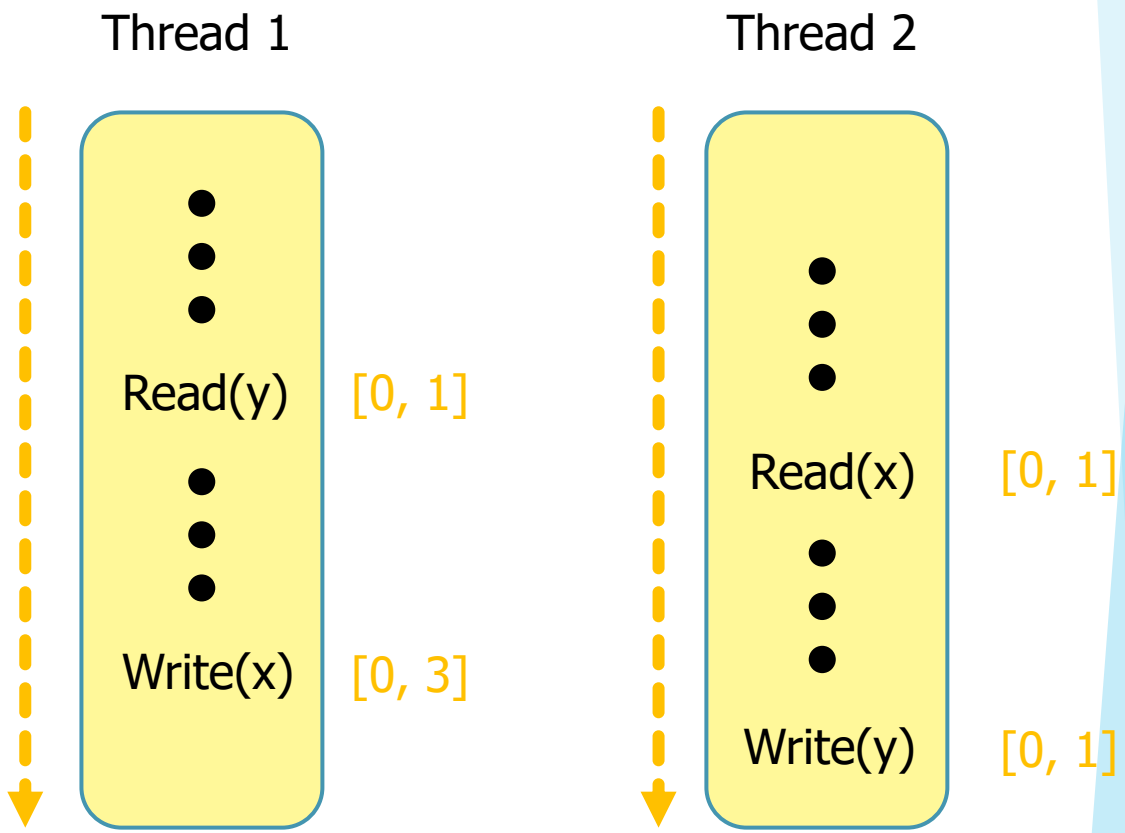


Modular abstract interpretation [Miné, VMCAI 2014]

Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or terminate

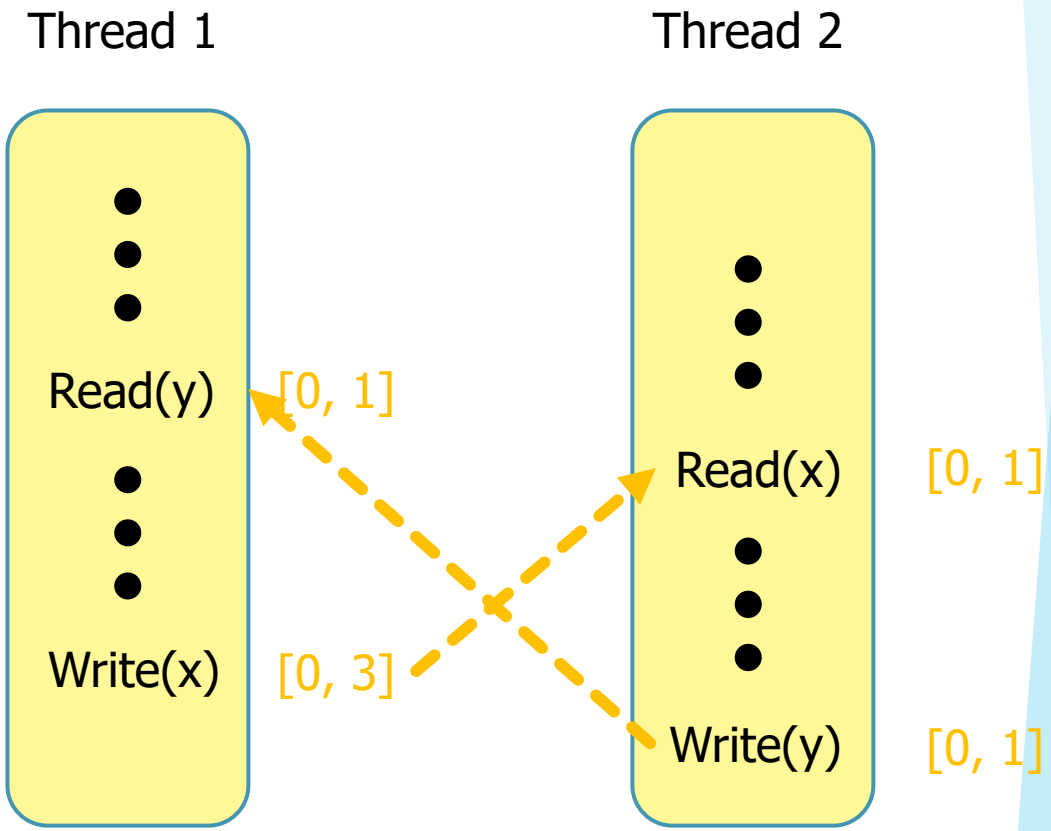


Modular abstract interpretation [Miné, VMCAI 2014]

Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or terminate

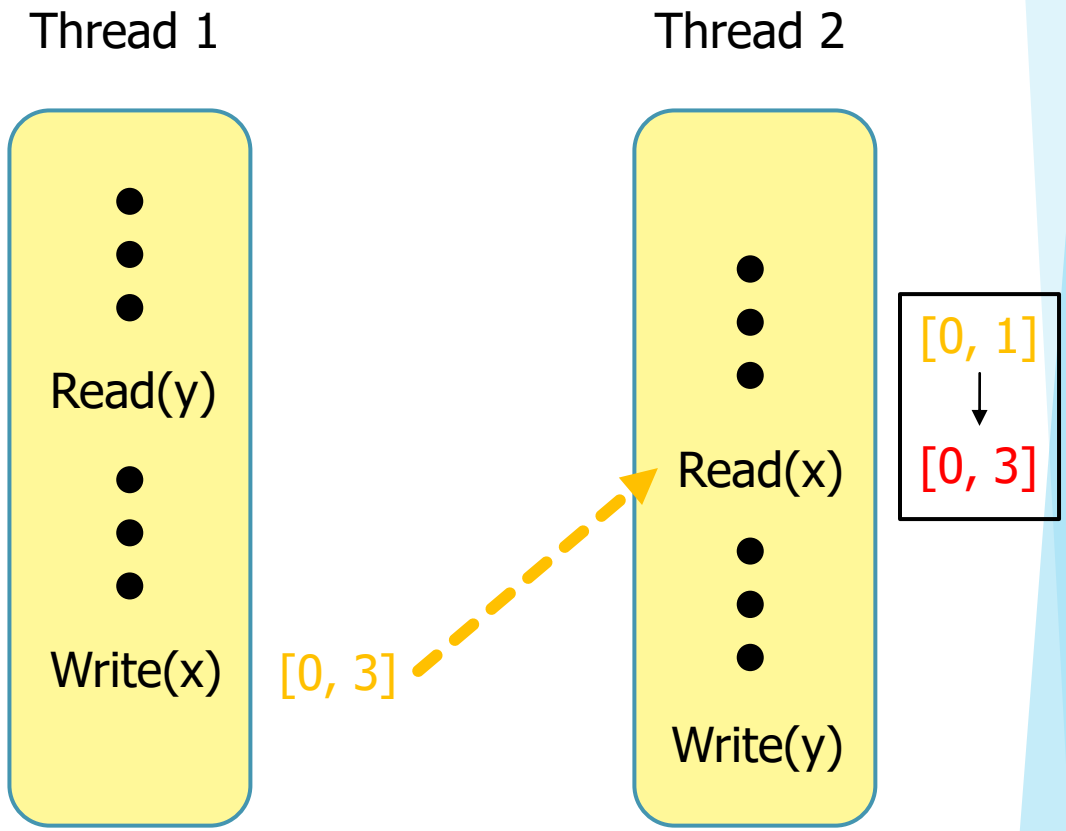


Modular abstract interpretation [Miné, VMCAI 2014]

Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or terminate

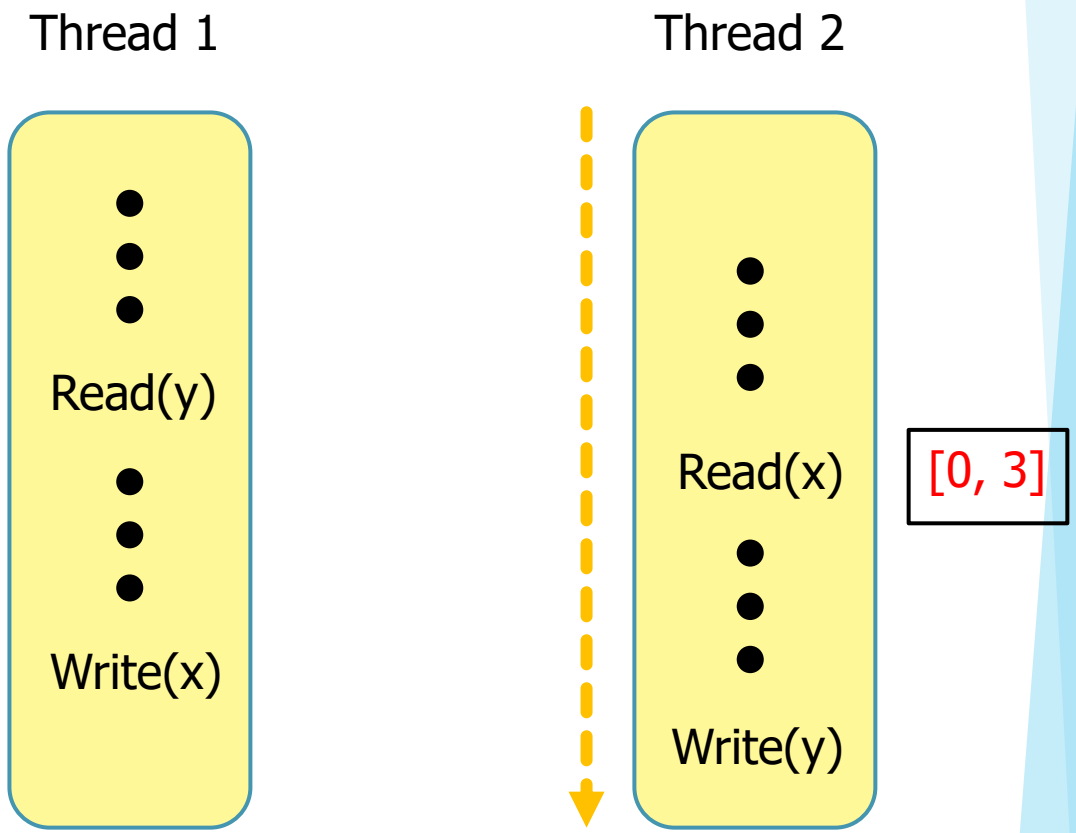


Modular abstract interpretation [Miné, VMCAI 2014]

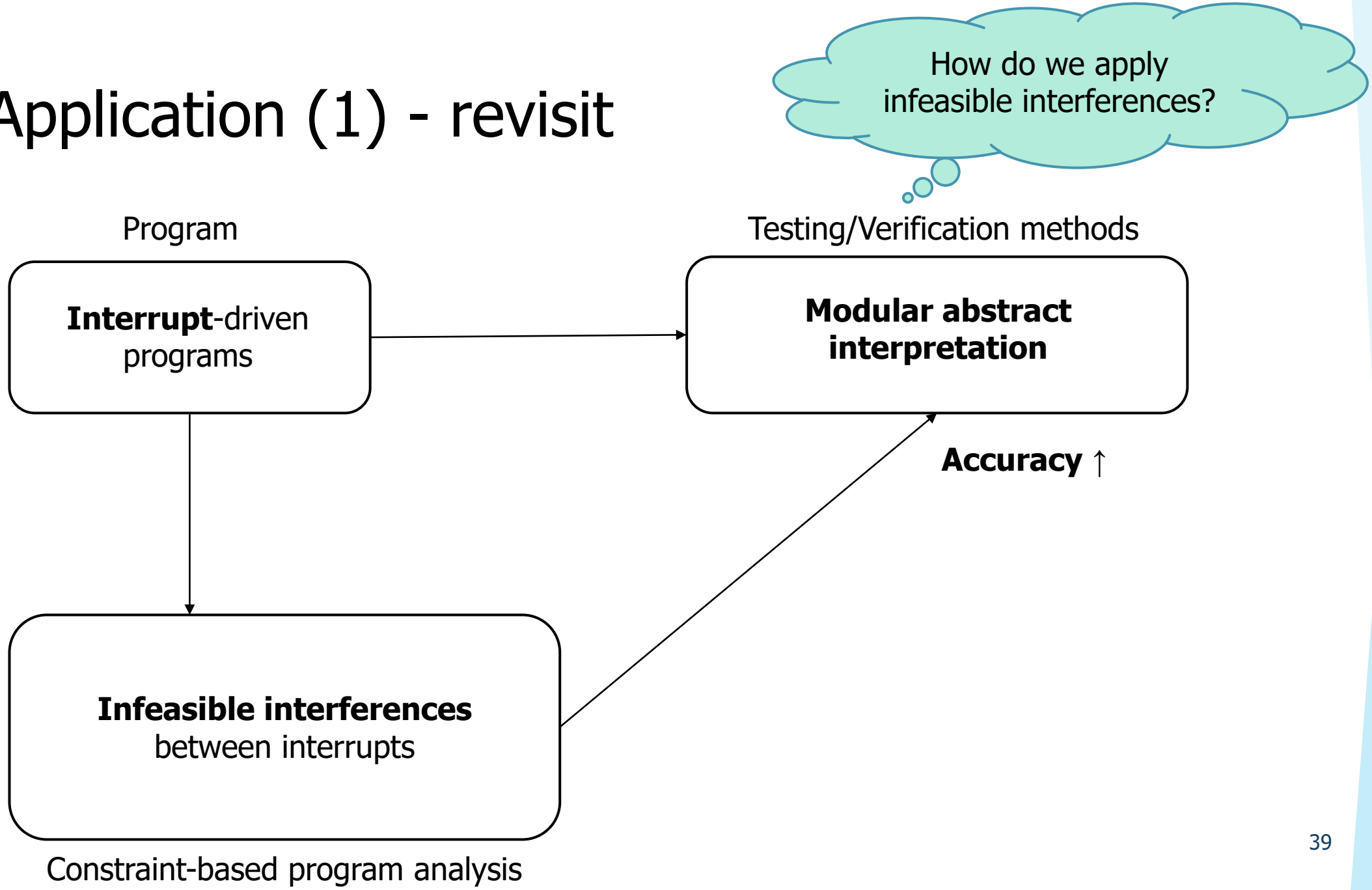
Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or terminate



