Fine-Grained Analyses for Evolution-Aware Runtime Verification

Pengyue Jiang
pj257@cornell.edu
Cornell University
Ithaca, New York, USA

Kevin Guan
kzg5@cornell.edu
Cornell University
Ithaca, New York, USA

Mahdi Khosravi
mahdi.khosravi@metu.edu.tr
Middle East Technical University
Ankara, Turkey

Moustafa Ismail
moustafa.ismail@metu.edu.tr
Middle East Technical University
Ankara, Turkey

Marcelo d'Amorim
mdamori@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Owolabi Legunsen
legunsen@cornell.edu
Cornell University
Ithaca, New York, USA

Abstract

Runtime verification (RV) found many bugs by monitoring passing tests in many open-source projects against formal specifications (specs). But, RV is often too slow for use in continuous integration. So, evolution-aware techniques were proposed to speed up RV by re-monitoring only a subset of specs affected by code changes. These techniques use coarse-grained class-level analyses, so they can sub-optimally and imprecisely re-monitor unaffected specs.

We propose FineMOP to speed up evolution-aware RV by using fine-grained analyses to re-monitor fewer unaffected specs. The key idea is simple: changes often do not require re-monitoring specs that are only related to unchanged parts of changed classes. We implement six variants of three fine-grained analyses in FineMOP and evaluate them on 1,104 revisions of 68 open-source Java projects. Compared with two class-level techniques, FineMOP is up to 4.86x faster, re-monitors up to 81.04% fewer specs per revision, and finds 99.68% of all new violations that these techniques find. Also, FineMOP and Regression Test Selection (RTS) are complementary: combining FineMOP with RTS is faster than FineMOP or RTS alone.

ACM Reference Format:
Pengyue Jiang, Kevin Guan, Mahdi Khosravi, Moustafa Ismail, Marcelo d'Amorim, and Owolabi Legunsen. 2026. Fine-Grained Analyses for Evolution-Aware Runtime Verification. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787833

1 Introduction

Runtime verification (RV) [45, 60, 95] checks running programs against formal specifications (specs). RV first uses instrumentation to re-write a program to signal spec-related events (*e.g.*, method calls) at runtime. Then, while running the instrumented program, traces—*i.e.*, sequences of events—are checked using monitors that are dynamically synthesized from specs. If a trace violates a spec, a monitor performs a user-specified action, *e.g.*, raising a warning.

RV of *passing* unit tests against specs of correct JDK API usage now scales to thousands of open-source projects [38, 53, 68, 70, 85] and helped find hundreds of bugs that testing alone missed. But, RV is often too slow for use in continuous integration (CI) [38]. So, evolution-aware RV techniques were proposed to speed up RV by re-monitoring only a subset of *affected specs* for which events can be signaled from code impacted by changes [71, 73, 112].

Evolution-aware RV addresses two main challenges. First, time to find and re-monitor affected specs must be less than time to simply re-monitor all specs across many revisions. Occasional slowdown is tolerable if most revisions see a speedup. Second, algorithms should be *safe*—defined as finding all new violations after a change [73].

All prior evolution-aware RV techniques use coarse-grained class-level analyses to find specs affected by changes [39, 73, 112]. So, these techniques can sub-optimally and imprecisely re-monitor unaffected specs. Suppose class `A` has two independent methods `m1` and `m2`, spec `S1` has events only in `m1` and spec `S2` has events only in `m2`. If only `m1` changes, current evolution-aware RV techniques correctly re-monitor `S1`, but they also imprecisely re-monitor `S2`. Finer-grained method-level analysis of this change will re-monitor only `S1`, thereby speeding up evolution-aware RV.

We propose FineMOP to speed up evolution-aware RV by using fine-grained analyses to re-monitor fewer unaffected specs. Fine-grained analyses can be unsound or slower than class-level ones [69]. But, two insights from recent fine-grained analyses for regression test selection (RTS) [78, 115, 117] inspire FineMOP: (i) extra time for fine-grained analysis is often offset by gains in lower end-to-end time across revisions; and (ii) unsoundness of fine-grained analysis is often mitigated by falling back to the class-level.

FineMOP embodies three fine-grained analyses: ① Mthd reasons about method-level changes; ② Hybrid also reasons about method-level changes, but falls back to the class level if doing so can be unsound; and ③ Fine uses field, method, and class-level reasoning to ignore a manually identified set of semantics-modifying changes that do not affect RV of specs in this paper, *e.g.*, only changing a method's access modifier. These three analyses still project their results to the class level: all specs with events in the class containing an impacted field or method are re-monitored. But, these three analyses can still re-monitor fewer unaffected specs than pure class-level analysis: they do not re-monitor specs that only have events in classes that do not transitively use any changed method.

FineMOP also embodies three variants of two of these analyses: ④ MthdNoP (a variant of Mthd) does not project method-level analysis results to the class level. Rather, it only re-monitors specs with events in methods impacted by changes. ⑤ HybridNoP (variant of Hybrid) also does not project the results of its analysis to the class level when that analysis itself does not fall back to the

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787833

class level. ⑥ Unlike the other five variants mentioned so far, which re-monitor entire classes with affected specs, MtHdFM (variant of MtHdNoP) only re-monitors affected specs in impacted methods.

Mthd, Hybrid, and Fine are inspired by RTS algorithms, and we are the first to adapt them for RV. MtHdNoP, HybridNoP, and MtHdFM are new in this paper. (§3 illustrates all six analyses.) More specifically, this paper goes beyond prior RTS work on using fine-grained analyses and semantics of changes in four ways: (i) FineMOP additionally computes and applies finer-grained mappings from programs to specs, while trying to satisfy design goals and constraints that RTS does not have (as discussed in §3.1); (ii) we design and apply a finer-grained monitoring algorithm in MtHdFM that is not applicable to RTS; (iii) FineMOP combines (i) and (ii) with reasoning about changes at different granularities; and (iv) we evaluate different analysis granularities for RV, not testing.

We compare all six analyses in FineMOP with ps_1^c and $ps_3^{c\ell}$, two class-level evolution-aware RV techniques proposed in prior work [73, 112]. Similar to FineMOP, ps_1^c and $ps_3^{c\ell}$ aim to speed up RV by re-monitoring only a subset of specs affected by code changes, but they use class-level analysis. ps_1^c is designed to be safe, so it uses an often slow conservative static analysis to find affected specs. On the other hand, $ps_3^{c\ell}$ is designed to trade safety for speed, so it uses a faster but less conservative analysis. §2 explains ps_1^c and $ps_3^{c\ell}$.

On 1,104 revisions of 68 GitHub projects, FineMOP is up to 4.86x or 45.02 minutes (avg: 1.7x or 5.68 minutes) faster, re-monitors up to 81.04% (avg: 33%) fewer specs than ps_1^c and finds 99.68% of new violations found by ps_1^c. Compared with $ps_3^{c\ell}$, which is quite fast, FineMOP is up to 4.47x or 41.85 minutes (avg: 1.18x or 1.14 minutes) faster, re-monitors up to 80.3% (avg: 29.91%) fewer specs, and finds 99.89% of new violations found by $ps_3^{c\ell}$. The few safety losses are due to unsoundness of method-level analysis. Summed across all projects, FineMOP is 6.44 hours faster than ps_1^c and 1.29 hours faster than $ps_3^{c\ell}$. So, fine-grained analysis is promising for speeding up evolution-aware RV. Future work must tackle the small safety loss.

We compare FineMOP, which re-runs all tests when re-monitoring a subset of specs, with combining RTS with RV to re-run a subset of tests while re-monitoring all specs. To do so, we compare all six FineMOP analyses against combining four RTS techniques with evolution-*unaware* RV, ps_1^c, and $ps_3^{c\ell}$: Ekstazi [33, 34] and STARTS [69, 72], which use class-level dynamic and static change-impact analyses respectively, and FineEkstazi and FineSTARTS [78], which are like Ekstazi and STARTS, but skip tests that only depend on classes where semantics-modifying changes cannot alter test outcome. FineMOP alone is 12.3x faster than RTS alone. Lastly, we combine all six FineMOP analyses with all four RTS techniques, to re-run subsets of tests while re-monitoring subsets of specs. FineMOP and RTS are complementary: combining them is 2.0 hours faster than FineMOP alone and 2.9 hours faster than RTS plus RV.

This paper makes the following contributions:

- ⋆ **Analyses.** FineMOP embodies the first six fine-grained analyses and their algorithms to further speed up evolution-aware RV.
- ⋆ **Comparison.** We compare FineMOP with four RTS techniques.
- ⋆ **Combination.** We combine all of FineMOP's analyses with RTS.
- ⋆ **Tool.** We implement all six FineMOP analyses and their integration with four RTS techniques in a Maven Plugin.

FineMOP's artifacts are at https://github.com/SoftEngResearch/FineMOP.

```
1 StringTokenizer_HasMoreElements(StringTokenizer i) {
2  event hasnexttrue after(StringTokenizer st) returning (boolean b):
3   (call(boolean StringTokenizer.hasMoreTokens()) || call(boolean
        StringTokenizer.hasMoreElements())) && target(st) && condition(b){}
4  event next before(StringTokenizer st):
5   (call(* StringTokenizer.nextToken()) || call(* StringTokenizer.nextElement())
        ) && target(st){}
6  ltl: [](next => (*) hasnexttrue)
7  @violation {/*print violation*/} }
```

Figure 1: The STHM MOP spec [94].

```
1 class E {
2   public static int m1(String s) {
3     StringTokenizer st = new StringTokenizer(s);
4     int l = st.nextToken().length(); /*INSTR: STHM.next*/
5     while (st.hasMoreTokens()) /*INSTR: STHM.hasnexttrue*/
6       l +=  st.nextToken().length(); /*INSTR: STHM.next*/
7     return l; }}
8 public class ETest {
9 @Test public void testM1(){assertEquals(3, E.m1("ab c"));}}
```

Figure 2: Example code and test.

2 Background and Running Example

We illustrate specs of correct JDK API usage, the RV style we use in this paper (*i.e.*, Monitoring-Oriented Programming, or MOP [16, 17, 19, 20]), and how state-of-the-art (SoTA) class-level evolution-aware RV works. §3 uses this running example to illustrate FineMOP.

A spec, and how MOP works. Figure 1 shows the MOP spec StringTokenizer_HasMoreElements (STHM); it has three parts. (i) Lines 2–5 define *events* `next` and `hasnexttrue`, which RV instruments in monitored programs (*e.g.*, *after* `hasMoreTokens()` calls on `StringTokenizer st` and *before* calls to `st.nextToken()`). (ii) Line 6 is a linear temporal logic (LTL) *safety property*: □(next ⇒ ⊙hasnexttrue)—"always ([]), a `next` event on `st` implies (=>) that the immediately preceding ((*)) event on `st` was `hasnexttrue`". MOP allows mixing past-time and future-time LTL operators, and using other formalisms, *e.g.*, finite state machines (FSM) [46], extended regular expressions (ERE) [96], context free grammars (CFG) [83], string rewrite systems (SRS) [82], etc. (iii) Line 7 is a handler that is triggered if LTL property is violated. Handlers can be any user-defined code, but for testing we print a message.

