Using Unit Tests to Validate Mined Specifications: A Study

Anonymous Author(s)

ABSTRACT

Specification mining infers specifications from software, but mined specifications are often spurious and produce many false alarms. Also, manual validation of the numerous mined specifications is often impractical. Automated validation techniques that filter out spurious specifications can help, but little is known on how well automated specification validation works, and how to improve it.

We study Deductive Specification Inference (DSI) with unit tests. DSI validates mined specifications using a simple idea: a specification is spurious if a program that violates the specification is correct. DSI dynamically mutates a program to violate a specification, but it only uses the absence of crashes to check program correctness.

To enable our study, we develop DSI+ by extending DSI to use unit tests, which encode developer intent in oracles that DSI can also use to check correctness. Also, we manually categorize [2001] mined specifications in [six] projects from GitHub. Then, we compare DSI+'s verdicts with our manual categorization. We find that oracles in unit tests help DSI+ find more true specifications. But, we also find that unit tests induce challenges to the accuracy of DSI+. We report on our qualitative analysis of why DSI+ does not always produce the correct verdict and how to improve it in the future.

1 INTRODUCTION

Many software engineering tasks require high-quality specifications, which formally capture expected program behavior. Such tasks include program understanding [7, 35, 42], static analysis [1, 43, 44, 50, 58], and runtime verification [3, 6, 11, 12, 22, 23, 32, 33, 38, 39, 51, 52]. But, developers often do not write specifications due to costs of evolving specifications, lack of expertise in specification languages, and business pressures to roll out features faster [46, 56].

Researchers proposed specification mining [2] to infer specifications from documentation [62, 64], source code [18, 41], program traces [18, 26, 30, 48, 60, 63], etc. Decades of progress on specification mining produced many techniques and miners [4, 10, 13, 17, 34, 40, 47, 49, 53, 61]. See Robillard et al.[54] for a survey.

Two problems in specification mining are that miners often produce false positives, i.e., *spurious specifications*, and that manual validation of the numerous mined specifications is usually impractical [19]. A reason why spurious are produced is that miners are often built on the premise that frequently-occurring code patterns are likely necessary for correctness [14]. Spurious specifications result from incidental patterns and are a challenge, despite the progress on using heuristics and statistics to reduce their occurrence *during* mining [8, 25, 29, 49, 58, 63].

Consider examples of the effects of these problems. Gabel and Su [19] say, "false positives dominated early [...] results" of their specification mining work despite their use of heuristics and statistics. Lee et al. told us informally that `JMiner` technique for specification mining [30] produces many spurious specifications. Also, software engineering tasks that use mined specifications yield many false alarms. For example, Legunsen et al. report a [97.8%] false alarm rate among runtime verification violations of mined specifications [32]. Lastly, we mine [103,264] specifications from [36] open-source projects projects (Section 5.4). It is impractical to expect developers to manually validate that many specs.

Gabel and Su [19] proposed deductive specification inference (DSI) as an automated specification validation technique that does not use heuristics or statistics. Despite its name, *DSI performs specification validation; it is not a specification miner.* DSI is miner agnostic; it takes a program (P), an input, and a mined specification (`s`). DSI gives a likely spurious or likely valid verdict for `s`.

DSI is based on a simple idea: a specification is spurious if a program that violates the specification is correct. DSI dynamically mutates P to violate the specification, producing P'. If P' crashes, then `s` is likely valid. Absent a crash, DSI classifies `s` as likely spurious. DSI uses dynamic rather than static mutation so that it can infer the nature of likely valid specifications. For example, if `s` is "method `a()` always precedes `b()`". If P' crashes when `a()` is called after `b()`, then DSI infers that `b()` may be a postcondition for `a()`.

DSI relies on crashes to approximate correctness, but crashes can be weak [36] or generic baseline [45] oracles—a program may not crash even if DSI's mutation induces an error. Also, DSI was only evaluated on the DaCapo benchmarks [5] and there is no ground truth for its verdicts. So, it is not known how effective DSI is, or how well it would work with unit tests in open-source projects. Yet, unit tests have oracles that DSI can also use, and validating project-specific mined specifications was a major motivation for developing DSI [19]. (Project-specific specifications specify a program, and not its third-party APIs [31, 41, 50, 54, 63, 64].)

We present the first in-depth study of specification validation using unit tests. To enable our study, we developed *DSI+* by extending DSI to use unit tests. We envision DSI+ usage in settings where developers run unit tests, e.g., during regression testing. That setting differs from the one in which DSI was evaluated. DSI+ allows us to investigate specification validation in a modern development environment, and to learn the challenges that arise in doing so.

Many miners can produce complex specifications, but we validate simple specifications of temporal order between two method calls—so-called "two-letter" specifications [18, 19, 27, 57]. We do so because automated validation of two-letter specifications *after* mining is still in its infancy. It is not yet known how to automatically validate more complex specifications, i.e., specification validation research lags specification mining research. We should understand current technology before working to validating more complex specifications. Also, two-letter specifications are a rich class of important temporal safety properties (Section 2). We refer to the first and second methods in a two-letter specification as `a()` and `b()`.

We perform our study in four steps. First, we run DSI+ on all [2001] specifications that a miner produced from developer-written tests in [six] well-tested and widely-used open-source projects. Second, we create a ground-truth dataset by manually inspecting all [2001] specifications and categorizing them as spurious or valid. Third, we compare our categorization with DSI+'s verdicts to

Table 1: Summary of our findings on DSI with unit tests, and their implications.

Findings about DSI with Unit Tests (DSI+)	Implications for Specification Validation
F.1 Methods in spurious specifications often do not share state.	**I.1** Future work should complement specification validation with state checking.
F.2 The return-value handling yields false positives and negatives.	**I.2** New analyses are needed to reduce misclassifications due to return values.
F.3 Order order of assertions yields contradictory DSI+ verdicts.	**I.3** Specification validation with unit tests should be aware of assertion order.
F.4 Expect exceptions are not handled; they induce false positives.	**I.4** New analysis should be developed for dealing with expected exceptions.
F.5 True specifications interfere with validation of spurious ones.	**I.5** A multi-stage validation technique can reduce these false positives.
F.6 DSI+'s mutation pollute state for subsequent test runs.	**I.6** Techniques for flaky test detection may help reduce misclassifications.
F.7 Weak test oracles lead to false negatives	**I.7** Ways to strengthen oracles without introducing false positives are needed.
F.8 Levels of test granularity yield different mined specifications.	**I.8** It seems better to validate use all granularity levels for validation.
F.9 Cross-granularity validation is costly and more inaccurate.	**I.9** Efficient improvement in accuracy of specification validation is needed.
F.10 DSI+ does not work across threads.	**I.10** Ways to detect and avoid validation across threads should be developed.

measure its accuracy, precision, and recall. Lastly, we qualitatively analyze all cases where DSI+ was wrong, to obtain insights that motivate future improvements to DSI+. We make the pragmatic choice to inspect *all* mined specifications from only [six] projects, rather than sampling specifications from many more projects. Doing so allows us to simulate the end-to-end developer experience that would result from using DSI+ today, and study specification validation in depth rather than in breadth.

We find that assertions in unit tests help DSI+ to find [1.8x%] more true specifications (confirmed by our manual inspection) than using crashes alone ([32] specifications vs. [18]). DSI+ has a higher precision ([87.8%]) and recall ([46.3%]) on spurious specifications than on likely true specifications (precision: [35.5%], recall: [23.9%]). So, DSI+ seems better at filtering out spurious specifications than at validating true specifications. But, DSI+' verdicts match ours only in [1087] of the [1987] cases that we conclusively categorize. Also, DSI+ overhead is high (average: [1313x], max: [4954x]).

Our qualitative analysis shows that unit tests induce challenges to the accuracy of DSI+. Table 1 summarizes these challenges and the insights that we gained towards addressing them. Section 2 provides examples of these challenges, and Section 6 provides more details and gives ideas for mitigating some of them. We hope that this paper spurs future research on specification validation, and ultimately reducing the number of specifications that users of specification mining must manually inspect after automated validation.

We make the following contributions:

- ⋆ **Study.** We conduct the first in-depth study of specification validation with unit tests, and the challenges entailed.
- ⋆ **Extension.** We develop DSI+ to enable our study; it extends DSI to also use unit tests when validating mined specifications.
- ⋆ **Insights.** We highlight some ideas for improving DSI+.
- ⋆ **Tools and Data.** We will make publicly available DSI+ and our data, including the manually labeled dataset.