Application (1) - revisit

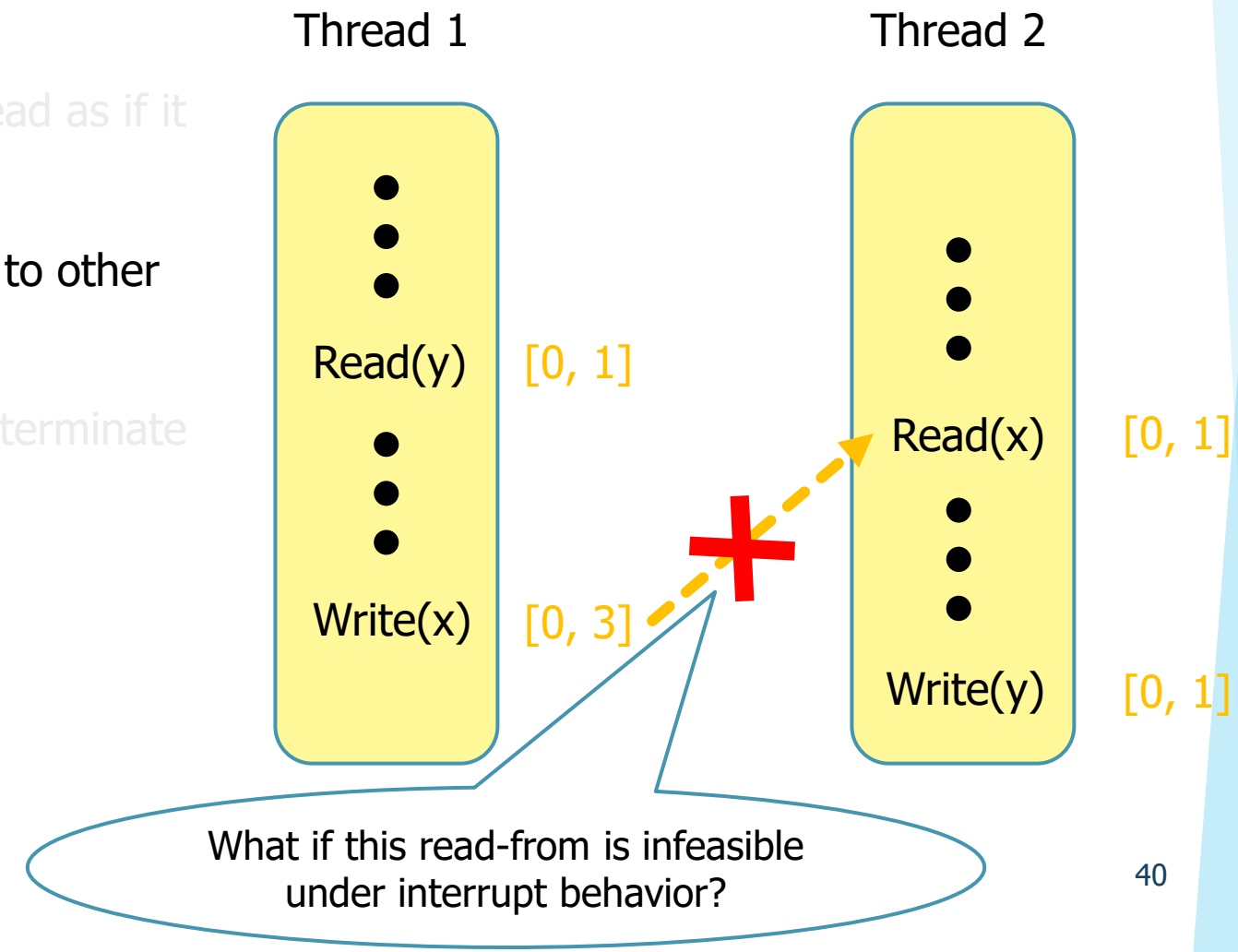


Modular abstract interpretation with infeasible interference

Step1
- Run abstract interpretation for each thread as if it is a sequential program

Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or terminate



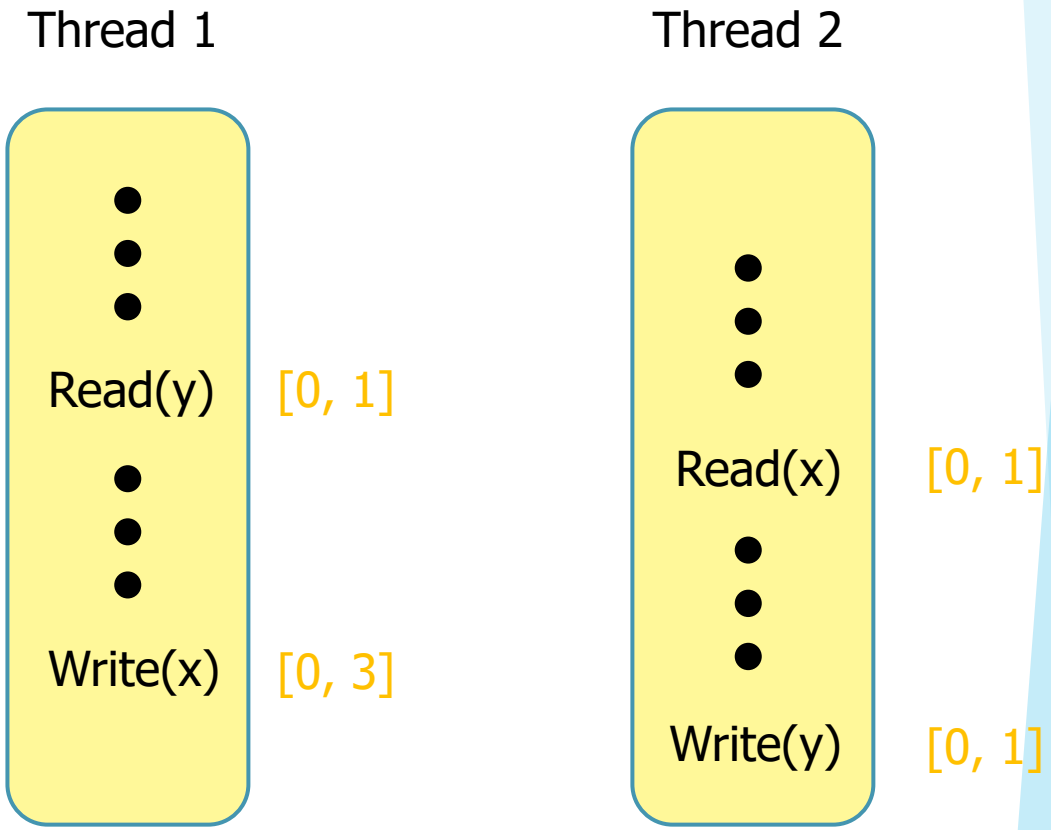
Modular abstract interpretation with infeasible interference

Step1
- Run abstract interpretation for each thread as if it is a sequential program

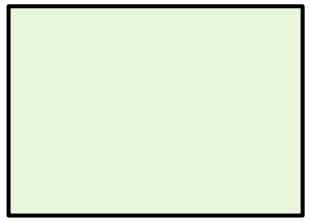
Step 2
- Propagate the abstract range of a value to other threads as if there is interference

Step 3
- If any value is changed, go to step 1 or **terminate**

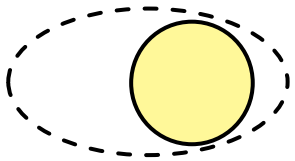
It will terminate with more accurate abstract value



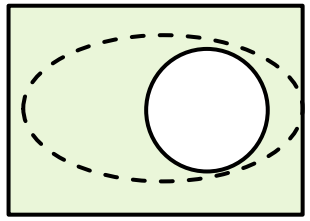
Under-approximation of infeasible interference



-> All visible interferences in thread models (considered for the previous verification method)

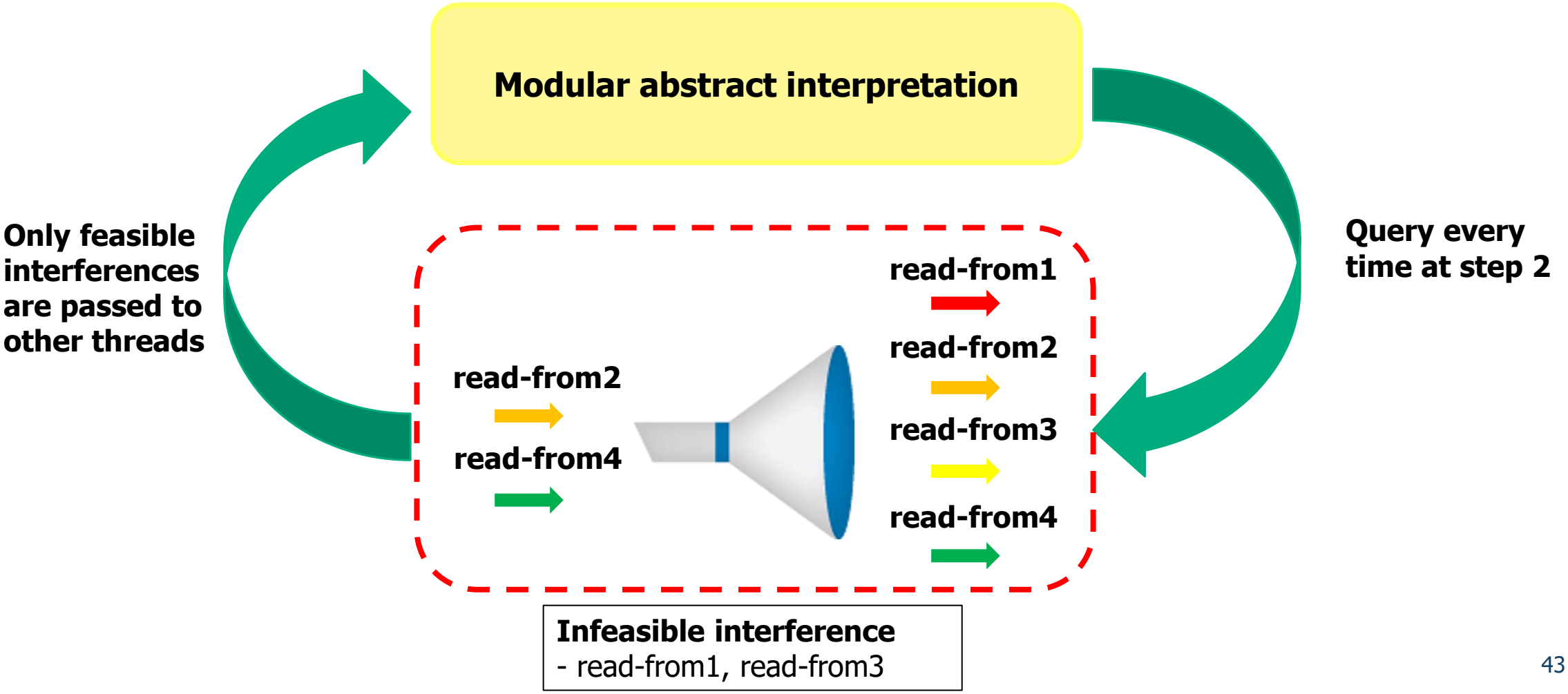


-> Infeasible read-from edges **with specific patterns (Our analysis)**
It is subset of all infeasible read-from edges (dashed circle)



-> Considering this space, the verification may not filter all infeasible read-from edges, but it **guarantees** to not miss feasible interferences.

Checking feasibility during the process



Experimental setup for Application (1)

- Purpose
  - Are there many infeasible edges?
  - Do we **improve accuracy** by removing false positives (and obtaining more proofs) ?