The toy example in Figure 2 shows how RV amplifies the bug-finding ability of unit tests. Method `m1` (lines 2–7) takes a string and returns its length after removing delimiters. Test `testM1` always passes, missing a subtle bug: `m1` crashes on empty strings because line 4 calls `st.nextToken()` without first checking if the tokenizer has more tokens. RV of `testM1` detects this bug. At line 4, when `nextToken` is called on `st`, RV signals a `next` event that triggers the violation handler: the `next` event is not immediately preceded by `hasnexttrue` on `st`. STHM helped find several confirmed bugs [68].

SoTA class-level evolution-aware RV. Techniques in this paper aim to speed up RV during CI by only re-monitoring specs affected by code changes. SoTA evolution-aware RV techniques first use *class-level* analysis to find classes impacted by changes. Then, these techniques find affected specs as those with events in impacted classes. To see how SoTA techniques work, consider the old and new revisions of hypothetical code in Figure 3. There, squares `A`, `B`, `C`, `D`, `E`, and `F` are classes, rectangles `m1` and `m2` are methods, and rhombi `S1` to `S6` are specs. Solid arrows show inheritance or use relationships, and dashed arrows show where events for each spec come from. These inter-class and class-to-spec relationships are found via static analysis. Colors show changes: (i) Green: semantics-modifying changes that cannot lead to new violations of our specs,

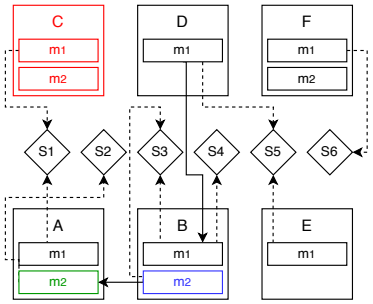

Figure 3: Example.

e.g., making a `private` method `public`. (We use 13 such changes from [78].) (ii) Red: new code. (iii) Blue: changes that can lead to new violations. §3.2 discusses deletions.

Given the changes in Figure 3, SoTA ps_1^c and $ps_3^{c\ell}$ in eMOP [73, 112] re-monitor specs S1–S5 in the new revision: S1–S4 have events in changed classes and S5 has events in D, which depends on changed class B. Spec S6 is not re-monitored: F, the only class with S6 events, is not impacted by changes. If finding affected specs is faster than re-monitoring S6, RV will be faster in the new revision. ps_1^c is slow but safe (within static analysis limitations), and uses a more conservative analysis that aims to find all affected specs. But, $ps_3^{c\ell}$ is unsafe by design; it uses a less conservative analysis that may miss affected specs.

Finer-grained analyses can provide more speedup by re-monitoring fewer unaffected specs. In Figure 3, our fine-grained analyses re-monitor only specs S1–S3; S4 and S5 cannot signal new events after the changes.

3 FineMOP

3.1 Design Considerations

Design goals. Evolution-aware RV has three conflicting goals. (i) *Efficiency*: analysis time plus time to re-monitor affected specs should be faster than re-monitoring all specs. (ii) *Safety*: finding all new violations after a change; and (iii) *Precision*: re-monitoring only affected specs. One conflict is that safe and precise analyses are slow. SoTA evolution-aware RV techniques address this conflict by using fast class-level static analyses to over-approximate the impact of changes, but they imprecisely find more unaffected specs. Note that dynamic analyses would require running tests twice—once to find inter-class and class-to-spec relationships and once to re-monitor affected specs—and incur an overhead of at least 2x.

Other trade-offs induced by finer-grained analyses.

1. Cost and safety of finer granularity levels. In Java, there are only three finer analysis granularity levels than classes: methods, statements, and fields. Analyses at these finer levels can be more precise, helping goal (iii). But, finer-grained analyses also tend to be more costly [34, 69], hurting goal (i), because dependency graphs can be substantially larger at finer granularity levels. For example, projects often have many more methods than classes. Also, finer-grained analyses can be more unsafe [34, 69], hurting goal (ii).

2. Reasoning about semantics of changes. Not all changes require re-monitoring specs that would otherwise be affected, *e.g.*, only changing a method's visibility or variable name does not require re-monitoring a JDK API spec. Ignoring such changes can be more precise, helping goal (iii), but incurs a cost, hurting goal (i).

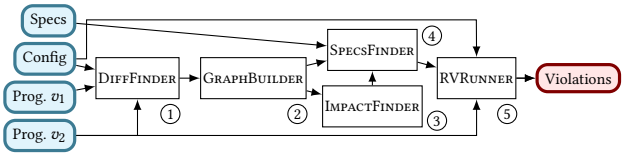

Figure 4: Workflow of evolution-aware RV in this paper.

3. Number of class-level analyses. There are **12** class-level analyses for evolution-aware RV [73] combining the following three factors: (i) the degree of conservativeness of change-impact analysis—1 (most conservative), 2, and 3 (least conservative); (ii) whether to instrument affected specs in 3rd-party libraries; and (iii) whether to instrument affected specs in classes not impacted by changes. So, the "1" subscript in ps_1^c means it uses the most conservative analysis and the "c" superscript means it does not instrument affected specs in non-impacted classes. Also, the "3" subscript in $ps_3^{c\ell}$ means it uses the least conservative analysis and the "ℓ" superscript means it does not instrument affected specs in libraries. The challenge is to speed up all class-level analyses without making them less safe.

Design justification. FineMOP's six analyses—Fine, Hybrid, HybridNoP, Mthd, MthdNoP, and MthdFM—reason about methods or fields. We do not yet explore statement-level reasoning because all statements are in methods or field declarations. To compensate for potential safety loss due to finer-grained analyses, three of FineMOP's six analyses "project" reasoning results onto classes. FineMOP integrates all 12 class-level analyses from prior work [73]. But, 72 combinations (six FineMOP analyses times twelve class-level analyses) are hard to evaluate at scale and present. So, we only evaluate FineMOP's analyses with ps_1^c—the fastest *safe*-by-design class-level analysis—and $ps_3^{c\ell}$—the fastest *unsafe*-by-design one. The rationale is that if FineMOP's analyses can safely speed up ps_1^c and $ps_3^{c\ell}$, our results are likely to generalize to the other slower safe- or unsafe-by-design class-level analyses. §4.2 makes recommendations on how to choose among FineMOP's analyses.

3.2 Overview

Figure 4 shows the high-level workflow of evolution-aware RV.

1. Compute changes. DiffFinder finds program elements that changed. It computes checksums of elements in the new revision, compares them with those in the old revision, and returns new, changed, deleted, or unchanged elements. DiffFinder cleans debug-related information in bytecode before computing checksums, so elements where, *e.g.*, only line numbers or white space were modified are ignored. Deleted elements have no checksums in the new revision, but do not require special handling: only elements present in the new revision are used in subsequent steps. Rationale: if events for spec s were signaled from element e in the old revision, deleting e means that there can be no new s events (or violations) from e in the new revision. If deleting e required modifying another non-deleted element e', that modification is handled like any other change. Moved elements are treated as deleted from the old location *and* added to the new. DiffFinder–used in Hybrid, HybridNoP, Mthd, MthdNoP, and MthdFM–does not handle refactoring, which is not precise. But, Fine performs additional reasoning over program changes, and handles some refactoring. Future work can use the checksum algorithm in refactoring-aware Reks [109] to handle more refactorings.

Algorithm 1 Generic evolution-aware RV algorithm for Figure 4

Inputs: $\mathcal{S}$: set of specs, $\mathcal{E}$: program elements in new revision,
$\mathcal{M}$: metadata, a (`elemToChecksum`, dg, `elemToSpecs`) triple where `elemToChecksum` = $\{e \rightarrow$ `checksum`$(e) \mid e \in \mathcal{E}\}$, dg = dependency graph, and `elemToSpecs` = $\{e \rightarrow S \mid e \in \mathcal{E} \wedge S \subseteq \mathcal{S}\}$,
v: a triple of binary values (`opt`, `excludeLib`, `excludeNonImpacted`)
Output: a set of violations

1: **procedure** `remonitor`($\mathcal{S}$, $\mathcal{E}$, $\mathcal{M}$, v):
2: $E_{\text{new}}, E_{\text{changed}}, E_{\text{all}} \leftarrow$ `getDiff` ($\mathcal{E}$, $\mathcal{M}$) ▷ Algo 2
3: $\mathcal{M}$.dg $\leftarrow$ `buildDependencyGraph` ($\mathcal{E}$)
4: $E_{\text{impacted}} \leftarrow$ `findImpacted`($E_{\text{new}} \cup E_{\text{changed}}$, $\mathcal{M}$.dg, v.opt) ▷ Algo 3
5: $S_{\text{affected}} \leftarrow$ computeAffectedSpecs(E_{impacted}, $\mathcal{M}$) ▷ Algo 4
6: $E_{\text{instr}} \leftarrow E_{\text{impacted}}$
7: **if** !v.excludeLib **then** $E_{\text{instr}} \leftarrow E_{\text{instr}} \cup$ getLibElements()
8: **if** !v.excludeNonAffected **then** $E_{\text{instr}} \leftarrow E_{\text{instr}} \cup (E_{\text{all}} \setminus E_{\text{impacted}})$
9: **return** monitor(S_{affected}, E_{instr})

In SoTA ps_1^c and $ps_3^{c\ell}$ (§2, §3.1), DiffFinder compares checksums for classes, and outputs classes. In Fine, DiffFinder compares checksums for fields, methods, and classes, but returns classes after reasoning about the semantics of the change. In Hybrid and HybridNoP, DiffFinder compares checksums for classes and methods; Hybrid returns classes, HybridNoP returns methods. Finally, in Mthd, MthdNoP, and MthdFM, DiffFinder compares checksums for methods and returns methods.

2. Build dependency graphs. GraphBuilder creates dependency graphs with edge from node `X` to `Y` if `X` depends on `Y`. In ps_1^c and $ps_3^{c\ell}$, GraphBuilder creates class-level graphs. In Mthd, MthdNoP, and MthdFM, GraphBuilder creates method-level graphs with edges from callers to callees. In Hybrid and HybridNoP, GraphBuilder creates method- and class-level graphs. In Fine, GraphBuilder creates a class-level dependency graph. Class-level graphs have an edge from class `X` to class `Y` if `X` uses or inherits from `Y`.

3. Find impacted program elements. ImpactFinder returns a change impact set consisting of program elements whose nodes reach nodes for changed or new elements in the reflexive and transitive closure of the dependency graph. In Hybrid, the impacted elements are computed using the method- and class-level graphs. Other analyses use the graph from GraphBuilder.

4. Find affected specs. SpecsFinder returns, as affected, the subset of specs whose events are generated in impacted elements. To do so, SpecsFinder maps each element e to specs instrumented into e in the new revision. SoTA ps_1^c and $ps_3^{c\ell}$, and Mthd, Hybrid, Fine, map classes to specs. Other analyses map methods to specs.

5. Configure and run RV. Based on Config, FineMOP configures RV and monitors affected specs in the new revision. All options instrument affected specs in impacted elements. Some options cause $X \in$ {Mthd, Hybrid, Fine, MthdNoP, HybridNoP, MthdFM} to *not* instrument (i) non-impacted elements in the code under test (CUT), denoted X^c; (ii) all elements in libraries, denoted X^ℓ; or (iii) non-impacted CUT elements and all elements in libraries, denoted $X^{c\ell}$. Lastly, FineMOP re-monitors only affected specs.

