Generalized Testing of Finite State Specifications

Anonymous Author(s)

***Abstract*—Many techniques automatically mine formal specifications (specs), which are important and rare in software engineering. Unfortunately, many mined finite state specs are buggy and miners can produce specs that are too numerous or complex to validate by hand. Despite these problems, the state-of-the-art automated technique for testing finite state specs has two big limitations: (1) it can only test specs involving two methods, i.e., two-letter specs; and (2) it is inaccurate—it gives correct verdicts only 39.4% of the time. We propose TEMARI, a general and more accurate technique for testing arbitrarily complex finite state specs. Such specs cannot always be decomposed into two-letter components. So, TEMARI first uses model checking to find two-letter constraints that hold on the spec. Then, it uses knowledge about program state to more accurately test those constraints. Lastly, it mutates the program to violate each constraint. If all tests still pass, the spec is buggy: it prescribes incorrect program constraints. We use TEMARI to test 1,726 specs that the state-of-the-art cannot test; they have up to 30 letters and 252 states. TEMARI finds bugs in 1,451 of them with 2x more accuracy than the state-of-the-art, at 78.9%.**

I. INTRODUCTION

Many software engineering tasks require high-quality specs, which formally capture expected program behavior. Such tasks include program understanding [1]–[3], static analysis [4]–[8], and runtime verification [9]–[20]. But, developers often do not write specs due to the costs of evolving them, inexperience with spec languages, and business pressures to roll out features faster [21]–[23]. So, researchers proposed many techniques to automatically mine specs from software [24]–[44].

We focus on behavioral API specs that are expressible as finite state machines (FSMs) over method calls; they are widely used in the literature on spec mining. Also, we only target one kind of bug: we say a spec is *buggy* if it prescribes incorrect temporal order among method calls in a program.

Unfortunately, there is growing evidence that mined finite state specs are often buggy. For example: (1) Legunsen et al. [9] report a 97.9% false alarm rate among runtime verification violations of mined specs. (2) Gabel and Su [45] say "false positives dominated [...] results" of their miner. (3) JMiner [41] produces many buggy specs [46].

To investigate mined spec bugginess more broadly, we run a preliminary study using seven spec miners on six open-source projects (Table I). The miners are variants of five tools [31], [34], [47]–[49] (Table II). We compare specs from these miners against ground truth that we obtain by manually inspecting 1,470 specs from an independent miner. We find that mined specs are often buggy, with an average percentage of buggy specs of 25.1% (across projects). Full manual inspection is impractical: these miners yield 48.7 million specs. Even if one samples a few specs to inspect, they are complex and hard to comprehend [9], [50]—average: 3.5 states, 10.5 transitions; maximum: 132,730 states, 158,859 transitions. In sum, mined specs are often buggy and too numerous or complex to inspect.

Automated spec testing [27], [45], [51] can reduce manual inspection burden. Deductive Specification Inference (DSI) [45] is the state-of-the-art technique for testing finite state specs. DSI mutates a program by re-ordering *two* methods in a spec and reports the spec as buggy if the mutated program's tests pass. The idea is that a spec is buggy if a program that violates the spec is correct. But, DSI has two big limitations: (1) it can only test *two-letter* specs involving two methods; and (2) we find that it is only 39.4% accurate.

We propose TEMARI (TEsting Mined specificAtions geneRallY)[1], a general and more accurate technique for testing *arbitrarily complex* finite state specs. One idea is to decompose complex specs into two-letter components that DSI can test. This idea is not general; complex specs cannot always be decomposed into two-letter components [34]. So, TEMARI reduces the problem of testing a complex spec to (1) finding two-letter temporal constraints that hold on the spec, and (2) testing if constraints on the spec also hold on the program. TEMARI is more accurate than DSI because it also uses a state-aware approach for testing. If a constraint holds on the spec but not on the program, TEMARI reports the spec as buggy.

In more detail, to find two-letter constraints, TEMARI first uses Linear Temporal Logic (LTL) templates to guess constraints on all distinct method pairs in a complex spec. Then, it calls a model checker to find which guessed constraints hold on the spec. To more accurately test if constraints on the spec also hold on the program, TEMARI first checks if the methods share program state that one of them modifies. Intuitively, methods that do not share state can be called in any order. So, if the methods do not share state, TEMARI reports the spec as buggy. If the methods share state, TEMARI reports the spec as buggy if DSI finds the constraint to be buggy.

Our evaluation results show that TEMARI is general: we use it to test 1,726 complex specs that DSI cannot directly test, having up to 30 letters, 252 states, and 4,385 transitions. TEMARI finds 1,451 of those specs to be buggy. Also, state-*aware* TEMARI is 78.9% accurate, compared to 39.4% for state-*unaware* DSI. Our manual inspection of DSI inaccuracy yields ideas for further improving TEMARI's accuracy.

This paper makes the following contributions:

- ⋆ **Study.** We show that DSI, the state-of-the-art technique for testing two-letter finite state specs, is only 39.4% accurate on 1,470 specs in the ground-truth dataset that we curate.
- ⋆ **Technique.** We propose TEMARI which uses model checking to test arbitrarily-complex specs and state-awareness to

[1]Pun on the Japanese toy: https://en.wikipedia.org/wiki/Temari_(toy)

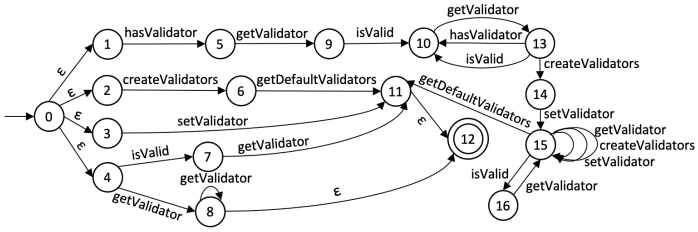

Fig. 1: DSM [48] mines a complex finite state spec that TEMARI can directly test but DSI [45] cannot.

improve accuracy of spec testing. Results are encouraging, showing that TEMARI is 78.9% accurate.

⋆ **Artifacts.** TEMARI and our data are publicly available [52].

II. ILLUSTRATIVE EXAMPLE

We illustrate how TEMARI tests complex specs. Figure 1 shows a spec that DSM [48] mined from validator [53]; it has 16 states, 23 transitions, and involves seven methods—it is a seven-letter spec. For testing, we add the initial state ⓪ and its four outgoing ϵ transitions. DSI can only test two-letter specs involving two methods; it cannot directly test complex specs like the one in Figure 1.

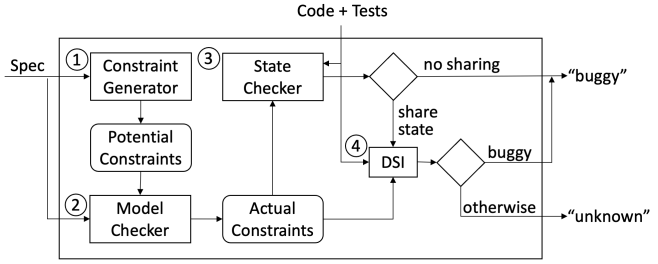

Fig. 2: TEMARI workflow.

TEMARI takes an arbitrarily-complex spec and outputs a verdict indicating if the spec is "buggy" or its status is "unknown". Figure 2 shows the TEMARI workflow for testing a complex spec in four steps.

Step ①: TEMARI generates *candidate* constraints using a set of Linear-Time Temporal Logic (LTL) templates over two symbols that were proposed in prior related work. It instantiates these templates by replacing the variables in the templates with method identifiers from the input spec.