- Benchmarks
  - Control software, firmware and device drivers considered in previous multi-threaded/interrupt verification approaches
    [Miné VMCAI 2014, Kusano et al. FSE 2017, Kroening et al. Date 2015]

Result of finding infeasible edges under interrupt behavior

Summary	
Num. of benchmarks	**35**
Total LOC	**22,541 lines**
Analysis Time	**64.21s**

Total num. of read-from edges (Thread behavior)	Num. of read-from edges filtered (Interrupt behavior)
5,116	**3,560**

69% of total edges are infeasible

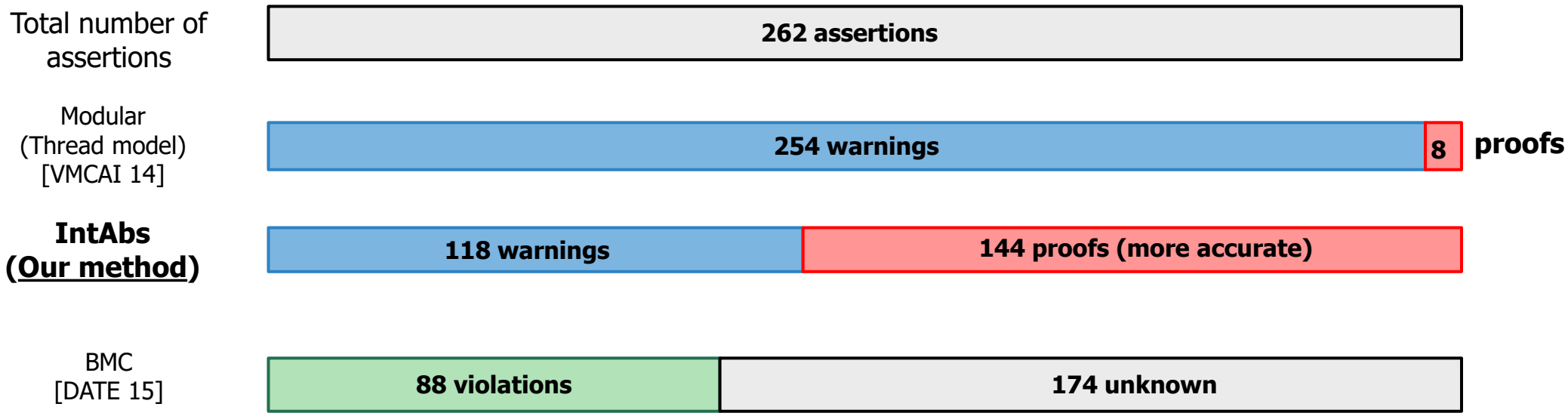
Prior works used for comparison

- Modular abstract interpretation for multi-threaded programs
  [Miné, VMCAI 2014]

- CBMC-based interrupt verification (reports only violations)
  [Kroening et al. Date 2015]

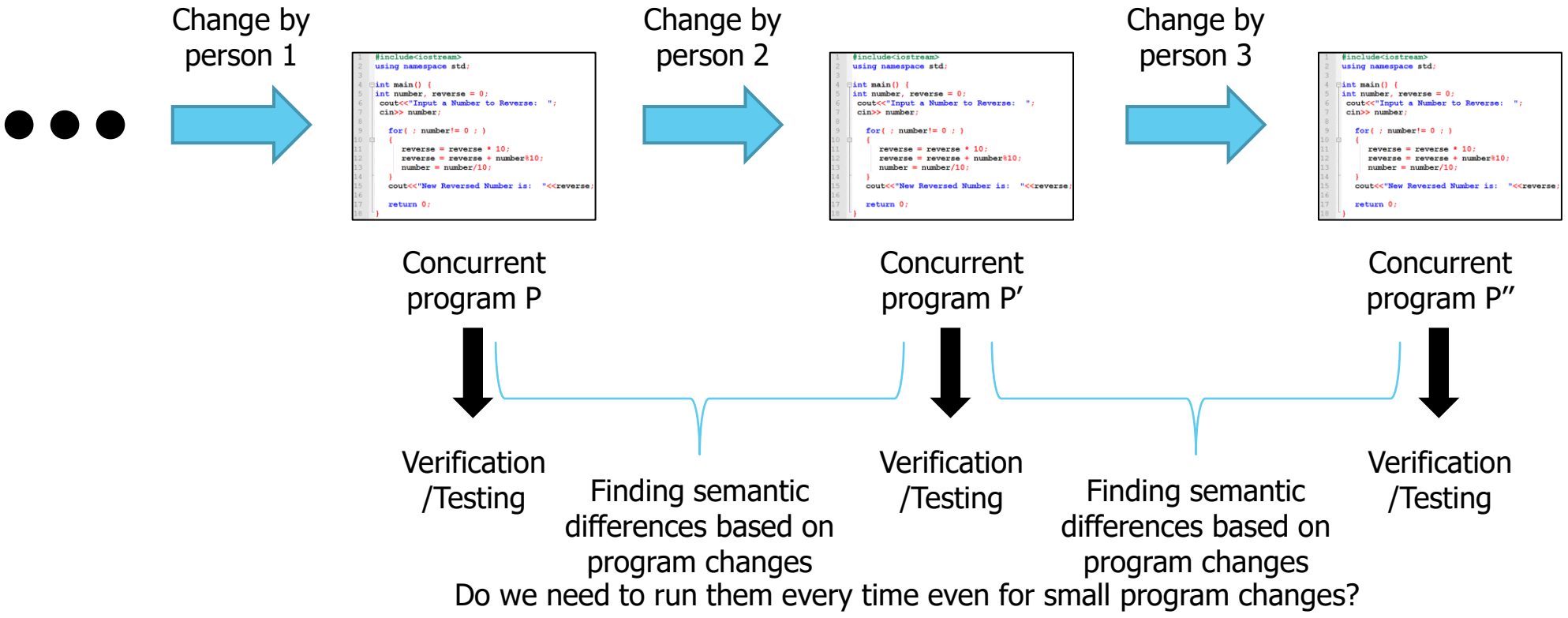
Result of accuracy in the verification of interrupts

- Proofs: the assertion is never violated
- Warnings: the assertion might be violated (it may include false positives)
- Violations: the assertion must be violated

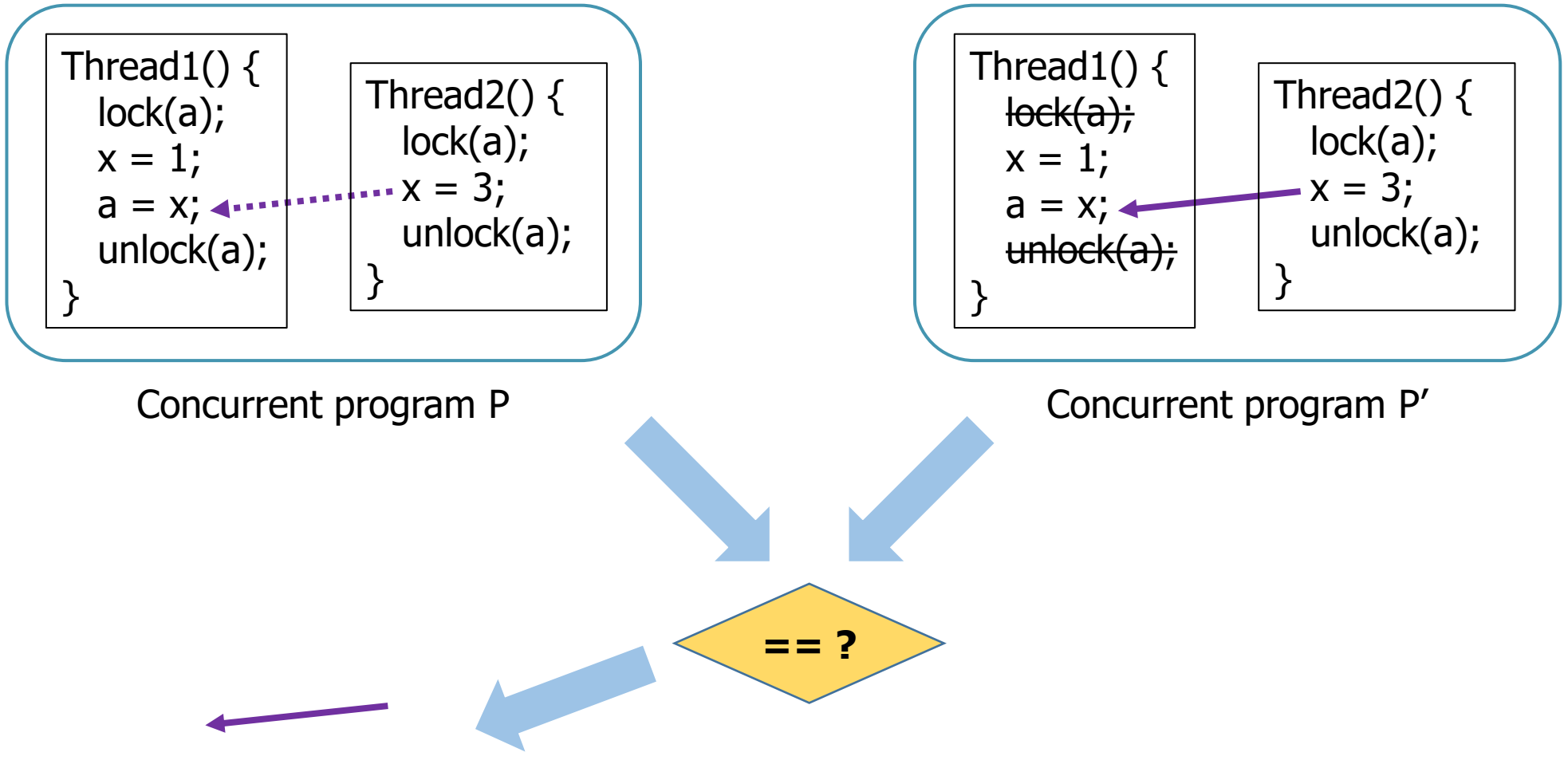


- **Background & Motivation**
- **Proposed method**
  - Customized constraint-based analysis for concurrent software
- **Three applications**
  - More accurate assertion checking for interrupt-driven software [Sung et al., ASE 2017]
  - **More scalable semantic diffing of multi-threaded programs [Sung et al., ASE 2018]**
  - More efficient testing web applications [Sung et al., FSE 2016]
- **Conclusion**