3.3 A generic evolution-aware RV algorithm

Algorithm 1 shows the entry point for analyses that implement the workflow in Figure 4; it is generic in terms of program elements

Algorithm 2 Procedure `getDiff`

Inputs: `prgmElmts`: a set of program elements, *e.g.*, classes or methods,
$\mathcal{M}$: (`elemToChecksum`, dg, `elemToSpecs`) triple
Output: E_{all}: all elements, E_{new}: added elements, E_δ: changed elements
Initialization: `newCksums` $\leftarrow \emptyset$; $E_{all} \leftarrow \emptyset$; $E_{new} \leftarrow \emptyset$, $E_\delta \leftarrow \emptyset$

1: **procedure** getDiff(`prgmElmts`, $\mathcal{M}$):
2: **for all** $e \in$ `prgmElmts` **do**
3: $E_{all} \leftarrow E_{all} \cup \{e\}$
4: **if** $e \in \mathcal{M}$.`elemToChecksum.keys()` **then** ▷ is e in old revision?
5: **if** `checksum` (e) is $\mathcal{M}$.`elemToChecksum`$[e]$ **then** ▷ true if no change
6: `newCksums` $\leftarrow$ `newCksums` $\cup \{e \rightarrow \mathcal{M}$.`elemToChecksum`$[e]\}$
7: **else**
8: $E_\delta \leftarrow E_\delta \cup \{e\}$
9: `newCksums` $\leftarrow$ `newCksums` $\cup \{e \rightarrow$ `checksum`$(e)\}$
10: **else** ▷ e was not in old revision
11: $E_{new} \leftarrow E_{new} \cup \{e\}$
12: `newCksums` $\leftarrow$ `newCksums` $\cup \{e \rightarrow$ `checksum`$(e)\}$
13: $\mathcal{M}$.`elemToChecksum` $\leftarrow$ `newCksums`
14: **return** $E_{new}, E_\delta, E_{all}$

and §3.4 describes how FineMOP's analyses instantiate it. Line 2 calls `getDiff` (explained shortly) to return, as changed, elements whose checksums differ in the old and new revisions. Procedure `getDiff` returns E_{new} (newly-added elements), E_δ (modified elements), and E_{all} (all elements). It is trivial to make `getDiff` return $E_{deleted}$ (deleted elements). But, we elide $E_{deleted}$ because it has no impact on evolution-aware RV speed, safety, or precision: only elements in the new revision ($\mathcal{E}$) have nodes in the dependency graph built on line 3. Next, line 4 computes as impacted $E_{impacted}$, the union of nodes that (i) are new, *i.e.*, E_{new}; (ii) changed, *i.e.*, E_δ; (iii) transitively or reflexively depend on E_δ; and (iv) to which E_δ can transitively pass data. Lines 5 and 6 compute affected specs ($S_{affected}$) and an initial set of elements to instrument with those specs (E_{instr}), respectively. Based on Config, elements in libraries or non-impacted elements (the complement of $E_{impacted}$) are added to E_{instr} on lines 7 and 8. Lastly, line 9 re-monitors $S_{affected}$ in E_{instr}.

In Algorithm 2, `getDiff` partitions `prgmElmts` into three sets, based on how their checksums (in $\mathcal{M}$) changed since the old revision. The loop on lines 2–12 classifies each element in `prgmElmts`. Line 3 adds each element e in the new revision to E_{all}. If the checksum of e's cleaned bytecode is unchanged since the old revision (line 5), that checksum is copied into `newCksums` on line 6. If e is in both revisions, but its checksums differ, e is added to E_δ and its new checksum is added to `newCksums` on line 9. If e is new, it is added to E_{new} and its checksum is added to `newCksums` on line 12. After the loop terminates, line 13 sets $\mathcal{M}$.`elemToChecksum` to `newCksums` for use in the next revision and line 14 returns `getDiff`'s outputs.

In Algorithm 3, `findImpacted` takes E_{diff} (changed or newly-added elements) and a dependency graph, and outputs $E_{impacted}$, elements whose behavior can differ after changes. There, elements that transitively or reflexively depend on E_{diff} are always included in $E_{impacted}$ (line 3). But, when applied to ps_1^c, line 5 also computes *dependees*—elements that E_{diff} transitively depends on—for safety: even if they are not changed, elements in E_{diff} or their dependents may pass data to dependees and alter RV outcomes.

Algorithm 4 shows how `computeAffectedSpecs` finds affected specs, $S_{affected}$. First, it instruments all classes containing elements

Algorithm 3 `findImpacted` subprocedure

Inputs: E_{diff}: set of changed or new elements,
dg: dependency graph and `opt`: closure option
Output: $E_{impacted}$: elements impacted by changes to E_{diff}
Initialization: $E_{impacted} \leftarrow \emptyset$

1: **procedure** `findImpacted`(E_{diff}, dg, opt):
2: $\mathsf{dg}^{-1} \leftarrow$ `invert`(dg) ▷ dg^{-1} is dg with all edge directions reversed.
3: $E_{impacted} \leftarrow$ `transitiveClosureOf`(dg^{-1}, E_{diff}) ▷ dependents of E_{diff}
4: **if** opt = ps_1^c **then** ▷ dependees of dependents of E_{diff}
5: $E_{impacted} \leftarrow$ `transitiveClosureOf`(dg, $E_{impacted}$)
6: **return** $E_{impacted}$

Algorithm 4 `computeAffectedSpecs` subprocedure

Inputs: $E_{impacted}$: impacted elements, $\mathcal{S}$: all specs,
$\mathcal{M}$: (`elemToChecksum`, dg, `elemToSpecs`) triple
Output: $S_{affected}$: affected specs
Initialization: $S_{affected} \leftarrow \emptyset$

1: **procedure** `computeAffectedSpecs`($E_{impacted}$, $\mathcal{M}$):
2: instrInfo $\leftarrow$ instrument($E_{impacted}$, $\mathcal{S}$)
3: **for** (srcFile, lineNumber, spec) $\in$ instrInfo **do**
4: $e \leftarrow$ elementAt(srcFile, lineNumber)
5: $\mathcal{M}$.elemToSpecs[e] $\leftarrow$ $\mathcal{M}$.elemToSpecs[e] $\cup$ {spec}
6: **for all** $e \in E_{impacted}$ **do** $S_{affected} \leftarrow S_{affected} \cup$ $\mathcal{M}$.`elemToSpecs`[e]
7: **return** $S_{affected}$

in $E_{impacted}$ with all specs (line 2) to obtain instrInfo—triples of (i) spec name; (ii) instrumented source file; and (iii) line number—per instrumentation site. Lines 3–7 update $\mathcal{M}$.elemToSpec and return $S_{affected}$ as specs instrumented in any $e \in E_{impacted}$. `elementAt` (not shown) returns the class or method containing its arguments.

SoTA ps_1^c and $ps_3^{c\ell}$ instantiate these generic algorithms straightforwardly, using only classes as program elements, and setting $v = (ps_1^c, 0, 1)$ and $v = (ps_3^{c\ell}, 1, 1)$, respectively in Algorithm 1. We discussed ps_1^c and $ps_3^{c\ell}$ in §2 and §3.1. Next, we discuss how FineMOP's analyses instantiate Algorithms 2–4.

3.4 Instantiating the generic algo in FineMOP

§3.4.1, §3.4.2, and §3.4.3 respectively discuss how Mthd (and its two variants), Hybrid (and its one variant), and Fine instantiate the generic algorithms, using Figure 3 as a running example.

3.4.1 Mthd and its two variants. Challenges of method-level spec selection include: (i) constructing and reasoning about method-level dependency graphs that are often much larger than class-level graphs for the same project; (ii) mapping specs to methods is costlier than mapping them to classes; and (iii) for some changes, analysis time is high enough to make method-level analysis slower than class-level analyses. We address these challenges using three analyses with different trade-offs: Mthd, MthdNoP, and MthdFM. For all three, $\mathcal{E}$ (Algorithm 1) and `prgmElmts` (other algorithms) contain only methods. Also, `elemToChecksum` in Algorithm 2 maps methods to their checksums, and `getDiff` and `findImpacted` output sets of methods. The main differences among Mthd, MthdNoP, and MthdFM are in `computeAffectedSpecs` (Algorithm 4):

1. Mthd: `elemToSpecs` maps each class that contains method $m \in E_{impacted}$ to specs that are instrumented into m's enclosing class. That is, `elementAt` on line 4 in Algorithm 4 returns the *class*

Algorithm 5 `getDiff` subprocedure for Fine

Inputs: $\mathcal{E}$: classes, $\mathcal{M}$: (`classToChecksum`, dg, `classToSpecs`) triple
Output: C_{all}: all classes, C_{new}: added classes, C_δ: changed classes
Initialization: newM $\leftarrow \emptyset$; $C_{all} \leftarrow \emptyset$; $C_{new} \leftarrow \emptyset$, $C_\delta \leftarrow \emptyset$

1: **procedure** getDiff($\mathcal{E}$, $\mathcal{M}$):
2: **for all** $c \in \mathcal{E}$ **do**
3: $C_{all} \leftarrow C_{all} \cup \{c\}$
4: newM $\leftarrow$ newM $\cup \{c \rightarrow$ `newMetadataForClass`(c)$\}$
5: **if** $c \in \mathcal{M}$.keys **then** ▷ c is in the old revision
6: **if** newM [c] = $\mathcal{M}[c]$ **then** ▷ true if no change
7: **else if** `isModified`($\mathcal{M}$, newM, c) **then** $C_\delta \leftarrow C_\delta \cup \{c\}$
8: **else** $C_{new} \leftarrow C_{new} \cup \{c\}$ ▷ Class did not exist in the old revision
9: $\mathcal{M} \leftarrow$ newM
10: **return** $C_{new}, C_\delta, C_{all}$

Inputs: $\mathcal{M}$: same as $\mathcal{M}$ in getDiff, C_{new}: same as $\mathcal{M}$, c: class
Output: $true$ if change can alter functionality, else $false$

11: **procedure** `isModified`($\mathcal{M}$, newM, c):
12: //Determine whether the change alters behavior.
13: **for all** $f \in$ `getFields`(c) **do**
14: **if** `fldChanged`($\mathcal{M}[c][f]$, newM [c][f]) **then return** `true`
15: **for all** $n \in$ `getConstructorsAndInits`(c) **do**
16: **if** `conChanged`($\mathcal{M}[c][n]$, newM [c][n]) **then return** `true`
17: **for all** $m \in$ `getMethods`(c) **do**
18: **if** `mtdChanged`($\mathcal{M}[c][m]$, newM [c][m]) **then return** `true`
19: **return** `false`

of each instrumentation site like `INSTR : STHM.next` on line 4 in Figure 2. The benefit of Mthd over ps_1^c and $ps_3^{c\ell}$ (§2, §3.1, [73, 112]) is that specs that only have events in classes that do not transitively depend on impacted methods are not re-monitored. In Figure 3, Mthd re-monitors only S1–S4, saving the cost to monitor S5 that ps_1^c and $ps_3^{c\ell}$ incur. Mthd re-monitors S4 because it projects method-level reasoning to classes to be safer, so it returns B as affected.