2 EXAMPLES

Crashes vs. oracles. Figure 1 illustrates DSI and DSI+, and shows a problem with DSI's reliance on crashes; `setX()`, `setY()`, `getX()`, and `getY()` have no unintended side-effects. Method `sums()` returns the sum of the most recent arguments to each of `setX()` and `setY()`. Assume that there is only one thread and that there are no overflows and underflows. Say specification `s` is "`getX()` always precedes `add()`". To validate `s`, DSI delays a call to `getX()` which used to be on line 3 to now be after a call to `add()`, i.e., on line 5 in the left snippet. DSI also replaces the return value of `getX()` on line 3 with `0` (Section 3). This mutation will not crash `sums()` if it is called via a `main` method with no `asserts`, so DSI classifies `s` as likely spurious, despite the error (`sums()` now returns `0+3`, not `2+3`). But, the assertion in `testSum()` fails. So, DSI+ (DSI with unit tests) correctly classifies `s` as likely valid. *We use the color coding and comments in* `sums()` *to show DSI+ mutations in the rest of this paper.*

```
1 int sums() {
2   int a = 0; int s;
3   a = 0;// a() @pre
4   s = add(a, getY());// b()
5   getX();// a() @post
6   return s;
7 }
```

```
1 @Test void testSum() {
2   /* Set vars*/
3   setX(2);
4   setY(3);
5   /* check result*/
6   assertEquals(5, sums());
7 }
```

Figure 1: Synthetic code showing benefits of oracles to DSI+.

The kinds of specifications that we evaluate. For reasons discussed in Section 1, we evaluate two-letter specs, which form a rich class of important temporal safety properties, e.g., (1) `a()` establishes a precondition for `b()`; (2) `b()` implements a postcondition for `a()`; (3) `a()` and `b()` are parts of a resource management specification like `acquire/release`, `lock/unlock`, `open/close`, `open/read`, etc. Also, two-letter specifications were easier for non-experts to reason about [57], so they aid our manual inspection. Lastly, two-letter specifications may be composed into more complex ones, e.g., (`getX()`, `add()`) and (`getY()`, `add()`) in Figure 1.

Challenges of using unit tests for specification validation. We evaluate DSI+ using tests at all levels of granularity: test method, test class, and test suite. It is not clear *a priori* what granularity level should be used. Consider a trace obtained from passing tests at the higher-granularity level of test suite (or test class): `T1 = ab|aba|bab`. Here, "`|`" symbols show the boundaries of three tests, `t1`, `t2`, and `t3` at the lower-granularity level of test class (or test method). Say a miner produces specification `s`: "`a()` always precedes `b()`" from `T1`. If `t1` fails when DSI+ calls `a()` after `b()`, then DSI+ classifies `s` as likely valid. But, that would be a misclassification if `t2` and `t3` pass when run alone and also represent correct program behavior.

Using only lower-granularity level tests can also lead to DSI+ misclassification. Consider trace `T2 = ab|a` with the same semantics as before, where there are now two lower-granularity level tests: `t4` and `t5`. A miner produces `s` from `t4`; no specification is mined from `t5`. If `t4` fails when DSI calls `a()` after `b()`, then DSI+ classifies the `s` as likely valid. But that could also be a misclassification if `T2` represents correct program behavior.

Our study also reveals other complications that arise from using unit tests to validate mined specifications. For example, in `T2`

```
@Test void testCodec() throws IOException { ...
    try { ... // a()@pre
    } finally { eof(); // b()
      write(StringUtils.getBytesUtf8(SF)); } // a()@post
    final String str = ...; // read written values
    assertEquals(SF.substring(1), str); } // fail
```

Figure 2: DSI+: likely valid; our inspection: true specification.

```
public String meta(final String txt) {
    switch(s) {
      case 'P': if (false) // a()@pre
          { code.append('F'); }
      case 'S': match = regionMatch(local,n,"SH"); // b()
          isNextChar(local,n,'H'); // a()@post
    }
    return code.toString(); }
@Test void testPF() {assertEquals("FX", meta("PHSH"));} // fail
```

Figure 3: DSI+: likely valid; our inspection: spurious specification.

```
@Test void testFormat() {
    Object test = null; // a()@pre
    assertNotNull("Test Date ", test); // fail
    assertEquals(..., validator.format(test)); // b()
    validator.parse(...); } // a()@post
```

Figure 4: DSI+: unknown; our inspection: spurious specification.

```
public final String encode(String name) {
    name = name; // a()@pre
    name = removeDoubleConsonants(name); // b()
    removeVowels(name); // a()@post
    return name; }
@Test void testMI() {assertEquals("M", encode("Mi"));} // fail
@Test void testMY() {assertEquals("MY", encode("My"));}
```

Figure 5: DSI+: Mixed; our inspection: spurious specification.

above, ab may be a true specification if t5 passed because it expects an exception that is thrown precisely because b() was not called. Section 6 details more challenges that we discovered in our study, plus our thoughts on how future work can tackle some of them.

Examples from open-source projects. In Figure 2, an assertion failure led DSI+ to correctly classify a true specification as likely valid: eof() puts the EOF character in the stream being written. When DSI+ calls the delayed write() after eof(), the stream is corrupted and different from the expected value. The assertion fails.

Figure 3 shows that assertions can mislead DSI+. There, DSI+ replaces isNextChar()'s return value with false, changing the control flow and corrupting the value stored in code so that it no longer matches the expected result of calling meta(). The assertion fails. So, DSI+ classifies the specification as likely valid although there is no restriction on the temporal order of a() and b(). In Figure 4, DSI+ cannot filter out a spurious specification; classifying it as "unknown". An assertion fails right after replacing the return value of a() with null, so DSI+ terminates before calling b().

Lastly, DSI+ produces contradictory verdicts on different tests in Figure 5. DSI+ replaces the return value of a() with the content of name. But, delaying a() means that name is not updated, and contains vowels that it should not. testMI() fails because its input value contains a vowel, but testMY() passes because its input value does not contain a vowel. DSI+ cannot decide on a verdict. Section 5 quantifies the impact of these challenges on DSI+, and Section 6 discusses some ideas for resolving them in the future.

Algorithm 3.1 Algorithm for DSI with Unit Tests

Inputs: P: a program, T: set of unit tests for P, $\mathsf{T}_{map} : \{t \rightarrow S\}$
Each S_i in T_{map} is the set of specifications mined from test t_i

Outputs: $S^f = \{s_1^f, s_2^f, ...\}$, a set of spurious specifications,
$S^t = \{s_1^t, s_2^t, ...\}$, a set of likely true specifications

```
1:
2:  procedure runDSI+(P, T, T_map)                          ▷ Main procedure
3:    results ← {}                                         ▷ Set of Tuples (S^f_test, S^t_test)
4:    for all ℓ in {all, class, method} do
5:      for all test in tests(ℓ, T) do                     ▷ tests(ℓ, T): tests at level ℓ
6:        S^f_test, S^t_test ← validate(test, T_map[test], P)
7:        results ← results ∪ (S^f_test, S^t_test)
8:    return S^f, S^t
9:
10: procedure validate(test, T_map[test], P)               ▷ DSI+ procedure
11:   S^f ← {}, S^t ← {} S^u ← {} ▷ S^f: spurious specifications, S^t: likely
      specifications, S^u: specifications that DSI could not invalidate
12:   for all s in T_map[test] do
13:     out1 ← run(P, test)                                ▷ out* ∈ {PASS, FAIL, CRASH}
14:     out2, trace1 ← instrRun(P, test)                   ▷ 1st instrumented run
15:     out3, trace2 ← instrRun(P, test)                   ▷ 2nd instrumented run
16:     if sanityCheck(s, out1, out2, out3, trace1, trace2) then
17:       exps ← experiments(s, trace)                     ▷ Algorithm 1 in [19]
18:     else
19:       S^u ← S^u ∪ {s}; continue
20:     runExperiments(s, P, test, S^u, S^f, S^t, exps)
21:   return S^f, S^t
22:
23: procedure runExperiments(s, P, I, S^u, S^f, S^t, exps)
24:   for all e : (idx, len) in exps do
        Violate s by delaying call at trace index 'e.idx' by 'e.len' calls
25:     out4, stage ← runExp(P, I, e)
26:     if stage == 3 ∧ out4 == PASS then
27:       S^f ← S^f ∪ {s}; continue
28:     if stage == 0 then S^u ← S^u ∪ {s}; continue
29:     else S^t ← S^t ∪ {s}
30:
31: procedure sanityCheck(s, o1, o2, o3, t1, t2)
32:   if o1 ∈ {FAIL, CRASH} then return false              ▷ Base run crash
33:   if o2 ∈ {FAIL, CRASH} then return false              ▷ Instrument fail
34:   if o2 ! = o3 then return false                       ▷ Nondeterministic
35:   if ! ((s in t1) ∧ (s in t2)) then return false       ▷ Traces omit s
36:   return true
```

3 DSI WITH UNIT TESTS

Algorithm 3.1 shows the DSI+ procedure, which extends DSI to also use assertions and to use multiple test runs. DSI+ takes a program P, a set of tests T, and a map from each test to the specifications that were mined from that test (T_{map}). The outputs are sets of likely spurious (S^f) and likely valid (S^t) mined specifications. It is trivial to extend a miner to produce T_{map}. We use the same version of BDDMiner [17] as in the original DSI work [19]; it is stripped of heuristics. Using this stripped-down miner helps (1) avoid interaction effects with heuristics, and (2) evaluate specification validation

using a basic miner so that we can obtain a ground truth and observe more challenges. DSI+ is miner agnostic; any miner that can produce the kinds of specifications that it checks can be used.