The need for computing semantic differences

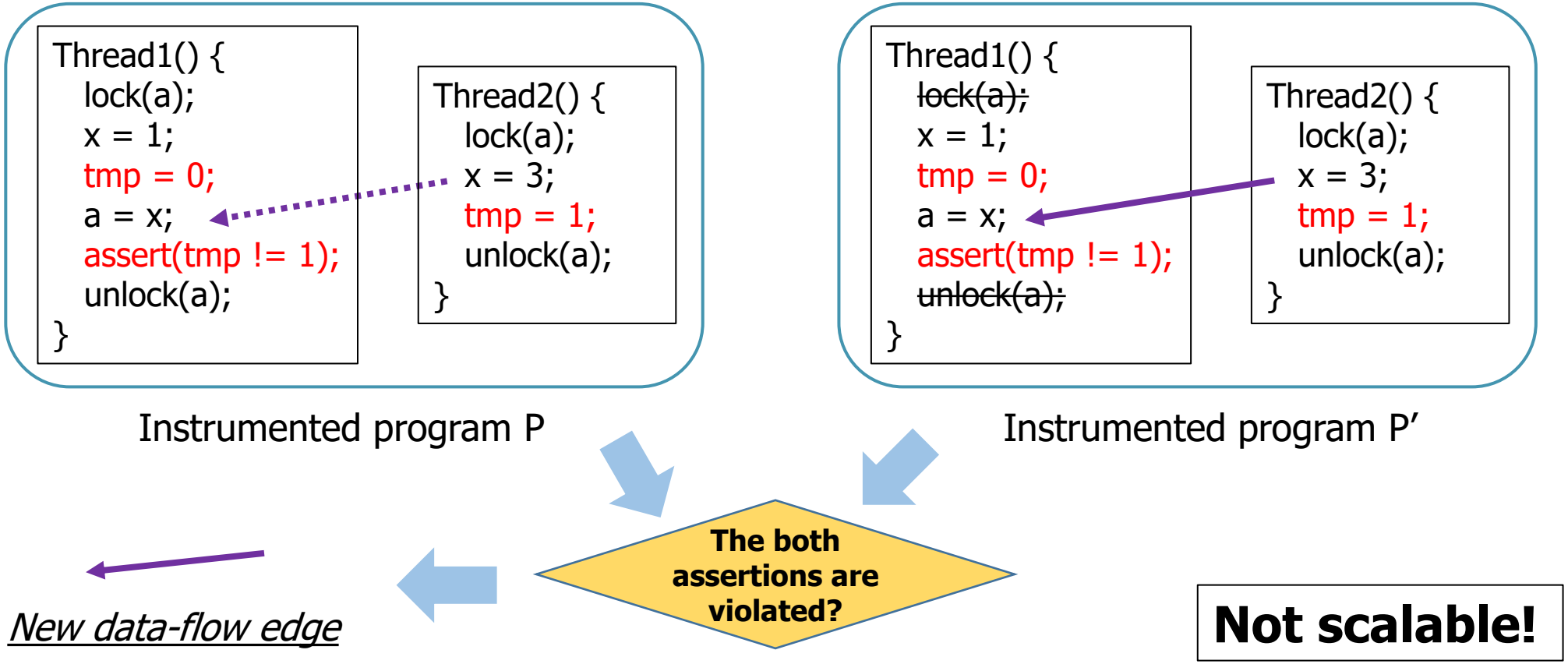


Semantic difference

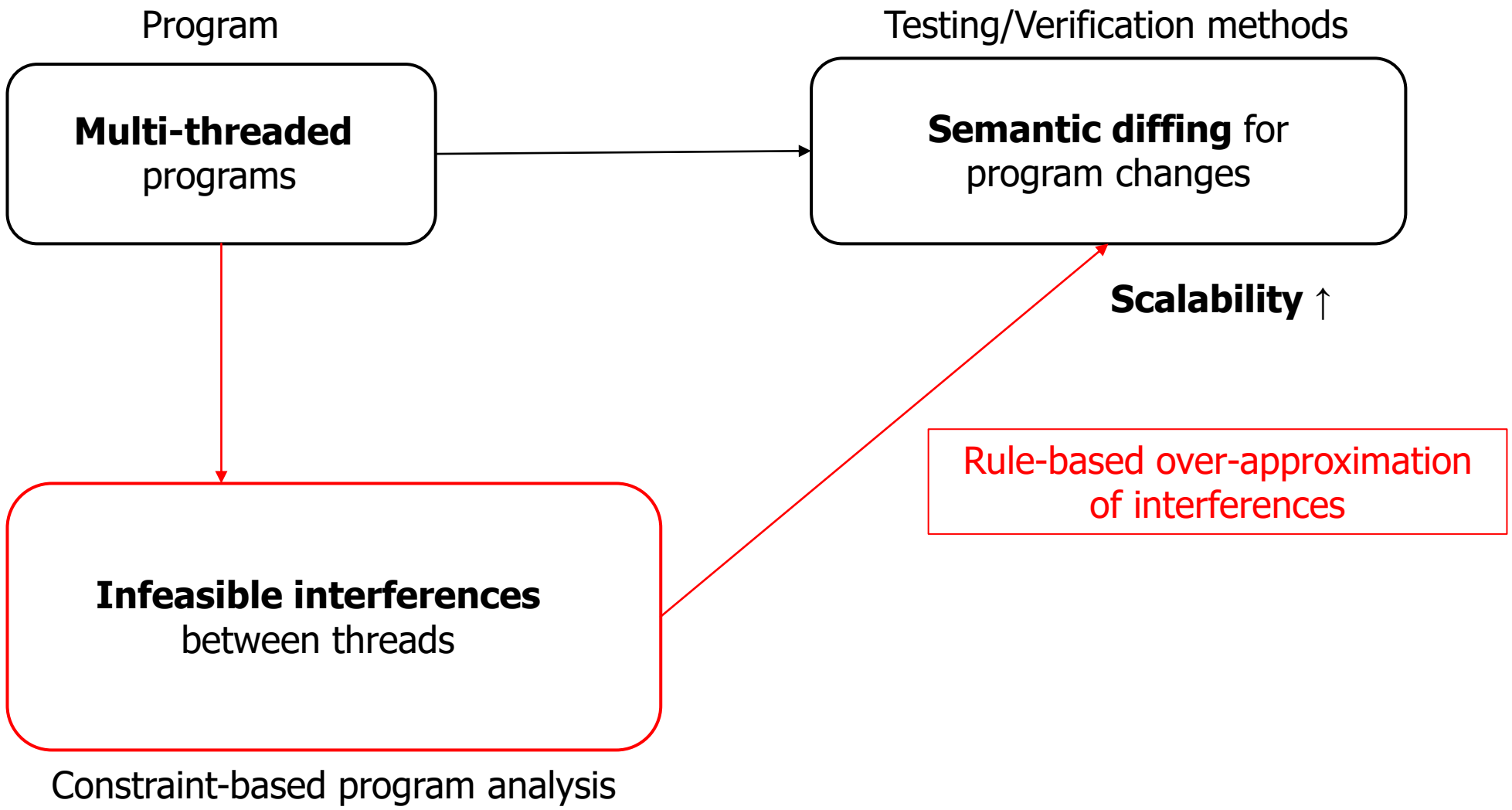


Semantic difference – prior work

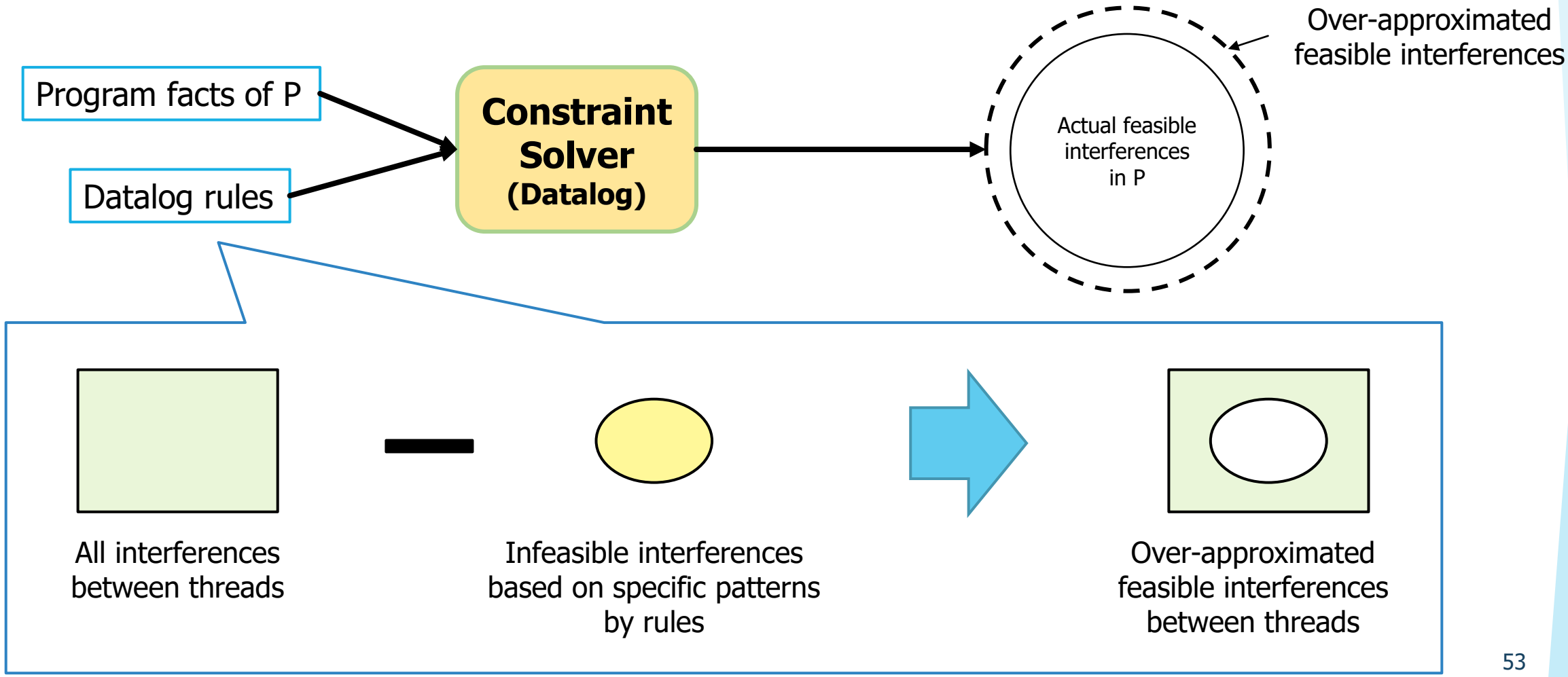
- Bounded Model Checking (BMC) based approach [Bouajjani et al. SAS 2017]



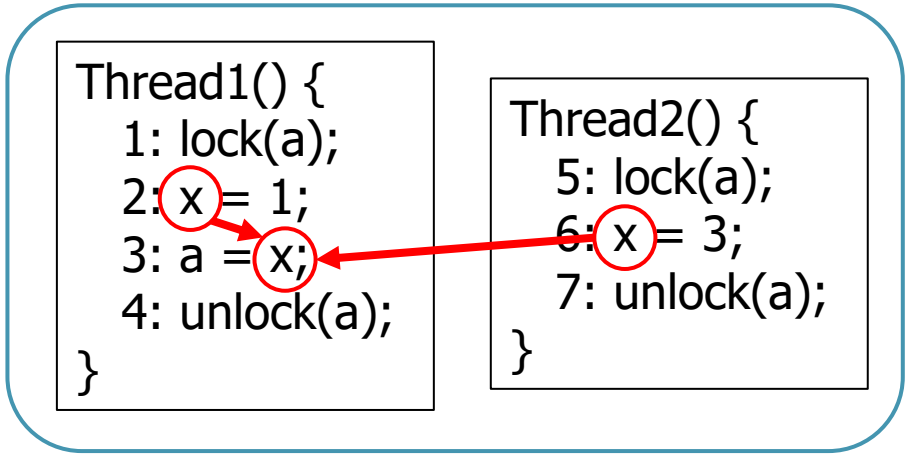
Scenario (2) – revisit



Over-approximated feasible interference for each program



Semantic difference – our approach

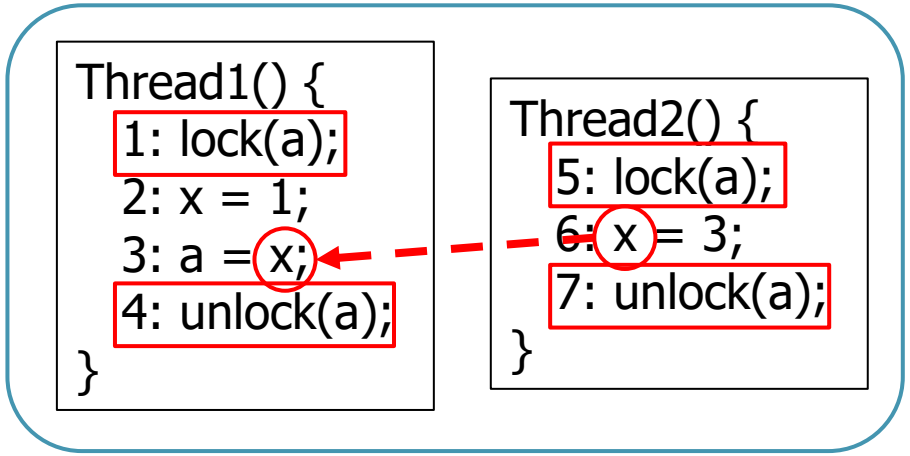


Concurrent program P

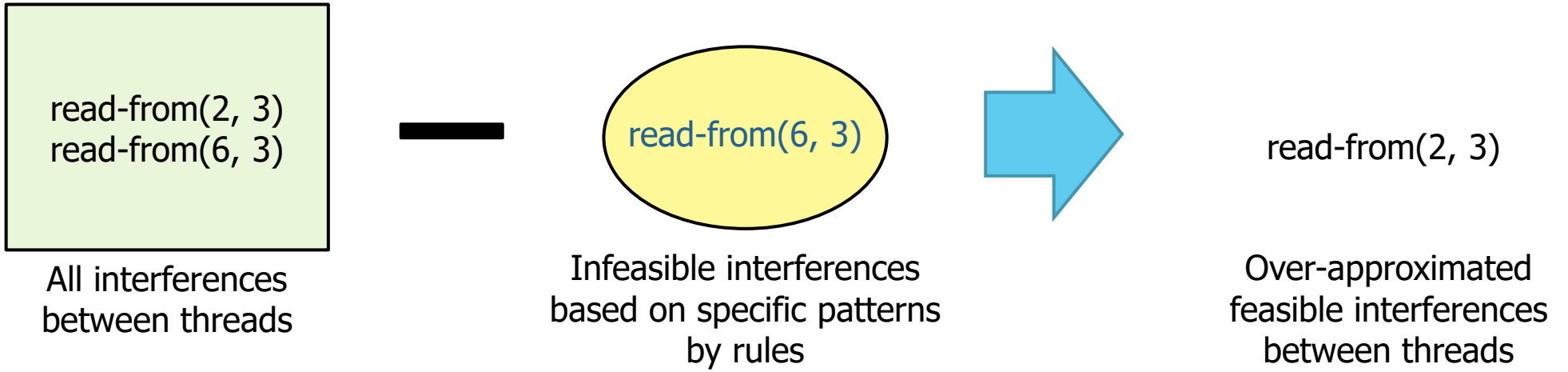
read-from(2, 3)
read-from(6, 3)

All interferences between threads

Semantic difference – our approach



Concurrent program P

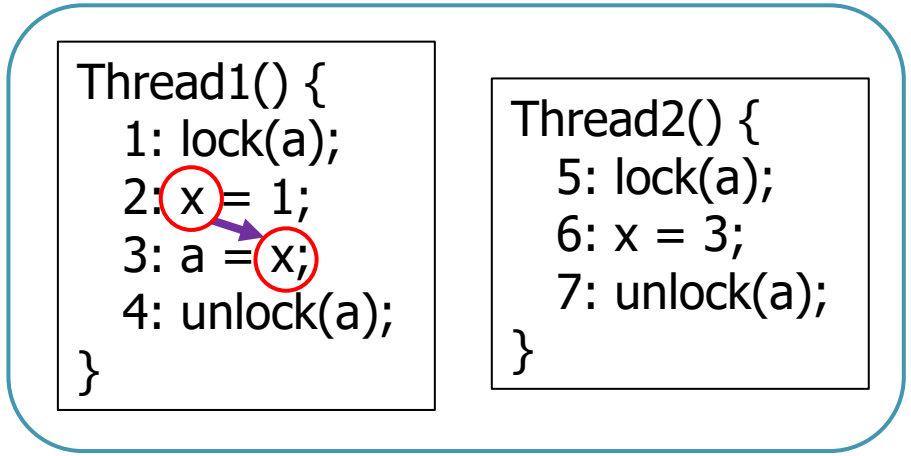


All interferences between threads

Infeasible interferences based on specific patterns by rules

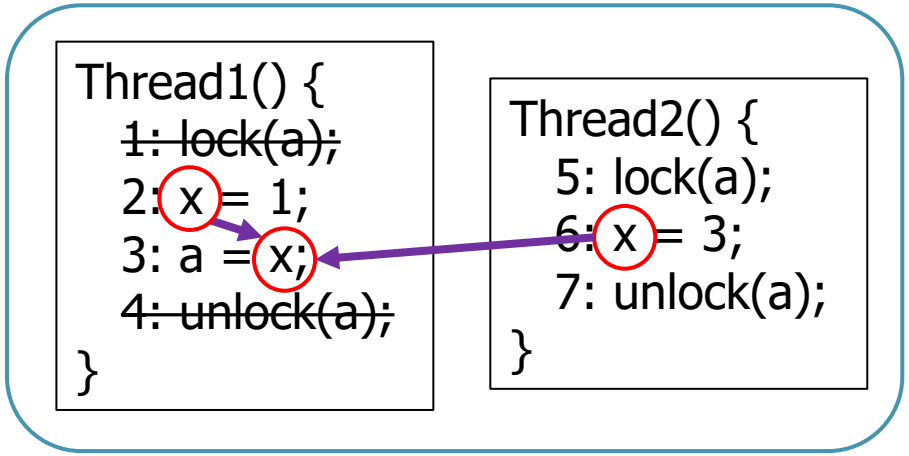
Over-approximated feasible interferences between threads

Semantic difference – our approach



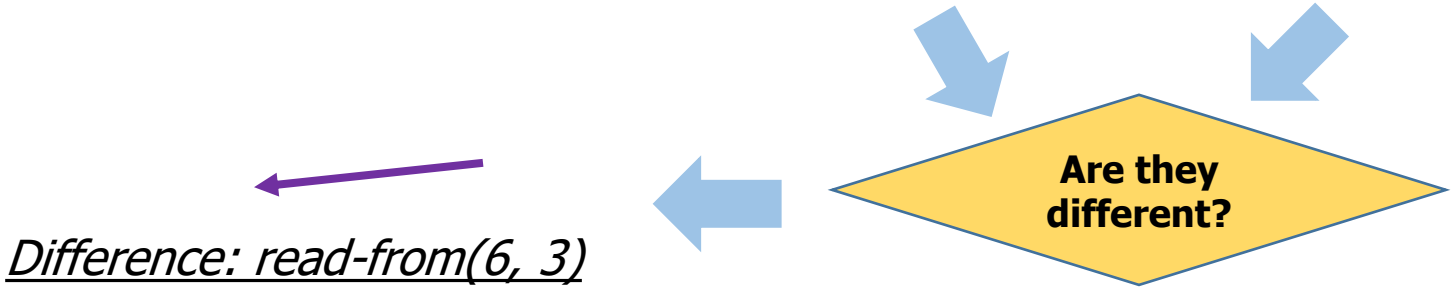
Concurrent program P

read-from(2, 3)

