On Using Unit Tests to Validate Mined Specifications

Anonymous Author(s)

ABSTRACT

Specification mining infers specifications from software, but mined specifications are often spurious and produce many false alarms. Automated validation techniques that filter out spurious specifications can help. But, *specification validation* research lags decades of progress on *specification mining*. Little is known on how well specification validation works in practice, 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 approximate correctness.

We develop DSI+ by extending DSI to use unit tests, which encode developer intent in oracles that DSI can also use. Also, we manually categorize [2001] mined specifications in [six] well-tested and widely-used open-source projects. Then, we compare DSI+'s verdicts on these [2001] specifications with our manual categorization. DSI+ is wrong [45.4%] of the time. So, we qualitatively analyze why DSI+ is often wrong and derive insights on how to improve DSI+. Our results motivate new techniques for reducing manual inspection burden by improving the accuracy of automated specification validation. We highlight several future research directions[1].

1 INTRODUCTION

Many software engineering tasks require high-quality specifications, which formally capture expected program behavior: program understanding [7, 34, 40], static analysis [1, 41, 42, 48, 55], runtime verification [3, 6, 11, 12, 20, 21, 30, 32, 36, 37, 49, 50], etc. But, developers often do not write specifications due to costs of evolving specifications with code, lack of expertise in specification languages, and business pressures to roll out features faster [44, 53]. So, researchers proposed specification mining [2] to infer specifications from documentation [59, 61], source code [17, 39], program traces [17, 24, 28, 46, 57, 60], etc. Decades of progress on specification mining produced many techniques and miners, e.g., [4, 10, 13, 16, 33, 38, 45, 47, 51, 58]. Robillard et al. provide a survey [52].

Unfortunately, miners often produce spurious specifications, especially when applied "in the wild", beyond research benchmarks. A reason 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 (e.g., [8, 23, 27, 47, 55, 60]).

A big negative impact of spurious specifications is that downstream techniques yield many false alarms, e.g., (1) Legunsen et al. report a [97.89%] false alarm rate among runtime verification violations of mined specifications [30]; (2) Gabel and Su [18] say, "false positives dominated our early... results" despite using heuristics/statistics; and (3) Lee et al. told us informally that `JMiner` [28] produces many spurious specs. Beyond miners and downstream techniques, recent proposals for crowdsourcing part of the mining task (e.g., [54]) may benefit from improved specification validation.

Gabel and Su [18] proposed Deductive Specification Inference[2] (DSI) as an automated specification validation technique that does not use heuristics and statistics. Despite its name, *DSI performs specification validation; it is not a specification mining technique.* DSI is miner agnostic; its input is a mined specification, and the output is a verdict on whether its input is spurious or likely valid.

DSI is based on a simple idea: a specification is spurious if a program P that violates the specification is correct. DSI dynamically mutates P to violate the specification and then checks if P crashes during or after the mutation. Consider an example. Given specification, s: "method `b()` is always preceded by `a()`", DSI attempts to delay calls to `a()` to be after calls to `b()`. If P crashes when `b()` is called after `a()` was delayed, or when `a()` is called after `b()`, then `s` is likely valid. Absent a crash, DSI adjudges `s` to be likely spurious.

DSI relies on crashes to approximate correctness, but crashes can be weak [35] or generic baseline [43] oracles—a program may not crash even if DSI's mutation induces an error. Also, DSI was only evaluated on the DaCapo benchmarks [5]. So, it is not known how effective DSI is when it is used 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 [18]. (Project-specific specifications specify a program, and not its third-party APIs [29, 39, 48, 52, 60, 61].)

We develop *DSI+* by extending DSI to use unit tests, which encode developer intent in oracles that DSI can also use. We envisage DSI+ usage in settings where developers execute software by running tests. So, we implement DSI+ inside Maven—a widely-used build system for Java. Our envisaged setting differs from the one in which DSI was evaluated so far: mined specifications were validated while running software on workloads in benchmarks that were originally curated for evaluating performance. DSI+ allows us to investigate specification validation in a modern development environment, and to learn the challenges that arise in doing so.

We perform the first in-depth study of specification validation using unit tests in four steps. First, we run DSI+ on all [2,001] specifications that a miner produced from developer-written tests in [six] well-tested and widely-used open-source projects. Second, we manually inspect all [2,001] specifications and categorize them as spurious or valid. Our "ground truth" data set can help evaluate future DSI+ improvements. Third, we compare our categorization with DSI+'s verdicts to 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+.