Step ②: TEMARI uses model checking to find which candidate constraints hold on the input spec. A constraint that holds captures temporal order that the spec prescribes on the program, even if those methods are not always labels of consecutive transitions in the FSM. Conceptually, such constraint is a higher-order spec of a complex spec. For example, in Figure 1, TEMARI finds that constraint `C1` holds: "`getDefaultValidators` is always preceded by `createValidators`". Note that methods in `C1` do not label consecutive transitions in the FSM sub-path 13, 14, 15, 11.

Step ③: TEMARI uses a two-step approach to check whether the program needs to satisfy constraints that hold on the spec for its tests to pass. First, it checks if the two methods share state that at least one of them modifies. If they do not share such state, then the methods can be called in any order and TEMARI reports that the complex spec is buggy. For example, TEMARI finds that the two methods in constraint `C1` above do not share state (we also confirmed by hand).

Step ④: if the methods share state, then TEMARI invokes DSI [45] to test the constraint. DSI mutates the program at runtime by swapping the order in which the two methods in the constraint are called. The expectation is that some test will fail on the mutated program. DSI takes the absence of a test failure on the mutant to indicate that the methods are unrelated and that the constraint is (likely) buggy. If DSI reports that the constraint is buggy, then TEMARI reports the complex spec as buggy. Otherwise, TEMARI proceeds to check the remaining constraints that hold on the complex spec.

If TEMARI finds no constraint where the methods do not share state, or that DSI reports to be buggy, then TEMARI reports an "unknown" status—it did not find a bug in the complex spec. Section IV describes the TEMARI workflow in more detail and Section III-B illustrates how DSI works.

TABLE I: Open-source projects that we evaluate in this paper.

ID	PROJECT	SHA	KLOC	TC	TM	SC	BC
P1	jtar [56]	4b669	1.0	2	8	.75	.67
P2	codec [57]	b959a	23.9	58	936	.97	.93
P3	validator [53]	72734	16.7	70	554	.86	.74
P4	exec [58]	2ca7c	3.6	13	89	.71	.61
P5	convert [59]	8f8bf	5.4	9	160	.76	.72
P6	fileupload [60]	55dc6	4.8	9	34	.81	.77

III. SPEC MINING AND TESTING: HOW FAR ARE WE?

This section reports on our preliminary study of the bugginess of mined specs (Section III-B) and the accuracy of DSI (Section III-C). It is important to assess both. Spec testing would be less relevant if mined specs are rarely buggy. Also, improving the accuracy of spec testing—one of TEMARI's goals—would be less relevant if DSI is highly accurate.

A. Preliminary Study Setup

Projects Studied. Table I shows the six projects that we evaluate; they were used in prior work [9], [54], [55]. "KLOC" is number of thousands of lines of code, "TC" is number of test classes, "TM" is number of test methods, and "SC" and "BC" are statement and branch coverage, respectively.

Obtaining a Ground-Truth. Studying mined spec bugginess and DSI accuracy "in the wild" requires a ground truth. But, no such ground-truth dataset exists that we know of. So, we curate a ground-truth dataset. To do so, we run a stripped-down version of a miner in our study (BDDMiner [34]) on the six projects and manually inspect *all* resulting specs. We use this stripped-down miner because (1) it is the only miner that DSI was evaluated on [45]; (2) we seek an unbiased set of specs; (3) we do not want to reuse the miners that we evaluate (Table II); (4) it is available in Gabel and Su's prototype [45]; and (5) it produces simple two-letter specs which are easier for humans to inspect [50].

Two co-authors of this paper manually classify 1,470 specs through multiple rounds of independent inspection and reviews, and joint discussion with a third co-author. They read

TABLE II: Specs mined during our preliminary study.

Miner	Year	Specs			
		Σ	?%	true %	buggy %
BDD-3	2008	3.5M	92.6	0.7	6.8
BDD-2	2008	54664	93.4	0.9	5.7
DICE	2021	163	58.9	2.5	38.7
DSM-Randoop	2018	161	61.5	2.5	36.0
DSM-Manual	2018	98	48.0	1.0	51.0
Javert	2008	382	69.6	1.1	29.3
Texada	2015	45.1M	87.6	3.9	8.6
AVG	-	-	73.1	1.8	25.1

the code and any documentation of methods in each spec, and manually check if the spec holds. They write a passing test that violates a suspected buggy spec as a counterexample to show that the spec is buggy. Lastly, they classify each spec as **true** (i.e., the temporal ordering is always or sometimes necessary for program correctness), or **buggy** (i.e., the temporal ordering is not required for program correctness). We estimate that manual inspection took four person months to complete.

B. On the bugginess of mined specs

Table II shows seven variants of five miners from the literature that we evaluate and the "Year" they were published. **BDDMiner** [34] uses a BDD-based algorithm to mine specs that match regular expression templates with two (BDD-2) or three (BDD-3) letters. **DICE** [47] first mines specs using traces from a test suite and then improves those specs by adversarially generating test cases that are counterexamples for the initially mined specs. **DSM** [48] uses deep learning to mine specs from traces of manually written (DSM-Manual) or automatically generated (DSM-Randoop) tests. **Javert** [49] composes two- and three-letter specs into complex specs. **Texada** [31] mines specs from LTL templates.

These miners produce specs that are expressible as FSMs. We run JavaMOP's [61] LTL-to-FSM translator on Texada specs and JavaMOP's ERE-to-FSM translator on BDDMiner specs. We use each miner's default configuration. Note that DSM and DICE mine few specs because they mine specs per class. Also, Javert produces much fewer specs than BDD-2 and BDD-3 because it chains specs from BDDMiner to form more complex specs and uses fewer templates in doing so. Lastly, Texada's `-no-vacuous-findings` optimization still produces over 30M specs, so we report the number from the default configuration to be consistent with other miners.

We want to estimate the extent to which the specs produced by these miners are likely buggy. For that, we use a heuristic based on Legunsen et al.'s finding that complex specs containing buggy consecutive transition pairs are often problematic [9]. Specifically, we compute all consecutive transition pairs in the mined specs, and compare them with our ground-truth dataset. If a complex spec has a consecutive transition pair that matches a buggy two-letter spec, we consider the complex spec as likely buggy—it may wrongly prescribe temporal order on two methods. If the only matches in our ground-truth dataset are true specs, then we have no evidence that the complex spec is likely buggy.

Mined spec bugginess. Table II shows the results ("Specs" column). Σ is the number of unique specs mined, ?% is the percentage of Σ for which we find no matching transition pairs, true % and buggy % are percentages of Σ where all matches that we find are true and buggy, respectively. On average, 25.1% of mined specs per project are likely buggy. Considering only the 5,882,455 mined specs with a match, we find that 70% are buggy. Our expensive manual inspection only helps find matches in 12% of all mined specs. The "buggy %" column shows in gray that projects with high ratios of matched pairs also specs have high ratios of buggy specs. The unmatched 88% may contain likely buggy specs, but users would need to expend huge efforts to manually inspect more. So, there is a need for better automated spec testing. In sum, we find that mined specs are often likely buggy, and that *all miners that we evaluate produce specs that are likely buggy*.

TABLE III: Mean number of states (States), transitions (Trans), letters (Letters), and consecutive transition pairs (TP) per mined spec.

Miner	States	Trans	Letters	TP
BDD-3	3.7	7.3	3.0	17.1
BDD-2	1.6	2.6	2.0	4.7
DICE	5169.7	5187.3	6.7	5297.4
DSM-Randoop	120.6	2585.4	7.5	53691.7
DSM-Manual	12.8	52.7	6.7	368.7
Javert	3.3	3.8	3.8	6.7
Texada	3.5	10.7	4.0	33.6

Mined spec complexity. Sampling a few of the mined specs to inspect may be a way to manually validate them. But, sampling may also be impractical because mined specs can be quite complex and hard to comprehend. For example, Table III shows the mean number of states, transitions, letters, and unique consecutive transition pairs for all mined specs. DICE, DSM-Manual, DSM-Randoop have high mean number of states and transitions, even though their mean number of letters is less than 10. Outliers play a role here. The maximum numbers of states and transitions are 132,730 and 158,859, respectively. Also, Texada mines many specs and makes the overall mean number of states and transitions look small. Still, prior work suggests that specs with this mean number of states and transitions could be challenging for humans to inspect [9], [50]. Generalized automated spec testing can help.

