

Evolution-Aware Monitoring-Oriented Programming (eMOP)

Owolabi Legunsen, **Darko Marinov**, and Grigore Roşu



CCF-1439957
CCF-1012759

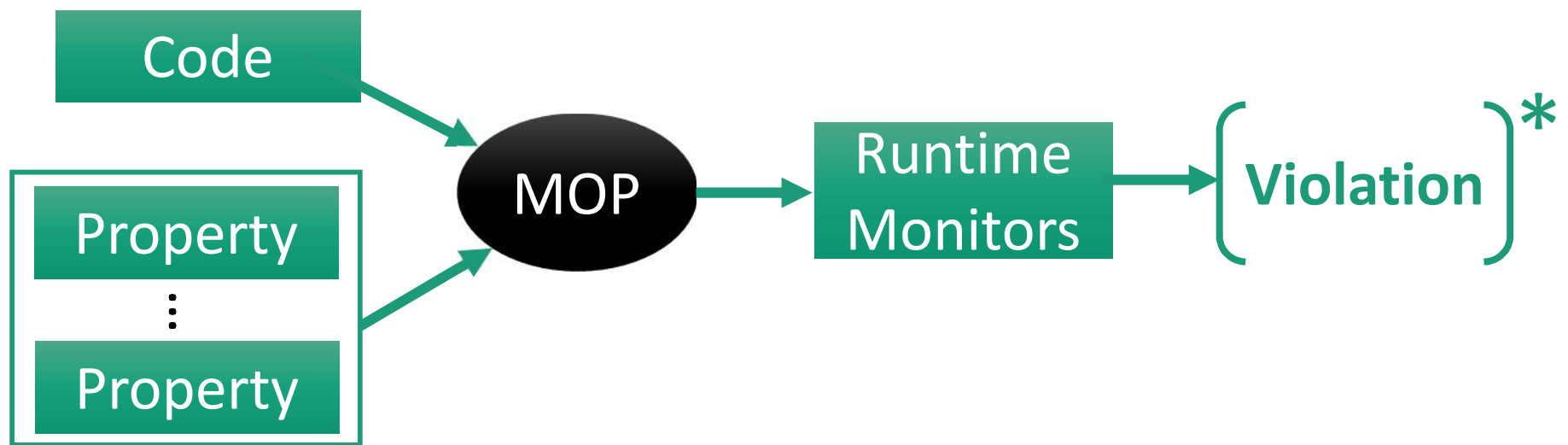
ICSE 2015 (NIER Track)
Florence, Italy
May 21, 2015



Monitoring-Oriented Programming (MOP)