Many miners today can produce complex specifications, but we validate simple specifications of temporal order between two method calls—"two-letter" specifications [17, 18, 25, 54]—for several reasons. First, automated validation of two-letter specifications

[1] 2022-03-14 17:40

[2] Gabel and Su's name for their technique; they acknowledge that it is not truly deductive—lack of specifications for reasoning about correctness necessitates mining.

is still in infancy and it is not known how to automatically validate more complex specifications. That is, *specification validation* research lags *specification mining* research. Second, all automated specification validation techniques that we know, including DSI, only validate such specifications. We seek to better understand current technology before making the leap to validating more complex specifications. Third, it allows easier comparison of DSI+ with DSI. Lastly, two-letter specifications encompass a rich class of important temporal safety properties (Section 2). We refer to the first method in a two-letter specification as `a()`, and the second method as `b()`.

We find that DSI+ is often wrong. DSI+' verdicts matched ours in [1087] of the [1987] cases that we conclusively categorize, so DSI+ was wrong [45.4%] of the time. DSI+ had 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. DSI+ accuracy, precision, and recall should be improved. Unit tests' assertions helped DSI+ to validate [23.70%] of true specifications [and to prune out [52.1%] of spurious specs]. Also, DSI+ overhead is high (average: [1,313x], max: [4,954x]).

Our qualitative analysis shows that DSI+' misclassifications are often due to the challenges of using unit tests to validate specifications. E.g., if `a()` is immediately followed by an assertion on its return value, then that assertion may fail right after DSI+ delays `a()`. So, DSI+ may be unable to classify a spurious specification. Also, DSI+ can conflict with itself on different tests, e.g., if the order of assertions in one test is as above, and another test asserts on `a()`'s return value only after calling `b()`. Section 2 exemplifies more challenges, and Section 6 discusses them, plus ideas for mitigating some of them.

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 (1) simulate the end-to-end developer experience that would result from using DSI+ today, and (2) study specification validation in depth rather than in breadth. We actually run DSI+ on [103,264] mined specifications from [36] open-source projects (Section 5 and Appendix). DSI and DSI+ are motivated in part by the infeasibility for developers to manually inspect tens of thousands of mined specifications. We hope that this paper spurs future work that builds on our findings to increase the confidence in DSI+' verdicts, and significantly reduce the number of specifications that developers must manually inspect.

We make the following contributions:

- ⋆ **Extension.** We extend DSI to also use unit tests when validating mined specifications, resulting in DSI+.
- ⋆ **Study.** We conduct the first in-depth study of specification validation with unit tests, using DSI+, and the challenges entailed.
- ⋆ **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 it 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()` must be called before `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()` 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 reported as being easier for non-experts to reason about [54], 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`: " "`b()` must only be called after `a()`" from `T1`. If `t1` fails when DSI+ calls `a()` after `b()`, then DSI+ adjudges `s` as likely valid. But, that would be a misclassification if `t2` and `t3` pass 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` passes and represents correct program behavior.

Our study also reveals other complications that arise from using unit tests to validate mined specifications. For example, in `T2` 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 these and other challenges that we discovered in our study, plus our thoughts on how future work can tackle them.

Examples from open-source projects. In Figure 2, an assertion failure led DSI+ to correctly classify a true specification as likely

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

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

```
1 public String meta(final String txt) {
2     switch(s) {
3       case 'P': if (false) // a()@pre
4           { code.append('F'); }
5       case 'S': match = regionMatch(local,n,"SH"); // b()
6           isNextChar(local,n,'H'); // a()@post
7     }
8     return code.toString(); }
9 @Test void testPF() {assertEquals("FX", meta("PHSH"));} // fail
```

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

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

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

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

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

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; see 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. `test_MI()` fails because its input value contains a vowel, but `test_MY()` 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.

3 DSI/DSI+ PRIMER AND IMPLEMENTATION

Algorithm 3.1 shows the DSI+ procedure, which extends DSI. [DSI+ amplifies DSI by leveraging assertions and including reasoning from multiple test executions.] 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` [16]