C. On the (in)accuracy of DSI

We conduct the first large-scale evaluation of DSI [45] accuracy. DSI's ideas, algorithms, and implementation inspired and enabled this paper. However, there was no large-scale evaluation of DSI accuracy on open-source projects. The DSI paper [45] reports finding no false positives from manual inspection of a small number of specs. But, we cannot tell how many specs (out of 8,000) were manually inspected (the paper mentions 25 and "a small, randomly sampled collection").

Miners produce complex specs, but DSI can only test two-letter specs that describe a temporal order between two methods. We write `a()` and `b()` to refer to the first and second methods, respectively. To test a two-letter spec, DSI mutates the program so that calls to `b()` come before calls to `a()`. If the mutated program crashes or some test fails, DSI reports the spec as true because violating the two-letter spec seems to make the program incorrect. Otherwise, if all tests pass and

TABLE IV: Results on DSI accuracy evaluation.

ID	Σ		Acc	True		Spurious	
	True	Spurious		pre[%]	rec[%]	pre[%]	rec[%]
P1	4	6	30.0	100.0	25.0	66.7	33.3
P2	26	237	31.6	35.1	50.0	94.6	29.5
P3	34	519	50.3	55.2	47.1	97.8	50.5
P4	56	195	37.5	29.4	8.9	89.0	45.6
P5	79	83	25.3	32.4	15.2	63.0	34.9
P6	23	208	34.6	22.7	21.7	97.4	36.1
ALL	222	1248	39.4	36.4	23.4	92.8	42.2
AVG	37.0	208.0	34.9	45.8	28.0	84.7	38.3

(a) (In)accuracy of DSI.

Reason	Σ	%Sp	%!∩
Oracle-Related Inaccuracy	403	70.2	30.5
Inaccuracy due to Return-Value Replacement	213	90.6	61.0
Inaccuracy due to Delay-Induced Interference	128	90.6	48.4
Inaccuracy due to Expected Exceptions	21	81.0	28.6
Inaccuracy due to DSI Limitations	97	82.5	30.9

(b) Causes of DSI inaccuracy.

there is no crash, DSI reports the spec as buggy because the program seems to remain correct when it violates the spec.

To evaluate accuracy, we compare DSI verdicts with our manual classification on 1,470 specs in our ground-truth dataset (Section III-A). Table IVa shows the results; Σ is the number of mined specs, Acc is the percentage of Σ for which DSI agrees with our manual classification, "True" columns (respectively "Spurious") show true (respectively buggy) specs, and pre[%] and rec[%] show precision and recall, respectively.

Table IVa shows that DSI is inaccurate—its verdicts are only correct for 39.4% of 1,470 specs in our ground-truth dataset (cell highlighted). The precision and recall for true specs, and the recall for buggy specs are also quite low, but DSI precision on buggy specs is high: 92.8% (cell highlighted). So, when DSI classifies a spec as buggy, it is almost always correct. We leverage this high precision on buggy specs in TEMARI.

To obtain insights on how to improve accuracy, we manually analyze all cases where DSI misclassifies a spec. Table IVb summarizes causes of DSI inaccuracy ("Reason") that we find, the number of specs impacted by each cause ("Σ"), and percentages of those specs that we manually find to be buggy ("**%Sp**"). Section IV discusses column "**%!∩**".

We conjecture that two differences between the original evaluation [45] and our study setup illuminate DSI inaccuracy: (1) Prior DSI evaluation used the DaCapo benchmarks (which have no unit tests), but we use unit tests in active open-source projects. Users may run DSI with unit tests, and modern spec miners use unit tests, e.g., [25], [47], [48], [51], [62]–[65]. (2) Prior DSI evaluation only used crashes as oracles to test program correctness after DSI mutation, but we use crashes *and* unit test failures. Unit tests can be stronger oracles—they capture the intended usage by the API's developers.

```
1 boolean execValMethod(Map<String, Object> params) {
2   if (this.valMethod == null) {
3     Loader loader = this.getLoader(params);
4     // this.loadClass(loader);// a()@before
5     this.loadMethod(loader); // b()
6     this.loadClass(loader);// a()@after
7   } ... }
```

Fig. 3: DSI correctly identifies a true spec.

Examples. We give examples from open-source projects of how DSI classifies two-letter specs, and causes of inaccuracy. Figure 3 shows a two-letter spec from validator [53] that we manually find to be a true spec. DSI correctly classifies that spec as true. In the rest of this paper, we use the format in Figure 3 to show *both* the original program and the mutant that DSI creates. The "// a()@before" comment indicates that line 4 is the old location of `a()` in the original program whereas "// a()@after" shows that line 6 is the new location of `a()` in the mutant. In Figure 3, `loadClass()` is the only location where field `validationClass` (not shown) is assigned a non-null value, and `loadMethod()` uses that field. At runtime, when DSI delays the call of `loadClass()` to be after the call of `loadMethod()`, `loadMethod()` throws a `NullPointerException`. So, DSI reports the spec as true—the temporal order of `loadClass()` and `loadMethod()` seems necessary for program correctness.

```
1 boolean execValMethod(Map<String, Object> params) {
2   if (this.valMethod == null) {
3     Loader loader = this.getLoader(params); ...
4     // this.loadMethod(loader);// a()@before
5   }
6   Object[] values = this.getParamVals(params); // b()
7   this.loadMethod(loader);// a()@after ... }
```

Fig. 4: DSI correctly identifies a buggy spec.

In Figure 4, DSI correctly classifies another spec mined from validator as buggy. There, `loadMethod()` sets field `validationMethod` (not shown), and `getParamVals()` returns an array constructed using another field `methodParameterList` (also not shown). We manually find no data dependencies between these methods, and no crash or test failure occurs when DSI delays `loadMethod()` to be after `getParamVals()`. So, DSI is correct: the methods can be called in any order.

```
1 String[] soundex(...) { ...
2   for (String rep : reps) {
3     // branch.processRep(rep);// a()@before
4   } ...
5   branch.finish(); // b()
6   branch.processRep(rep);// a()@after
7   return branch.toString(); }
```

Fig. 5: DSI misclassifies a true spec as buggy (false negative).

Figure 5 shows a true spec in codec [57] that DSI misclassifies as buggy (false negative). There, `processRep()` ensures that field `builder` (not shown) is below its maximum length, but `finish()` pads `builder` to fill that maximum length. So, `processRep()` should be called before `finish()`. But, we find that the test oracle in this case is weak and did not crash or fail after DSI mutation.

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

Fig. 6: DSI misclassifies a buggy spec as true (false positive).

Figure 6 shows a buggy spec in codec that DSI misclassifies as true (false positive). After DSI delays `isNextChar()`, it also replaces its return value in the original location with `false`. (Line 3 changes from "`if (isNextChar(...))`" to "`if (false)`"). Return-value replacement alters control flow and corrupts `code`. So, `meta`'s return value is not the expected value, the test fails, and DSI misclassifies the spec as true; we find that `a()` and `b()` can be called in any order.

Weak oracles and return-value replacement are two of the most prevalent causes that we find for DSI inaccuracy (see Table IVb). The ability of TEMARI to check whether the methods `a()` and `b()` in a spec share state significantly improves the accuracy of classification *even in the presence of these causes of inaccuracy*.

IV. TECHNIQUE

We describe TEMARI rationale, design, and implementation.