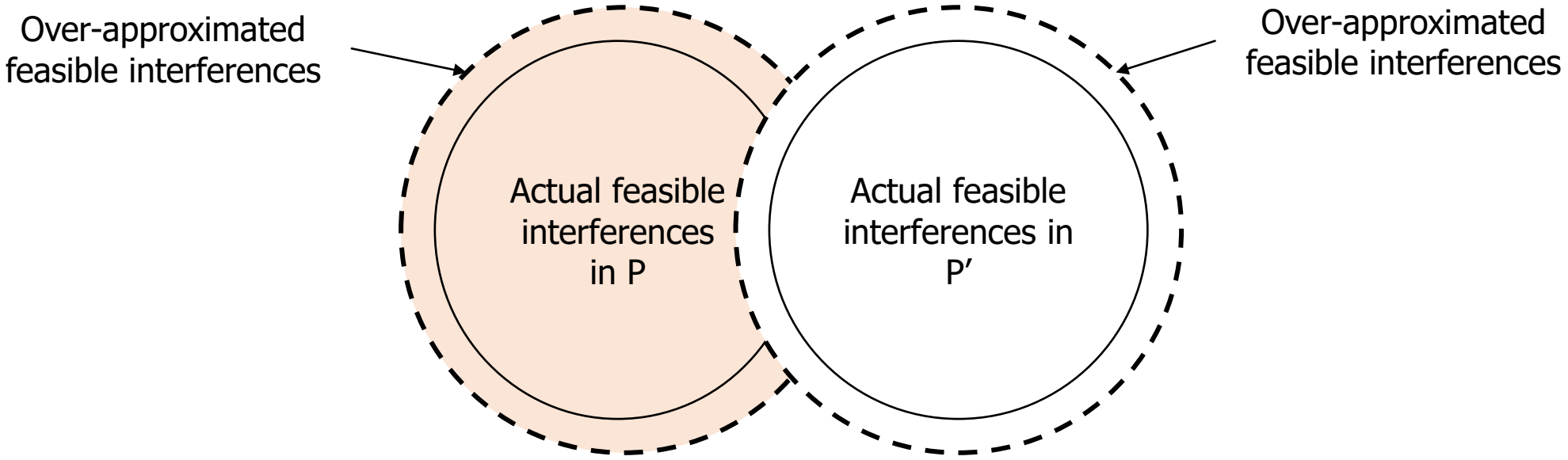


Concurrent program P'

read-from(2, 3),
read-from(6, 3)



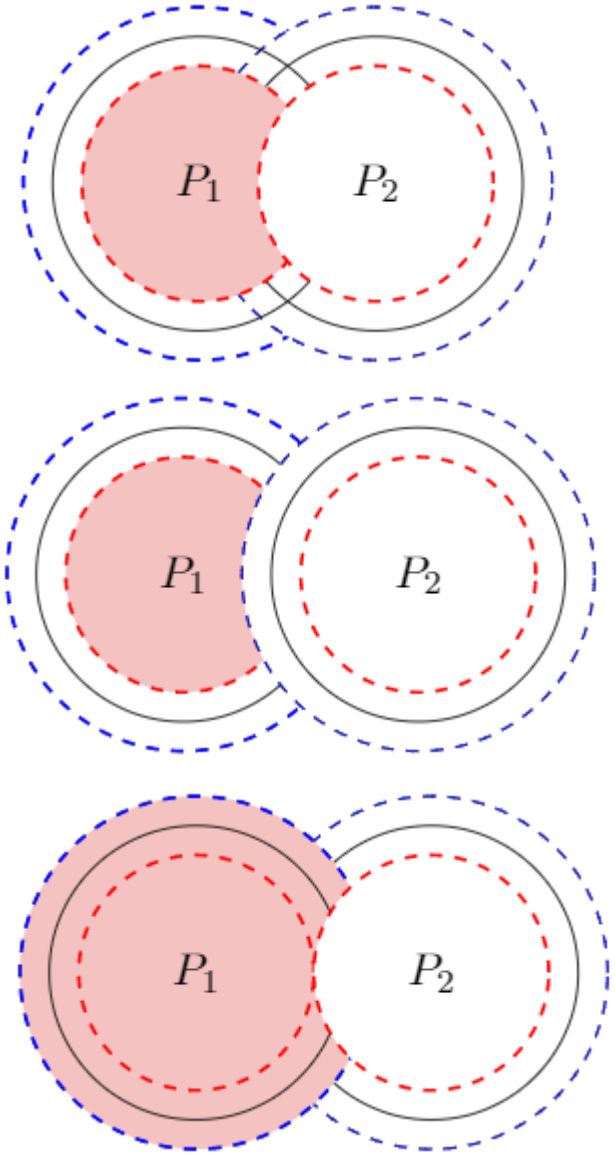
Diffing two sets of over-approximated interferences



Comparison of two over-approximated sets may lose soundness,
but we show (empirically) the changes related to concurrency are valid
(Thread-creation order, Cond-wait statements and Statement order)

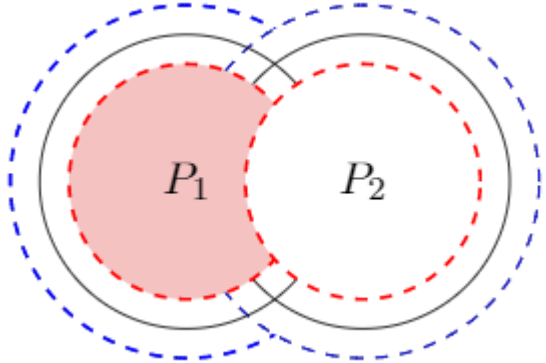
Possible differences (1)

- Under-approximation ($P1^{-}$) \ Under-approximation ($P2^{-}$)
- Under-approximation ($P1^{-}$) \ Over-approximation ($P2^{+}$)
- Over-approximation ($P1^{+}$) \ Under-approximation ($P2^{-}$)

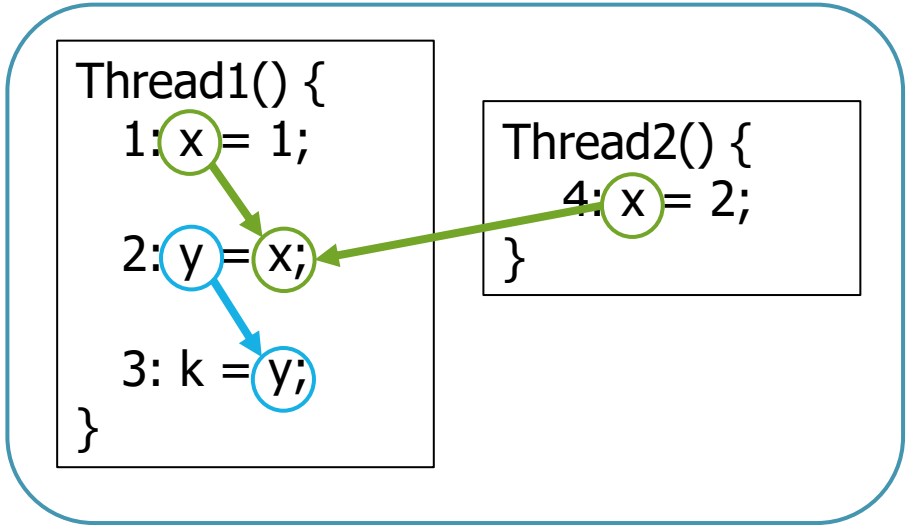


Possible differences (2)

- Under-approximation ($P1^-$) \ Under-approximation ($P2^-$)



It is not easy to compute under-approximated behaviors with Datalog rules since the rules only captures possible partial traces, not traces that must appear in all feasible traces



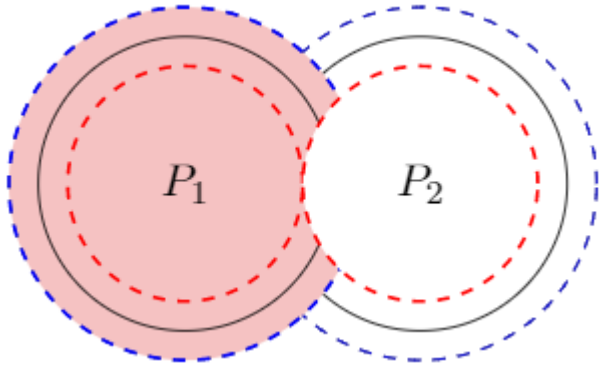
- Under-approximated read-from
  - ✓ {(2, 3)}
- Over-approximated read-from
  - ✓ {(1, 2), (4, 2), (2, 3)}

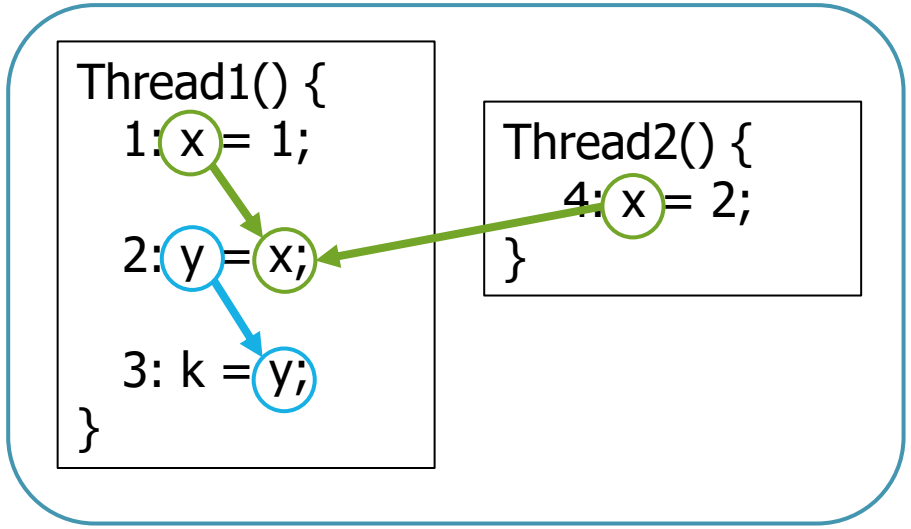
Cannot happen in the same execution

Possible differences (3)

- Over-approximation ($P1^{+}$) \ Under-approximation ($P2^{-}$)

It might produce differences for the same program





- Under-approximated read-from
  - ✓ {2, 3}
- Over-approximated read-from
  - ✓ {(1, 2), (4, 2), (2, 3)}

Result of Over-approximation ($P1^{+}$) \ Under-approximation ($P2^{-}$): {(1, 2), (4, 2)}

Experimental setup for Application (2)

- Purpose
  - Do we get **scalable diffing results** by finding **valid** semantic differences?
  - Specific to concurrency-related changes: Thread-creation order, Cond-wait statements and Statement order

- Benchmarks
  - Concurrent program bug patches and changes
    [Yu et al. ACM SIGARCH 2009, Yu et al. Tech Report 2008, Bouajjani et al. SAS 2017, Beyer TACAS 2015, Bloem CAV 2014, Lu et al. ACM SIGARCH 2008]
  - Concurrent patches from bug reports (Gcc, Glib, Jetty and LLVM)

Scalability improvement for diffing

- Execution time improvement

The small-sized benchmarks	
Num. of benchmarks	41
Total LOC	5,546 lines
Types	Sync, Th.Order, St.Order, Cond

Execution time of previous approach [Bouajjani et al. SAS 2017]	Execution time of our approach
> 3 hours	**15.57 s**

The large-sized benchmarks	
Num. of benchmarks	6
Total LOC	7,986 lines
Types	Th.Order, Cond

Execution time of previous approach [Bouajjani et al. SAS 2017]	Execution time of our approach
Not available	**140.28 s**

Emprical accuracy: the differences our approach found are ***all valid (inter-thread) dataflows***

- **Background & Motivation**
- **Proposed method**
  - Customized constraint-based analysis for concurrent software
- **Three applications**
  - More accurate assertion checking for interrupt-driven software [Sung et al., ASE 2017]
  - More scalable semantic diffing of multi-threaded programs [Sung et al., ASE 2018]
  - **More efficient testing web applications [Sung et al., FSE 2016]**
- **Conclusion**

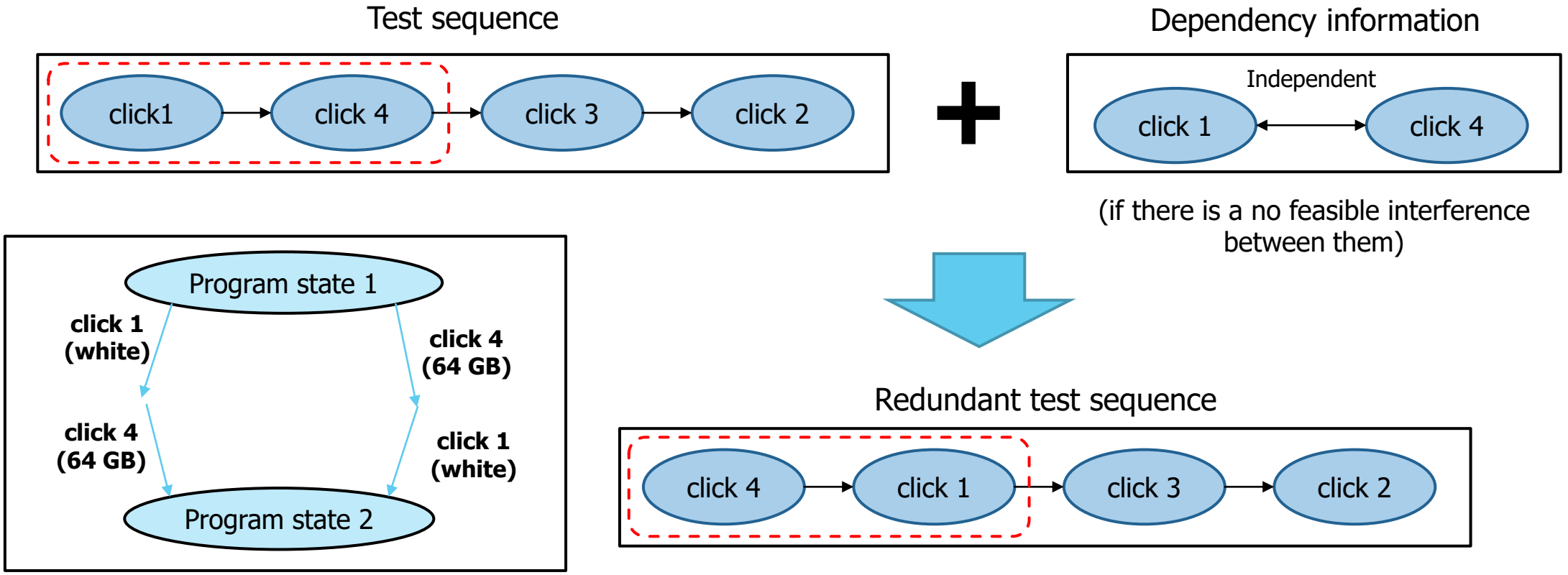
Testing web applications

Should we enumerate all possible sequences of clicks?