2. MthdNoP: `elemToSpecs` maps from each method $m \in E_{impacted}$ to specs that are instrumented into m, *i.e.*, `elementAt` returns the enclosing *method* of each instrumentation site. The benefit of MthdNoP over Mthd is that specs that only have events in methods that are not in $E_{impacted}$ are not re-monitored. In Figure 3, by avoiding such projection, MthdNoP re-monitors only S1–S3; `B.m1` that has S4 events is not affected by the change.

3. MthdFM: `elemToSpecs` is same as MthdNoP's, but MthdFM applies an optimization to MthdNoP: by default, `monitor` instruments all classes containing an $e \in E_{impacted}$. But MthdFM resets Config on the fly (not shown) to only signal events from $E_{impacted}$ at runtime. The benefit of MthdFM over MthdNoP is that events in methods that are not in $E_{impacted}$ are not signaled at runtime, saving the time to monitor them. In Figure 3, MthdFM also re-monitors S2, but it re-monitors S1 only in method `C.m1` and S3 only in method `B.m2`, because the other two methods, `A.m1` and `B.m1`, that may generate events for S1 and S3 are not in $E_{impacted}$.

3.4.2 Hybrid and its one variant. The main difference between Hybrid and Mthd is in `remonitor` (Algorithm 1). There, Hybrid invokes `getDiff` twice, once with $\mathcal{E}$ as the set of classes, and once again with $\mathcal{E}$ as the set of methods. Hybrid mixes analysis granularities based on the change, for efficiency, *e.g.*, if classes are added or deleted, class-level reasoning is likely faster. But, if only methods

are modified, method-level may be more precise and faster than the purely class-level analyses that ps_1^c and $ps_3^{c\ell}$ use.

Both calls from HYBRID to `getDiff` return six sets: C_{all} (all classes), C_{new} (newly-added classes), C_δ (changed classes), M_{all} (all methods), M_{new} (newly-added methods), and $M_{changed}$ (changed methods). These are used in `findImpacted` to find $E_{impacted}$. HYBRID projects analysis results to the class level, but HYBRID's variant—HYBRIDNOP—does not. Also, HYBRID's `elemToSpecs` maps classes to specs, but HYBRIDNOP's `elemToSpecs` maps methods to specs. HYBRIDNOP does not project method-level reasoning to classes, so it can be faster than HYBRID by not re-monitoring affected specs in unaffected methods. In Figure 3, HYBRID re-monitors `S1`–`S4`, but HYBRIDNOP only re-monitors `S1`–`S3`: `B.m1` is unchanged.

3.4.3 FINE. Unlike all other FineMOP analyses, FINE uses `getDiff` in Algorithm 5 because its metadata $\mathcal{M}$ is different and maps each class C to a triple (F, N, M), that maps fields and *callables*—constructors, initializers, and methods—in C to custom data structures [78]. FINE is the first to use these data structures to determine if the semantics of bytecode-level changes can alter RV outcomes.

Algorithm 5 takes $\mathcal{E}$ as the set of classes. But, when checking if C is modified (line 7), bytecode-modifying changes to each class member are analyzed against 13 rules from [78] on lines 13–18 to see if those changes can alter RV outcomes. If all changes to fields and callables in C cannot alter RV outcomes, C is treated as not changed (line 19). Doing so can be more beneficial than other algorithms if these are the only kind of changes in a revision. In Figure 3, FINE re-monitors `S1`, `S3`, `S4`, and `S5`; unlike ps_1^c and $ps_3^{c\ell}$, it does not re-monitor `S2` because the change to `A.m2` cannot lead to new violations. FINE does not have variants. We hypothesize that such FINE variants would be slower than FINE, as they must also additionally collect method-level dependencies.

Some interdependencies make it hard to instantiate the generic algorithm in other ways. For example, "`NOP`" requires a method-level dependency graph, but FINE reasons about method-level changes without using a graph. So, FINE cannot be combined with "`NOP`". Also, "`FM`" depends on "`NOP`", so FINE plus "`FM`" is hard.

3.4.4 Summary: features of class-level and FineMOP analyses. Table 1 summarizes features (header row) in all analyses (first column) in this paper. Each FineMOP analysis is annotated with the same superscripts and subscript as the class-level analyses (ps_1^c or $ps_3^{c\ell}$) that it is applied to. We will use this annotation for these FineMOP analyses in the rest of this paper. §3.1 and §3.2 describe superscripts c and l, and introduce subscripts 1 and 3 as the most and least conservative change-impact analyses for evolution-aware RV [73], respectively. The change-impact analysis marked as 1 leads to re-monitoring specs with events in (i) changed program elements (E_δ), (ii) dependents of E_δ, (iii) dependees of E_δ, and (iv) dependees of dependents of E_δ. In contrast, the change-impact analysis marked as subscript 3 leads to re-monitoring only specs with events in (i) E_δ and (ii) dependents of E_δ.

3.5 Implementation

We implement FineMOP as a Maven plugin. We extend STARTS [72, 105] using JavaParser [52] and ASM [7] to reason about method-level, field-and-method-level, and semantics-modifying changes.

Table 1: Features of all analyses in this paper. ps_1^c and $ps_3^{c\ell}$ are SoTA class-level analyses we evaluate. Other rows mark FineMOP's six analyses with the same superscripts and subscript (explained in the text) as the class-level analyses that we apply them to. ✓: feature is present. ✗: feature is absent.

	Class level analysis	Method level analysis	Library instrumentation	Semantics reasoning	Finer spec mapping	Finer monitoring
ps_1^c	✓	✗	✓	✗	✗	✗
FINE$_1^c$	✓	✗	✓	✓	✗	✗
HYBRID$_1^c$	✓	✓	✓	✗	✗	✗
HYBRIDNOP$_1^c$	✓	✓	✓	✗	✓	✗
MTHD$_1^c$	✗	✓	✓	✗	✗	✗
MTHDFM$_1^c$	✗	✓	✓	✗	✓	✓
MTHDNOP$_1^c$	✗	✓	✓	✗	✓	✗
$ps_3^{c\ell}$	✓	✗	✗	✗	✗	✗
FINE$_3^{c\ell}$	✓	✗	✗	✓	✗	✗
HYBRID$_3^{c\ell}$	✓	✓	✗	✗	✗	✗
HYBRIDNOP$_3^{c\ell}$	✓	✓	✗	✗	✓	✗
MTHD$_3^{c\ell}$	✗	✓	✗	✗	✗	✗
MTHDFM$_3^{c\ell}$	✗	✓	✗	✗	✓	✓
MTHDNOP$_3^{c\ell}$	✗	✓	✗	✗	✓	✗

Table 2: Statistics on 68 projects that we evaluate: no. of test methods (#tests), test time w/o RV in seconds (t), test time with RV in seconds (t^{rv}), lines of code (SLOC), % statement coverage (cov^s), % branch coverage (cov^b), no. of commits (#SHAs), years since first commit (age), and no. of stars (#★).

	#tests	t	t^{rv}	SLOC	cov^s	cov^b	#SHAs	age	#★
Mean	223.9	7.1	77.7	11,672.8	64.4	56.0	476.2	10.1	475.3
Med	83.5	4.1	65.0	3,775.5	68.1	60.0	199.0	10.0	63.0
Min	2	2.2	7.7	312	0.4	0.3	10	2	6
Max	4,232	36.4	546.8	2.1×10^5	99.2	99.3	4,890	27	12,292
Sum	15,222	481.7	5,280.4	7.9×10^5	n/a	n/a	n/a	n/a	32,320

FineMOP uses STARTS to find classes or methods impacted by changes and uses AspectJ to find specs to re-monitor. Lastly, FineMOP updates AspectJ's config file to avoid re-monitoring non-affected specs. For MTHDFM, FineMOP uses AspectJ to dynamically exclude un-impacted methods from monitoring.

4 Evaluation

We organize our evaluation around five research questions (RQs):

RQ1. What is FineMOP's overhead vs. JavaMOP, ps_1^c, and $ps_3^{c\ell}$?
RQ2. How precise is FineMOP compared with ps_1^c, and $ps_3^{c\ell}$?
RQ3. How safe is FineMOP compared with ps_1^c, and $ps_3^{c\ell}$?
RQ4. How does FineMOP compare with RTS?
RQ5. How beneficial is combining FineMOP with RTS?

RQ1 compares FineMOP's overheads vs. those of SoTA evolution-*unaware* JavaMOP and evolution-*aware* class-level analyses, ps_1^c and $ps_3^{c\ell}$ (§2, §3.1, [73, 112]). RQ2 investigates the degree to which FineMOP reduces ps_1^c's and $ps_3^{c\ell}$'s affected specs, impacted classes, monitors, events, and instrumentation. RQ3 evaluates if FineMOP's speedups and precision are at the expense of missing violations that are new after a change. RQ4 compares overheads of FineMOP alone vs. those of combining four RTS techniques with ps_1^c and $ps_3^{c\ell}$. Lastly, RQ5 evaluates the benefits of combining FineMOP with RTS.

4.1 Experimental Setup

Evaluation subjects and revisions. We use 1,104 revisions of 68 open-source projects; Table 2 summarizes these projects' statistics

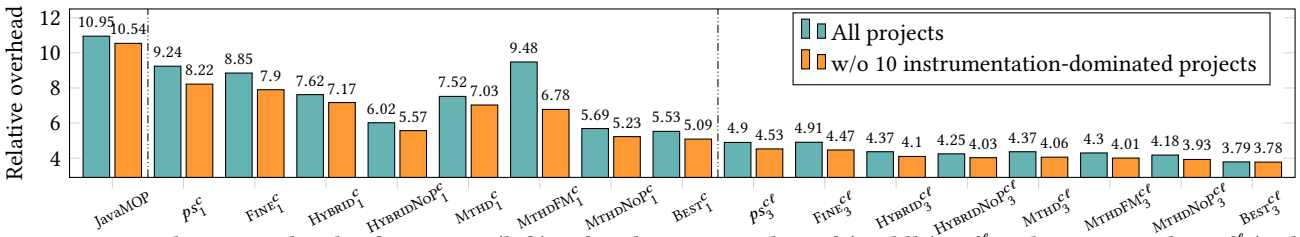

Figure 5: Cummulative overheads of JavaMOP (left), ps_1^c and FineMOP plus ps_1^c (middle), $ps_3^{c\ell}$ and FineMOP plus $ps_3^{c\ell}$ (right), relative to running tests without RV, for all projects (teal) and with 10 instrumentation-dominated projects excluded (orange).

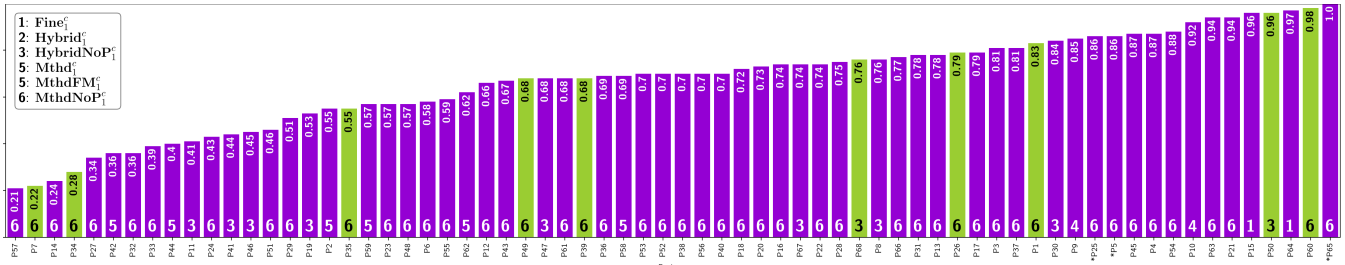

Figure 6: Best-performing analysis applied to ps_1^c vs. ps_1^c (1.0 on y-axis). Green: 10 projects exluded from orange bars in Figure 5.

