Evolution-Aware Monitoring-Oriented Programming (eMOP)

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

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

CCF-1439957
CCF-1012759

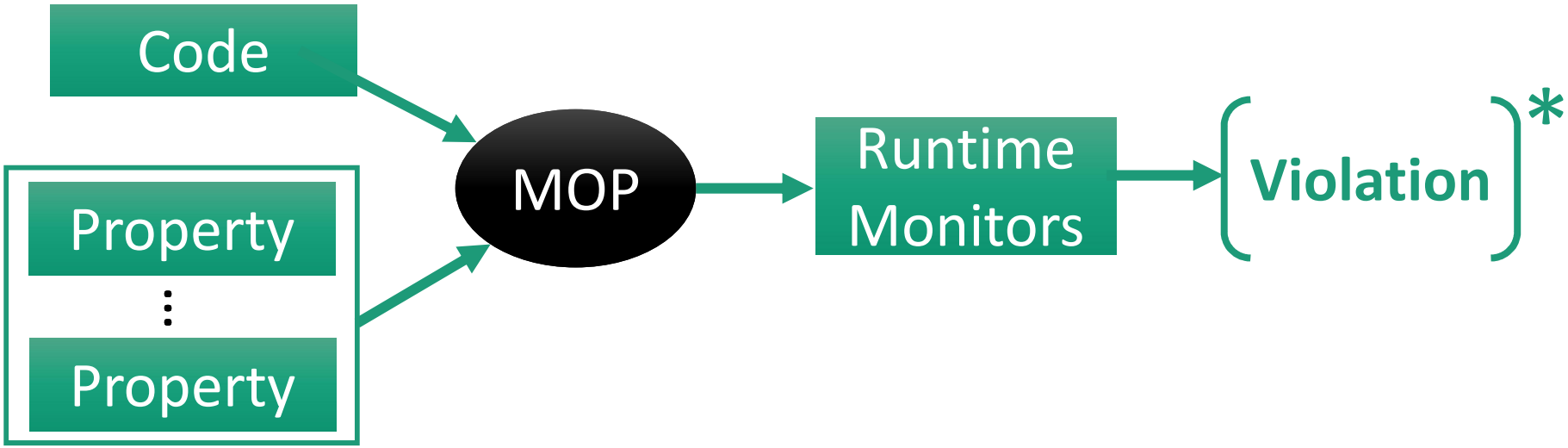
ICSE 2015 (NIER Track)
Florence, Italy
May 21, 2015

ITI RPS #28

Monitoring-Oriented Programming (MOP)