A. Rationale

DSI has two big limitations: (1) it can only test two-letter specs, although miners often produce more complex specs (Section III-B); and (2) it is inaccurate (Section III-C). TEMARI leverages two insights to address these limitations.

Insight #1: Complex Specs can have two-letter constraints. DSI can only test two-letter specs, and no technology today can directly test complex specs having more than two letters.

To bridge the gap, one possible design choice is to extend DSI's algorithm to create one mutant per permutation of methods in a complex spec. But, such an approach would be too expensive since there would be too many permutations per spec. For example, there are at least $P_7^7 = 5,040$ mutants just for the spec in Figure 1, and it is not clear how one should mutate the program to violate a 7-letter spec.

We follow the opposite approach: *"reduce" the problem of testing a complex spec "s" to the problem of testing simple two-letter constraints entailed by "s", which DSI can check*. Although there is proof that decomposing specs into two-letter components is not always possible [34], we note that complex specs can have two-letter constraints, even when the methods involved are not always labels of consecutive transition pairs in the FSM. An example is the constraint `C1` in Figure 1, which we describe in Section II. Section IV-B1 and IV-B2 describe how we obtain two-letter constraints from complex specs.

Insight #2: State-awareness can improve DSI accuracy. During manual analysis of DSI's inaccuracy, we observe that *when `a()` and `b()` do not intersect on program P's state that one of them modifies, the spec likely is buggy*.

Table V shows the breakdown of state-related cases that we manually find; Σ is the number of specs that we label as state-related and **%Sp** is the percentage of Σ that we manually find to be buggy. As an illustration, the first row of Table V shows our finding that all 350 specs where `a()` and `b()` modify different parts of P's state are buggy. But DSI is inaccurate on 169 of these 350 specs. Other rows are similar. Cases in "Pure setter and Stateful method" and "Only one method alters P state" do not have 100% because some specs share state. The

TABLE V: Relating `a()`, `b()`, state, and buggyness.

Description	Σ	%Sp
`a()` and `b()` are stateful but unrelated	350	100.0
Neither `a()` nor `b()` alters P state	88	100.0
Only one method alters P state	241	96.3
Pure setter and Stateful method	182	86.3

%!∩ column in Table IVb shows the percentage of Σ where both methods in the spec do not share state. The numbers in that **%!∩** column also show that state-awareness can help *even in the presence of* all the causes of inaccuracy that we find. Based on these observations, we conjecture that checking if `a()` and `b()` share state *before* performing DSI mutation would help to more accurately classify specs than running DSI alone. We leverage this insight in the design of TEMARI, which we describe next.

B. Design

The inputs to TEMARI are spec S, program P, a set of tests T for P, and a set of templates which are LTL formulas over two symbols. TEMARI outputs "buggy" if it detects that S wrongly prescribes temporal order on methods in the program. Otherwise, TEMARI outputs "unknown" to indicate that it did not find a bug in S. (TEMARI does testing; it only shows the presence of bugs in specs, not the absence of bugs.)

Figure 2 showed the TEMARI workflow, which we now discuss in three parts: (1) guessing constraints that hold on the input spec (Step ① in Figure 2, Section IV-B1), (2) using a model checker to find guessed constraints that hold on the spec (Step ② in Figure 2, Section IV-B2), and (3) using our state-aware version of DSI to check if constraints that hold on the spec also hold on the program (steps ③ and ④ in Figure 2, Section IV-B3). The rest of this section describes Algorithm 1, which shows a detailed design view of TEMARI.

Algorithm 1 TEMARI algorithm

Inputs: S: a spec, P: a program, T: a set of tests for P,
Templates: a set of LTL formulas on two symbols x, y
Output: "buggy": S prescribes a buggy constraint, or
"unknown": no buggy constraint is found in S

```
1:  procedure main(S, P, T, Templates)                ▷ Main procedure
2:    if isTwoLetterSpec(S) then return DSI(P, T, S)
3:    C_holds ← {}                                    ▷ Constraints that hold on S
4:    M_pairs ← allDistinctPairs(S)                   ▷ All distinct pairs of labels in S
5:    for all (m1, m2) in M_pairs do
6:      for all t in Templates do                     ▷ guess a constraint on S
7:        constr(m1, m2) ← t.replaceAll(x, m1).replaceAll(y, m2)
8:        if S ⊨ constr(m1, m2) then
            ▷ a model checker confirms that S entails constr(m1, m2)
9:          C_holds ← C_holds ∪ {constr(m1, m2)}
10:   for all constr(m1, m2) in C_holds do
11:     dsiOutcome ← DSI(P, T, constr(m1, m2))
12:     if dsiOutcome == "buggy" then return "buggy"
13:   return "unknown"
14: procedure DSI(P, T, S)                            ▷ Improved DSI
15:   (m1, m2) ← getMethods(S)
16:   if !shareState(m1, m2) then return "buggy"
17:   if runDSI(P, T, S) == "buggy" then return "buggy"
18:   return "unknown"
```

1) Constraint Generation (lines 4–7): TEMARI checks whether two-letter constraints that hold on a complex spec S also hold on a program P. To obtain those two-letter constraints, TEMARI generates all distinct pairs of methods that are labels of transitions in S (line 4) and "plugs" them into LTL templates to create *candidate constraints* (line 7). Section IV-B2 discusses how TEMARI uses model checking to filter out candidate constraints that do not hold on S (line 8).

Generating all pairs of methods in S is guaranteed to terminate—FSMs have finite transitions. As templates for candidate constraints, TEMARI uses these four two-symbol LTL formulas that we obtain from prior work [47], [50], [66]:

(a) x is always followed by y: $\square(x \Rightarrow \circ\Diamond y)$
(b) x is always immediately followed by y: $\square(x \Rightarrow \circ y)$
(c) x is always preceded by y: $(!x\ W\ y)$
(d) x is always immediately preceded by y:$\Diamond x \Rightarrow x\, U (y \wedge \circ x)$

We choose these four templates from the six that are used in those prior works because the other two contain negation and DSI does not handle negation. For each pair (m_1, m_2) of methods generated from S, TEMARI replaces all x with m_1, and all y with m_2 to obtain a candidate constraint.

2) Constraint Filtering via Model Checking (line 8): Model checking is a method for checking whether a system model satisfies a spec [67]. If the model does not satisfy the spec, the model checker says "no" and produces a counterexample. Otherwise, the model checker says "yes". To filter out candidate constraints (Section IV-B1) that do not hold on the input spec S, TEMARI treats S as the model and each candidate constraint as a spec of S. Conceptually, each constraint is a higher-order spec of the system. Then, TEMARI calls a model checker to verify if S satisfies a constraint (line 8). If the model checker says "yes", TEMARI adds that constraint to set C_{holds}, which are constraints that hold on S (line 9).

Note that Le et al. [66] first used model checking to obtain two-letter constraints from complex specs. Their goal is to find two-letter components that can be combined across specs from multiple miners to obtain an improved spec. Differently, we are the first to use model checking for testing specs, and our goal is to obtain a principled approach for generalizing the testing of finite state specs beyond the two-letter specs that current techniques [27], [45] are limited to.

3) Do constraints on S also hold on P? (lines 10–18): By the time execution reaches line 10, TEMARI has filtered out all the candidate constraints that do not hold on S. The next step is to check whether the constraints in C_{holds} also hold on the program P. If P does not satisfy a constraint $\mathsf{constr}(\mathsf{m_1}, \mathsf{m_2})$ in C_{holds}, then TEMARI reports the spec as "buggy". The reason is that S is overly strict—it incorrectly prescribes temporal order between m_1 and m_2. TEMARI terminates and reports an "unknown" status on S if: (1) C_{holds} is empty, (2) C_{holds} is not empty and all constraints in C_{holds} also hold on P, or (3) TEMARI cannot verify that at least one constraint in C_{holds} does not hold in P, e.g., if DSI returns "unknown" for all constraints in C_{holds}.