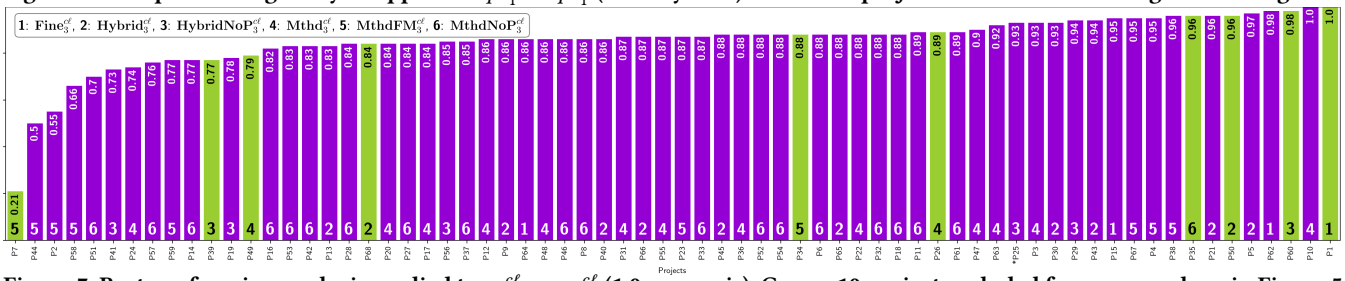

Figure 7: Best-performing analysis applied to $ps_3^{c\ell}$ vs. $ps_3^{c\ell}$ (1.0 on y-axis). Green: 10 projects exluded from orange bars in Figure 5.

("n/a" are meaningless sums). We start with all 80 projects in [38] where monitoring (not instrumentation) dominates RV overhead and JavaMOP time is greater than 10 seconds. We use only 33 of these. Of those excluded, 28 have fewer than three historical revisions where JavaMOP works, three fail to compile, and tests fail in 16. We also use 35 projects from [39] where RV overhead is most dominated by instrumentation. For all 68 projects, we choose up to 20 historical revisions where Java file(s) changed, code compiles, and tests pass with and without JavaMOP.

Specs. We use 160 specs of correct JDK API usage that were formalized by Lee et al. [67, 80] and used in all recent work on RV of software tests [37–40, 53, 61, 62, 68, 70, 71, 73, 85, 112].

Baseline Tools. We use the modernized and refactored JavaMOP in the TraceMOP repository [40, 106] which also merges JavaMOP [49, 55] and RV-Monitor [50, 80] code, fixes a 13-year old performance bug [51], and reduces duplication. We use ps_1^c and $ps_3^{c\ell}$ from eMOP [27, 112], but we extend them to add library classes to dependency graphs. We use Ekstazi [33, 34], FineEkstazi [78], FineSTARTS [78], and STARTS [69, 72] for RTS experiments.

Running Experiments. We write Maven extensions and scripts to integrate FineMOP in evaluated projects. We run all experiments in Docker containers; our repository has our Dockerfiles. We run all experiments for each project on one of three machines: (i) Intel® Xeon® w9-3475X CPU, 128 GB RAM; (ii) Intel® Xeon® Gold 6348 machine, 512 GB RAM; (iii) AMD EPYC™ 9654 machine, 1.5 TB RAM, running Java 8 and Maven 3.9 on Ubuntu (20.04 or 24.04).

4.2 RQ1: Overheads

Figure 5 shows cumulative relative overheads of JavaMOP (left), ps_1^c and FineMOP applied to ps_1^c (middle), and $ps_3^{c\ell}$ and FineMOP applied to $ps_3^{c\ell}$, respectively. Cumulative relative overhead is the sum of time to run RV in all 1,104 revisions of 68 projects divided by sum of time to run tests without RV in these versions.

FineMOP analyses vs. JavaMOP, ps_1^c, and $ps_3^{c\ell}$. The teal bars in the middle of Figure 5 show that, cumulatively, MthdNoP$_1^c$ is fastest when applied to ps_1^c; it is ~2x faster than JavaMOP. Per project, MthdNoP$_1^c$ is up to 6.77x faster than JavaMOP (avg: 2.09x) and up to 4.86x faster than ps_1^c (avg: 1.7x). Teal bars in the right of Figure 5 show that MthdNoP$_3^{c\ell}$ is also the fastest cumulatively when applied to $ps_3^{c\ell}$, taking only 38.0% of JavaMOP time. Per project, MthdNoP$_3^{c\ell}$ is up to 7.31x faster than JavaMOP (avg: 2.87x) and up to 4.47x faster than $ps_3^{c\ell}$ (avg: 1.18x). Overall, FineMOP speeds up ps_1^c more than $ps_3^{c\ell}$, which is quite fast. MthdFM$_1^c$ is the slowest cumulatively when applied to ps_1^c; it is slower than ps_1^c because for 10 projects where instrumentation dominates RV overhead, the extra cost to disable monitoring in methods $m \notin E_{impacted}$ is high. Orange bars in Figure 5 show results without these 10; the across-the-board speedup shows a need for future work to make FineMOP faster in instrumentation-dominated projects.

Best-performing FineMOP analysis per project. We run all FineMOP analyses in all revisions of each evaluated project, and select the fastest and safe one as *best-performing* for that project.

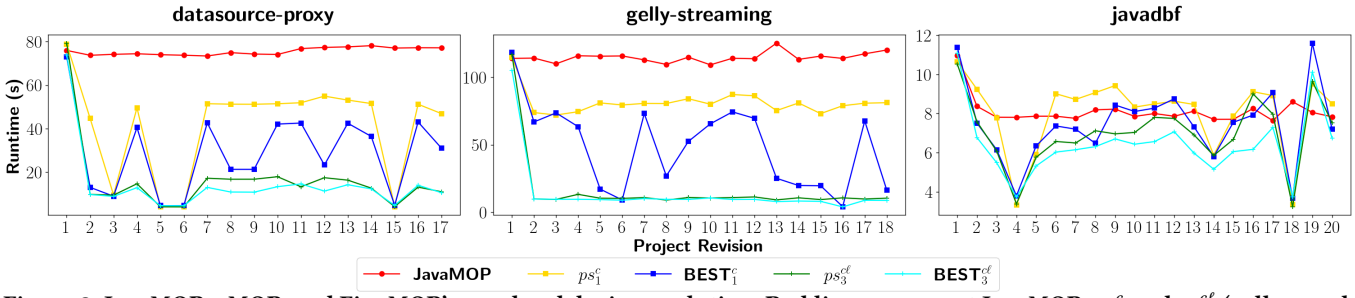

Figure 8: JavaMOP, eMOP, and FineMOP's overhead during evolution. Red lines represent JavaMOP, ps_1^c and $ps_3^{c\ell}$ (yellow and green), and best performing FineMOP techniques $\textsc{Best}_1^c$ and $\textsc{Best}_3^{c\ell}$ (blue and cyan).

Rank	Fine	Hybrid	HybridNoP	Mthd	MthdFM	MthdNoP
1	2 (4)	0 (12)	10 (7)	2 (14)	6 (10)	48 (20)
2	3 (0)	4 (15)	41 (14)	0 (9)	4 (12)	14 (15)
3	2 (5)	8 (9)	10 (13)	25 (18)	21 (2)	0 (18)

Figure 9: How frequently each FineMOP analyis is one of the three fastest. Numbers outside (inside) parentheses are counts when FineMOP analyses are applied to ps_1^c ($ps_3^{c\ell}$).

When applied to ps_1^c, $\textsc{Best}_1^c$ in Figure 5 is the cumulative relative overhead of FineMOP's best-performing analysis per project; it is 9.56 hours faster than JavaMOP and 6.44 hours faster than ps_1^c. Also, $\textsc{Best}_3^{c\ell}$—analogous to $\textsc{Best}_1^c$—saves 12.3 and 1.29 hours, compared to JavaMOP and $ps_3^{c\ell}$, respectively. Figure 6 shows the ratio of $\textsc{Best}_1^c$ to ps_1^c time per project; lower is better. The $y = 1.0$ line represents ps_1^c. The number at the bottom of each bar (and mapped in the legend) shows the $\textsc{Best}_1^c$ analysis for each project. Also, green bars show the 10 projects that we exclude from the orange bars in Figure 5. In 50 of 68 projects, the best FineMOP algorithm takes less than 80% of ps_1^c time, and for 14 out of 68 projects, FineMOP is at least 2x faster than ps_1^c. Figure 7 shows analogous results ratio for $\textsc{Best}_3^{c\ell}$. There, only 13 of 68 projects take less than 80% of $ps_3^{c\ell}$ time, but two projects still see at least a 2x speedup.

Which FineMOP analysis should be the default? Figure 9 shows how often each FineMOP analysis performs best (1), second (2), or third (3) when applied to ps_1^c ($ps_3^{c\ell}$). MthdNoP performs best most often for ps_1^c and $ps_3^{c\ell}$. When FineMOP is applied to ps_1^c, $\textsc{Fine}_1^c$, $\textsc{HybridNoP}_1^c$, $\textsc{Mthd}_1^c$, $\textsc{MthdFM}_1^c$, $\textsc{MthdNoP}_1^c$ are the fastest in 2, 10, 2, 6, 48 projects, respectively. ps_1^c and $\textsc{Hybrid}_1^c$ are never the fastest in our evaluation. When FineMOP is applied to $ps_3^{c\ell}$, $\textsc{Fine}_3^{c\ell}$, $\textsc{Hybrid}_3^{c\ell}$, $\textsc{HybridNoP}_3^{c\ell}$, $\textsc{Mthd}_3^{c\ell}$, $\textsc{MthdFM}_3^{c\ell}$, and $\textsc{MthdNoP}_3^{c\ell}$ are the fastest for 4, 12, 7, 14, 10, and 20 projects, respectively. $ps_3^{c\ell}$ is fastest for one project. FineMOP's less conservative change-impact analysis misses violations in only one project (P25). Each FineMOP analysis is the best-performing at least twice when applied to ps_1^c or $ps_3^{c\ell}$, further justifying our design choice to implement multiple analyses in FineMOP. Overall, based on these findings, we recommend $\textsc{MthdNoP}_1^c$ as the default FineMOP analysis: it optimizes safe-by-design ps_1^c and it is most often the fastest without safety issues in our experiments.

How to choose FineMOP analysis for a new project? We recommend to run all FineMOP analyses in the first revision, and choose the best-performing analysis in that revision subsequently. Our per-project analysis (our appendix has details) suggests two reasons why this strategy might work well in practice. First, the best-performing analysis in the first revision remains the best or second best in all subsequent revisions in 48 of 68 evaluated projects. Second, even when the best-performing analysis in the first revision later becomes second best, the loss in speedup is only 9.3 percentage points on average per project, or 3.50 seconds per revision. Users who prefer not to incur the one-time cost of finding the best-performing analysis for their project can use our recommended default: $\textsc{MthdNoP}_1^c$.