The entry point to Algorithm 3.1 is `runDSI+` (lines 2–8). For all tests at the test method, test class, and test suite granularity levels, `runDSI+` calls `validate` (lines 10–21) to check whether each specification that is mined from that test is likely spurious or likely valid. (See Section 2 for why we mine and validate at all granularity levels.) For each mined specification in its input, the `validate` procedure (lines 10–21) works in three phases: *preamble* (lines 13–16), *experiment selection* (line 17), and *experiment execution* (line 20).

Preamble. Sanity checks in DSI+ ensure that traces obtained from running a test is consistent with the specification, `s`, being validated. If a sanity check fails, DSI+ terminates and the outcome on `s` is `unknown`—`validate` internally tracks `unknown` specifications in a set, S^u, that can be exposed for debugging. The sanity checks run each test thrice (lines 13–15) and compare the outcomes (lines 16, 31–36). The first run (line 16) executes P on test and records whether the program crashes. The second and third runs (line 14 and line 15) execute a version of P that is instrumented to collect a new execution trace for the test; both runs record the test result (`PASS`, `FAIL`, or `CRASH`) and the trace. Procedure `sanityCheck` (lines 31–36) takes these, and returns `false` if a sanity check failed and `true` otherwise.

The `sanityCheck` procedure returns `false` if (1) the first run does not `PASS`, so DSI+ cannot proceed (line 32); (2) the first run `PASS`es but the second run does not, so instrumentation induced a failure (line 33); (3) the test outcomes in the second and third runs differ, so the runs are nondeterministic (line 34); and (4) the traces from the second and third runs do not satisfy `s`, so `s` may be spurious (line 35). The reasons for sanity check failures in our evaluation are concurrency-related issues, state pollution among tests [21], and very few instrumentation failures .

Experiment Selection. DSI+ creates *experiments* that mutate P to violate the specification by delaying `a()` and invoking it after `b()` (line 17). We use DSI's experiment selection procedure, which aims to perform delays on P's execution that are minimally disruptive, i.e., delays should have minimal side effects (Algorithm 1 in [19]). It takes a trace and a specification, and it produces a set of experiments. Each experiment is a pair ⟨idx, len⟩, where delaying the invocation of `a()` at idx by len steps in the trace is minimally disruptive. We highlight three important aspects of experiment selection:

(1) Return values. Method `a()` may return a value which intervening code between `a()` and `b()`, inclusive, may rely on. To reduce crashes due to missing return values of the delayed method, DSI+ fills in the nearest value of a variable of the same type. If no such variable exists, then DSI+ uses a default value. Replacing return values needs improvement in DSI+ (Section 6.1). But, the current scheme works quite well in most cases even when in [1597] of the [2001] specifications that we inspect, `a()` does not return `void` (Section 5.4).

(2) Locks. Experiments may require delaying `a()` and invoking it at a later location, after needed locks were released. DSI+ records such locks and tries to reacquire them before invoking `a()`. Runs that deadlock are terminated after a timeout. We find other concurrency-related problems that DSI+ cannot handle (Section 6.6).

Table 2: Projects that we inspected. **KLoC**: thousands of lines of code, **TC**: no. of test classes, **TM**: no. of test methods, **SC**: statement coverage, **BC**: branch coverage, **USE**: no. of clients.

ID	Project	SHA	KLoC	TC	TM	SC	BC	USE
P1	jtar [24]	4b669	1.0	2	8	.75	.67	24
P2	codec [9]	b959a	23.9	58	936	.97	.93	10618
P3	validator [59]	72734	16.7	70	554	.86	.74	1387
P4	exec [?]	2ca7c	3.6	13	89	.71	.61	753
P5	convert [?]	8f8bf	5.4	9	160	.76	.72	852
P6	fileupload [15]	55dc6	4.8	9	34	.81	.77	2312

(3) Multiple call sites. Traces can have multiple pairs of `a()` and `b()` with different call sites; unique call sites are called *usage scenarios*. DSI+ creates an experiment per usage scenario.

Experiment Execution. DSI+ uses the outcome of running experiments to classify a specification (lines 20, 23–29). DSI+ runs experiments like so: at location idx, it captures the calling context of `a()` in a *thunk* (i.e., a function object) and attempts to force the execution of the thunk at the program point that is offset at len from idx. If the thunk runs successfully and the test `t` does not `FAIL` or `CRASH`, then specification `s` is likely spurious—it may not be necessary for P's correctness. So, DSI+ puts `s` in S^f (line 27). Otherwise, DSI+ uses the stage of `t` `FAIL` or `CRASH` to classify `s`:

- **Stage 0:** After delaying `a()`, before calling `b()`—cannot run experiment and `s` cannot be validated; add `s` to S^u (line 28).
- **Stage 1:** While running `b()`, after delaying `a()`—method `a()` likely establishes a precondition for `b()`.
- **Stage 2:** After `b()`, but before `a()` is called–`b()` puts the program in a state in which `a()` cannot be run.
- **Stage 3:** After `b()` and `a()` run in that order—violating `s` breaks P.

Stages 1, 2, or 3 mean that `s` is a likely valid; put it S^t (line 29).

DSI+ Implementation. We build DSI+ on top of Gabel and Su's DSI prototype, after we used their DSI prototype to reproduce their results [19]. We developed Maven plugins that embody different components of DSI+: trace collection, specification mining (which can be customized to use any miner), the validate procedure from Algorithm 3.1, and a test failure listener (which provides feedback on DSI+-generated experiments and tracks the stages at which those experiments failed). For our experiments, we configure our plugin to exclude methods defined in a test class from the specifications.

4 METHODOLOGY

4.1 Study Setup

We answer the following research questions:

RQ1 What is the accuracy, precision, and recall of DSI+?

RQ2 How much do test oracles contribute to DSI+ verdicts?

RQ3 What is the runtime overhead of DSI+?

RQ4 How do specifications and tests interact in DSI+?

We run DSI+ on [six] open-source projects and recorded its verdicts and overheads. Then we manually inspect [2001] specifications that were mined from these projects, to obtain a "ground truth" for evaluating DSI+, and to generate more data about specification validation. Finally, we compare DSI+ with our manual validation and analyze data from DSI+ and the manual inspections.

The rest of this section describes our experimental settings. We answer RQ1—RQ4 in Section 5 and discuss the results in Section 6.

Projects Studied. Table 2 shows details on the projects that yielded the specifications that inspect; the caption describes the columns. We select these [six] projects from a larger set of [36] open-source projects that we run DSI+ on. [25] of those [36] projects were used in recent evaluations of runtime verification [32] or regression testing [55, 65]. We reuse them as they are likely to have tests that run and are less susceptible to instrumentation failures (techniques that they were evaluated on perform instrumentation). The other [11] projects were selected from GitHub based on [Ayaka to describe]. Appendix **??** provides data about all [36] projects.

We select [six] of [36] based on a combination of their (1) statement coverage (at least 70%); (2) most recent commit (being after December [2019]); (3) number of clients on Maven Central [37] (a higher number increases the chance that true specs that we find will be useful); and (4) having a number of mined specifications that we could inspect within our time budget (for reasons given in Section 1, we inspect *all specifications in few projects*, rather than *few specifications from many projects*). All experiments were run on [Ayaka to finalize]. [Include note about how exec got stuck on some tests so we had to cut them out.]