Runtime monitoring of software against formal properties

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

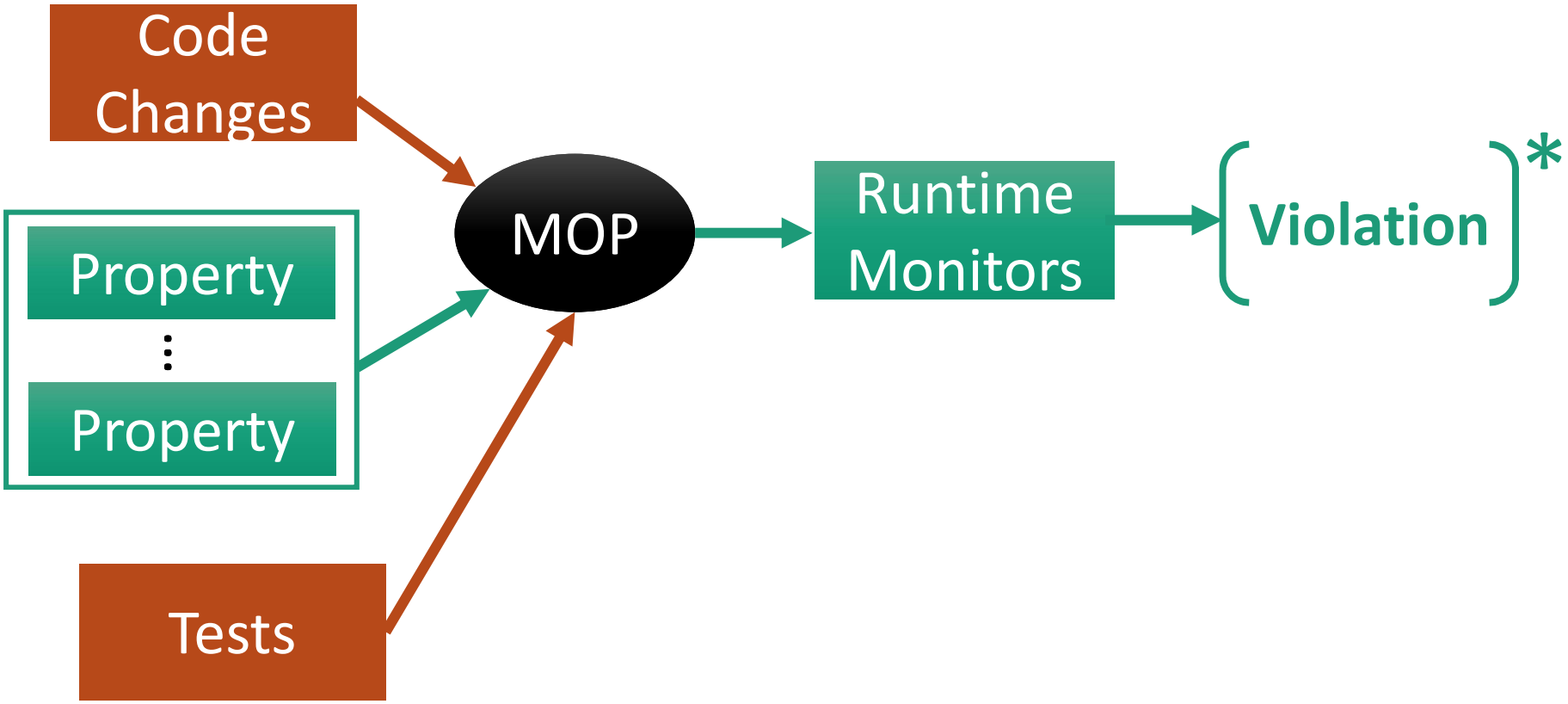


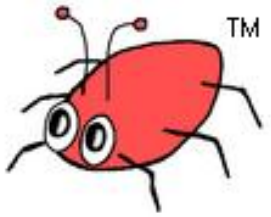
Problems: High overhead and too many violations shown during evolution across many versions