FineMOP's runtime overhead as code evolves. Figure 8 shows the overheads of JavaMOP, ps_1^c, $ps_3^{c\ell}$, and FineMOP applied to ps_1^c and $ps_3^{c\ell}$ for each evaluated revision in three projects (our appendix has plots for all projects). The area under each curve is the total time across all revisions per technique. Older revisions are to the left of newer ones. In `datasource – proxy` (left), FineMOP's best-performing analysis is almost always faster than ps_1^c and $ps_3^{c\ell}$. For `gelly – streaming` (middle), FineMOP outperforms ps_1^c, but is only slightly faster than $ps_3^{c\ell}$. In `javadbf` (right), FineMOP is occasionally slower than ps_1^c and $ps_3^{c\ell}$, and FineMOP, ps_1^c, and $ps_3^{c\ell}$ are often slower than JavaMOP. But, FineMOP is faster than ps_1^c and $ps_3^{c\ell}$ overall. Evolution-aware RV often performs poorly when JavaMOP is fast, when a project makes frequent major changes, or due to regular library updates. 25, 16, and 4 projects have similar trends as those on the left, center, and right, respectively.

When does each FineMOP analysis perform best? We conduct a preliminary qualitative analysis by manually analyzing 146 revisions in 43 projects where a FineMOP analysis outperforms others. Fine tends to perform best when most or all of the semantics-modifying changes in a revision cannot lead to new violations. When Hybrid and HybridNoP perform best, they do so only marginally. But, if (i) method-level dependencies are complex and (ii) the changes mix class-level changes (like class deletion or addition) with finer-granularity changes, Hybrid and HybridNoP tend to outperform other FineMOP analysis by avoiding unnecessary and costly traversal of the method-level graph, and outperform ps_1^c and $ps_3^{c\ell}$ by precisely re-monitoring fewer unaffected specs.

MthdFM tends to perform best when (i) many events and monitors are generated in methods that MthdFM does not instrument with affected specs or (ii) the project has few instrumentation locations, especially if frequent changes are made to methods with events for many affected specs. In such cases, MthdFM's high precision pays off. But, MthdFM performs poorly when the absolute reduction in events and monitors is small or when a project has many instrumentation points. In such cases, MthdNoP tends to be the better choice. Mthd is generally slower than MthdNoP. In the

Table 3: Average percentage of ps_1^c (left) and $ps_3^{c\ell}$ (right) peak memory used (Mem.), affected specs (Specs), impacted classes (Classes), monitors synthesized (Mon.), events signaled (Events), and lines instrumented (Instr.) by FineMOP's analyses.

	Fine$_1^c$	Hybrid$_1^c$	HybridNoP$_1^c$	Mthd$_1^c$	MthdFM$_1^c$	MthdNoP$_1^c$	Fine$_3^{c\ell}$	Hybrid$_3^{c\ell}$	HybridNoP$_3^{c\ell}$	Mthd$_3^{c\ell}$	MthdFM$_3^{c\ell}$	MthdNoP$_3^{c\ell}$
Mem.	99.39	97.70	78.42	97.59	84.43	76.86	99.38	99.33	102.30	103.15	111.20	109.37
Specs	96.87	96.17	70.2	94.29	67.0	67.0	96.14	92.47	73.52	89.98	70.09	70.09
Classes	93.59	86.97	86.97	72.21	72.21	72.21	88.84	69.44	69.44	57.8	57.8	57.8
Mon.	96.57	97.75	70.27	97.5	64.22	66.42	99.0	99.16	99.26	99.19	99.1	99.22
Events	96.51	97.93	70.02	97.78	63.79	66.47	98.97	99.19	99.22	99.18	99.11	99.24
Instr.	96.19	96.14	70.81	95.14	66.58	66.58	96.93	95.28	90.13	96.18	90.09	90.09

few cases where Mthd appears faster than MthdNoP, the difference is only a few seconds and may be due to experimental noise. In sum, MthdFM tends to perform best for small projects where developers make frequent and localized changes in methods containing many events. Otherwise, MthdNoP typically outperforms MthdFM. HybridNoP generally performs better than Hybrid. Just like with Mthd and MthdNoP, the reason is that the "NoP" optimization yields more precision that pays off in majority of projects.

Memory overheads. The first row in Table 3 shows average memory overhead of FineMOP's analysis relative to ps_1^c and $ps_3^{c\ell}$. Applying FineMOP to ps_1^c saves more memory than applying FineMOP to $ps_3^{c\ell}$. In fact, FineMOP uses more memory than $ps_3^{c\ell}$ in most cases. The most memory efficient FineMOP analysis reduces ps_1^c's peak memory use by 2.05 GB. But, the analysis with the worst additional memory use increases $ps_3^{c\ell}$'s peak memory by 532.48 MB.

4.3 RQ2: Precision

We measure the degree to which FineMOP reduces the number of affected specs that class-level analyses (ps_1^c and $ps_3^{c\ell}$) re-monitor after changes. We also measure the effect of that reduction on the number of impacted classes, monitors created, events signaled, and instrumented code locations in FineMOP vs. ps_1^c and $ps_3^{c\ell}$.

The second row in Table 3 shows the average percentage of affected specs in ps_1^c (left) and $ps_3^{c\ell}$ (right) that all six FineMOP analyses find. (MthdNoP and MthdFM are equal). There, MthdNoP$_1^c$ and MthdNoP$_3^{c\ell}$ re-monitor the lowest proportion of ps_1^c and $ps_3^{c\ell}$ affected specs—only 67.0% and 70.09%, respectively, on average. MthdNoP finds 28.1% fewer specs (17.8 per revision) than Mthd, HybridNoP finds 26.5% fewer specs (16.9 per revision) than Hybrid, and Hybrid and Mthd select a similar number of specs.

The third row in Table 3 shows the average percentage of impacted classes in ps_1^c (left) and $ps_3^{c\ell}$ (right) that FineMOP finds. Hybrid and HybridNoP are equal, as are Mthd, MthdNoP, and MthdFM: elements in these sets use the same change-impact analyses. Mthd$_1^c$ and MthdNoP$_3^{c\ell}$ find the fewest impacted classes on average, only 72.21% (min: 32.02%) of ps_1^c's, and 57.8% (min: 16.76%) of $ps_3^{c\ell}$'s. That is, reasoning about changes at the method-level is more precise than reasoning with Fine and Hybrid.

The fourth and fifth rows in Table 3 show the average percentage of monitors created and events signaled in ps_1^c (left) and $ps_3^{c\ell}$ (right) that FineMOP analyses find. MthdFM$_1^c$ creates the fewest monitors and signals the fewest events, yet it is the slowest when FineMOP is applied to ps_1^c because the extra cost to disable monitoring (see §4.2) outweighs savings from signaling fewer events. MthdNoP$_1^c$ has the next fewest monitors created and events signaled, which is notable because it achieves the greatest overall time savings relative to ps_1^c. We see also that FineMOP only marginally reduces $ps_3^{c\ell}$'s monitors and events on average. This marginal reduction likely contributes to the similarity in the greater distribution of best-performing FineMOP analyses when applied to $ps_3^{c\ell}$ in Figure 9.

Lastly, row six in Table 3 shows average percentages of locations instrumented by ps_1^c (left) and $ps_3^{c\ell}$ (right) that FineMOP instruments. The differences among FineMOP analyses almost mirror those for affected specs (second row). Since only affected specs are used for instrumentation in a new program revision, the fewer the affected specs, the fewer code locations are instrumented.

4.4 RQ3: Safety

Setup. An evolution-aware RV technique is safe if it finds all *new* violations after changes [73]. The assumption is that users only want to see new violations, and are aware of old ones. To compute the ratio of new violations that FineMOP analyses find to those of ps_1^c and $ps_3^{c\ell}$, we use Violation Message Suppression (VMS) [73] in eMOP [112], as the ground truth. VMS aims to filter out old violations as those that are the same in the old and new revisions, after syntactically mapping lines across versions. Since syntactic line mapping [2, 86] is unsound in general, and semantic line mapping is a hard problem [3, 104], VMS often reports old violations as new. So, we report the number of new violations found by VMS but missed by a FineMOP analysis before (pre) and after (post) manual inspection, during which we filter out seemingly missed new violations that are due to (i) VMS limitations or bugs, (ii) non-deterministic test executions, and (iii) a known bug in eMOP [36].

Results. Table 4 shows safety results for all six FineMOP analyses when applied to ps_1^c. The first, third, and fifth rows show numbers of missed violations, numbers of revisions with a missed violation, and numbers of projects with an unsafe revision *before* inspection, respectively. The second, fourth, and sixth rows show these numbers *after* inspection. First column parentheses show totals.

Across 1,104 revisions of 68 projects, VMS finds 943 new violations. Before manual inspection, Fine$_1^c$ seems to miss 70 violations that ps_1^c finds in 40 revisions of 15 projects. Our inspection shows that Fine$_1^c$ finds the same new violations as ps_1^c. The least safe, HybridNoP$_1^c$, MthdFM$_1^c$, and MthdNoP$_1^c$ only miss two new violations that ps_1^c finds. The percentages of missed new violations (≤0.32%) and revisions where FineMOP's analyses are unsafe (≤0.27%) are small. So, we have initial confidence that finer-granularity analysis speeds up evolution-aware RV without impacting safety much. §5 discusses future work on improving safety.

We find that all violations missed by some FineMOP analyses are caused by missing edges in dependency graphs, which happen because eMOP, which we extend, does not re-instrument libraries when finding affected specs. So, FineMOP can miss violations if all events in a violating trace are in a library. Theoretically, graphs can be incomplete if dynamic language features like reflection make static analysis miss edges. But, we do not see missed violations due

Table 4: Safety of FineMOP relative to ps_1^c ($ps_3^{c\ell}$).

	Fine	Hybrid	HybridNoP	Mthd	MthdFM	MthdNoP
Missed violations pre (of 943)	70 (118)	76 (119)	93 (133)	75 (118)	100 (136)	101 (134)
Missed violations post (of 943)	0 (0)	1 (0)	2 (1)	1 (0)	2 (1)	2 (1)
Unsafe revisions pre (of 1,104)	40 (67)	45 (68)	61 (76)	45 (68)	62 (77)	63 (76)
Unsafe revisions post (of 1,104)	0 (0)	1 (0)	2 (1)	1 (0)	2 (1)	2 (1)
Unsafe projects pre (of 68)	15 (19)	14 (18)	17 (21)	15 (18)	18 (21)	17 (21)
Unsafe projects post (of 68)	0 (0)	1 (0)	2 (1)	1 (0)	2 (1)	2 (1)

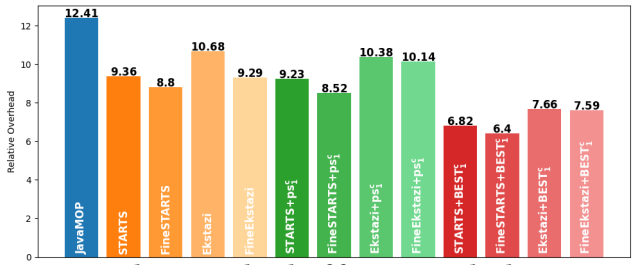

Figure 10: Relative overheads of four RTS tools plus JavaMOP (orange bars), ps_1^c (green bars), and Best$_1^c$ (pink bars).

to dynamic features, and prior work suggests such misses are rare in practice (only 0.2% of 985 versions of 22 projects on average [69]).