Inspection Process, Data, and Duration. We assign a co-author as the primary inspector of all specifications mined from each of the [six] projects. The primary inspector's tasks are to (1) manually determine the necessity of each specification for correctness (i.e., is it a true specification?); (2) compare their manual findings with DSI+'s; (3) determine why manual findings differed from DSI+'s; (4) find problems that arise when using unit tests to validate mined specifications; and (5) record extensive information (including relevant code) about each specification so that any other person can check their work only by looking at their records. Our replication package contains `json` files with these data, plus schema for validating their structural consistency, and scripts to process them.

After the primary inspector finished the first round of inspections, they made a pull request which a secondary inspector (another co-author) checked thoroughly and sent for revision. The secondary inspector double-checked the specification categorization and pointed out ways to improve the recorded information. Multiple rounds of updates and reviews were made before the pull request was merged. Finally, the primary and secondary instructors had several meetings to try to decide on specifications that were difficult to figure out earlier. We estimate that it took **8 person months** to complete our inspection process, from creating the `json` templates and schemas to inspection, review, and resolution.

5 RESULTS

5.1 RQ1: Accuracy, Precision, and Recall

We compare DSI+ verdicts with our manual inspection decisions. Table 3 shows DSI+ and manual inspection outcomes for all [2001] specifications. There, rows show DSI+ verdicts: LV means likely valid, **LS** means likely spurious, U means "unknown" (DSI+ could not classify them), Mixed means that DSI+ verdicts differ across multiple tests, and Σ is the sum across all *manual* verdict categories. Columns in Table 3 show our manual inspection decisions: TS are true specifications, NS are spurious, NBP are those that DSI+ *cannot* violate because `a()` transitively calls `b()` (it means "no-break-pass" [19]), US are those where we could not conclude, and Σ is the sum across all *DSI+* verdict categories. Unlike DSI+, our manual inspection revealed two kinds of true specifications: Must are always necessary for P's correctness, while May are only necessary for P's correctness under certain conditions. We provide examples and describe these two kinds of true specifications in Section 6.6.

Table 3: DSI+ vs. manual inspection outcomes across all projects.

DSI+ \ Man	TS Must	TS May	NS	NBP	US	Σ
LV	32	21	90	6	2	151
LS	17	37	564	24	1	643
U	15	7	425	468	4	919
Mixed	34	60	183	4	7	288
Σ	98	125	1262	502	14	2001

Table 4: DSI+ vs. manual inspection outcomes for all Mixed cases.

DSI+ \ Man	TS Must	TS May	NS	NBP	US	MC	Σ	Acc [%]
LV-**LS**	22	56	91	0	3	169	172	98.3
LV-U	6	1	6	0	0	7	13	53.8
LS-U	1	0	73	4	3	77	81	95.1
LV-**LS**-U	5	3	13	0	1	21	22	95.5
Σ	34	60	183	4	7	274	288	95.1

Table 5: Accuracy, precision, and recall of DSI+.

ID	Σ	Acc	Acc [%]	Precision[%] **TP**	Precision[%] **TN**	Recall[%] **TP**	Recall[%] **TN**
IP1	11	7	63.6	100.0	66.7	25.0	66.7
IP2	442	267	60.4	30.2	93.4	50.0	31.4
IP3	659	383	58.1	55.2	97.4	47.1	50.9
IP4	331	186	56.2	35.3	72.9	10.5	63.3
IP5	210	84	40.0	32.4	63.0	15.2	33.0
IP6	334	206	61.7	22.7	97.4	23.8	40.3
ALL	1987	1133	57.0	35.5	87.8	23.9	46.3
AVG	331.2	188.8	56.7	46.0	81.8	28.6	47.6

Colored cells in Table 3 show how often DSI+ was correct. Summing those values shows that DSI+ was only correct [44.5% (885÷1,987)] of the time. [(We ignore cases that we could not decide.)] Four other observations from Table 3: (1) DSI+ incorrectly classifies [65% (98÷149)] of spurious specifications as likely valid; (2) DSI+ correctly classifies [88% (564÷642)] spurious specifications, so it seems better at filtering out spurious specifications than at confirming likely valid ones; (3) DSI+ correctly classifies only [51% (468÷915)] of the U cases as NBP; and (4) DSI+ produced Mixed verdicts [14% (281÷1,987)] of the time. Section 6 shows that $U \setminus NBP$ cases and Mixed cases are mostly due to challenges of specification validation with unit tests.

Table 4 shows how much "signal" is in the "noise" of the Mixed cases. Rows are DSI+ outcomes. LV-**LS** means that DSI+ classifies a specification as LV using some tests and **LS** using other tests; other rows can be similarly translated. The first four column headers mean the same as in Table 3. MC shows how many Mixed cases have at least one correct verdict. There is no clear pattern that a user can use to make (heuristic) decisions without manual inspections. For example, DSI+ results also show no useful pattern when we

take the majority votes to be correct [(Appendix ?? Table 16)]. Also, the fact that mixed cases often contain at least one correct verdict is only obvious in retrospect, after manual inspection. Mixed cases are a problem for DSI+ that DSI's evaluation [19] did not show. [how many cases are mixed among all 36 projects?] In the rest of this section, we treat Mixed cases as DSI+ being wrong and we omit the small number of cases that we could not manually decide.

We use these quantities to compute DSI+ accuracy, precision, and recall relative to our manual inspection shown in Table 5: (1) **TP** (true positives): DSI+ classifies as likely valid and manual found a true specification; (2) **FP** (false positives): DSI+ classifies as likely valid and manual found a spurious specification; (3) **TN** (true negatives): DSI+ classifies as likely spurious and manual found a spurious specification; (4) **FN** (false negatives): DSI+ classifies as likely spurious and manual found a true specification; (5) **UP**: DSI+ classifies as likely valid, but manual found NBP; (6) **UN**: DSI+ classifies as spurious but manual found NBP; (7) $\mathbf{U}^{T}$: DSI+ could not validate, and manual found NBP; (8) $\mathbf{U}^{fn}$: DSI+ could not validate, but manual found a true specification; (9) $\mathbf{U}^{tn}$: DSI+ could not validate, but manual found a true specification; (10) $\mathbf{M}_T$: DSI+ gave mixed verdicts, but manual found a true specification; and (11) $\mathbf{M}_N$: DSI+ gave mixed verdicts, but manual found a spurious spec. We show only the results of our computation in Table 5; Table **??** in the Appendix shows a per-project view of all quantities.

We use these formulas for multi-class classification to compute the accuracy (Acc [%]), macro-Precision (Precision[%]), macro-recall (Recall[%]), and macro-F1 score ([macro?]) [20] of DSI+ in Table 5: Acc = $[addformula]$, Acc [%] = $[addformula]$, Precision[%] **TP** = $\frac{\mathbf{TP}}{\mathbf{TP+FP+UP}}\times 100$, Recall[%] **TP** = $\frac{\mathbf{TP}}{\mathbf{TP+FN+U}^{fn}\mathbf{+M}_T}\times 100$, Precision[%] **TN** = $\frac{\mathbf{TN}}{\mathbf{FN+TN+UN}}\times 100$, Recall[%] **TN** = $\frac{\mathbf{TN}}{\mathbf{FP+TN+U}^{tn}\mathbf{+M}_N}\times 100$, F1 score **TP** = $[addformula]$, formulas are applicable here because DSI+ classifies specifications into more than two categories (LV, **LS**, U, Mixed).

Intuitively, higher Precision[%] on **TP** means that more specifications that DSI+ classified as likely valid were true specifications, and higher Precision[%] on **TN** mean that more specifications that DSI+ classified as likely spurious were spurious specifications. Similarly, higher Recall[%] on **TP** means that DSI+ identified more true specifications as likely valid, and a higher Recall[%] on **TN** dsi identified more spurious specifications as likely spurious. So, the numbers in Table 5 show that the accuracy (Acc [%]), precision (Precision[%]), and recall (Recall[%]) of DSI+ are quite low, but its precision **TN** specifications is higher. Thus, subject to the generality of our findings, we conclude that DSI+ needs more improvements before it can become an effective technique for validating whether mined specifications are true specifications, but it already shows promise filtering out spurious mined specifications. Section 6 discusses the problems of specification validation using unit tests, and some ideas for how to solve them.

5.2 RQ2: Test Oracles

We measure the contributions of test oracles to DSI+ verdicts, compared with exceptions/crashes, and report our findings on the impact of the weak oracles that we observed during our inspection.