"click 1 (color white) -> click 4 (64 GB) -> click 6 (Add to Bag)" => Add to Bag (white + 64 GB)
"click 4 (64 GB) -> click 1 (color white) -> click 6 (Add to Bag)" => Add to Bag (white + 64 GB)

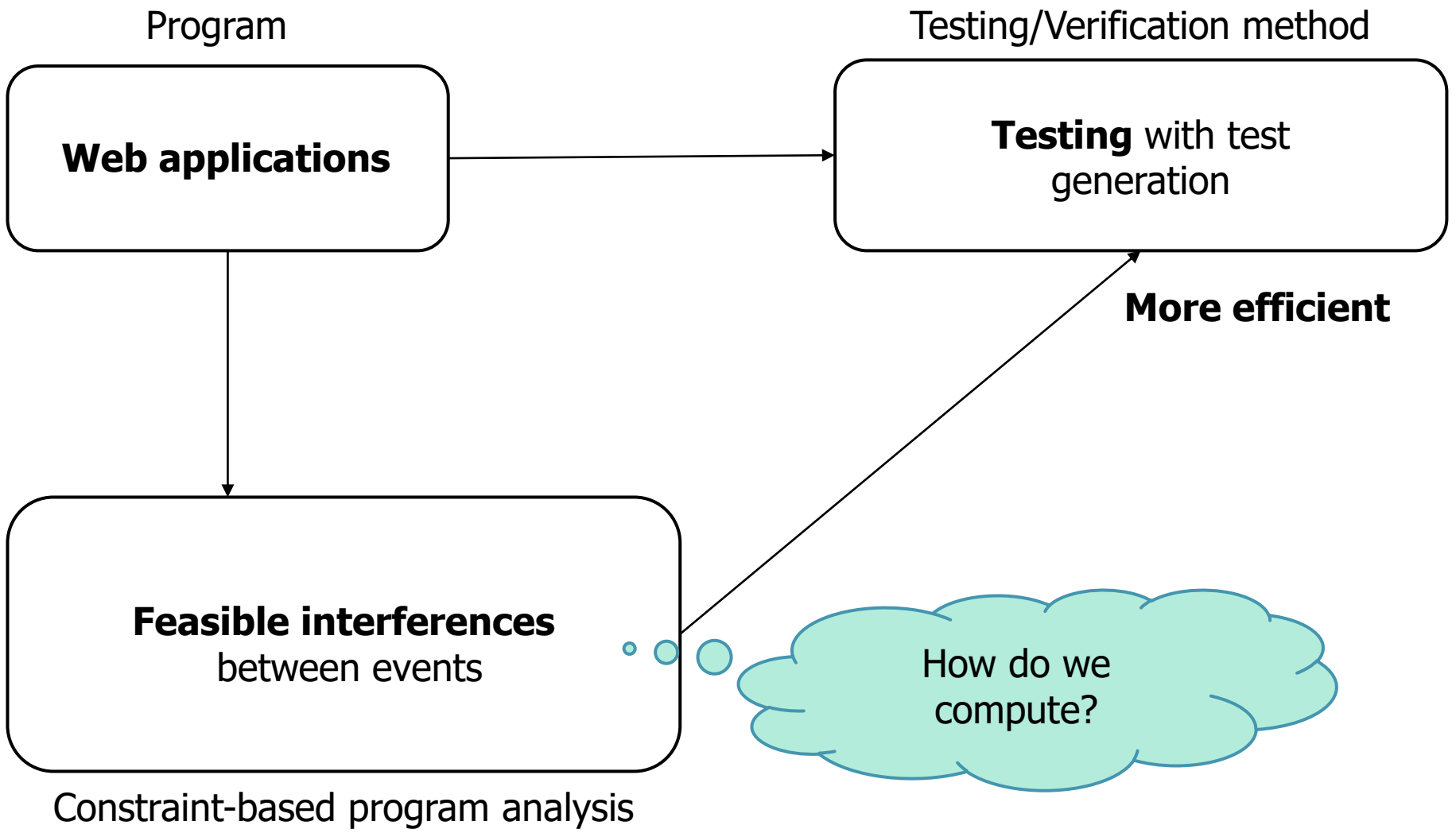
Finding redundant test sequence

- inspired by Partial Order Reduction [Godefroid, Springer 1996]

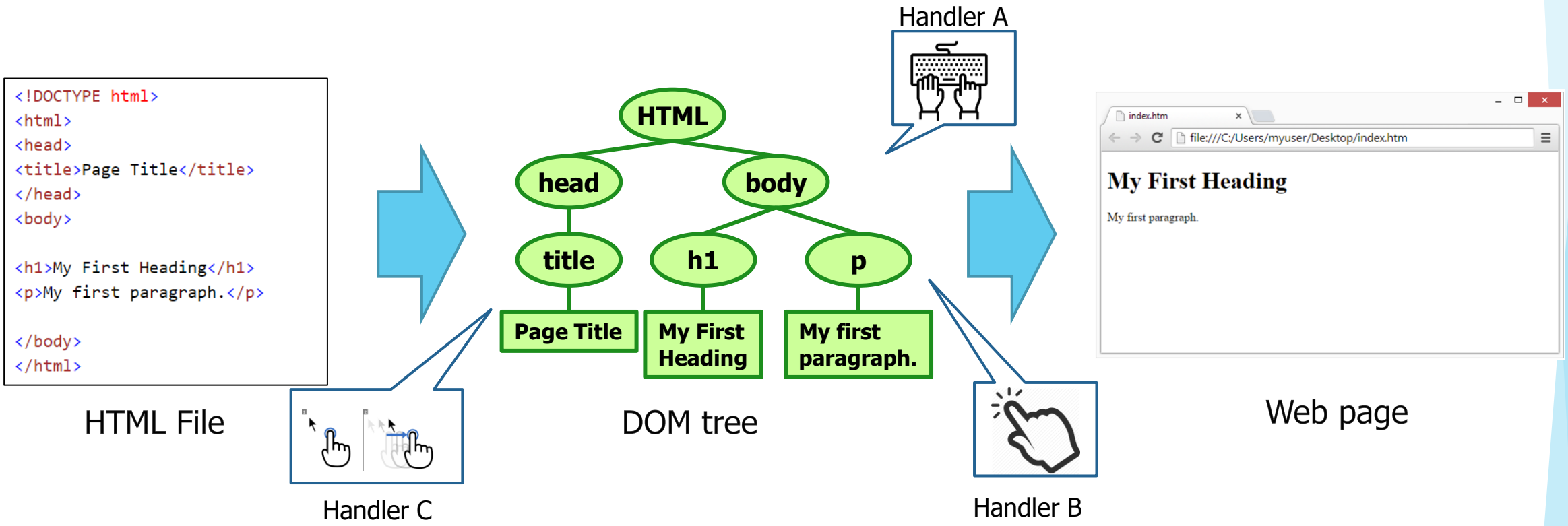


Running click1->click4 and click4->click1 from the same program state will reach the same program state since running the events are independent

Application (3) - revisit

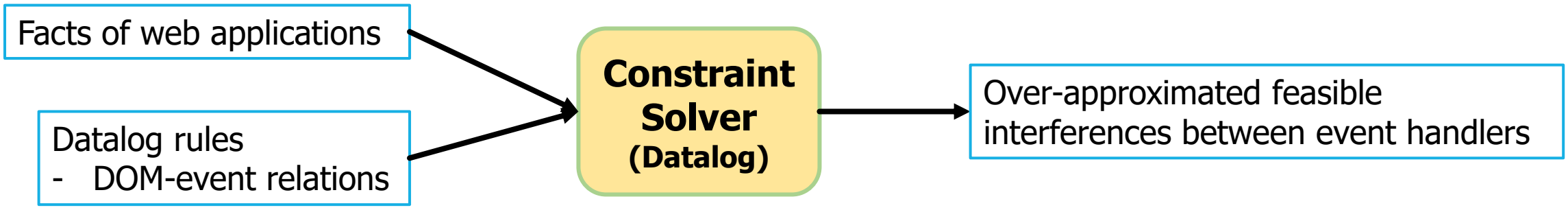


Understanding concurrent events in web applications

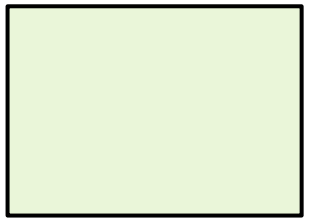


Event handlers are attached to DOMs -> We analyze dependency between DOM-events

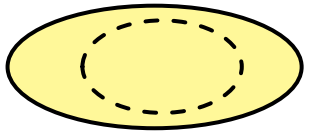
Computing interference between events



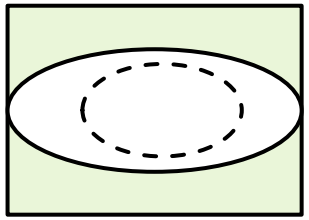
Over-approximation of feasible interferences



-> All event pairs considered in sequence generation for the testing (previous method)

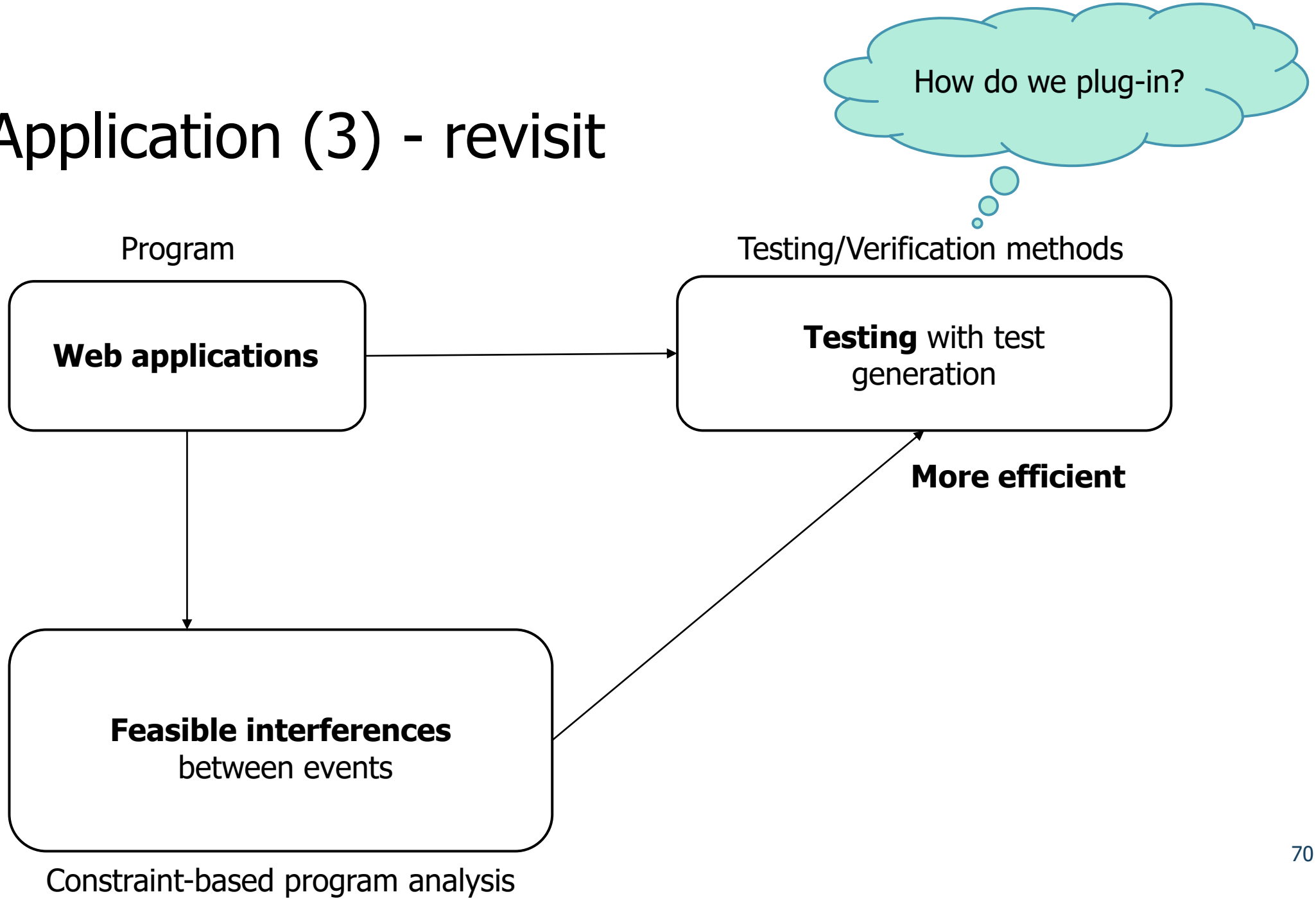


-> Event pairs which have over-approximated feasible interferences **(Our analysis)**
It subsumes event pairs which have actual feasible interferences (dashed circle)



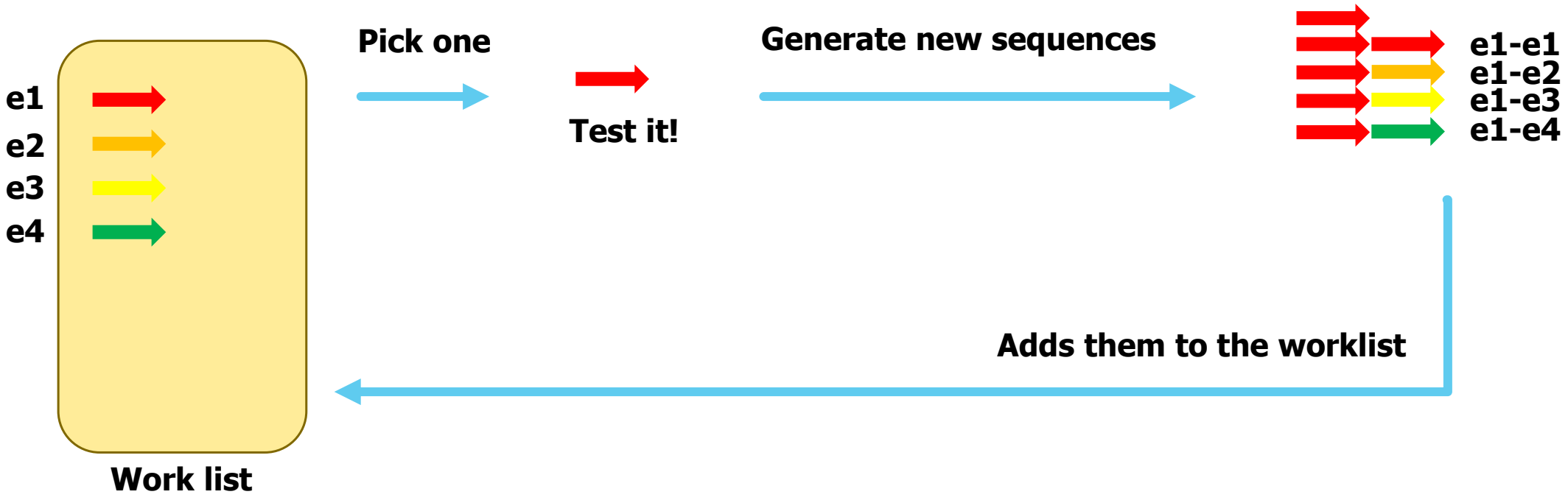
-> Considering this space for sequence generation may include actual infeasible interferences, but do not miss any feasible interferences in sequence generation.