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()**

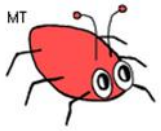
3. Handler code

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();){
        // event: "before Iterator.next()"
6       ... i2.next() ...
7     }
8   } return ...
9 }
```



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**

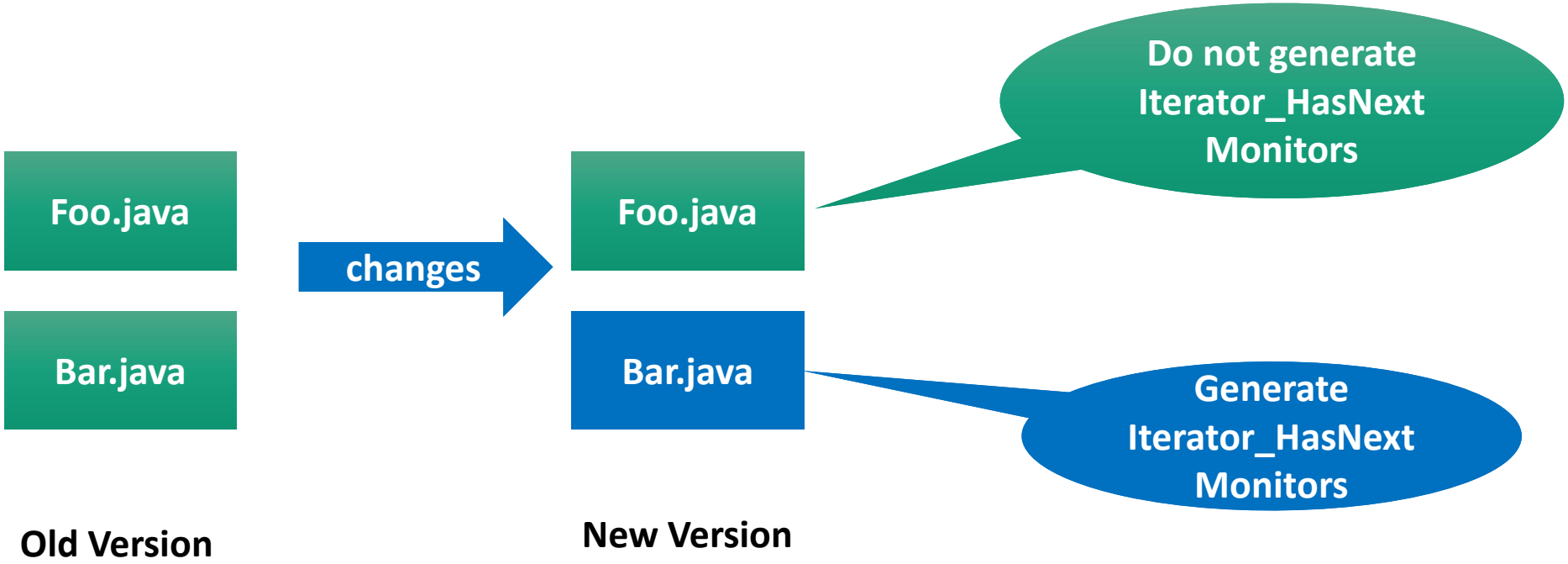
Technique: Property Selection

- 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**

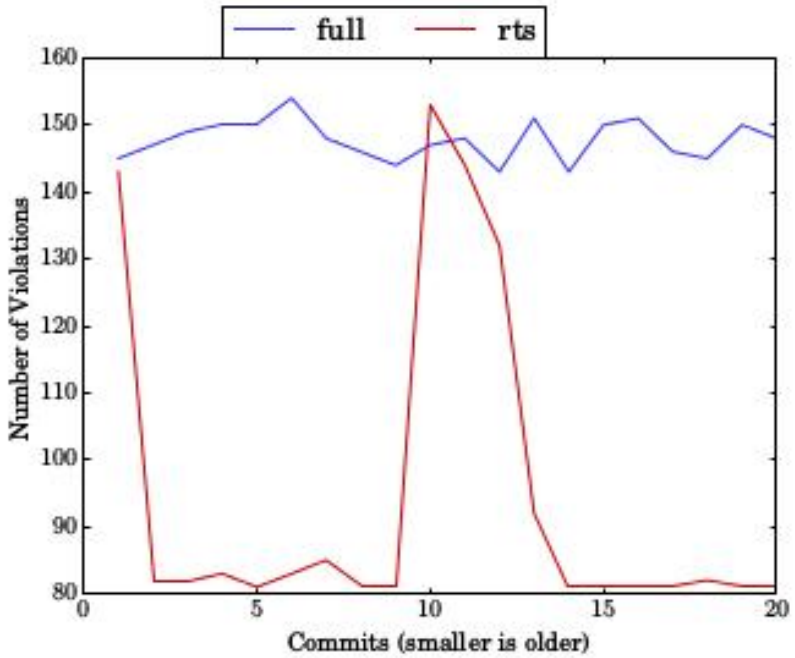
Technique: Monitor Selection

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

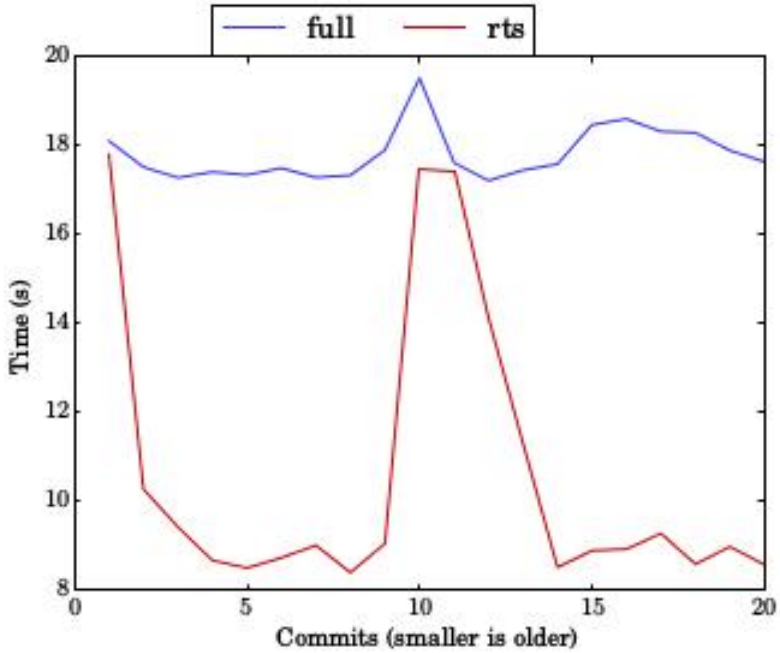


Technique: Test Selection (MOP + RTS)