Assertions vs. Exceptions Table 6 shows the contributions of assertions vs. exceptions to DSI+ verdicts on the true positive (**TP**) specifications. The "Assert" shows tests that failed, "Except" shows executions that crashed, "Σ" sums the first two columns, "% Assert" is the proportion of failed assertions in the sum (% Assert = $\frac{Assert}{Total}\times 100$), DSI+ validates a specification on all tests that mined it, so sums of Assert and Except are greater than **TP** specifications. To normalize, we compute % Assert p.spec as the ratio of assertion failures per **TP** specification (% Assert p.spec = $(\sum_{spec=1}^{n}\frac{Assert(spec)}{Assert(spec)+Except(spec)}\times 100)\div n$) where n is the # of **TP** specifications and % Assert p.spec is the number of assertions that failed per such specification. For example, suppose DSI+ verdict is LV for **TP** specification X using tests T1, T2, T3, but an exception is thrown with T1, and an assertion fails with T2 and T3. Suppose also that **TP** DSI+ verdict is LV for specification Y tests T4 and T5, but an exception is thrown with T4, and an exception fails with T5. Then, "% Assert" is $\frac{2+1}{5}$ ($\frac{T2+T3+T5}{T1+T2+T3+T4+T5}$) and "% Assert p.spec" is $\frac{\frac{2}{3}+\frac{1}{2}}{2}$ (average of $\frac{T2+T3}{T1+T2+T3}$ for X, and $\frac{T5}{T4+T5}$ for Y). ["Only Assert" shows the number specifications that were validated only by assertions.]

Table 6: Assertions vs exceptions/crashes for all true positives.

ID	Assert	Except	Σ	Only Assert	% Assert	% Assert p.spec
IP1	0	3	3	0	0.00	0.00
IP2	14	7	21	9	66.67	69.23
IP3	172	147	319	8	53.92	48.28
IP4	3	11	14	1	21.43	16.67
IP5	0	12	12	3	0.00	0.00
IP6	0	10	10	0	0.00	0.00
ALL	189	190	379	21	49.80	-
AVG	31.5	31.7	63.2	3.5	23.70	22.40

Table 7: Impact of Weak Oracles.

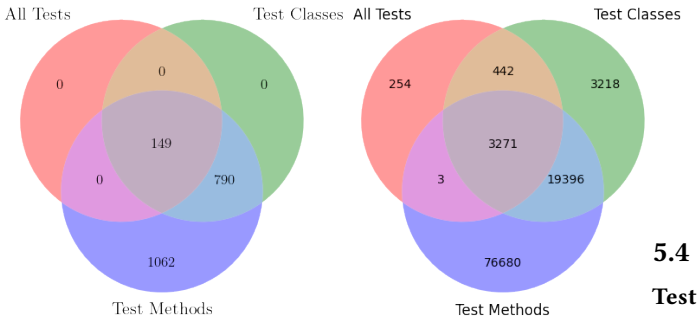
ID	TS	NS	US	Σ	% TS	% NS	% US
IP1	0	0	0	0	0.00	0.00	0.00
IP2	1	16	1	18	5.56	88.89	5.56
IP3	4	67	0	71	5.63	94.37	0.00
IP4	5	32	0	37	13.51	86.49	0.00
IP5	3	0	0	3	100.00	0.00	0.00
IP6	2	44	0	46	4.35	95.65	0.00
ALL	15	159	1	175	8.57	90.85	0.57
AVG	2.5	26.5	0.2	29.2	21.50	60.90	0.90

The results in Table 6 show that assertions can be useful for specification validation even when the program does not crash. On average across all [six] projects, assertions contribute almost equally as exceptions. [21] specifications would not have been validated if it were not for assertions. Not all projects benefitted from assertions and assertions contribute more than exceptions in [two] of [six] projects. So, assertions should be used to complement exceptions/crashes.

Weak Oracles If DSI+ mutation causes a program to be incorrect without crashing, then DSI+ may classify a valid specification as likely spurious if the test oracle is weak. Table Table 7 shows a breakdown of the [175] specifications for which we observed a weak oracle in at least one test that DSI used for validation, and the impact on DSI+. We do not inspect all tests that DSI+ used to validate each specification, rather we inspect at least one test per verdict. So, there could be more weak oracle cases in our dataset. The column headers mostly follow those in Table 3. "% TS", "% NS", "%

Table 8: Overhead of DSI+. **S#**: number of specifications. **T#**: number of tests. **SM[s]**: time in seconds for mining. **bl(s)**: time in seconds to run all tests without DSI+. **Run Time[h]**: total DSI+ time in hours. **Wall**: runtime in wall clock time. **CPU**: sequential runtime. **OH[x]**: Run Time[s] $\div$ $bl(s)$.

ID	S#	T#	SM[s]	bl(s)	Run Time[h]		OH[x]	
					Wall	CPU	Wall	CPU
P01	11	11	9	1	0.0	0.0	39	153
P02	445	995	3861	23	1.5	23.1	231	3616
P03	664	625	254	3	4.1	328.2	4954	393813
P04	391	103	176	74	1.1	18.1	53	879
P05	210	170	73	1	0.6	42.7	2041	153595
P06	337	44	302	2	0.3	3.7	565	6710
SUM	2058	1948	4675	104	8	416	-	-
AVG	343	324.7	779.2	17.3	1.3	69.3	1313	93127



(a) [Six] inspected projects. **(b)** All [36] projects.

Figure 6: Mined specifications at different levels of granularity.

US" are percentages of weak oracles that we found while inspecting a, respectively, true, spurious, or "unknown" specifications. Observe that [8.5%] of identified weak oracle cases are in true specifications, while most cases [90.5%] are in spurious specifications. That is, the oracles seem adequate for specification validation but if the oracles had been stronger then DSI+ may classify more spurious specifications as being likely valid.

5.3 RQ3: Runtime Overhead

Measuring the runtime overhead gives a sense of (1) DSI+ costs relative to the two weeks of CPU time reported for running DSI on the DaCapo benchmarks [19], and (2) how well DSI+ may scale to bigger projects than those that we evaluated. Table 8 shows the overhead of DSI+; the caption describes the column headers. We measure mining time separately because specifications are inputs to DSI+. Clearly, the overheads in Table 8 are very high. Considering that we evaluate relatively small open-source projects, we anticipate that larger projects will be much more costly. In fact, the average overhead on the [31] projects that we did not inspect is [add]. Future work should investigate how to speed up DSI+ without degrading its classification accuracy.

Table 9: Return values of `a()` and `b()` across all inspected projects.

b() \ a()	V	NV	Σ
V	276	128	404
NV	539	1058	1597
Σ	815	1186	2001

Table 10: Return values of `a()` and `b()` across all inspected projects (true specs).

b() \ a()	V	NV	Σ
V	42	20	62
NV	128	33	161
Σ	170	53	223

Table 11: Return values of `a()` and `b()` across all inspected projects (spurious specs).

b() \ a()	V	NV	Σ
V	195	83	278
NV	358	626	984
Σ	553	709	1262

5.4 RQ4: Specification and Test interaction

Test Granularities. We use all test granularity levels during DSI+ experiments because it is not clear *a priori* what is the right granularity to use (Section 2). The Venn diagram in Figure 6a shows that method-level mining and validation would have sufficed the projects and their versions that we evaluate. (All tests that were mined at higher levels of granularity were also mined at the method level). But Figure 6b shows the same specification-to-granularity relationship for all [36] projects that we ran DSI+ on. There, many specifications were mined at higher-granularity levels that were not mined at the method level. So, the method-level sufficiency among the [six] inspected projects is co-incidental. There is no "**best**" granularity level to use for specification validation, all three levels can produce unique specifications and should be used.

Multiple call sites of `a()` and `b()`. The methods in a specification can have multiple unique call sites—called "usage scenarios" in DSI+ (Section 3)—in a trace. So, the DSI+ experiment that is created for each scenario can interfere with each other. Worse, the results from these experiments can contradict one another since the DSI+ mutation in an earlier experiment can cause a state corruption-that affects the outcome of a later experiment. We measure the frequency of mutiple usage scenarios in our experiments. [[306] specifications had multiple usage scenarios, out of which [109] were miscategorized.] . Multiple usage scenarios are more common at higher granularity levels ([proportion?]), but they also occur at test method level too ([proportion?]). We discuss more on the challenge of multiple usage scenarios and some insights on how to resolve them in Section 6.