To check if a constraint in C_{holds} also holds on P, TEMARI uses our version of DSI that is improved in two ways. The first improvement is shown on line 16: TEMARI first checks if m_1 and m_2 share state that m_1 or m_2 modifies. That is, TEMARI checks if there is a read/write, write/read, or write/write relation on program state. Intuitively, if m_1 and m_2 do not intersect on any part of program state, or if they only have a read/read relation on state that they share, then m_1 and m_2 can be called in any order. So, the complex spec S on which the constraint holds is buggy. Our design decision to first check for shared state is grounded in our preliminary study (Section III), which shows that 96.1% of two-letter specs with methods that do not share state are buggy.

The second improvement that we make to the original DSI algorithm is to make it use unit tests. The original DSI algorithm (Section III) used program crashes as oracles. We have modified the runDSI algorithm that TEMARI invokes on line 17 to run each test in T. It aggregates the results across all those test runs to determine whether a two letter spec (or constraint) is buggy. If no related test fails after mutating P to violate the spec under test, then DSI reports the spec as buggy. If all related tests fail, then DSI reports the spec as true. Lastly, if some related tests pass and others fail, then DSI conservatively reports the specs as unknown. Due to space limitation, we elide the details of our enhanced DSI algorithm that also uses unit tests.

If DSI finds that some constraint that holds on S does not hold on P, it has found evidence that spec S is buggy. In that case, TEMARI terminates and outputs "buggy". Otherwise, if all constraints in C_{holds} are reported as true or unknown by DSI, then TEMARI outputs "unknown". We base this design decision on our preliminary study results, which show that DSI is 92.8% precise when classifying buggy specs (Table IVa).

C. Implementation

Generating all pairs of methods in a complex spec S is trivial. For model checking, we use SPIN [68]. We automate the conversion of specs produced by the miners in Section III into PROMELA, check all candidate constraints on each S, and collect the constraints that hold. We implement our two enhancements (state-awareness and utilizing unit tests) to the original DSI algorithm on top of Gabel and Su's prototype [45] and implemented a Maven plugin that enabled us to integrate with a modern testing framework. To check if two methods share state, we implement a dynamic approach based on AspectJ [69] to collect all the values that a method reads or writes at runtime. AspectJ cannot track modification to array indices [70]. So, if both methods access the same array, the dynamic state checker in TEMARI conservatively reports that two methods share state so that DSI can be invoked.

V. EVALUATION

A. Quantitative Results

We answer the following research questions:

RQ1 How generalizable is TEMARI?

RQ2 What is the accuracy of TEMARI, compared to the accuracy of DSI?

RQ3 What are the runtime costs of TEMARI, and how do they compare to those of DSI?

Experimental Setup. We evaluate TEMARI on the six open-source projects shown in Table I, using specs produced by the miners shown in Table II. All timed experiments are run on an Intel® Xeon® Gold 6348 machine with 512GB of RAM running Ubuntu 20.04.4 LTS, using 96 cores.

1) Answering RQ1—Generalization: RQ1 evaluates the ability of TEMARI to generalize to complex specs, i.e., its ability to test specs that are beyond DSI's capability. To that end, we first perform a "limit study" by to better understand TEMARI's limitations and correctness. To do so, we use TEMARI to test increasingly complex specs that DSI cannot test because it is limited to testing two-letter specs. To show that TEMARI is miner independent, we run this study on complex specs from all but one of the miners from Table II. We do not test Texada specs as none of Texada's templates entail constraints that match any of the four LTL templates that we use for candidate constraint generation.

In our limit study, we use TEMARI to test 100 complex specs that DSI cannot test, having up to 3,480 candidate constraints, 30 letters, 252 states, and 4385 transitions. (Average: 493.8 candidate constraints, 45.1 states, 233.4 transitions, and 9.4 letters.) We select these 100 specs as follows. We start with the smallest number of states in a mined FSM and select a fixed number of specs having that number of states across the miners. Then, we increment the number of states and repeat the process until we reach 255, the maximum that SPIN's `mtype` allows us to model.

TEMARI finds 59 of the 100 complex specs in our limit study to be buggy. Two co-authors of this paper manually inspect 29 buggy constraints in 15 of these buggy complex specs that have up to five total constraints. For each, we analyze the FSM to check if the constraint holds and then we manually inspect the code to check if the constraint is buggy. We did not inspect more constraints because specs with more than five constraints tend to have FSMs that are too complex for us to understand. We find so far that TEMARI is correct during our limit study. In sum, our limit study shows that TEMARI is feasible for correctly testing arbitrarily complex specs up to the limits of the model checker.

To evaluate TEMARI generalization more broadly, we also use it to test 1,626 other complex specs that are not in our limit study. These specs have up to 3,248 candidate constraints, 29 letters, 179 states, and 570 transitions. (Averages: 56.4 candidate constraints, 5.1 states, 11.0 transitions, and 3.5 letters.) These are all the specs obtained from DICE, DSM-Manual, DSM-Randoop, and Javert that have less than 255 states (626 specs), and 1,000 randomly selected specs from BDD-3. We do not inspect the buggy constraints as we did in the limit study, since our goal is to run TEMARI on more complex specs. TEMARI finds 1,451 of the 1,626 specs in our breadth study to be buggy, showing the potential to test many more mined specs.

Overall, across our limit and breadth studies, TEMARI is able to test 1,726 complex specs that DSI cannot directly test, having up to 30 letters, 252 states, and 4385 transitions. TEMARI finds bugs in 1,451 of those complex specs.

TABLE VI: Accuracy comparison of TEMARI and DSI.

ID	Σ	# Accurate Spec		Acc	
		DSI	TEMARI	DSI	TEMARI
P1	10	3	7	30.0	70.0
P2	263	83	209	31.6	79.5
P3	553	278	512	50.3	92.6
P4	251	94	168	37.5	66.9
P5	162	41	88	25.3	54.3
P6	231	80	176	34.6	76.2
ALL	1470	579	1160	39.4	78.9

2) Answering RQ2—Accuracy: Section V-A1 provides evidence that TEMARI is general and allows to test many complex specs that DSI cannot directly test. So, we next turn our attention to comparing the accuracy of TEMARI and DSI. Since DSI can only test two-letter specs and TEMARI can also test two-letter specs (see line 2 in Algorithm 1), we compare the tools on the 1,470 specs from our ground-truth dataset. Table VI shows the results; "Σ" is the number of mined specs, "# Accurate Spec" (respectively, "Acc") is the absolute number (respectively, percentage) of specs that DSI and TEMARI correctly classify. The results in Table VI shows that TEMARI is 39.5 percentage points more accurate than DSI—78.9% vs. 39.4%. That is, TEMARI correctly classifies more than twice as many specs as DSI—1160 vs. 579.

3) Answering RQ3—Runtime Costs: DSI is very expensive to run [45]. So, it is important to measure the costs of the TEMARI approach to generalize testing of complex finite state specs. Since a direct comparison between DSI and all TEMARI features is not possible due to the mismatch in the complexities of specs that they can test, we first report on the runtime costs of TEMARI. Then, we report the costs of dynamic state measurement in TEMARI and compare the overall cost of running TEMARI with the cost of running DSI on the 1,470 specs in our ground-truth dataset.

In our limit study, the average total time to test a spec is 417.1 seconds. The average model checking time per constraint is 0.6 seconds (there are 49384 candidate constraints in total). The spec that took TEMARI the most time to test—20,685—had 133 states, 2983 transitions, and 27 letters. Our implementation is not optimized, so it should be possible to reduce these costs.

TABLE VII: Runtime costs of TEMARI vs. DSI in minutes.