- 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

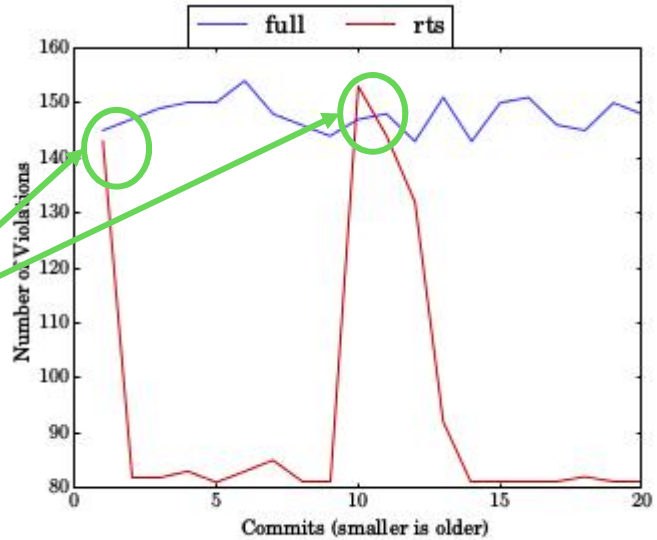


(b) Monitoring Times for one project

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**

Backup Slides

How MOP Works

Monitor

Monitor

Run Time

Compile Time

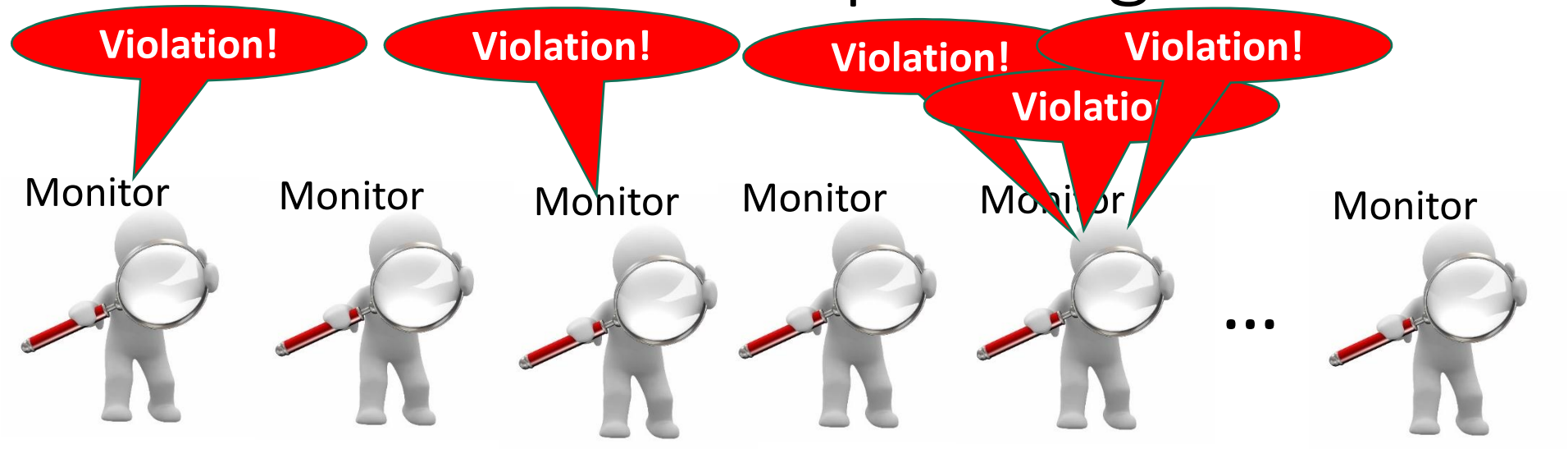
```
1 public boolean findFiles(List files, List dirs){
2   File file, dir; int count = 0;
3   for(Iterator iter = files.iterator(); iter.hasNext();){
4     file = (File) iter.next();
5     for(Iterator iter2=dirs.iterator(); iter.hasNext();){
6       dir = (File) iter2.next();
7       if (new File(dir, file.getName()).exists()){
8           count++; break;} //file is in dir
9     }}  return count == files.size(); }
```