Return Values. We investigate the impact of return values to DSI+. The results in Table 9 show that among all specifications, `a()`

Table 12: Return values of `a()` and `b()` across all inspected projects (FP specs).

b() \ a()	V	NV	Σ
V	30	5	35
NV	33	22	55
Σ	63	27	90

Table 13: Return values of `a()` and `b()` across all inspected projects (FN specs).

b() \ a()	V	NV	Σ
V	13	2	15
NV	38	1	39
Σ	51	3	54

```
1  @Test void testSafe() throws Exception { // LS
2    String plain = ...; String encoded = plain; // a()@pre
3    assertEquals(plain, encoded);
4    assertEquals(plain, urlCodec.decode(encoded)); // b()
5    urlCodec.encode(plain); } // a()@post
6  @Test void testBasic() throws Exception { // U
7    String plain = ...; String encoded = plain; // a()@pre
8    assertEquals("Hello+there\%21", encoded); // fail
9    assertEquals(plain, urlCodec.decode(encoded)); // b()
10   urlCodec.encode(plain); } // a()@post
```

Figure 7: DSI+ is misled by return-value replacement.

does not return `void` [[59.2]%, ([1186]÷[2001])] of the time. In that table, columns show how often `a()` returns a `void` (V) or a non-`void` (NV) value, and rows show the same for `b()`. Also, [86]% of [2001] specifications have at least one method that returns a non-`void` value. [Describe the true specifications, spurious specifications, false positives, and false negatives cases.] These results show the importance of the correctness return-value replacement. Section 6.1 discusses current challenges in return-value replacement and some ideas for resolving them. [Validating specifications for which `a()` returns `void` is non-trivial as well: out of the [815] specifications which `a()` returns `void`, [63] were false positives, and [51] were false negatives. The majority ([31] specifications) of the false negatives were due to weak oracles, discussed in Section 6.4.]

6 ANALYSIS OF RESULTS

To find insights about how to improve DSI+, we put our observations from manual inspection into the six categories in Table 14. There, **%NS** = $\frac{\mathbf{NS}}{\Sigma - (\mathbf{NBP})}$ is the proportion of manually confirmed spurious specifications in each category and Σ is the number of specifications that we observed. We discuss some insights and ideas for improvement, based on some of the observations.

6.1 Needed: Better Return-Value Replacement

DSI+ replaces return values of `a()` with the value of the nearest same-typed variable, or the type's default value (Section 3). Replacing `a()` with an unexpected value can lead to unintended assertion failures and exceptions (false positives), and replacing `a()` with an expected value can hide incorrect state changes caused by DSI+'s mutation (false negatives). For example, in Figure 7, two tests check the specification `s` with `encode()` and `decode()` as `a()` and `b()`. We manually found `s` to be spurious. Using `testSafe()`, DSI+ replaces `a()`'s return value with the value of `plain`, which is also the value that `a()` returns. So, the test passes and DSI+ correctly classifies `s` as likely spurious. But using `testBasic()`, `plain`'s value is also the replacement but it is *not* the expected value. So, the assertion on line 9 fails, and DSI+ classifies `s` as likely valid. There are two problems: (1) contradictory DSI+ verdicts, and (2) without `testSafe()`, DSI+ would miss the correct verdict entirely.

An improved return-value replacement scheme could be one that is based on a static analysis of how `a()`'s return value, henceforth called **`AReturn`**, is used. If **`AReturn`** is assigned to a variable, then the thunk (Section **??**) could also capture that variable. Then, when `a()` is later called, its return value is also assigned to the captured variable. Doing so will help in cases like Figure 5 (but not Figure 7). Developing such an improved scheme will involve dealing with the challenges of (1) multiple (and possibly long) def-use paths

Table 14: Categorization of observations from manual inspection. **`AReturn`**: the replacement value used when `a()` is delayed.

Category	Description	%NS	Σ
Return	`a()` returns void	76	807
	`AReturn` ≠! = test's expected value	96	313
	`AReturn` == null causes NPE	85	253
	`AReturn` unchecked	98	125
	`AReturn` == test's expected value	94	94
	`AReturn` == `a()`'s return is default value	100	89
	Discards `a()`'s return value	52	72
	`AReturn` guards path; `b()` not called	97	40
Relation	`a()` and `b()` are stateful but unrelated	100	307
	`a()` XOR `b()` does not alter *P* state	95	290
	`a()` XOR `b()` is a pure setter	86	221
	`a()` and `b()` are connected	100	174
	Neither `a()` nor `b()` alters *P* state	100	89
	`a()` and `b()` are both pure setters	100	43
	`a()` and `b()`'s caller form true specification	100	12
	`a()` and `b()` modify disjoint part of *P* state	100	2
Exception	Test expects an exception	66	269
	Expected exception not thrown; test fails	70	58
	Exception swallowed in catch block	66	30
	Expects exception	50	20
	Delaying `a()` causes unexpected exception	80	10
Oracle	Test has weak oracle	84	304
	Order of assertions affects DSI+ outcome	99	167
Delay	DSI+ pollutes state for tests	83	87
	Delayed call of `a()` restores *P*'s state	97	86
	Delay causes assertion to fail	78	39
	Delay causes a timeout	44	9
	Delay corrupts state for `b()`	75	8
Misc	`a()` and `b()` called in different threads	74	73
	Dynamic dispatch slowed down inspection	91	70
	Bug causing sanity-check-failure	95	54
	A NBP that goes beyond method-call chains	100	42
	Javadoc confirms manual inspection	66	12
	Configurations influenced DSI+ outcome	90	12
	`b()` calls `a()`	88	10

```
1 @Test void testZeroSum() {
2   assertFalse(false); // a()@pre
3   try { routine.calculate(zeroSum); // b()
4     routine.isValid(zeroSum) // a()@post
5     fail("Zero Sum - expected exception");
6   } catch (Exception e) {
7     assertEquals("Invalid code", e.getMessage()); } }
```

Figure 8: Expected Exception misleads DSI+ towards **FP**.

starting at the original location of `a()`; (2) changes in variable scope between when the thunk is captured, and when `a()` is eventually called (the captured variable may now be out of scope), etc. [Run random replacement? After deadline: how many cases benefit?]

6.2 Use(?): Relationship of `a()` and `b()` with State

The "Relation" row in Table 14 show that when `a()` and `b()` do not intersect on the state of program P that they touch, the specification `s` is almost always spurious. For example, our discovery that `a()` and `b()` in Figure 3 do not intersect on state.[Out of the [307], [27] were Mixed specifications that we found spurious.] Future research could investigate how to leverage this fact, for example to reach verdicts when DSI+ produces Mixed verdicts. Such improvement should be used to complement DSI+'s experiments, rather than to heuristically replace it. A reason is that `s` may be spurious even if `a()` and `b()` intersect on state, as in Figure 7. There both `public` methods in the specification are called on the same string, but it is not required to always `encode()` and `decode()` the same `String` or to do so in some order.

6.3 Needed: Handling Expected Exceptions

DSI+ inherits DSI's use of exceptions for finding likely valid specifications. But, tests with expected exceptions can mislead DSI+. We use the spurious specification `s` in Figure 8 to illustrate. There, `b()` throws an exception regardless of where `a()` is called but the test passes because it expects the exception. So, by the `runExperiments` procedure in Algorithm **??** (lines **??**—**??**), DSI+ wrongly classifies `s` as likely valid. We found [52] specifications that were wrongly classified due to DSI+'s lack of handling of expected exceptions. Also, in Figure 8, DSI+'s mutation can cause an exception to be thrown other than the one thrown when not running with DSI+. But the two exceptions would be indistinguishable to DSI+ because the `catch` clause uses the most general exception class in Java.

Future work could capture the stack trace and P's state at the point where the exception is being thrown during sanity checks, and compare them with those obtained during DSI+'s mutation. If these are equal, then the specification is likely spurious or such tests are not providing good signals to DSI+. Better handling of expected exceptions will require (1) detecting beforehand that DSI+'s mutations will occur across the boundaries of a test that expects exceptions; and (2) ensuring that the "exception capture and compare" mechanism is not influenced by the outcome of the tests or introduce new false positives or false negatives. Gabel and Su's DSI experiment selection algorithm (Algorithm 1 in **??**) could be a good basis for a detection technique.