ID	Specs	State-no-DSI	State-with-DSI	DSI
P1	10	2.3	3.0	1.6
P2	266	484.8	557.8	414.2
P3	558	1106.2	3010.2	15693.1
P4	254	187.3	376.4	687.4
P5	162	819.0	1604.0	1896.1
P6	234	330.6	371.1	152.3
ALL	1484	2930.3	5922.5	18844.7
AVG	247.33	488.4	987.1	3140.8

Table VII shows the runtime costs of TEMARI and DSI in *minutes*. Recall that TEMARI first checks if the two methods share state, and then runs DSI *only on the specs for which it*

finds that the two methods share state. "Specs" is the number of specs per project. "State-no-DSI" is the time that TEMARI incurs in checking whether the two methods in the spec share state. "State-with-DSI" is the sum of the TEMARI times to perform state checking and to invoke DSI on specs that it finds to share state. "DSI" is the time to run simply run state-unaware DSI on the specs. We incorporate state-awareness in TEMARI to improve classification accuracy. Surprisingly, comparing the AVG of the "State-with-DSI" with that of the "DSI" column, we see that checking program state before invoking DSI is $3.2x$ faster than only running state-unaware DSI on the two-letter specs in our ground-truth dataset. This speedup does not include model-checking costs. So, there is room to further reduce overall TEMARI runtime costs.

B. Qualitative Analysis

This section discusses the results of a manual analysis of DSI's misclassifications. We discuss the five sources of inaccuracy that we observed (Table IVb).

a. Oracle-Related Inaccuracy. Figure 5 shows a weak oracle caused inaccuracy, discussed in Section III-C. We also find other cases where test oracles are too weak to detect errors induced by DSI's mutation, including cases where a test passes even if `a()` or `b()` is not called. Separately, we find that the locations of JUnit assertions cause DSI inaccuracy. For example, the assertion on line 4 in Figure 7 fails before DSI can even begin its mutation. So, DSI classifies this spurious spec as "unknown". If line 4 had been after lines 6 and 7, then the test would have passed after DSI's mutation, and DSI would correctly classify the spec as buggy.

```
1 @Test void testExecute() { ...
2   // int exitValue = exec.execute(cl);// a()@before
3   int exitValue = 0;
4   assertEquals("FOO..", baos); // fail
5   assertFalse(exec.isFailure(exitValue));
6   assertEquals(new File("."), exec.getWorkDir());// b()
7   exec.execute(cl); // a()@after
8 }
```

Fig. 7: The order of assertions causes DSI inaccuracy.

b. Inaccuracy due to Return-Value Replacement. Figure 6 shows an example DSI's return-value replacement leads to an unintended test failure that causes DSI to misclassify a buggy spec as true (false positive). We also find that DSI can misclassify specs if return-value replacement causes one test to fail because the actual value in an assertion no longer matches the expected value, but another test passes. So, DSI gets contradictory signals and classifies the spec as unknown even if it is buggy or true.

c. Inaccuracy due to Expected Exceptions. Tests that expect exceptions mislead DSI, because it treats all exception as program crashes. Treating all exceptions as crashes may have worked well for the DaCapo benchmarks on which DSI was originally evaluated, but it does not work well when using unit tests to validate specs. Consider buggy spec `s` in Figure 8: `b()` throws an exception regardless of `a()`'s call site, but the test passes as it expects an exception. DSI wrongly classifies `s` as likely true (false positive) because it successfully delays `a()` but cannot run `a()` after `b()` due to the exception that `b()` throws. In other cases, DSI's mutation causes an expected exception to not be thrown, so the test fails and a buggy spec is misclassified as likely true (false negative).

```
1  public class Base16 {
2    void decode(...) {
3      // byte[] buf = ensureBSize(); // a()@before
4      byte[] buf = null;
5      if (...) { validateTC(); } // b()
6      ensureBSize(); // a()@after //unreachable
7      ... }
8    void validateTC() { if (isStrict()) {
9      throw new IllegalArgumentException("strict"); } } }
10
11 @Test(expected=IllegalArgumentException.class)
12 void testDecoding() { b16.decode(...); }
```

Fig. 8: Expected exception causes DSI false positive.

d. Inaccuracy due to Delay-Induced Interference. Delaying `a()` while testing a spec `s` interferes with testing other specs due to state pollution, or because testing `s` requires violating a true spec. In one case, state pollution resulted from delaying `a()` while testing one spec. An assertion right fails after delaying `a()`. Method `b()` should delete a temporary file, `tempDir`, but `b()` is not called because of the assertion failure and no other test re-initializes `tempDir`. So, when DSI later uses a test that assumes `tempDir`'s initial state to test other specs, the test fails before DSI's mutation. So, DSI misclassifies all subsequent specs as "unknown".

Inaccuracy due to interference also occurs if delaying `a()` while using DSI to test a buggy spec `s2` causes a true spec `s1` to be violated (resulting in a crash or test failure). In Figure 9, `s1=(setOut(), start())` is a true spec, `s2=(setOut(), stop())` is a buggy spec, and `s2` is being tested. Delaying `setOut()` violates `s1`, causing an assertion failure that misleads DSI to wrongly classify `s2` as a likely valid spec.

```
1 int execute(...) { ...
2   try { // streams.setOut(os); // a()@before
3   } catch (...) {...}
4   streams.start(); // (setOut(), start()) form true spec
5   streams.stop(); // (setOut(), stop()) form buggy spec
6   streams.setOut(os); // a()@after }
```

Fig. 9: Spec inteference.

e. Inaccuracy due to DSI Limitations. DSI cannot test specs where `a()` and `b()` are called in different threads or processes; it classifies them as "unknown". DSI also classifies conditionally true specs as "unknown. As an example conditional spec, say `a()`: `setWatchdog()` sets `Watchdog`s to monitor timeouts, and `b()`: `execute()` attaches `Watchdog`s to processes. A `Watchdog` need not be set, but `a()` must be called before `b()` *if* one is set. These two limitations were not described [45]; we leave them as open problems.

VI. FUTURE DIRECTIONS

A. Ideas for further accuracy improvement

1) Reducing Oracle-Related Inaccuracy: Techniques for detecting and strengthening tests with weak oracles [51], [71], [72] could be applied to help improve accuracy. For inaccuracies due to the order and location of assertions, one possible mitigation strategy could be to delay not only `a()` but also

all "affected" assertions on `a()`'s return value that are called before `b()`. Doing so would lead to accurate classification in several cases that we analyze. Some challenges that would be involved are how to (1) let TEMARI know in advance (and accurately) which assertions are affected by `a()`'s return value or by state changes that `a()` induces, and (2) modify the DSI algorithm to account for delaying multiple statements.

2) Reducing Inaccuracy due to Return-Value Replacement: Future work could implement an improved return-value replacement scheme based on static analysis of how `a()`'s return value, henceforth called `AReturn`, is used. If `AReturn` is assigned to a variable, then TEMARI could also capture that variable. Then, when `a()` is called, `AReturn` is also assigned to the captured variable. Developing this improved scheme will involve (1) analyzing multiple def-use paths starting at `a()`'s original location; and (2) dealing with variable scope changes between `a()`'s location before and after TEMARI's mutation (as the captured variable may go out of scope).

3) Reducing Inaccuracy due to Expected Exceptions: Future work could capture the stack trace and program state at exception points during normal runs, and compare them with those obtained during DSI's mutation. If these are equal, then the spec is likely buggy or such tests are not providing good signals to DSI. Better handling of expected exceptions will require (1) ahead-of-time analysis to check that DSI's mutations may involve tests that expect exceptions; and (2) ensuring that the "exception capture and compare" mechanism is not influenced by test outcomes and that no new false positives or false negatives are introduced.