Algorithm 3.1 DSI algorithm

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: $S^f = \{s^f_1, s^f_2, ...\}$, a set of spurious specifications,
$S^t = \{s^t_1, s^t_2, ...\}$, 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
6:              S^f_test, S^t_test ← validate(test, T_map[test], P, T)
7:              results ← results ∪ (S^f_test, S^t_test)
8:      return S^f, S^t
9:
10: procedure validate(test, T_map[test], P, T)             ▷ 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 [18]
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 out4 == PASS then S^f ← S^f ∪ {s}; continue
27:         if stage == 0 then S^u ← S^u ∪ {s}; continue
28:         if stage ∈ {1, 2, 3} then S^t ← S^t ∪ {s}
29:
30: procedure sanityCheck(s, o1, o2, o3, t1, t2)
31:     if o1 ∈ {FAIL, CRASH} then return false ▷ Base run crash
32:     if o2 ∈ {FAIL, CRASH} then return false ▷ Instrument fail
33:     if o2! = o3 then return false                      ▷ Nondeterministic
34:     if (s ∉ t1) ∨ (s ∉ t2) then return false           ▷ Traces omit s
35:     return true
```

as in the original DSI work [18]; it is stripped of heuristics [in order to (1) avoid interaction effects with heuristics, and (2) assess specification mining at its most raw form to observe challenges]. Note that 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 level, `runDSI+` calls the `validate` procedure (lines 10–21) to check whether each specification that is mined from that test is likely spurious or likely valid. (See Section 2) for an explanation on why we mine and validate at all granularity levels.) That is, each test is used to

validate the specifications that were mined from its traces. 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, 30–35). 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 30–35) 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 31); (2) the first run `PASS`es but the second run does not, so instrumentation induced a failure (line 32); (3) the test outcomes in the second and third runs differ, so the runs are nondeterministic (line 33); and (4) the traces from the second and third runs do not satisfy `s`, so `s` may be spurious (line 34). The reasons for sanity check failures in our evaluation are concurrency-related issues, state pollution among tests [**?**], and very few instrumentation failures [(Section **??**)].

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 [18]). 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 the default value. Replacing return values needs improvement in DSI+ (Section 6.1). But, the current scheme works quite well in most cases although in [1597] of the [2,001] specifications that we inspect, `a()` does not return `void` (Section 5.4).

(2) Locks. Experiments may require delying `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 **??**)].

(3) Multiple call sites. Traces can have multiple pairs of `a()` and `b()` with different call sites; we refer to the unique call site for each pair as a *usage scenario*. DSI+ creates one experiment per usage scenario. [These experiments are all performed in a single test execution, which introduces interference described in Section 6.]

Experiment Execution. DSI+ uses the outcome of running experiments to validate a specification (lines 20, 23–28). 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 spurious—it is not necessary for P's correctness. So, DSI+ pust `s` in S^f (line 26). Otherwise, DSI+ uses the stage when test `t` `FAIL`s or `CRASH`es to decide the validity of `s`:

Stage 0: After delaying `a()` but before calling `b()`—the experiment could not run and `s` cannot be validited; add `s` to S^u (line 27).

Stage 1: While running `b()` after delaying `a()`—method `a()` likely establishes a precondition for `b()`.

Stage 2: After `b()` ran 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 28).

DSI+ Implementation. We build DSI+ on top of Gabel and Su's DSI prototype, after we used their DSI prototype to reproduce their results [18]. We developed Maven plugins that embody different components of DSI+: trace collection, specification mining (which can be customized to use any existing specification 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). [Users can prune out mined specifications for which `a()` or `b()` is defined in a test class. The specifications we evaluate in this paper were mined with this configuration.]

Table 1: 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 [22]	4b669	1.0	2	8	.75	.67	24
P2	codec [9]	b959a	23.9	58	936	.97	.93	10618
P3	validator [56]	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

4 METHODOLOGY

4.1 Study Setup

We answer the following research questions:

RQ1 What is the accuracy, precision, and recall of DSI+, compared with manual validation?

RQ2 How much do test oracles contribute to DSI+ verdicts, compared with crashes?

RQ3 What is the runtime overhead of DSI+?

RQ4 How do specifications and tests interact during specification validation?