6.4 Needed: Handling Oracle-Related Problems

The efficacy of specification validation with unit tests depends on the quality and order of test oracles. For example, Figure 9 shows

```
1  int execute(CommmandLine cl) {
2    ... // a()@pre
3    streams.start(); // b()
4    streams.setProcessErrorStream(...); ... } // a()@post
5  @Test void testStreams() throws Exception { // LV
6    setStreamHandler(new StreamHandler(baos, baos, fis));
7    final int exitValue = execute(cl);
8    assertTrue(baos.toString().indexOf("read") > 0); } // fail
9  @Test void testExecute() throws Exception { // LS
10   final int exitValue = execute(cl);
11   assertEquals("FOO..", baos.toString()); }
```

Figure 9: Weak Oracles mislead DSI+ towards **FN** and Mixed.

```
1 @Test void testAboveThreshold() { ...
2   File sl = null; // a()@pre
3   assertNotNull(sl); // fail
4   delete(); // b()
5   getStoreLoc(); } // a()@post
```

Figure 10: The order of assertions mislead DSI+ towards $\mathbf{U}^{fn}$.

```
1 a();
2 getA(); // true specification with a()
3 b();
```

Figure 11: Synthetic code motivating multi-stage validation.

two tests used by DSI+ to validate a true specification `s`; it specifies the necessity of propagating `stderr` output from a subprocess to a stream. Test `testStreams()` checks the output of a bash script (not shown) which prints to `stderr`, and DSI+'s mutation corrupt that output, leading to a likely valid classification. But, `testExecute()` checks the output of another script that does *not* print to `stderr`. So, `testExecute()`'s oracle is weak for validating `s`, and it causes DSI+ to wrongly classify `s` as likely spurious.

Figure 9 shows two oracle-related problems: (1) a weak oracle caused contradictory verdicts, and (2) if `testStreams()` did not exist, then `s` would be wrongly classified as likely spurious (not Mixed). In fact, `testStreams()` was the *only* test (out of [39]) that classified `s` as likely valid. We found [32] false negatives due to similarly weak oracles. Other forms of weak oracles that we observed involved tests that pass even when `a()` or `b()` is not called, and those that perform a[simple containment/existence check], which is not enough to capture state corruption caused by DSI+'s mutation. So, detecting and strengthening tests that are too weak to catch differences in P's state during DSI+ mutation could help.

Another oracle-related problem is that the order of assertions can mislead DSI+. For example, DSI+'s mutation in Figure 10 fails the `assertNotNull()` on line 3 even before `b()` is called (DSI+ replaces `a()`'s return value with `null`). So, DSI+ classifies this true specification as "unknown". We found [155] such misclassifications that were due to the order of assertions. One way to reduce noise caused by order of assertions could be to delay not only `a()` but also all "affected" assertions on `a()`'s return value that are checked before calling `b()`. Some challenges would be how to (1) let DSI+ know in advance which assertions will be affected by either `a()`'s return value or state changes that `a()` would have induced if it was not delayed, and (2) perturb the program minimally (Section **??**) when multiple statements are being delayed at once.

6.5 Needed: Fixing Mutation-induced Problems

If the violation of a true specification, `s1`, causes a test failure while validating a spurious specification, `s2`, then DSI+ wrongly classifies `s2` as likely valid. We say that `s1` *interferes* with the validation of `s2`.

```
1 @Test void testSpecifiedRepository() throws IOException {
2   tempDir.mkdir(); File sl = ... // a()@pre
3   assertEquals(..., sl.length()); // fail
4   item.delete(); // b()
5   getStoreLoc(); // a()@post
6   assertTrue(tempDir.delete()); } // cause sanity-check-fail
```

Figure 12: DSI+ can misclassify due to test state pollution.

For example, in Figure 11, if s1 is (a(), [getA()]) and the s2 being validated is (a(), b()). Delaying a() violates s1 and can cause test failures or exceptions, even when the order of a() and b() does not matter. We observe [X] specifications that suffer from interference.

A new *multi-stage approach* for specification validation can help reduce interference. Using Figure 11, let's say s2 is the input to DSI+. In the first stage, DSI+ evaluates s1. If DSI+ classifies s1 as likely valid, then it can check s3=(getA(), b()) in a second stage. If s3 is likely valid, then DSI+ can conclude that its input s2 is also likely valid (by transitivity) without violating the true specifications1. But, if s3 is likely spurious, then DSI+ can delay *both* a() and getA() to be after after b() in a third stage. Note that s1 and s3 are not inputs to DSI+ in this example, so DSI+ need to compute them on the fly. Some challenges to be resolved in multi-stage DSI+: (1) s1 and s3 can be long call chains, so efficient algorithms will need to be devised to avoid brute-force; (2) multi-stage DSI+ can introduce redundant computations while validating a set of mined specifications, so those computations will need to be shared.

Delaying a() during DSI+ mutation of one specification can prevent correct validation of other specifications. Recall that DSI+ uses each test to validate all specifications that were mined from its trace (Section 3). Figure 12 shows how state pollution can result from delaying a() while running a test on one specification. There, the assertion on line **??** causes the test to fail right after delaying a(). Method b(), which deletes the temporary file tempDir, does not get called. So, when DSI+ runs the same test to validate other specifications, sanity check failures (Section **??**) occur. The reason is that line **??** will always subsequently fail because tempDir is not empty and no other test restores tempDir to its initial state. So, DSI+ classifies all subsequent specifications as "unknown". We observed state pollution only in IP6, but [90] specifications were affected. It may be possible to leverage recent techniques for detecting, managing, and fixing state-polluting flaky tests [] to help with state pollution in DSI+. These flaky-test related techniques are sometimes costly, so the cost of combining them with the already expensive DSI+ is one challenge that would need to be resolved.

6.6 Discovered: Some Limitations of DSI+

DSI+ cannot handle the case where a() and b() are called in different threads or processes. Even if there is a constraint on the temporal order of a() and b(), DSI+ currently classifies such specifications as "unknown", since it cannot delay calls to a() to be after calls to b(). A user then has to manually check these. DSI+ can also not correctly classify specifications that are only true under some conditions. For example. consider the methods a(): setWatchdog(), which sets up a Watchdog object to deal with timeouts, and b(): execute(), which connects the Watchdog to a new process and launches the process. A watchdog does not always have to be set, but a() needs to be called before b() when the user **intends** to impose a timeout. These limitations were not described in the original DSI work and we do not know solutions; we leave them as open problems in specification validation.

7 DISCUSSION

Specification Validation using Other Miners. Section 3 discusses why we use a miner that is stripped of its heuristics. We perform a small experiment to see if DSI+ provides value for validating specifications that two full-fledged miners produce. We ran BDDMiner with its heuristics [17] and Javert [16] on the same [five] projects and their tests that we use in our evaluation. To see if DSI+ can also help these tools, we check if some spurious specifications that we manually discover are also in these miners' output.

Javert mined [382] specifications from all our projects and tests—its heuristics filtered out [1619] specifications that the stripped-down miner produced. [220] of these [382] specifications that Javert mined are automata with more than two states and [71] of them had at least one transition that was a confirmed spurious specification in our evaluation. The other [162] of [382] are two-letter specifications, and [22] of them were spurious. [how many true specifications does Javert filter out?] These results show that specification validation can help even in the presence of heuristics and that there is a need to extend DSI+ to specifications with greater than two letters.

Table 15: Results of an ineffectual approach to improving DSI+.

DSI+ \ Man	TS Must	TS May	NS	NBP	US	Σ
LV	4	1	12	0	0	17
LS	74	114	785	33	7	1013
U	20	10	465	469	7	971
Σ	98	125	1262	502	14	2001

An ineffectual approach for improving DSI with unit tests. DSI+ produces different outcomes from different tests, so it may seem that cross-validation across tests and across test granularity levels could be a good way to improve DSI+ effectiveness. We implemented such an approach and found that it is not effective. For ease of reference and due to space limits, let's call this ineffectual approach DSI++; it validates specifications that were mined at one level of test granularity against tests from other granularity levels. If a specification is classified as spurious at any level, DSI++ classifies it as spurious. [DSI++' algorithm is given in the Appendix.]

Table 15 compares DSI++ verdicts with manual inspection for the [2001] specifications in our evaluation; the rows and columns are the same as in Table 3. DSI++ correctly filtered out more spurious specifications than DSI+ ([785] vs. [564]). But, it only reported [5] likely valid specifications, compared to [53] reported by DSI+. Also, DSI++ has an even higher overhead (average: [1941x], total: [11646x]) compared to DSI+ (average: [1313x], max: [4954x]). These overheads remained high even with an optimization that is [4x] faster than DSI++. So, DSI++ prunes out specifications too eagerly, and is much costlier than DSI+. Better solutions are needed for the problem of high DSI+ are needed.

Threats to Validity. Our results might not generalize to other Java projects since we focus on an in-depth study using a few projects. But our goal is to learn how specification validation works and how to improve it. Extending to other projects in the future will

help us learn more, but doing so will not invalidate the challenges that we discovered in our study. Our results might be affected by errors in our implementation of DSI+ or in our experimental infrastructure. To mitigate this threat, we tested our tools and manually inspected the results. Finally, we also performed a manual analysis of the specifications validated by DSI+, and this analysis might be characterized by divergent understanding among the researchers involved in the analysis. So, we had multiple rounds of review, via pull requests in order to find agreement wherever possible.