4) Reducing Interference-Related Inaccuracy: A new *multi-stage approach* for testing specs could reduce inaccuracy due to interference. In Figure 9, say `s2` the input to TEMARI. A first stage tests `s1`. If `s1` is found to be true, then a second stage can check `s3=(start(), stop())`. If `s3` is also true, the input `s2` is also true by transitivity. Note that this conclusion is reached without spec `s1` interfering. But, if `s3` is buggy, then a third stage delays *both* `setOut()` and `start()` till after `b()`. Since `s1` and `s3` are not inputs, a multi-stage approach would need to compute them on the fly. Also, `s1` and `s3` could be long method-call chains. So, efficient algorithms will be needed to avoid brute force search. Techniques for detecting, managing, and fixing state-polluting flaky tests [73]–[76] could be used to deal with state pollution. These techniques can be costly, so the cost of combining them with TEMARI would need to be addressed.

B. Static State Checking

Table VII shows a 3.2x speedup of TEMARI over DSI. But this speedup does not include model checking costs. So, future work should further speed up TEMARI. One idea is to use static analyses to check if two methods share state. Static analysis can produce false positives, but it does not rely on code coverage and it could be faster than a dynamic analysis. We implement a small proof of concept in which we use purity analysis [77] to check if methods in the specs in our ground-truth dataset share state. We find 89 specs that do not share state. The purity analysis analyzed all 1,470 specs in 40 minutes, compared to the 2930.3minutes to run dynamic state checking. So, (1) static analyses could be used to complement dynamic state checking; (2) more advanced analyses could statically detect additional specs where methods do not share state; and (3) static analyses with low false positive rates could help to further speed up TEMARI.

VII. DISCUSSION

A. Case study: a possible application of TEMARI

We conduct a small case study to investigate the utility of TEMARI for runtime verification, given the high ratio of buggy complex specs that it finds. Specifically, we use TEMARI to repair a buggy spec from our evaluation to see if the repaired version gives fewer false alarms among runtime verification violations. The spec is mined by DSM-Manual from the `LogOutputStream` class in exec [58]; it has 4 letters, 9 states, and 9 transitions.

First, we manually convert the spec to a JavaMOP spec and monitor it on 41 clients of exec that use `LogOutputStream` on GitHub. One client, PhantomJS-Wrapper [78], generates 57 violations. Our manual inspection shows that several violations are false alarms due to omitting a state and two transitions. We repair the spec to address these violations, and then re-monitor it. This time, there are 51 violations, and three of them are false alarms due to a constraint that TEMARI finds to be buggy. So, we repair the spec again to eliminate the buggy constraint. The resulting spec yielded 48 violations.

Our case study shows that TEMARI seems promising for helping to improve mined spec quality. Iterative and semi-automated repair will require viewing mined specs as intermediate products to be improved, rather than as final products. TEMARI can help fuel that paradigm shift, with the ultimate goal to improve spec quality. We plan to conduct a larger study on the utility of TEMARI for aiding repair.

B. What about NBPs?

We exclude a challenging type of spec from our ground-truth dataset: those that are vacuously true because `a()` transitively calls `b()`. Gabel and Su [45] call such specs "no-break-pass", or NBPs; they arise because flat traces—sequences of method calls—are commonly used for spec mining. Correctly filtering out callees from traces is challenging [34].

We manually inspect all 2,001 specs (including NBPs) that the independent miner produced and then developed a static checker to detect NBPs. We omit all 517 NBPs that the checker finds (after manually verifying its correctness) and use the checker to filter out NBP constraints during TEMARI runs. We also omit the 14 specs that are too difficult for us to manually inspect. Our accuracy results are based on the remaining 1,470 specs. If we include NBPs, then DSI and TEMARI accuracy are 53.8% and 84.4%, respectively. So, excluding NBPs does not affect our high-level conclusion about accuracy, but dealing with NBPs remains a challenge.

C. Testing manually written specs

TEMARI is agnostic to the origin of finite state specs. So, conceptually, it can also test manually written ones. Legunsen et al. [9] observe that mined finite state specs tend to suffer more than manually-written ones from "errors of commission"—they prescribe temporal order on methods that can be called in any order. Our analysis of their data also shows that manually-written specs suffer more from "errors of omission" by not including some methods that should have a temporal ordering relation with those that are in the spec. A different approach than TEMARI is needed to test for errors of omission, but if manually written finite state specs are buggy, TEMARI should be able to detect those.

D. Limitations

TEMARI is designed for testing specs that are expressible as FSMs; it is not intended for testing other kinds of specs, such as value-based invariants [51], [62], [71] or behavioral API specs that are written in non-regular languages like context-free grammars [79], or string rewrite systems [80].

In theory, TEMARI can test arbitrarily complex specs. But, in practice, the complexity of the specs that DSI can test is limited the model checker's capability. We note that TEMARI is agnostic to the model checker being used and future work can explore the limits on the complexity of specs that can be tested when using other model checkers other than SPIN.

Other limitations are the costs, false positives, and false negatives in our dynamic approach for checking if methods share state. If better and faster analyses are used in the future, TEMARI accuracy and performance will likely also improve.

E. Threats to validity

Our results may not generalize to other open-source projects beyond the six that we evaluate. Evaluating more projects in the future will be useful, but doing so will likely not invalidate the generalization and improved accuracy of TEMARI over DSI. Our results may be affected by errors in our implementation of TEMARI or in our experimental code. To mitigate this threat, we tested our tools and manually inspected the results. Also we make our tools and datasets publicly accessible for independent verification [52]. Our manual analysis of the specs that we test may suffer from divergent understanding among the co-authors involved. So, we had multiple rounds of review, via pull requests and in-person discussions to find consensus where possible. Finally, TEMARI uses the same assumption as spec mining and DSI that program P is correct and that tests exercise P correctly. We use this assumption because specs are needed to formally prove programs correct, and programs that are proven correct must already have specs.

VIII. RELATED WORK

Spec testing and debugging. Significance testing [27] can also only test two-letter specs; it mutates programs by statically deleting `a()`. Static mutation may be fast, but DSI's dynamic mutation yields useful knowledge about when the mutated program fails: right after `a()` is removed from its original location, when `b()` is invoked, when `a()` is invoked after `b()`, or much later. Such knowledge helps to better understand specs. Also, DSI can test more kinds of specs than significance testing. Separately from DSI and significance testing, TEMARI can test arbitrarily complex finite state specs.

`OASIs` [51], [71] uses program mutation to improve value-based specs in the form of test oracles. The ideas in `OASIs` are similar to those in DSI. Differently, TEMARI and DSI target finite state behavioral API specs. Nimmer et al. [81] use static program verification to validate value-based invariants produced by Daikon [82], but TEMARI uses program mutation to test specs of temporal relationships among method calls.

Ammons et al. [83] use concept analysis to reduce the number of traces that a user has to manually inspect when debugging a mined spec. TEMARI automates the process of detecting if a spec is buggy; a debugging approach could still be used if TEMARI does not find a spec to be buggy.

Unit testing and spec mining. Dallmeier et al. [25] use automatic test generation to enrich traces with the goal to improve mined spec quality. Pradel and Gross [32] mine specs from passing automatically generated tests, and check if those mined specs are violated during runs of failing generated tests. DICE [47] first mines specs and then uses guided test generation to find counterexample traces that it then uses to improve those mined specs. Deep Specification Miner (DSM) [48] also uses automatic test generation to obtain traces on which it uses deep learning to mine specs. Differently from these approaches, TEMARI uses unit tests to validate, rather than obtain, specs. Also, TEMARI is complementary: it can be used to test specs produced by these mining approaches.

Reducing false positives among mined specs. TEMARI is orthogonal and complementary to work on reducing false positives *during* spec mining, e.g., [6], [84]. That is, TEMARI can be used to test specs "post-mining", but these approaches filter out false positives "intra-mining". TEMARI should be used with these approaches, to further improve spec quality.