To answer these research questions, we run DSI+ on [six] open-source projects and recorded its verdicts and overheads. Then we manually inspect all the [2001] specifications that were mined from these projects, to obtain some "ground truth" for evaluating DSI+, and to generate more data about how 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 1 shows details about the projects that yielded the specifications that we manually inspected; the caption describes the column headers. We selected those [six] projects from among a larger set of [36] open-source projects that we successfully run DSI+ on. [25] of those [36] projects that were recently used in evaluations of runtime verification [] or regression testing []. We reused them because they are more likely to have tests that run and are less susceptible to instrumentation failures (the techniques that they were evaluated on perform instrumentation). The other [11] were selected from GitHub based on [Ayaka to describe]. Section **??** provides details about all [36] projects.

We finally selected [six] of these [36] based on a combination of their (1) statement coverage (at least 70%); (2) most recent commit (must be after December [2019]); (3) number of clients that they have on the Maven Central repository [?] (a higher number increases the chance that true specs that we find will be useful more broadly); and (4) having a manageable number of mined specifications that we could inspect within our time budget (for reasons given in Section 1, we chose to 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 one co-author as the primary inspector of all specifications mined from each of the [six] projects. The primary inspector's main tasks were to (1) manually determine how necessary each specification is for program correctness (i.e., whether it is a true specification or not); (2) compare their findings with that of DSI+; (3) determing why manual findings differed from DSI+ in terms of the code under test and in terms of how DSI+ operated in those cases; (4) find problems that arise when using unit tests to validate mined specifications; and (5) record extensive information (including the code snippet) about each specification such that any other person can check their work only by looking at their records. [Our replication package contains all the `json` files that contains all this information, together with `json` schemas for valiging their consistency and scripts for processing them.]

Once the primary inspector was done with the first round of inspections, they made a pull request which a secondary inspector (another co-author) checked thoroughly and sent back for revision. The secondary inspector double-checked the specification categorization and pointed out ways to improve the quality of the recorded information. Multiple rounds of updates and reviews before the pull request was merged. Finally, the primary and secondary instructors had several meetings to try to come to a decision 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.

[Notes: (1) we should say somewhere that this is the most detailed evaluation of DSI, as Mark did not really do any inspections]

Table 2: 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 3: 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

5 RESULTS

5.1 RQ1: Accuracy, Precision, and Recall

We compare DSI+ verdicts with our manual inspection decisions. Table 2 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 2 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" [18]), 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 6.7].

Colored cells in Table 2 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 2: (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 3 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 2. 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

Table 4: Accuracy, precision, and recall of DSI+.

ID	Σ	Acc	Acc [%]	Precision[%]		Recall[%]	
				TP	**TN**	**TP**	**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

take the majority votes to be correct [(Appendix ??)]. 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 [18] 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 4: (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+ classifies as DSI 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 4; 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?]) [19] of DSI+ in Table 4: 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 4 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.

Table 5: Assertions vs exceptions/crashes for all true positives.

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

Table 6: 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

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 5 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 (per spec) as the ratio of assertion failures per **TP** specification (% Assert (per 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 (per spec) is the number of assertions that failed per such specification. For example, suppose DSI+ verdict is LV for 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 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 (per 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).

The results in Table 5 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. 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. [How many TP would we miss if oracles did not fail?]