Application (3) - revisit



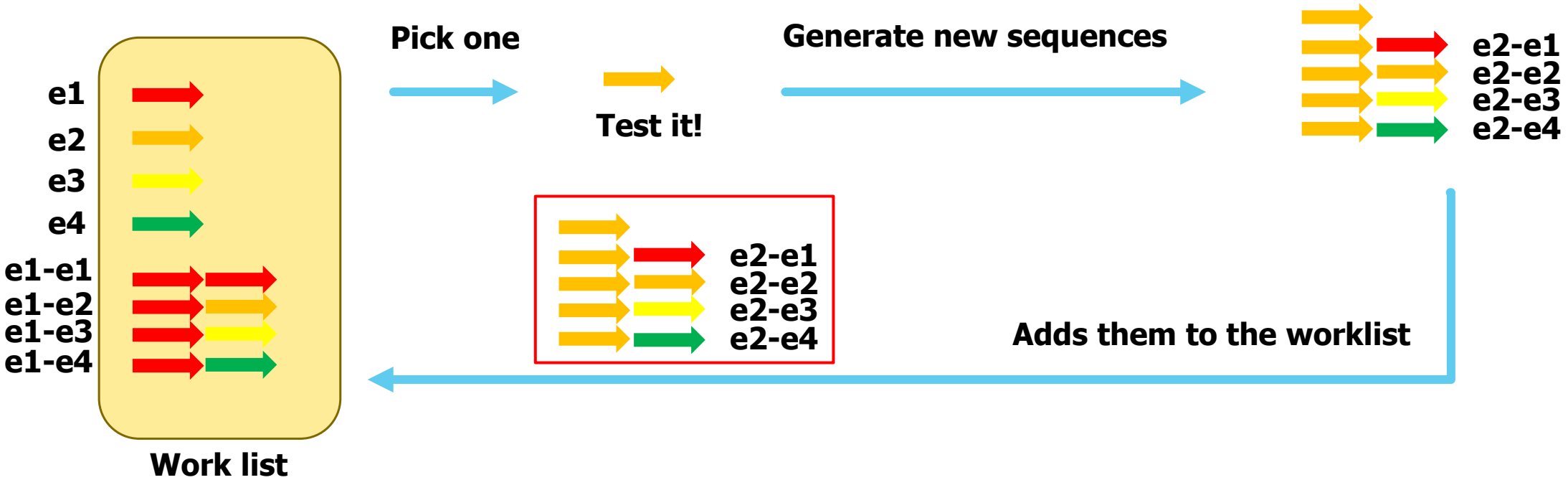
Artemis (testing tool) [Artzi et al., ICSE 2011]

Each iteration (one cycle)



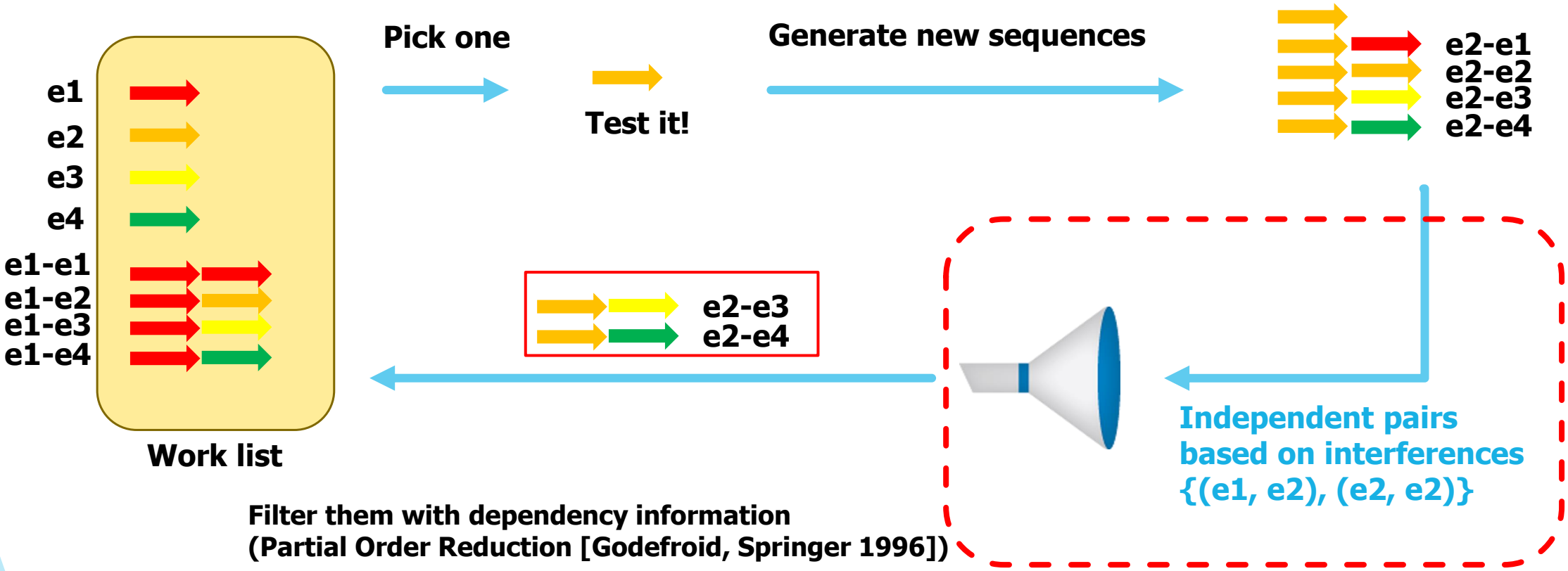
Artemis (testing tool) [Artzi et al., ICSE 2011]

Each iteration (one cycle)



Interference-enabled filtering redundant sequences (our improvement)

Each iteration (one cycle)



Experiments setup for Application (3)

- Purpose
  - Are there many independent (no interference) pairs of events?
  - Do we get **better test coverage (more efficient)** by filtering redundant test sequences?

- Benchmarks
  - JavaScript-based games from web
  - Benchmarks used from Artemis
    [Artzi et al., ICSE 2011]

Result of dependent event pairs

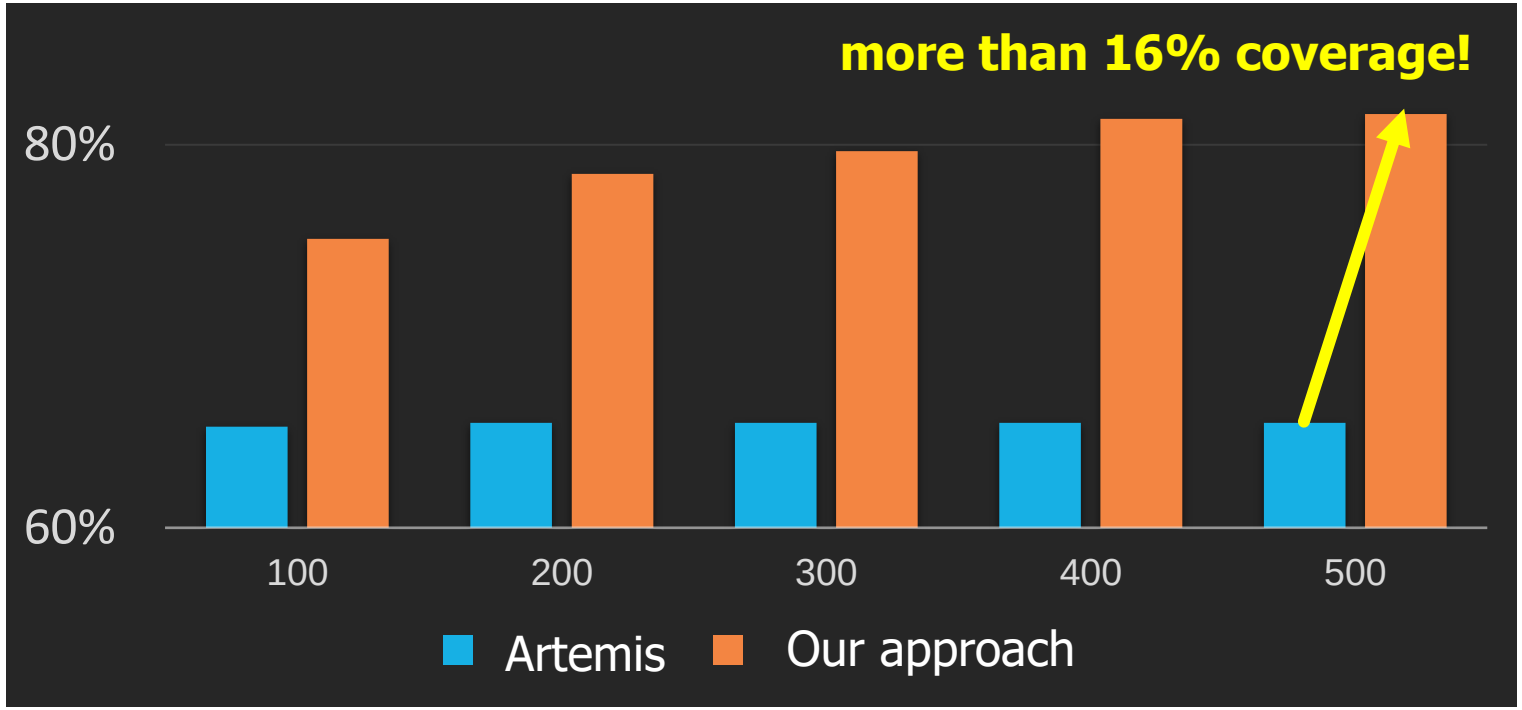
Summary	
Num. of benchmarks	**21**
Total LOC	**18,599 lines**
Constraints	**50,246**
Analysis Time	**50.11s**

Dep. pairs / Indep. pairs (Previous Artemis)	Dep. pairs / Indep. pairs (Our approach)
3,898 / 0	**2,120 / 1,778**

54% of max deps. are dependent

Result of test coverage in testing

- Branch coverage w.r.t. the number of iterations

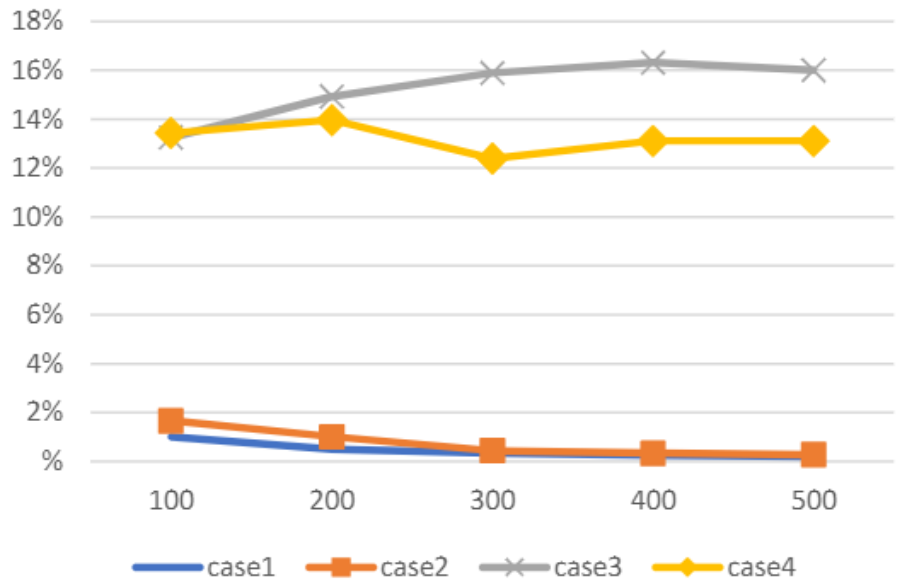


Y-axis: Branch coverage
X-axis: the number of iterations in Artemis

Further experiments

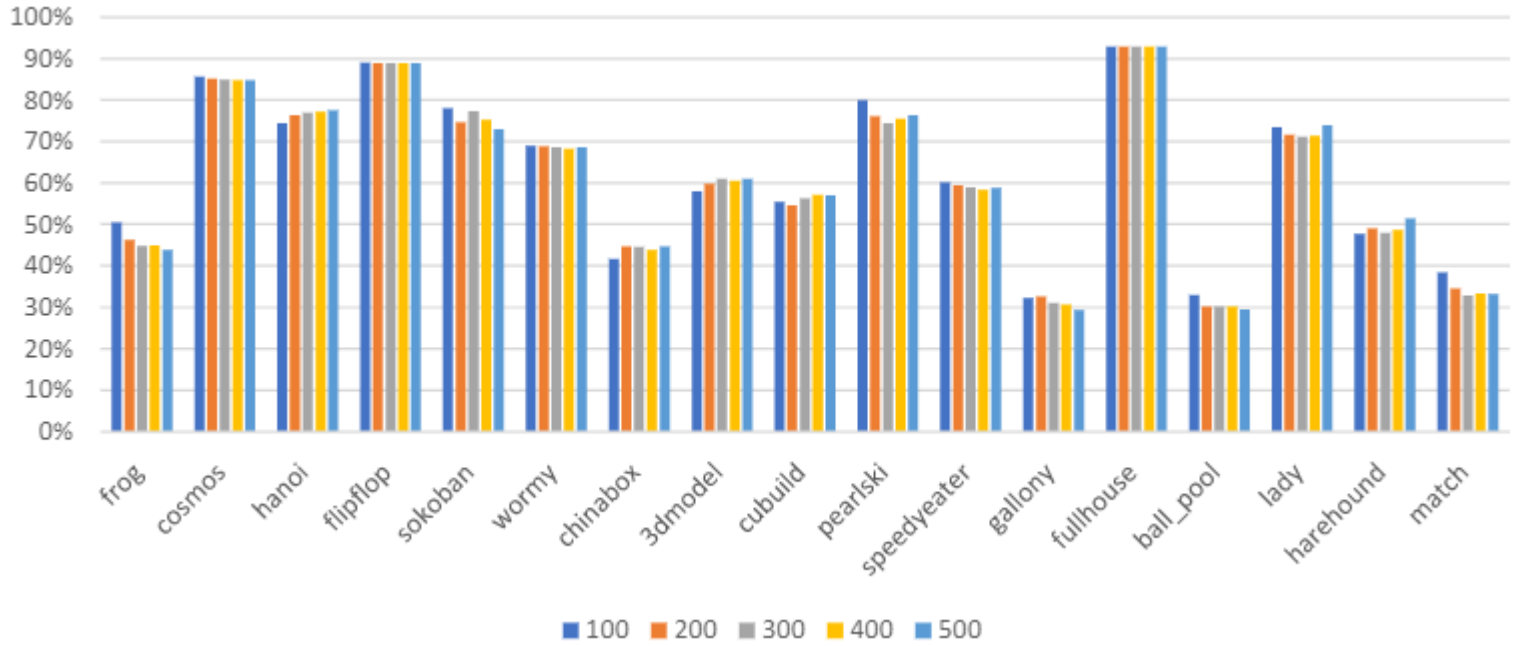
- Purpose
  - Does our approach reduce the size of worklist efficiently?
  - Does our approach have large overhead in terms of execution time?
    - What's the factors that affect testing time?
      - Length of test sequences
      - Code coverage and read/written properties in testing