+

```
1 Iterator_HasNext(Iterator i) {
2   event hasnexttrue after(Iterator i) returning(boolean b):
3     call(*Iterator+.hasNext())&&target(i)&&condition(b){}
4   event next before(Iterator i):
5     call(*Iterator+.next()) && target(i){}
6   ltl: [](next => (*) hasnexttrue)
7   @violation {...}
8 }
```

Code **Property**

MOP Costs can be quite high



Run Time

Compile Time

```
1 public boolean findFiles(List files, List dirs){
2   File file, dir; int count = 0;
3   for(Iterator iter = files.iterator(); iter.hasNext();){
4     file = (File) iter.next();
5     for(Iterator iter2=dirs.iterator(); iter.hasNext();){
6       dir = (File) iter2.next();
7       if (new File(dir, file.getName()).exists()){
8           count++; break;} //file is in dir
9     }}  return count == files.size(); }
```

+

```
1 Iterator_HasNext(Iterator i) {
2   event hasnexttrue after(Iterator i) returning(boolean b):
3     call(*Iterator+.hasNext())&&target(i)&&condition(b){}
4   event next before(Iterator i):
5     call(*Iterator+.next()) && target(i){}
6   ltl: [](next => (*) hasnexttrue)
7   @violation {...}
8 }
```

Code **Properties**

No.	Project	LOC	Tests	Violations	BaseTime(s)	MOPTime(s)	Overhead(%)
1	FXForm2	5355	34	1008	5.2	20.9	304.1
2	JAQ-InABox	3570	1	1	5.1	5.5	7.4
3	JSqlParser	7786	232	27895	8.9	342.2	3733.2
4	ObjectLayout	1305	19	0	9.0	22.4	148.8
5	androlog	2532	8	19	4.0	5.6	41.1
6	apache.commons-lang	63425	2497	0	23.8	29.9	25.4
7	asterisk-java	35659	215	145	10.5	17.5	65.9
8	bcel	35827	73	27	7.2	12.2	70.9
9	commons-beans	33007	1269	0	43.4	962.6	2120.1
10	commons-cli	6292	364	0	4.7	8.2	73.8
11	commons-codec	16160	616	0	9.7	11.7	20.7
12	commons-collections4	52040	13702	0	25.1	32.5	29.7
13	commons-discovery	2588	14	0	4.6	7.5	62.5
14	commons-fileupload	4316	71	0	5.7	9.9	73.8
15	commons-imaging	36657	93	0	30.0	38.9	29.6
16	commons-lang3	63425	2497	0	23.1	30.8	33.2
17	commons-validator	11982	416	0	7.3	11.0	51.0
18	compile-testing	1813	22	149	3.9	19.0	385.2
19	compress	28931	466	0	9.4	16.1	70.2
20	connector4java	1928	36	470	4.1	22.8	458.1
21	dbcp	18759	480	28	67.8	73.2	8.0
22	dropwizard-todo	796	13	5674	8.9	79.9	797.7
23	functor	21688	1134	0	10.7	18.9	76.4
24	hivemall	5360	24	224	4.7	9.4	100.1
25	htrace	1531	9	68	16.0	23.4	45.8
26	invokebinder	2818	97	12	3.9	6.1	57.2
27	jblas	12570	120	0	5.3	10.3	94.2
28	jline	3419	22	5707	3.5	62.9	1674.5
29	joda-time	82998	4057	3504	13.9	53.2	282.7
30	jpatterns	2604	33	4	3.8	7.4	94.7
31	jsoup	13556	413	48385	6.1	366.3	5891.9
32	junit	25916	867	0	12.9	29.1	125.3
33	laforge49	7245	56	185357	4.5	2133.0	47800.9
34	logback-encoder	637	18	1812	4.2	20.1	371.9
35	math	186796	5943	0	120.9	123.6	2.2
36	ogrisel.pignlproc	2296	19	4200	64.7	92.1	42.3
37	paper2ebook	142	2	9329	2.9	136.7	4623.2
38	scribe-java	5344	99	24	13.0	16.5	27.2
39	zookeeper-utils	455	4	1297	8.5	18.3	116.1
40	zxing	42493	373	457094	33.8	4399.0	12904.0
	Total	852021	36428	752433	654.8	9306.6	82941.3
	Arith. Mean	**21300.5**	**910.7**	**18810.8**	**16.4**	**232.7**	**2073.5**

Figure 1: Time overhead of monitoring each project