Table 7: Overhead of DSI+. **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] ÷ $bl(s)$. **SM[s]**: time in seconds for mining.

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

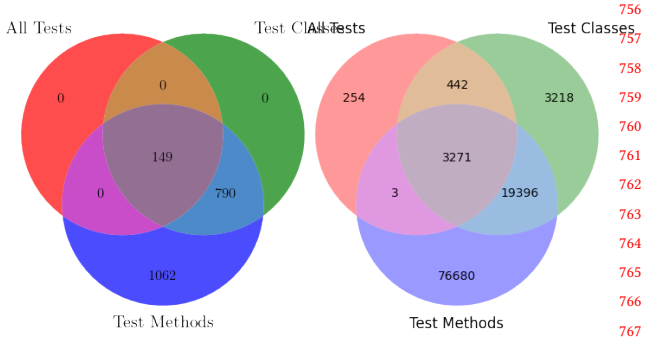
Weak Oracles If DSI+ mutation causes a program to be incorrect without crashing, then DSI+ may classify a likely valid specification as likely spurious if the test oracle is weak. Table Table 6 shows a breakdown of the [175] specifications for which we observed a weak oracle in at least one test that 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 2. "% TS", "% NS", "% 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 [18], and (2) how well DSI+ may scale to bigger projects than those that we evaluated. Table 7 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 7 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.

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



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

Figure 6: Mined specifications at different levels of granularity.

Table 8: 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 9: All

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

Table 10: TS

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

Table 11: NS

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

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 pollution that affects the outcome of a later experiment. We measure the frequency of mutiple usage scenarios in our experiments. [Add: for how many specifications do we have multiple usage scenarios? And how many of those were problematic?]. 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. Given the return-value replacement that DSI+ performs when delaying `a()` (Section 3), we check whether DSI+ can work because `a()` or `b()` often returns `void`. The results in Table 9 show that among all specifications, `a()` does not return `void` [X%, ([1186]÷[2001])] of the time. In that table, columns show

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

Table 14: List of trends across all inspections.

Category	Description	% NT	% NS	Σ
Return	a() returns void.	76.7	76.5	807
	Replacement was unequal to expected value.	96.9	96.9	313
	DSI+ replaced with null; NullPointerException thrown.	85.5	85.5	253
	Replacement not checked in assertion but equal to a()'s return.	98.2	98.2	125
	Replacement was equal to expected value.	95.0	94.9	94
	Default replacement value equal to original return.	100.0	100.0	89
	Caller of a() discards return value.	52.4	52.4	72
	Replacement alters conditional; b() is not called.	97.4	97.4	40
	Replacement in small range that causes failure.	100.0	100.0	2
Relationship	a() and b() are stateful but unrelated.	100.0	100.0	307
	Either a() or b() does not alter state.	95.4	95.7	290
	Either a() or b() is a pure setter.	86.1	86.4	221
	a() and b() are connected	100.0	100.0	174
	Neither method changes state.	100.0	100.0	89
	a() and b() are both pure setters.	100.0	100.0	43
	a() and b()'s caller construct true specification.	100.0	100.0	12
	a() and b() modify disjoint state.	100.0	100.0	2
Exception	Test expects an exception.	65.7	66.7	269
	Expected Exception not thrown; test fails.	76.7	70.8	58
	Exception swallowed by catch block.	66.7	66.7	30
	Expects Exception but does not check.	50.0	50.0	20
	Delay of a() causes unexpected exception.	80.0	80.0	10
Oracle	Test has weak oracle.	84.8	84.8	304
	The order of assertions influenced experiment result.	99.4	99.4	167
Delay	DSI+ experiment caused state pollution.	82.5	83.6	87
	Delayed call of a() restores state.	97.7	97.7	86
	Delay causes output-dependent assertion to fail.	78.6	78.6	39
	Delay causes a timeout value to be exceeded.	44.4	44.4	9
	Delay will cause corruption in the data that method-b works on.	75.0	75.0	8
Misc	Multiple usage scenarios.	86.8	86.9	306
	a() and b() called in different threads.	81.7	74.5	73
	a() and b() defined as being from different classes.	91.7	91.8	70
	Bug causing sanity-check-failure.	0.0	95.8	54
	A NBP that goes beyond method-call chains.	100.0	100.0	42
	a() and b() are the same method; dynamic dispatch.	100.0	100.0	13
	Manual inspection strengthened by/dependent on JavaDOC comments.	66.7	66.7	12
	A configuration file impacted the outcome of the test.	90.9	90.9	12
	b() calls a().	90.0	88.9	10
	Simple and uninteresting specification.	0.0	0.0	3
	A relevant set of instructions to the spec was not covered in the test suite.	50.0	50.0	2

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.

6 ANALYSIS OF RESULTS

We discuss key challenges, insights, and impacts of DSI+. Table 14 shows common trends observed in manual inspection. We compute **% NT** as the frequency of the trend among non-TS, non-$\mathbf{U}^T$ cases (**% NT** = $\frac{\mathbf{FP+UP+TN+UN+U}^{tn}}{\Sigma-\mathbf{U}^T}$), and **% NS** among NS cases in non-U cases ($\frac{NS}{\Sigma-(U)}$). Sections 6.1 to 6.6 discuss challenges and solutions within each category. Section 6.7 describes the nature and implications of the true specifications found. We chose simple examples due to space limitations. We include more examples in the appendix,