Notably, for method-level dependency graphs, our early implementation included variants that use fields. We only present results for variants that do not use fields because they are more efficient and we do not observe missed violations due to field exclusion.

Numbers in parentheses in Table 4 show safety results for FineMOP analyses when applied to $ps_3^{c\ell}$. Our inspection shows that Fine$_3^{c\ell}$, Hybrid$_3^{c\ell}$, and Mthd$_3^{c\ell}$ find all violations that $ps_3^{c\ell}$ finds. But, HybridNoP$_3^{c\ell}$, MthdFM$_3^{c\ell}$, and MthdNoP$_3^{c\ell}$ each miss one violation that $ps_3^{c\ell}$ finds. FineMOP's analyses miss ≤0.11% of new violations that $ps_3^{c\ell}$ finds in only ≤0.09% of revisions; our inspection shows that these misses have the same causes as those in ps_1^c.

Performance-safety trade off. Users who are only concerned with speed, *e.g.*, during pre-commit testing or debugging, can use the fastest technique for their project, since safety issues are rare in our experiments (at its worst, FineMOP only misses 0.32% of violations). But, when safety and performance are both essential, the fastest *and* safe technique for a project should be used.

4.5 RQ4: Comparing with RTS

We compare the overheads relative to running tests without RV of the best-performing FineMOP analysis (Best$_1^c$) per project with those of combining JavaMOP with four regression test selection (RTS) tools: STARTS [69, 72], Ekstazi [33, 34], FineSTARTS, and FineEkstazi [78]. RTS aims to speed up regression testing by re-running only tests impacted by code changes. We evaluate 51 of 68 projects. We could not evaluate the rest due to a known bug in Ekstazi or FineEkstazi [108]. The relative overheads of JavaMOP, STARTS, FineSTARTS, Ekstazi, FineEkstazi, and Best$_1^c$ are 12.4, 9.4, 8.8, 10.7, 9.3, and 5.8, respectively. Combining JavaMOP with all four RTS tools is faster than running JavaMOP alone. But despite running all tests in each revision, FineMOP still outperforms all combinations of RTS with JavaMOP. Best$_1^c$ is up to 12.3x (avg: 1.4x) faster than the most efficient RTS tool used. Across the 51 projects, FineMOP is faster than all RTS tools in 34 projects, and faster than an RTS tool in 43 projects. We conclude that RTS alone does not provide as much speedup as FineMOP alone. But, both approaches could be complementary. So, we next evaluate their combination.

4.6 RQ5: Combining with RTS

We combine the four RTS tools in RQ4 with the best-performing FineMOP analysis per project, as well as class-level ps_1^c and $ps_3^{c\ell}$. Figure 10 shows the resulting relative overheads. There, the first bar is JavaMOP alone. The orange bars show overheads of JavaMOP plus each RTS tool. The green bars show overheads of ps_1^c plus each RTS tool, and the red bars show overheads of Best$_1^c$ plus RTS. Combining FineMOP with RTS yields even more speedups in Figure 10. Combining Best$_1^c$ with FineSTARTS has the best speedup, from 8.8x to 6.4x, and is the fastest on average for all combinations in Figure 10. On average, Best$_1^c$ plus RTS is 1.2x (max 3.5x) faster than the fastest RTS tool alone. Across the 51 projects, FineMOP plus RTS outperforms the RTS-only counterpart in 31 projects. Compared with STARTS alone, combining Best$_1^c$ with STARTS saves 2.5 hours across all projects, Best$_1^c$ plus FineSTARTS saves 2.5 hours, Best$_1^c$ plus Ekstazi saves 2.9 hours, and Best$_1^c$ plus FineEkstazi saves 1.5 hours. These speedups come at little additional cost due to RTS' analysis, which account for 1.2% of the end-to-end time of Best$_1^c$ plus RTS runtime and 1.5% of Best$_3^{c\ell}$ plus RTS end-to-end time.

Combining FineMOP with RTS yields, on average, 32.2% fewer events, 32.7% fewer monitors, and 29.3% fewer instrumented locations, compared with RTS alone. Also, FineMOP plus RTS yields 22.6% fewer events, 26.0% fewer monitors, and 14.6% fewer instrumented locations compared with FineMOP alone. On average, FineMOP alone yields 12.4%, 9.1%, and 17.2% fewer events, monitors, and instrumented locations, respectively, than RTS alone. We find that combining FineMOP with RTS benefits less from RTS selecting fewer tests and more from FineMOP re-monitoring fewer specs.

One concern is that, by combining FineMOP and RTS, their potential for unsafety might be compounded. We see no such compounding in our experiments: the unsafe cases from FineMOP plus RTS are the same as those in Table 4 after inspection.

5 Discussion

Why some projects see no speedups. The main reason is frequent third-party library updates. By default, ps_1^c and $ps_3^{c\ell}$ re-monitor *all* specs in *all* classes when a library changes, because reasoning about the impact of such changes is prohibitive [72, 112]. FineMOP inherits this limitation. In projects P10, P60, and P65, roughly 35%, 75%, and 50% of revisions, respectively, update a library. So, FineMOP's best-performing analyses in these projects are slower than ps_1^c and $ps_3^{c\ell}$. But, FineMOP's speedups across many revisions in all but one project outweigh costs of library changes in few revisions. Another reason some projects see minimal speedup, especially when applying FineMOP to $ps_3^{c\ell}$, is that RV is already very fast for them. So, FineMOP's extra analysis costs do not pay off.

Limitations and future work. Some FineMOP variants use method-level static analysis, which can be unsound if the dependency graph is incomplete. But, we find that for 66 (of 68) projects and 1,102 revisions, FineMOP's method-level analyses find all new violations found by class-level ps_1^c and $ps_3^{c\ell}$. FineMOP's static analysis can be unsound in the presence of dynamic features like reflection [65]. Other static analyses (including eMOP [73, 112] and STARTS [69, 72]) have this limitation too, and future work can learn from techniques for making them "soundier" [79] in the presence of such dynamic features [10, 14, 74–77, 99, 102, 107].

We evaluate FineMOP on monitoring- and instrumentation-dominated projects. But, for instrumentation-dominated projects, FineMOP alone may not offer as much speedup as iMOP, the SoTA instrumentation-driven evolution-aware RV technique [39]. More research and development is needed to combine iMOP and FineMOP. To see why, consider a hypothetical example with two specs: `SpecA` and `SpecB`. iMOP must instrument `SpecA` and `SpecB` into the entire codebase in the first revision. If in a second revision FineMOP finds only `SpecA` to be affected by changes, there is no way to (safely) undo all the instrumentation of `SpecB` that iMOP did in the first revision. So, if FineMOP is combined with iMOP today, FineMOP will still re-monitor `SpecA` and `SpecB`.

FineMOP is designed to reduce RV overhead only during testing, before deployment. FineMOP aims to find all new violations, not all violations, so it is not suitable for use in deployment, where RV aims to find all violations. Also, this paper is on overhead reduction only; it is not concerned with other challenges of using RV, like imprecise specs, debugging violations, or inferring specs.

Lastly, in our safety evaluation, we manually inspect hundreds of new violations that were seemingly missed. But, outside research, RV users will not need to do as much manual inspection since they will likely not be comparing analyses as we do. Better tools and techniques for reducing manual inspection are needed, but this is research on its own [70], beyond our paper's scope. We perform manual inspection mainly due to (i) test nondeterminism and (ii) bugs in eMOP's VMS, which we use to evaluate safety. If a violation occurs non-deterministically when running the same technique on the same program and tests always pass, there is no automated technique to find the root cause. So, we use manual inspection. As eMOP improves, the need to manually inspect some of its outputs will also naturally reduce.

Threats to validity. FineMOP's results may not generalize beyond the 68 projects and their 1,104 GitHub revisions that we evaluate. To mitigate this threat, we use many projects and revisions from prior evolution-aware RV work and a recent RV study. Due to bugs and limitations in VMS, we manually inspect some violations. This manual process may lead to misclassifications, a threat that we mitigate by having a co-author review all inspection results. The scripts and Maven plugin that we use may also contain bugs. To reduce this threat, the scripts and plugin are reviewed by multiple authors for validation. We also build experimental infrastructure on top of more mature tools such as STARTS and eMOP, and our artifact is publicly available on GitHub for external validation.

6 Related Work

RV during software testing. It is well-known in the RV community that RV can be used to find bugs during testing. For example, in very early work, Artho et al. [5, 6] combined automated test generation with RV. But, with the emergence of CI [26, 31, 47, 48, 84, 103, 113], and its rapid code-change cycles, more recent work started to investigate how to make RV practical for use during modern regression testing in CI. Such works include those showing that RV's runtime overhead during testing in CI is likely still too high [38, 53, 68, 70]. To speed up RV in CI settings, Legunsen et al. [71, 73, 112] proposed evolution-aware RV, and techniques that use coarse-grained class-level analysis to re-monitor only specs related to code affected by changes. We discuss and compare with class-level evolution-aware RV throughout this paper. But, FineMOP is the first to leverage fine-grained reasoning, including about the semantics of changes, to further reduce the overhead of evolution-aware RV. Recent work took an orthogonal, complementary approach to reduce instrumentation costs during evolution [39]. FineMOP speeds up monitoring portions of RV overhead, which are beyond the scope of an instrumentation-driven approach. Beyond overhead reduction, some prior work explored using machine learning to classify spec violations as true bugs or false alarms (due to bugs in the specs or RV) [85]. Future work can investigate if violations reported by FineMOP are true bugs.

RV research more broadly. Several surveys and tutorials outline the tremendous progress made in the RV community over the last few decades [9, 11, 15, 29, 30]. But, leveraging software evolution as FineMOP does is a recent research direction. Some earlier directions include those that make monitoring algorithms more efficient [18, 21, 23], improve performance of monitor garbage collection [54, 56, 80], propose new data structures [22, 23, 80, 89], infer specs manually or automatically [32, 66, 67, 88], and develop new frameworks and tools [4, 13, 20, 44, 55, 58, 112].

Regression testing. RTS [28, 34, 35, 41, 43, 64, 69, 78, 87, 91, 92, 99, 101, 110, 115–117, 120] speeds up regression testing by re-running only a subset of tests affected by code changes. RTS inspired evolution-aware RV [71]. Prior work [39, 73, 112] showed that RTS can be used to reduce RV overhead, but RTS by itself does not provide as much speedup as evolution-aware RV techniques. Our results (RQ4 and RQ5) support these earlier findings about RTS vs. evolution-aware RV and show that FineMOP and RTS are complementary. FineMOP is inspired by recent RTS approaches [78, 115, 117] that use finer-grained analyses to speed up RTS, but those techniques are not concerned with RV. Future work can explore combining FineMOP with other regression testing techniques such as test-suite reduction [1, 12, 42, 57, 63, 81, 90, 97, 98, 100, 111, 118, 119] and test-case prioritization [8, 24, 25, 59, 93, 114].