Runtime monitoring of software against formal properties

- **Existing technique** targeted at single program version

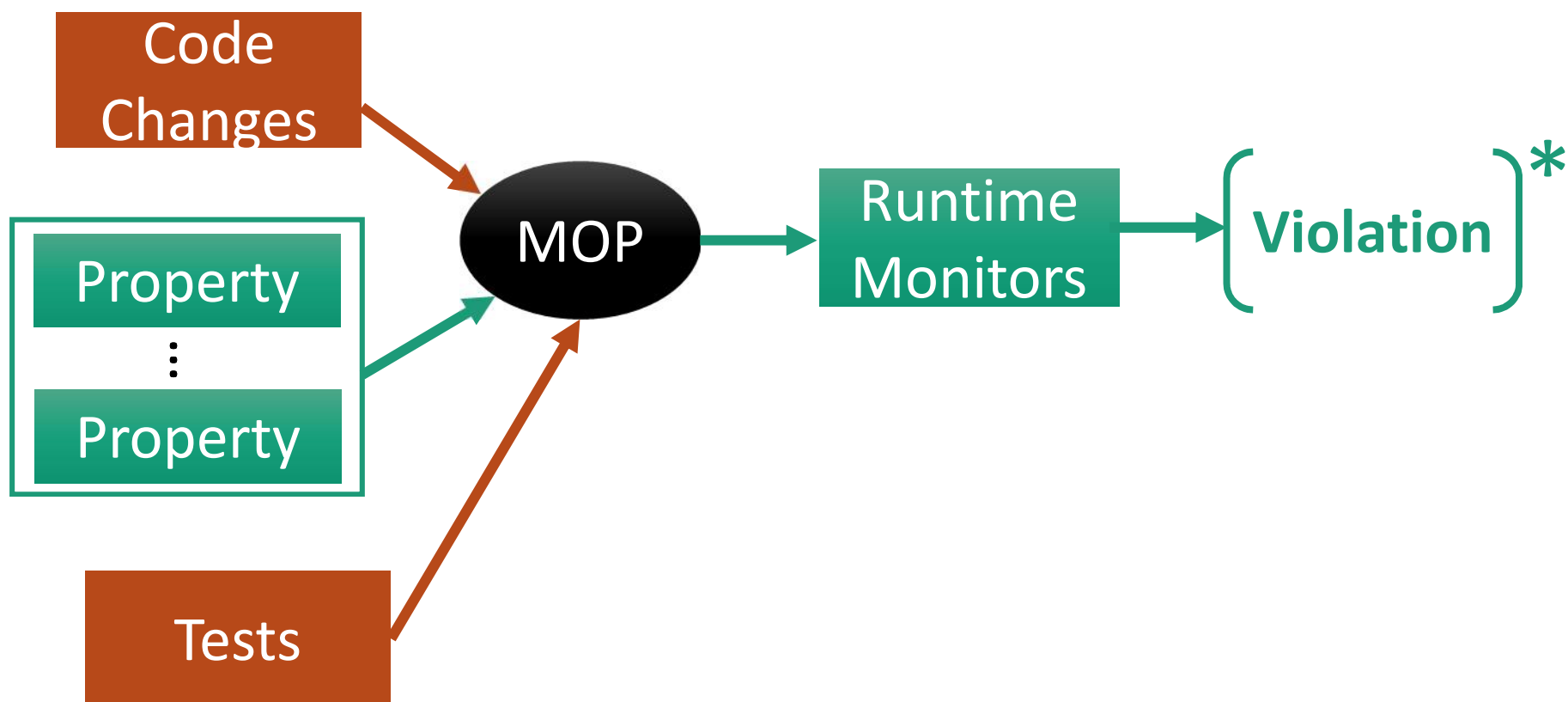


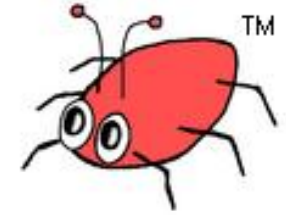
Problems: High overhead and too many violations shown during evolution across many versions

Evolution-Aware MOP (eMOP)

Make MOP **faster** and **show fewer violations** during evolution

- **Proposed**





Input: (Potentially Buggy) Code

```
1 public boolean m(List a, List b) {
2   ...
3   for(Iterator i = a.iterator(); i.hasNext();){
4     ...
5     for(Iterator i2 = b.iterator(); i.hasNext();){
6       ... i2.next() ...
7     }
8   } return ...
9 }
```

Line 5 should be *i2.hasNext()*

Mimics two real bugs found in older AspectJ code

Input: Formally Specified Properties

1. When to fire Events

after `Iterator.hasNext() == true`, before `Iterator.next()`

2. Specification over Events

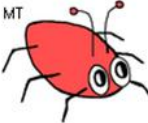
`Iterator.hasNext() == true` precedes every `Iterator.next()`

3. Handler code

User-defined action when specification is violated

Many properties can be monitored at once

Output

```
1 public boolean find(List a, List b) {
2     ...
3     for(Iterator i = a.iterator(); i.hasNext();){
4         ...
5         for(Iterator i2 = b.iterator(); i.hasNext();){
6              // event: "before Iterator.next()"
7             ... i2.next() ...
8         }
9     } return ...
10 }
```

Violation: *next()* was called without calling *hasNext()*

Current State of MOP Research

- Many papers, focus on reducing runtime overhead
- Many bugs found in well-used, well-tested code
- **All prior research focused on one version**
 - Recurring costs of monitoring are high, e.g.,

Run	Properties Monitored	Total Violations	Time(s)
No MOP v1	n/a	n/a	8.4
MOP v1	180	27,895	164.1
MOP v2	180	27,904	231.8

Evolution-Aware MOP (eMOP)

- Improve MOP during software evolution
 - **Faster:** re-monitor based on parts affected by changes
 - **Show fewer violations:** show only violations due to changes
- We propose three techniques
 - Can be used separately or combined
 - **Property selection**
 - **Monitor selection**
 - **Test selection**