```
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: Example of DSI+ being misled by return value replacement.

and all code examples of inspected specifications are available in the replication package.

6.1 Return Values

DSI+ replaces the return value of a() with either the nearest value of a variable of the same type, or a default value. Replacing a() with an incorrect value can lead to unintended assertion failures and exceptions, and replacing a() with a correct value can mask incorrect state changes caused by DSI+'s delay. So, DSI+'s replacement scheme can result in both **FP** and **FN** outcomes. For example, Figure 7 shows two tests that check the encoding and decoding of a String. Our manual inspection concluded encode() and decode() do not form a true specification. The DSI+ experiment using testSafe() replaces the return value of a() with plain, which is the expected return value of a(). So, the test passes and DSI+ categorized the specification as **LS**. plain is also chosen as the replacement value for testBasic(), but it is **not** the expected value. Hence, the assertion on line 9 fails, and DSI+ classifies this specification as U. This is problematic for two reasons: (1) this replacement of return values caused Mixed conclusions, and (2) if testSafe() didn't exist, then DSI+ would not have concluded **LS** at all. [DSI+'s replacement mechanism could be improved by being *case-dependent*, based on how the return value of a() is being used.]

[When the return value of a() is assigned to a variable, DSI+ does not restore the state of this variable when the delayed call of a() is performed. So, the test continues its execution with the replacement value assignment even after the delayed call to a() is performed. One approach to solve this problem is to make DSI take note of the variable that the return value of a() would be assigned to, and replace the value of the variable when the delayed call to a() is performed. However, this is also not a perfect solution, as there may be other variables that were impacted by a(), and the original variable that a()'s return value would have been captured in may not be in scope by the point at which the delayed call to a() is performed.]

6.2 State Relationship

Knowledge of the state relationship between a() and b() can indicate of whether the two methods form a true specification. However, DSI+'s current design does not take this into account. For example, we observed that the state used and modified respectively by a() and b() in Figure 3 did not intersect. We observe that [43] **FP** specifications consist of methods that do not share state. A heuristic "two methods that do not intersect in the state that they access and/or modify do not compose a true specification" can aid DSI+ in filtering out such cases.

```
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**.

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

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

However, DSI+ experimentation is still necessary, as two methods that share state do not necessarily compose a true specification. For example, the methods `encode()` and `decode()` in Figure 7 share state because the return value of `a()` is the argument to `b()`. Given this common use case, we believe that such a spurious specification would be likely mined by state-of-the-art miners. However, it is not always necessary to encode before decoding, so this is a spurious specification. DSI+'s **LS** classification of this specification using `testSafe()` shows potential for DSI+ to filter out such spurious specifications. So, we believe that both DSI+ experimentation and state relationship are necessary to effectively filter out spurious specifications.

6.3 Exceptions

Exceptions can help DSI+ determine that a specification is LV. However, tests that expect exceptions to be thrown can mislead DSI+. For example, DSI+'s experimentation in Figure 8 concluded that the specification was LV because an exception was thrown by `b()`. However, the test passed because it expected `b()` to throw an exception. We found [52] **FP** specifications to share this same trait. We classified this specification as NS because `a()` and `b()` were unrelated, and because `b()` threw the same exception regardless of whether `a()` was called beforehand or not. Leveraging this knowledge, we propose to improve DSI+ to track and compare the exception stack trace and the state observed at the point of the exception for such exceptions. If the stack trace and state are equal between the DSI+ experiment and a normal execution, then the specification is likely spurious. This has three challenges: (1) detecting tests that expect exceptions is nontrivial; (2) determining whether such "exception tracking" overrides the outcome of the tests; (3) ensuring that this extension does not introduce new **FP** or **FN** cases.

[Add EXPECTED_EXCEPTION_NOT_THROWN if we have space?]

6.4 Oracles

DSI+ relies on the quality of the unit tests that it uses. However, weak oracles in tests can mislead DSI+ experiments. For example, Figure 9 show two tests used by DSI+ to validate a true specification that is necessary to propagate `std err` output from a subprocess. `testStreams()` runs and checks the output of a script that produces `std err` output, and DSI+ using this test concludes LV. However, `testExecute()` runs and checks the output of a script that does **not** produce `std err` output. So, the oracle is weak for validating this specification, and misleads DSI+ towards a **LS** classification. This is problematic because (1) the weak oracle caused contradictory classifications, and (2) if `testStreams()` did not exist, then this specification would only be errneously categorized as **LS**. [In fact, `testStreams()` was the **only** LV test out of [39] tests that DSI+ used to validate this true specification.] We found [at least [32]] **FN** specifications impacted by weak oracles. So, detecting and strengthening tests that are too weak to catch state differences caused by violating true specifications are imperative to improving DSI+. These are tests that still pass even when `a()` or `b()` is ommitted, and those that perform a simple containment/existence check, which is not enough to capture the state corruption caused by the DSI+ experiment.

```
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 b();
2 int a = a();
3 assertEquals(100, a);
```

Figure 11: Synthetic code for order of assertions solution.