Result of the size of worklist (1)



Y-axis: relative size of worklist in our approach compared to Artemis
X-axis: the number of iterations in Artemis

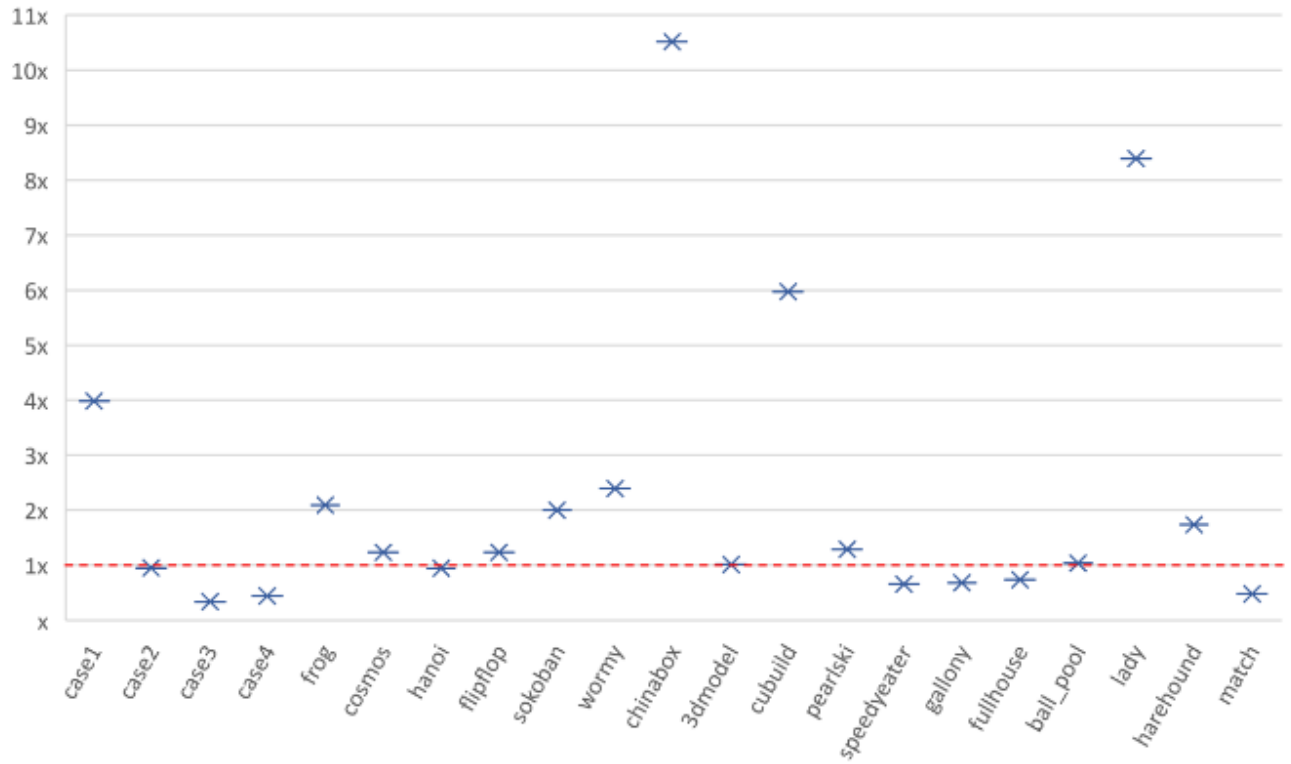
Result of the size of worklist (2)



Y-axis: relative size of worklist in our approach compared to Artemis
X-axis: benchmarks

The relative size of worklist does not change as we filter out dependencies linearly
-> The better testing coverage is possible if we further reduce the size of worklist

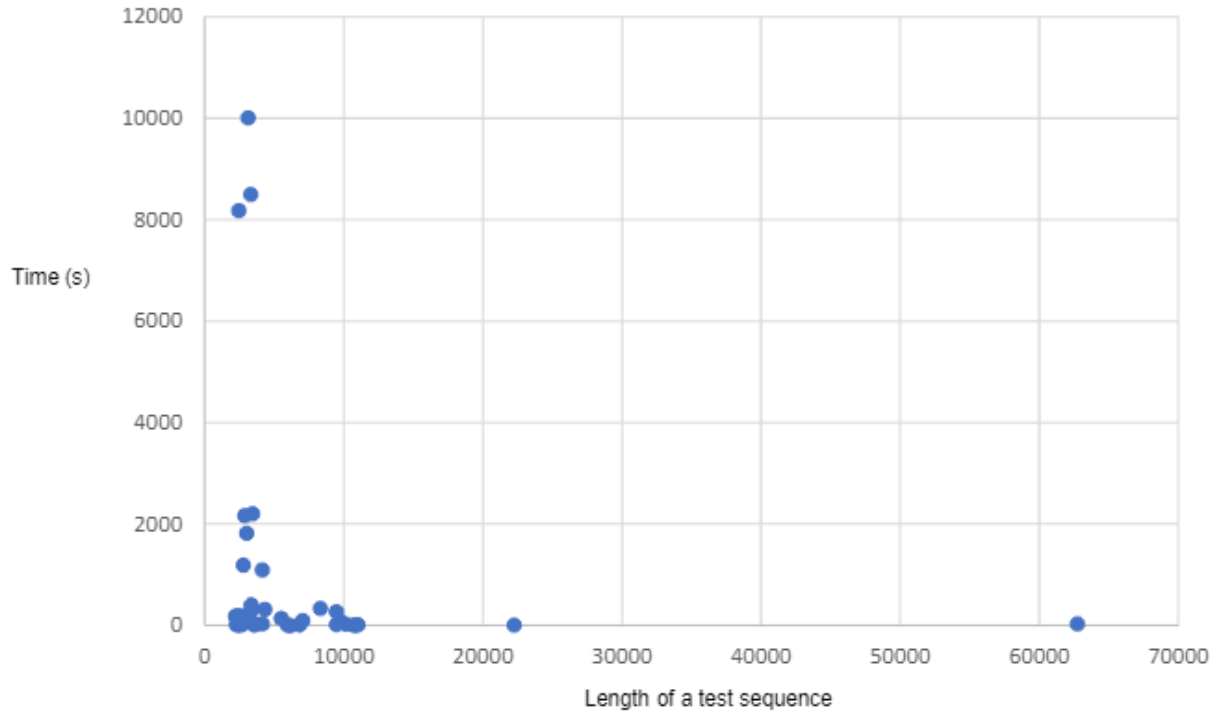
Result of execution time after 500 iterations



No evidence that our approach is always slow

Y-axis: normalized execution time of our approach compared to Artemis
X-axis: benchmarks

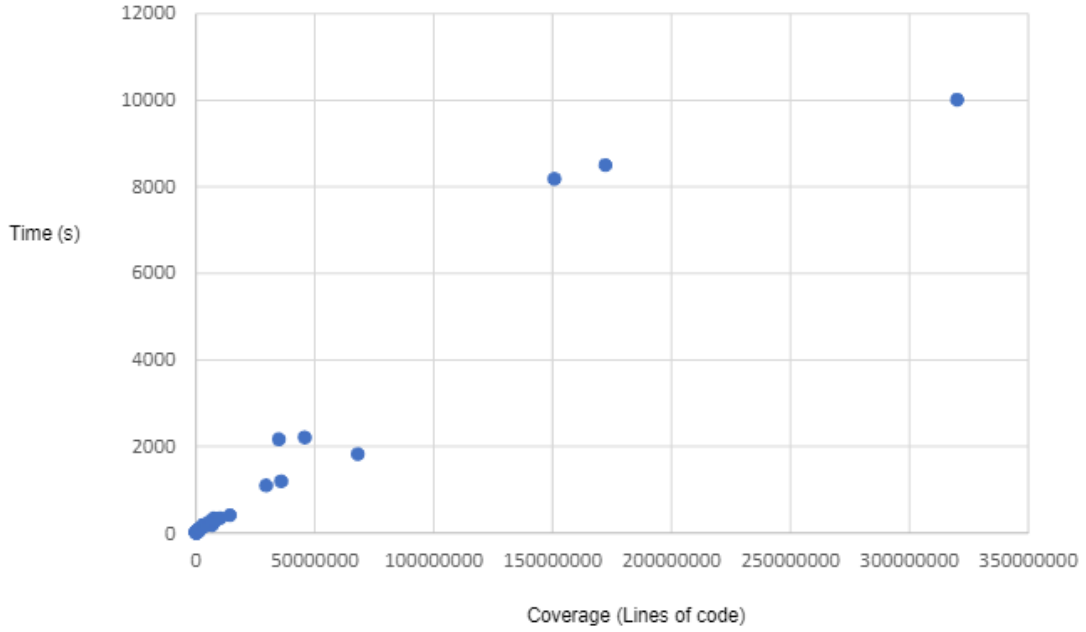
Possible factors for execution time
– length of test sequences



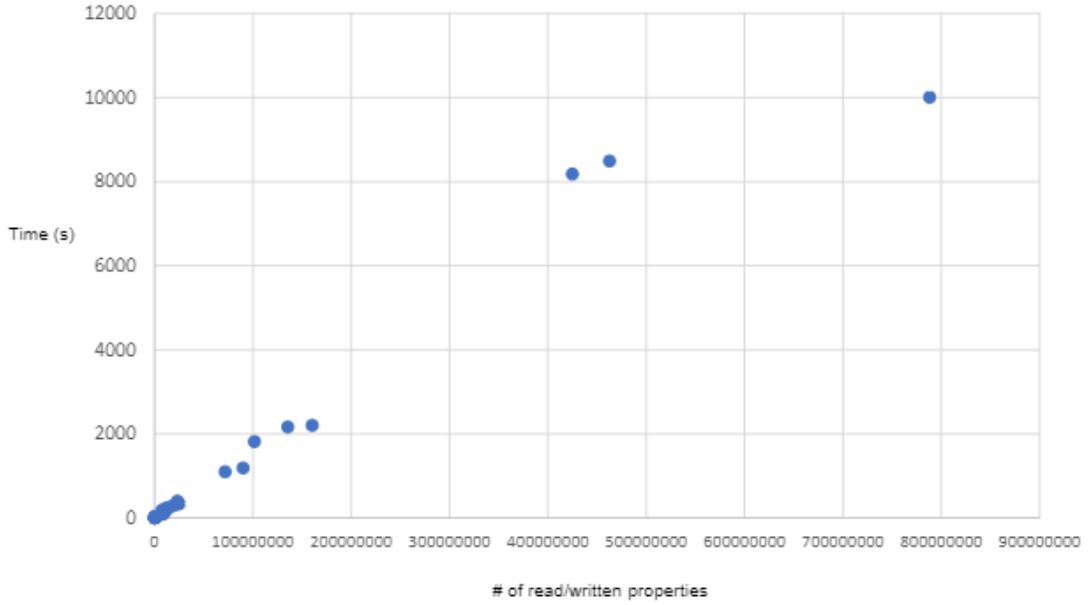
We are not able to conclude that there is a correlation between the length of test sequences and the execution time since the plot does not show any tendency

Y-axis: execution time
X-axis: length of test sequences

Possible factors for execution time
– coverage and read/written properties



(a) The execution time by covered lines of code including overlaps.

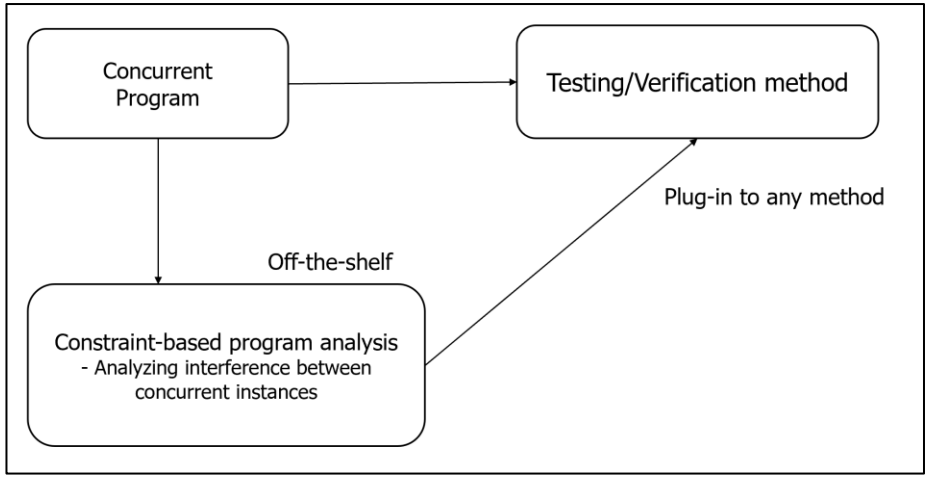


(b) The execution time by the number of read/written properties.

Testing more lines of code by reading and writing more variables affects the testing time
-> testing time can be drastically reduced if we cache some program states corresponding to some of the common test subsequences

Conclusion

- Our constraint-based analysis of interferences between concurrent events significantly improves the performance of concurrent testing/verification techniques
  - Method
    - Customized constraint-based program analysis to analyze interferences between concurrent events
  - Results
    - More accurate verification of interrupt software
    - More scalable semantic diffing of multi-threaded programs
    - More efficient testing of web applications



Thank you!