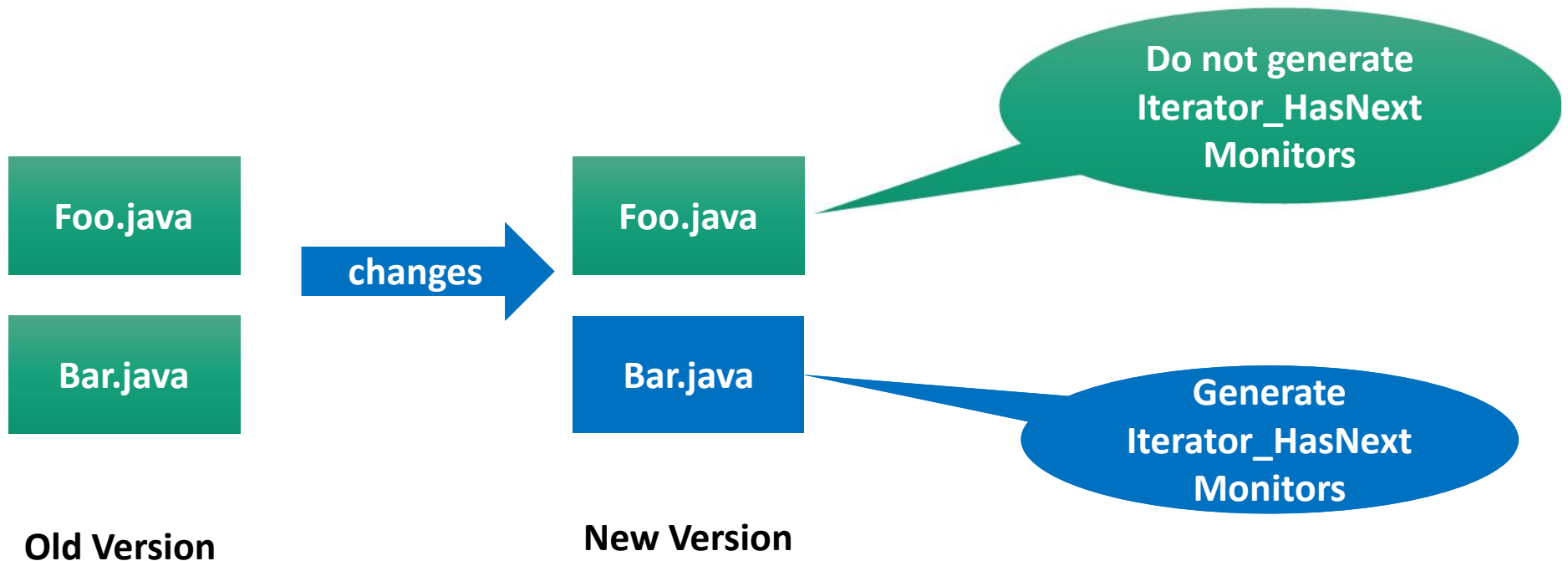
Technique: Property Selection

- What subset of properties to re-monitor in new version?
- Preliminary evaluation by seeding **i2.next()** bug :
 - Only *Iterator_HasNext* is affected by changes

Run	Properties Monitored	Properties Violated	HasNext Violations	Total Violations	Time(s)
No MOP v1	n/a	n/a	n/a	n/a	8.4
MOP v1	180	6	0	27,895	164.1
MOP v2	180	7	9	27,904	231.8
eMOP v2	1	1	9	9	8.8