```
1 methodA(); // a()
2 getA(); // true \spec with a()
3 methodB(); // b()
```

Figure 12: Synthetic code motivating multi-stage validation.

DSI+ also relies on the ordering of assertions within the test, which can be arbitrary and misleading. For example, the DSI+ experiment in Figure 10 fails in the `assertNotNull()` on line 3 before `b()` is called[because DSI+ replaces the return value of `a()` with `null`]. So, DSI+ concludes this true specification is U. We found 155 miscategorizations by DSI+ due to the order of assertions. To reduce this noise in assertion failures, we propose DSI+ to delay not only the call to `a()` but also all affected assertions, as illustrated in Figure 11. This is challenging because DSI+ must know in advance which assertions will be impacted by either the return value of `a()` or the state changes that `a()` makes, and preserve the semantics of the program as much as possible.

6.5 Effects of Delay

[Replace below opener with"when evaluating a specification, you can get interference if the evaluation involves breaking a true specification"?] The delay of `a()` can cause a different true specification to be violated, misleading DSI+ to classify a spurious specification as LV. Figure 12 shows a contrived example. Suppose `methodA()` and [`getA()`] form a true specification. Then, validating `methodA()` and `methodB()` will violate this true specification, causing the test to fail. Then, DSI+ will mistakenly conclude that `methodA()` and `methodB()` will form a true specification. We observe [X] specifications that follow this trend. We propose a *multi-stage approach* towards DSI+ validation. In the example, DSI+ can first evaluate `methodA()` and `getA()`. If they form a LV specification, then DSI+ can evaluate `getA()` and `methodB()`; if they form a LV specification, then DSI+ can conclude that `methodA()` and `methodB()` transitively form a LV specification as well. If not, DSI+ can delay

```
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 13: Test causing and impacted by DSI+ state pollution.

both `methodA()` and `getA()` after `methodB()`, to prevent the interference. Two large challenges are: there can be multiple methods between `a()` and `b()` (candidates for `getMethod()`); this proposal introduces dependencies among specifications to DSI+'s algorithm, which could potentially be **much** more expensive.

DSI+'s delay of `a()` can also effect classification of specifications other than the one being validated. DSI+ iteratively runs experiments for each specification mined by a single test. Figure 13 shows unintended state pollution resulting from a single delay of `a()`. DSI+ first attempts to validate the specification, but the assertion on line 4 causes the test to fail before reaching `b()`, which deletes a temporary file. Since `b()` was not called, the temporary file was not deleted. This caused line 6 to fail on all subsequent runs of the same test, because `tempDir` will no longer be empty. This causes the sanity checks for all subsequent specifications to fail, misleading DSI+ to classify them as U. While we only observed this state pollution from IP6, [90] specifications were affected. [insert proposal for fix here]

6.6 Unintended Consequences

[Include example for multiple perturbs?] In the inspected projects, there were a total of [777] specification-test combinations that featured multiple usage scenarios. In [182] of these [777] specification-test combinations, at least one experiment was not performed. In [160] out of these [182] combinations, the first experiment caused the test to fail/crash, preventing the subsequent experiment from being performed. This is harmful because it masks insights DSI+ could get from the other usage scenarios. One (naive) solution is to isolate each experiment. This potential solution has the challenges: (1) DSI+ is already costly, and isolating each DSI+ experiment would significantly increase this cost; (2) there will be a new layer of contradition, where even the DSI+ experiment results within a single test-specification combination contradict. We will have to find more efficient, concise ways of resolving disagreements in perturbations. [Potential other idea: we might want to apply some program transformations that allow us to separate the test into p chunks? Sometimes the multiple perturbations arise from the test performing similar computations using different inputs (so, p invocations of `a()` and `b()` simply because the developer lumped multiple "tests" within one test.)]

6.7 The nature of True Specs that we found

As introduced in Section 1, the specification pattern used in this paper is a simple two-letter specification. We identify how (1) validation of these simple two-letter specifications can lead to validating larger kinds of specifications mined by today's state-of-the-art miners, and (2) this specification pattern has multiple interpretations.

6.7.1 *Larger than 'ab'.*

Table 15: DSI+ vs manual inspection outcome across all projects [DSI+].

DSI+ \ Man	TS		NS	NBP	US	Σ
	Must	May				
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

6.7.2 *Not just 'a' must be called before 'b'.* **Must specifications**

We further analyzed the set of true specifications found from manual inspection to understand their nature. Following from [cite], we establish six categories of intent: (a) `a()` is always followed by `b()`, (b) `b()` is never followed by `a()`, (c) `b()` is always preceded by `b()`, (d) `b()` is never immediately followed by `a()`, (e) `a()` is always immediately followed by `b()`, and (f) `b()` is always immediately preceded by `a()`. We found that [53] specifications were in the (c) category, meaning that `b()` depends on `a()`. [Finish looking through the true specifications and say more stats here. Might be helpful to have a table but I don't think we will have space for it.] We aim to explore techniques to automatically idenitfy the precise nature of a DSI+ validated true specification to build more precise understanding and better encode true specifications.

May specifications Our inspections revealed a class of true specifications that conditionally hold. That is, the specification pattern we use is too stringent[, and would produce many false alarms when the specification is used with RV]; however, there is a temporal relationship between `a()` and `b()` that ought to be followed in certain 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 and launches a new 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.