8 RELATED WORK

Other approaches for validating mined specifications. We view Nguyen and Khoo's approach [40] (which was proposed around the same time that Gabel and Su proposed DSI) as a different implementation of the same idea as DSI. They also use mutations to cause a program to violate a specification and determine that the specification is "significant" only if the mutated program crashes. The main differences with DSI are: (1) Nguyen and Khoo's approach performs mutations by statically deleting `a()` to see if `a()` establishes a precondition for `b()`. Static mutation may be faster and easier to apply, but it misses the opportunity to gather finer-grained dynamic information about when the program crashes. As we discussed in Section 3, knowing when the program crashes in DSI is helpful for obtaining more information about the nature of (non-spurious) specifications. (2) DSI's dynamic mutation for re-ordering method calls allows to validate more kinds of specifications than the preconditions that Nguyen and Khoo's implementation supports. Separately from DSI and Nguyen and Khoo's approach, we propose and enhance the use of oracles that exist in unit tests to validate mined specifications. It will be interesting in the future to see if and how the use of unit tests impacts the Nguyen and Khoo approach.

Unit testing and specification mining. Others have used unit tests in specification mining research, but they have done so for obtaining rather than validating the specifications. Dallmeier et al. [10] use automatic test case generation to produce enriched traces that enhance the quality of mined specifications. Pradel and Gross [49] mine specifications from automatically generated tests that pass, and check if those mined specifications are violated during the execution of automatically generated tests that fail. DICE [25] uses guided automated test generation to improve the quality of mined specifications. First, DICE mines specifications. Then, it uses guided automated test generation to find counterexample traces to the mined specifications. Finally, the traces from which the specifications were originally mined are combined with the counterexample traces to produce higher-quality specifications. The Deep Specification Miner (DSM) [28] also uses automatic test generation to come up with traces from which to mine specifications using deep learning.

DSI+ is complementary because it can be used to validate the output from these specification mining approaches. Our work on DSI+ is also different; (1) we use the oracles that are present in developer-written tests and not automatically generated tests. (2) We evaluate on open-source projects and not on benchmarks or individual classes. Since automatically generated tests are often unit tests, it will be interesting to compare and maybe combine developer-written and automatically-generated tests in DSI+.

Reducing false positives among mined specifications. DSI+ is orthogonal and complementary to approaches for reducing the number of spurious specifications that are mined, e.g., [29, 58]. Specifically, DSI+ can be applied to perform specification validation "post-mining", whereas these approaches are concerned with filtering out spurious specifications "intra-mining". These approaches should be used together, to reduce the false positives that result from checking mined specifications.

9 CONCLUSIONS

[Mined specifications are often spurious, and specification validation techniques lag behind decades of specification mining research.] We presented DSI+, an approach and a system for using the existing and abundant unit tests for validating mined specifications. We assessed DSI+ by comparing DSI+ and manually inspected categorizations of [2001] mined specifications in [six] open-source projects. Finally, we qualitatively analyzed the causes of DSI+ miscategorzation, and highlight several future research directions. Our work evaluates the current state of *specification validation* and provides a foundation on which other researchers can build in the quest to obtain better specifications for use in software engineering.

REFERENCES

[1] M. Acharya, T. Sharma, J. Xu, and T. Xie. Effective generation of interface robustness properties for static analysis. In *ASE*, pages 293–296, 2006.
[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
[3] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pages 143–162, 2008.
[4] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *PLDI*, pages 211–221, 2011.
[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
[6] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE*, pages 36–47, 2008.
[7] A. J. Brown. Specifications and reverse-engineering. *Journal of Software Maintenance: Research and Practice*, 5(3):147–153, 1993.
[8] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: A new approach to revealing neglected conditions in software. In *ISSTA*, pages 163–173, 2007.
[9] commons-codec. https://github.com/apache/commons-codec.git.
[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA*, pages 85–96, 2010.
[11] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma. Runtime monitoring with union-find structures. In *TACAS*, pages 868–884, 2016.
[12] M. B. Dwyer, R. Purandare, and S. Person. Runtime verification in context: Can optimizing error detection improve fault diagnosis? In *RV*, pages 36–50, 2010.
[13] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *ICSE*, pages 779–790, 2014.
[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
[15] commons-fileupload. https://github.com/apache/commons-fileupload.
[16] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008.
[17] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, pages 51–60, 2008.
[18] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE*, pages 15–24, 2010.
[19] M. Gabel and Z. Su. Testing mined specifications. In *FSE*, pages 1–11, 2012.
[20] M. Grandini, E. Bagli, and G. Visani. Metrics for multi-class classification: an overview. *ArXiv*, abs/2008.05756, 2020.
[21] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*, pages 223–233, 2015.
[22] S. Hussein, P. Meredith, and G. Roşu. Security-policy monitoring and enforcement with JavaMOP. In *PLAS*, pages 1–11, 2012.

[23] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *PLDI*, pages 415–424, 2011.
[24] kamranzafar.jtar. https://github.com/kamranzafar/jtar.git.
[25] H. J. Kang and D. Lo. Adversarial specification mining. *TOSEM*, 30(2):1–40, 2021.
[26] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *FSE*, pages 178–189, 2014.
[27] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh. Synergizing specification miners through model fissions and fusions (t). In *ASE*, pages 115–125, 2015.
[28] T.-D. B. Le and D. Lo. Deep specification mining. In *ISSTA*, pages 106–117, 2018.
[29] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS*, pages 292–306, 2009.
[30] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, pages 591–600, 2011.
[31] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical report, Computer Science Dept., UIUC, 2012.
[32] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE*, pages 602–613, 2016.
[33] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov. Techniques for evolution-aware runtime verification. In *ICST*, pages 300–311, 2019.
[34] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining. In *ASE*, pages 81–92, 2015.
[35] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE_Software*, 26(1):15–35, 2000.
[36] N. Li and J. Offutt. An empirical analysis of test oracle strategies for model-based testing. In *ICST*, pages 363–372, 2014.
[37] Maven central repository. https://mvnrepository.com.
[38] P. Meredith and G. Roşu. Efficient parametric runtime verification with deterministic string rewriting. In *ASE*, pages 70–80, 2013.
[39] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *RV*, pages 208–222, 2011.
[40] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *ICFEM*, pages 472–488, 2011.
[41] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of APIs in large-scale code corpus. In *FSE*, pages 166–177, 2014.
[42] J. W. Nimmer. *Automatic generation and checking of program specifications.* PhD thesis, Massachusetts Institute of Technology, 2002.
[43] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.
[44] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking. In *FSE*, pages 11–20, 2002.
[45] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *ICST*, pages 342–351, 2013.
[46] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 262–271, 2013.
[47] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *ICSM*, pages 1–10, 2010.
[48] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.
[49] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*, pages 288–298, 2012.
[50] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE*, pages 925–935, 2012.
[51] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *OOPSLA*, pages 270–285, 2010.
[52] R. Purandare, M. B. Dwyer, and S. Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *ISSTA*, pages 280–290, 2013.
[53] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *ASE*, pages 658–663, 2013.
[54] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *TSE*, 39(5):613–637, 2013.
[55] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen. Reflection-aware static regression test selection. In *OOPSLA*, pages 187:1–187:29, 2019.
[56] J. Spolsky. Painless functional specifications part 1: Why bother? In *Joel on Software*, pages 45–51. Springer, 2004.
[57] P. Sun, C. Brown, I. Beschastnikh, and K. T. Stolee. Mining specifications from documentation using a crowd. In *SANER*, pages 275–286, 2019.
[58] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE*, 2009.
[59] commons-validator. https://github.com/apache/commons-validator.git.
[60] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE*, pages 295–306, 2009.
[61] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, pages 461–476. 2005.
[62] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Iterative mining of resource-releasing specifications. In *ASE*, pages 233–242, 2011.
[63] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.
[64] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.
[65] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. A framework for checking regression test selection tools. In *ICSE*, pages 430–441, 2019.

Table 16: Breakdown of cases where DSI experiments of tests disagreed, based on the majority votes of DSI outcomes. Rows are the categories of disagreements. Columns are the majority vote between the tests of each specification within the disagreement category.

DSI+ \ Majority	LV	**LS**	U	Σ
LV-LS	42	130	0	172
LV-U	6	0	7	13
LS-U	0	38	43	81
LV-LS-U	0	11	11	22
Σ	48	179	61	288

10 APPENDIX