7 Conclusions

FineMOP speeds up RV by using fine-grained analyses to re-monitor fewer unaffected specs as code evolves. Prior evolution-aware RV techniques used coarse-grained, class-level analysis to find a subset of affected specs to re-monitor after code changes. But, these techniques are imprecise and often re-monitor unaffected specs for which there can be no new violation after code changes. FineMOP's more precise analysis considers fewer classes as impacted and re-monitors fewer specs compared to the SoTA. FineMOP outperforms class-level evolution-aware RV (eMOP) and JavaMOP on 1,104 revisions of 68 projects, and finds 99.68% of all new violations found by eMOP. Future work is needed to address this small safety loss.

Acknowledgments

We thank Saikat Dutta, Shinhae Kim, Elaine Yao, and the anonymous reviewers for their help, comments, and feedback. This work is partially supported by an Intel Rising Star Faculty Award, a Google Cyber NYC Institutional Research Award, and the US NSF under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2319472, and CCF-2349961.

References

[1] Marwah Alian, Dima Suleiman, and Adnan Shaout. 2016. Test case reduction techniques-survey. *Int. J. of Adv. Comp. Sci. and App.* 7, 5.

[2] Dale Anson and Andre Kaplan. 2025. jEdit JDiff Plugin. http://plugins.jedit.org/plugins/?JDiffPlugin.

[3] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A Differencing Algorithm for Object-Oriented Programs. In *ASE*.

[4] Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *OOPSLA*.

[5] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, et al. 2005. Combining test case generation and runtime verification. *TCS* 336, 2-3.

[6] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser. 2003. Experiments with test case generation and runtime analysis. In *Abstract State Machines*.

[7] ASM Team 2025. ASM. http://asm.ow2.org/.

[8] Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement learning for test case prioritization. *TSE* 48, 8.

[9] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. 2009. Rule systems for runtime verification: A short tutorial. In *RV*.

[10] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. In *ASE*.

[11] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. 2014. First International Competition on Software for Runtime Verification. In *RV*.

[12] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-Criteria Models for All-Uses Test Suite Reduction. In *ICSE*.

[13] Eric Bodden. 2011. MOPBox: A Library Approach to Runtime Verification. In *RV Demo*.

[14] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *ICSE*.

[15] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A survey of runtime monitoring instrumentation techniques. *arXiv preprint arXiv:1708.07229*.

[16] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2004. A Formal Monitoring-Based Framework for Software Development and Analysis. In *ICFEM*.

[17] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2006. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *RV*.

[18] Feng Chen, Patrick O'Neil Meredith, Dongyun Jin, and Grigore Roşu. 2009. Efficient formalism-independent monitoring of parametric properties. In *ASE*.

[19] Feng Chen and Grigore Roşu. 2007. MOP: An efficient and generic runtime verification framework. In *OOPSLA*.

[20] Feng Chen and Grigore Roşu. 2003. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. In *RV*.

[21] Feng Chen and Grigore Roşu. 2008. *Parametric trace slicing and monitoring*. Technical Report UIUCDCS-R-2008-2977. Computer Science Dept., UIUC.

[22] Christian Colombo and Yliès Falcone. 2016. Organising LTL monitors over distributed systems with a global clock. *FMSD* 49, 1.

[23] Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2016. Runtime Monitoring with Union-Find Structures. In *TACAS*.

[24] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *ISSTA*.

[25] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey Malishevsky. 2004. Selecting a Cost-Effective Test Case Prioritization Technique. *SQJ* 12, 3.

[26] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE*.

[27] eMOP Team. 2025. eMOP. https://github.com/SoftEngResearch/emop.

[28] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: A systematic review. In *ESEM*.

[29] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*.

[30] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2018. A Taxonomy for Classifying Runtime Verification Tools. In *RV*.

[31] Martin Fowler. 2006. Continuous Integration. http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf

[32] Mark Gabel and Zhendong Su. 2012. Testing Mined Specifications. In *FSE*.

[33] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *ICSE Demo*.

[34] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*.

[35] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An empirical evaluation and comparison of manual and automated test selection. In *ASE*.

[36] Kevin Guan. 2025. eMOP safety issue. https://github.com/SoftEngResearch/emop/issues/97.

[37] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster explicit-trace monitoring-oriented programming for runtime verification of software tests. In *OOPSLA*.

[38] Kevin Guan and Owolabi Legunsen. 2024. An In-depth Study of Runtime Verification Overheads during Software Testing. In *ISSTA*.

[39] Kevin Guan and Owolabi Legunsen. 2025. Instrumentation-Driven Evolution-Aware Runtime Verification. In *ICSE*.

[40] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *FSE Demo*.

[41] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*.

[42] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-Demand Test Suite Reduction. In *ICSE*.

[43] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *OOPSLA*.

[44] Klaus Havelund and Grigore Roşu. 2001. Monitoring Java Programs with Java PathExplorer. In *RV*.

[45] Klaus Havelund and Grigore Roşu. 2001. Monitoring Programs Using Rewriting. In *ASE*.

[46] Klaus Havelund and Grigore Roşu. 2002. Synthesizing Monitors for Safety Properties. In *TACAS*.

[47] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *FSE*.

[48] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *ASE*.

[49] Runtime Verification Inc. 2025. JavaMOP. https://github.com/runtimeverification/javamop.

[50] Runtime Verification Inc. 2025. RV-Monitor. https://github.com/runtimeverification/rv-monitor.

[51] JavaMOP Regression Commit 2025. Performance regression that we find in JavaMOP. https://github.com/runtimeverification/rv-monitor/commit/884f9622f.

[52] JavaParser Team 2025. JavaParser - Home. https://javaparser.org.

[53] Omar Javed and Walter Binder. 2018. Large-Scale Evaluation of the Efficiency of Runtime-Verification Tools in the Wild. In *APSEC*.

[54] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. 2011. Garbage Collection for Monitoring Parametric Properties. In *PLDI*.

[55] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE Demo*.

[56] Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Scalable Parametric Runtime Monitoring*. Technical Report. Computer Science Dept., UIUC.

[57] James A. Jones and Mary Jean Harrold. 2001. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage, In ICSM. *TSE* 29, 3.

[58] Murat Karaorman and Jay Freeman. 2004. jMonitor: Java runtime event specification and monitoring library. In *RV*.

[59] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*.

[60] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 1999. Formally specified monitoring of temporal properties. In *ECRTS*.

[61] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2025. Faster Runtime Verification during Testing via Feedback-Guided Selective Monitoring. In *ASE*.

[62] Shinhae Kim, Saikat Dutta, and Owolabi Legunsen. 2026. Valg: A Fast Reinforcement Learning-Based Runtime Verification Tool for Java. In *ICSE Demo*.

[63] Bogdan Korel, Luay Ho Tahat, and Boris Vaysburg. 2002. Model based regression test reduction using dependence analysis. In *ICSM*.

[64] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. 1994. Change impact identification in object oriented software maintenance. In *ICSM*.

[65] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *ICSE*.

[66] Choonghwan Lee, Feng Chen, and Grigore Roşu. 2011. Mining Parametric Specifications. In *ICSE*.

[67] Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, and Grigore Roşu. 2012. *Towards Categorizing and Formalizing the JDK API*. Technical Report. Computer Science Dept., UIUC.

[68] Owolabi Legunsen, Nader Al Awar, Xinyue Xu, Wajih Ul Hassan, Grigore Roşu, and Darko Marinov. 2019. How Effective are Existing Java API Specifications for Finding Bugs During Runtime Verification? *ASE Journal* 26, 4.

[69] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*.

[70] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE*.

[71] Owolabi Legunsen, Darko Marinov, and Grigore Roşu. 2015. Evolution-aware monitoring-oriented programming. In *ICSE NIER*.

[72] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE Demo*.

[73] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for Evolution-Aware Runtime Verification. In *ICST*.

[74] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: Taming reflection to support whole-program analysis of Android apps. In *ISSTA*.

[75] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Reflection-aware static analysis of Android apps. In *ASE*.

[76] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *ECOOP*.

[77] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *SAS*.

[78] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *ISSTA*.

[79] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2.

[80] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *RV*.

[81] Scott McMaster and Atif Memon. 2007. Fault Detection Probability Analysis for Coverage-Based Test Suite Reduction. In *ICSM*.

[82] Patrick Meredith and Grigore Roşu. 2013. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *ASE*.

[83] Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *ASE*.

[84] Mathias Meyer. 2014. Continuous Integration and Its Tools. *IEEE Software* 31, 3.

[85] Breno Miranda, Igor Lima, Owolabi Legunsen, and Marcelo d'Amorim. 2020. Prioritizing Runtime Verification Violations. In *ICST*.

[86] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1.

[87] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*.

[88] Michael Pradel and Thomas R Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE*.

[89] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2013. Optimizing Monitoring of Finite State Properties Through Monitor Compaction. In *ISSTA*.

[90] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*.

[91] Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *ICSM*.

[92] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *TOSEM* 6, 2.

[93] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*.

[94] Runtime Verification Inc. 2025. STHM spec from JavaMOP. https://github.com/SoftEngResearch/tracemop/blob/master/scripts/props/StringTokenizer_HasMoreElements.mop.

[95] Fred B. Schneider. 2000. Enforceable Security Policies. *TISSEC* 3, 1.

[96] Koushik Sen and Grigore Roşu. 2003. Generating optimal monitors for extended regular expressions. In *RV*.

[97] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *FSE*.

[98] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-suite Reduction in Real Software Evolution. In *ISSTA*.

[99] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. In *OOPSLA*.

[100] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *FSE*.

[101] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *ISSRE*.

[102] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *APLAS*.

[103] Sean Stolberg. 2009. Enabling Agile Testing through Continuous Integration. In *Agile Conference*.

[104] William N. Sumner and Xiangyu Zhang. 2013. Comparative Causality: Explaining the Differences Between Executions. In *ICSE*.

[105] STARTS Team. 2025. STARTS—A tool for STAtic Regression Test Selection. https://github.com/TestingResearchIllinois/starts.

[106] TraceMOP Team. 2024. TraceMOP: A Trace-Aware Runtime Verification Tool for Java. https://github.com/SoftEngResearch/tracemop.

[107] Andreas Thies and Eric Bodden. 2012. RefaFlex: Safer refactorings for reflective Java programs. In *ISSTA*.

[108] KD tuition. 2025. mockito is not working #61. https://github.com/gliga/ekstazi/issues/61.

[109] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *ICSE*.

[110] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database Driven Applications. In *ICSM*.

[111] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2.

[112] Ayaka Yorihiro, Pengyue Jiang, Valeria Marques, Benjamin Carleton, and Owolabi Legunsen. 2023. eMOP: A Maven Plugin for Evolution-Aware Runtime Verification. In *RV*.

[113] Nathan York. 2011. Tools for Continuous Integration at Google Scale. https://www.youtube.com/watch?v=b52aXZ2yi08.

[114] Ke Zhai, Bo Jiang, and W. K. Chan. 2014. Prioritizing Test Cases for Regression Testing of Location-Based Services: Metrics, Techniques, and Case Study. *TSC* 7, 1.

[115] Guofeng Zhang, Luyao Liu, Zhenbang Chen, and Ji Wang. 2024. Hybrid Regression Test Selection by Integrating File and Method Dependences. In *ASE*.

[116] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-based and Learning-based Regression Test Selection. In *AST*.

[117] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE*.

[118] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *ISSRE*.

[119] Hao Zhong, Lu Zhang, and Hong Mei. 2008. An Experimental Study of Four Typical Test Suite Reduction Techniques. *IST* 50, 6.

[120] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*.