A co-author further inspected the May specifications, identified the conditions for which they are necessary for correctness, and created new formulae for them accordingly.

7 DISCUSSION

7.1 Monitoring True Specs

7.2 Impact on Other Miners

7.3 Ineffectual approaches

DSI+ and DSI++

7.4 Threats to Validity

As is the case with most empirical evaluations, there are threats to the validity of our results. In terms of external validity, our results might not generalize to other Java applications. To mitigate this threat, we considered applications with different functionalities that were used in research on testing Java programs and on runtime verification [30–32]. In terms of construct validity, our results might be affected by errors in the implementation of DSI+ or our experimental infrastructure. To mitigate this threat, we tested our

tools and manually inspected our 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. We are confident of the reliability of our analysis as we systematically analyzed the specifications and used negotiated agreements to draw conclusions.

8 RELATED WORK

Other approaches for validating mined specifications. We view Nguyen and Khoo's approach [38] (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 [47] 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 [23] 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) [26] 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., [27, 55]. 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. Symbolic mining of temporal specifications. In *ICSE*, pages 51–60, 2008.
[17] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE*, pages 15–24, 2010.
[18] M. Gabel and Z. Su. Testing mined specifications. In *FSE*, pages 1–11, 2012.
[19] M. Grandini, E. Bagli, and G. Visani. Metrics for multi-class classification: an overview. *ArXiv*, abs/2008.05756, 2020.
[20] S. Hussein, P. Meredith, and G. Roşu. Security-policy monitoring and enforcement with JavaMOP. In *PLAS*, pages 1–11, 2012.
[21] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *PLDI*, pages 415–424, 2011.
[22] kamranzafar.jtar. https://github.com/kamranzafar/jtar.git.
[23] H. J. Kang and D. Lo. Adversarial specification mining. *TOSEM*, 30(2):1–40, 2021.
[24] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *FSE*, pages 178–189, 2014.
[25] 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.

[26] T.-D. B. Le and D. Lo. Deep specification mining. In *ISSTA*, pages 106–117, 2018.

[27] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS*, pages 292–306, 2009.

[28] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, pages 591–600, 2011.

[29] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical report, Computer Science Dept., UIUC, 2012.

[30] 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.

[31] O. Legunsen, A. Shi, and D. Marinov. STARTS: STAtic Regression Test Selection. In *ASE*, pages 949–954, 2017.

[32] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov. Techniques for evolution-aware runtime verification. In *ICST*, pages 300–311, 2019.

[33] C. Lemieux, D. Park, and I. Beschastnikh. General LTL specification mining. In *ASE*, pages 81–92, 2015.

[34] N. G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE_Software*, 26(1):15–35, 2000.

[35] N. Li and J. Offutt. An empirical analysis of test oracle strategies for model-based testing. In *ICST*, pages 363–372, 2014.

[36] P. Meredith and G. Roşu. Efficient parametric runtime verification with deterministic string rewriting. In *ASE*, pages 70–80, 2013.

[37] 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.

[38] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *ICFEM*, pages 472–488, 2011.

[39] 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.

[40] J. W. Nimmer. *Automatic generation and checking of program specifications.* PhD thesis, Massachusetts Institute of Technology, 2002.

[41] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 229–239, 2002.

[42] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking. In *FSE*, pages 11–20, 2002.

[43] F. Pastore, L. Mariani, and G. Fraser. CrowdOracles: Can the crowd solve the oracle problem? In *ICST*, pages 342–351, 2013.

[44] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. What good are strong specifications? In *ICSE*, pages 262–271, 2013.

[45] 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.

[46] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.

[47] 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.

[48] 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.

[49] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *OOPSLA*, pages 270–285, 2010.

[50] R. Purandare, M. B. Dwyer, and S. Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *ISSTA*, pages 280–290, 2013.

[51] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *ASE*, pages 658–663, 2013.

[52] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *TSE*, 39(5):613–637, 2013.

[53] J. Spolsky. Painless functional specifications part 1: Why bother? In *Joel on Software*, pages 45–51. Springer, 2004.

[54] P. Sun, C. Brown, I. Beschastnikh, and K. T. Stolee. Mining specifications from documentation using a crowd. In *SANER*, pages 275–286, 2019.

[55] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE*, 2009.

[56] commons-validator. https://github.com/apache/commons-validator.git.

[57] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE*, pages 295–306, 2009.

[58] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, pages 461–476. 2005.

[59] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Iterative mining of resource-releasing specifications. In *ASE*, pages 233–242, 2011.

[60] 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.

[61] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, pages 307–318, 2009.