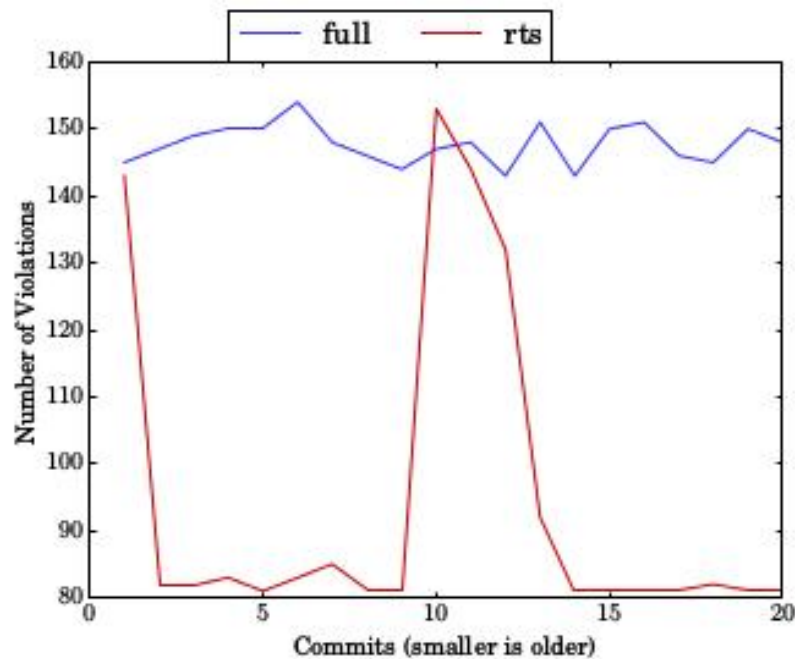
Technique: Monitor Selection

- Generate monitors for parts of code affected by change
- Example: *Foo.java* and *Bar.java* both use Iterator

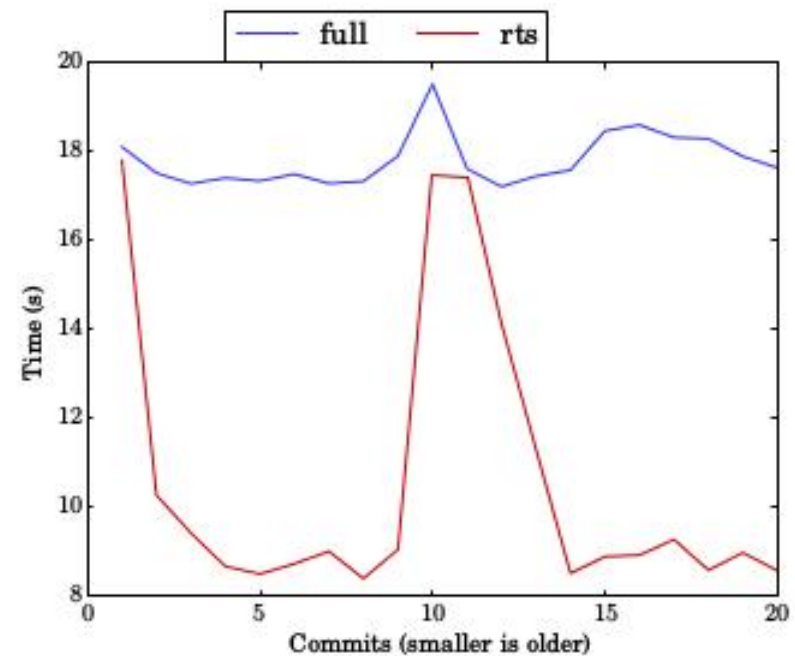


Technique: Test Selection (MOP + RTS)

- In eMOP we monitor execution of tests
 - RTS selects **subset** of tests that can be affected by code changes
 - If fewer tests are run, fewer violations and less overhead



(a) Violation Counts for one project

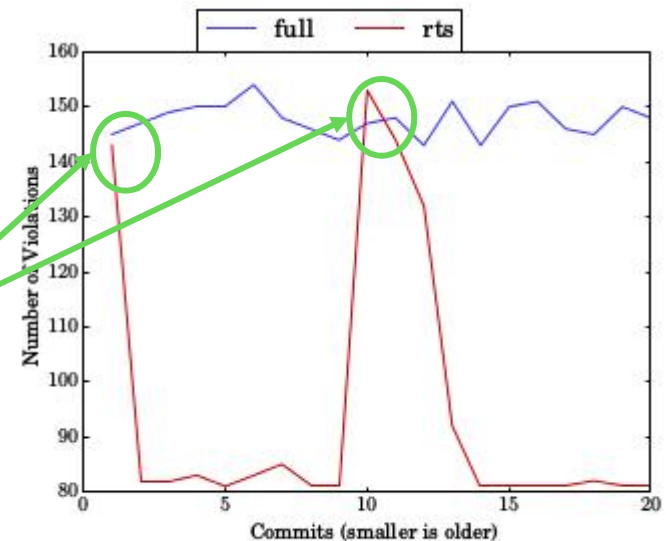


(b) Monitoring Times for one project

Some Challenges

- Safely determining properties/monitors/tests that can't have new violations
- Non-determinism, e.g.,

In these versions, the same tests are run, but different number of violations



(a) Violation Counts for one project

Conclusions

- All prior research on MOP targeted single code versions
- **eMOP** aims to adapt MOP to software evolution
 - Make MOP **faster** between versions of software
 - **Show only violations** due to changes between versions
- We proposed three techniques for eMOP
 - **Property selection**
 - **Monitor selection**
 - **Test selection**