Model Checking and Spec Mining. Ghezzi et al. [85] use model checking to *mine* user-interaction specs in the form of Markov models from log files. Differently, TEMARI uses model checking to *test* finite state specs.

IX. CONCLUSIONS AND FUTURE WORK

We study and improve automated testing of behavioral API specs that are expressible as FSMs. DSI, the state-of-the-art technique, can only test specs involving two methods. Also, we find that DSI is only 39.4% accurate. So, we propose TEMARI, a more general technique that uses model checking to test arbitrarily complex specs and state-awareness to improve accuracy. We demonstrate generality by using TEMARI to test 1,726 complex specs involving up to 30 methods, and having up to 252 states and 4385 transitions. TEMARI finds 1,451 of those 1,726 to be buggy. TEMARI is 78.9% accurate, and we provide ideas that can be used to further improve accuracy in the future. We believe that our results and artifacts can help to spur a new era of research on testing finite state specs.

Data Availability. All code and data are available [52].

REFERENCES

[1] J. W. Nimmer, "Automatic generation and checking of program specifications," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[2] A. J. Brown, "Specifications and reverse-engineering," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 3, pp. 147–153, 1993.

[3] N. G. Leveson, "Intent specifications: An approach to building human-centered specifications," *IEEE_Software*, vol. 26, no. 1, pp. 15–35, 2000.

[4] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *ISSTA*, 2002, pp. 229–239.

[5] ——, "Invariant inference for static checking," in *FSE*, 2002, pp. 11–20.

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

[7] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *ICSE*, 2012, pp. 925–935.

[8] M. Acharya, T. Sharma, J. Xu, and T. Xie, "Effective generation of interface robustness properties for static analysis," in *ASE*, 2006, pp. 293–296.

[9] 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*, 2016, pp. 602–613.

[10] R. Purandare, M. B. Dwyer, and S. Elbaum, "Optimizing monitoring of finite state properties through monitor compaction," in *ISSTA*, 2013, pp. 280–290.

[11] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *PLDI*, 2011, pp. 415–424.

[12] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *FSE*, 2008, pp. 36–47.

[13] M. B. Dwyer, R. Purandare, and S. Person, "Runtime verification in context: Can optimizing error detection improve fault diagnosis?" in *RV*, 2010, pp. 36–50.

[14] S. Hussein, P. Meredith, and G. Roşu, "Security-policy monitoring and enforcement with JavaMOP," in *PLAS*, 2012, pp. 1–11.

[15] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *ASE*, 2013, pp. 70–80.

[16] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010, pp. 270–285.

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

[18] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *RV*, 2011, pp. 208–222.

[19] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," in *OOPSLA*, 2008, pp. 143–162.

[20] N. Decker, J. Harder, T. Scheffel, M. Schmitz, and D. Thoma, "Runtime monitoring with union-find structures," in *TACAS*, 2016, pp. 868–884.

[21] J. Spolsky, "Painless functional specifications part 1: Why bother?" in *Joel on Software*. Springer, 2004, pp. 45–51.

[22] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *ICSE*, 2013, pp. 262–271.

[23] L. Teixeira, B. Miranda, H. Rebêlo, and M. d'Amorim, "Demystifying the challenges of formally specifying api properties for runtime verification," in *ICST*, 2021, pp. 82–93.

[24] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *POPL*, 2002, pp. 4–16.

[25] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *ISSTA*, 2010, pp. 85–96.

[26] W. Weimer and G. Necula, "Mining temporal specifications for error detection," in *TACAS*, 2005, pp. 461–476.

[27] A. C. Nguyen and S.-C. Khoo, "Extracting significant specifications from mining through mutation testing," in *ICFEM*, 2011, pp. 472–488.

[28] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *ASE*, 2013, pp. 658–663.

[29] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *ICSM*, 2010, pp. 1–10.

[30] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *PLDI*, 2011, pp. 211–221.

[31] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *ASE*, 2015, pp. 81–92.

[32] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *ICSE*, 2012, pp. 288–298.

[33] 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*, 2014, pp. 779–790.

[34] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *ICSE*, 2008, pp. 51–60.

[35] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Iterative mining of resource-releasing specifications," in *ASE*, 2011, pp. 233–242.

[36] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, 2009, pp. 307–318.

[37] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *FSE*, 2014, pp. 166–177.

[38] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *FSE*, 2014, pp. 178–189.

[39] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE*, 2009, pp. 295–306.

[40] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009, pp. 371–382.

[41] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *ICSE*, 2011, pp. 591–600.

[42] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *ICSE*, 2010, pp. 15–24.

[43] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE*, 2006, pp. 282–291.

[44] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *TSE*, vol. 39, no. 5, pp. 613–637, 2013.

[45] M. Gabel and Z. Su, "Testing mined specifications," in *FSE*, 2012, pp. 1–11.

[46] "Grigore Roşu's Open Problems and Challenges (Archived)," https://web.archive.org/web/20190218131557/http://fsl.cs.illinois.edu/index.php/Open_Problems_and_Challenges#mining.

[47] H. J. Kang and D. Lo, "Adversarial specification mining," *TOSEM*, vol. 30, no. 2, pp. 1–40, 2021.

[48] T.-D. B. Le and D. Lo, "Deep specification mining," in *ISSTA*, 2018, pp. 106–117.

[49] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *FSE*, 2008, pp. 339–349.

[50] P. Sun, C. Brown, I. Beschastnikh, and K. T. Stolee, "Mining specifications from documentation using a crowd," in *SANER*, 2019, pp. 275–286.

[51] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *ISSTA*, 2016, pp. 247–258.

[52] "Tools and data for this paper," https://github.com/dsiTester/general-spec-testing.

[53] "commons-validator," https://github.com/apache/commons-validator.git.

[54] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," in *OOPSLA*, 2019, pp. 187:1–187:29.

[55] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *ICSE*, 2019, pp. 430–441.

[56] "kamranzafar.jtar," https://github.com/kamranzafar/jtar.git.

[57] "commons-codec," https://github.com/apache/commons-codec.git.

[58] "exec," https://github.com/apache/commons-exec.git.

[59] "convert," https://github.com/JodaOrg/joda-convert.git.

[60] "commons-fileupload," https://github.com/apache/commons-fileupload.

[61] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "Javamop: Efficient

parametric runtime monitoring framework," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1427–1430.

[62] F. Molina, M. d'Amorim, and N. Aguirre, "Fuzzing class specifications," in *ICSE*, 2022, pp. 1008–1020.

[63] Z. Cao and N. Zhang, "Deep specification mining with attention," in *International Computing and Combinatorics Conference*. Springer, 2020, pp. 186–197.

[64] Y. Gao, M. Wang, and B. Yu, "Dynamic specification mining based on transformer," in *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2022, pp. 220–237.

[65] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 288–298.

[66] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (t)," in *ASE*, 2015, pp. 115–125.

[67] E. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking, second edition*, ser. Cyber Physical Systems Series. MIT Press, 2018. [Online]. Available: https://books.google.com.br/books?id=OJV5DwAAQBAJ

[68] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[69] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *European Conference on Object-Oriented Programming*, 2001, pp. 327–354.

[70] "157031—array element get/set pointcut," https://bugs.eclipse.org/bugs/show_bug.cgi?id=157031.

[71] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Oasis: Oracle assessment and improvement tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 368–371.

[72] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 621–631.

[84] C. Le Goues and W. Weimer, "Specification mining with few false

[73] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *ISSTA*, 2015, pp. 223–233.

[74] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *ICSE*, 2014, pp. 550–561.

[75] J. Bell, G. E. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *ESEC/FSE*, E. D. Nitto, M. Harman, and P. Heymans, Eds., 2015, pp. 770–781.

[76] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE*, 2019, pp. 545–555.

[77] A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2005, pp. 199–215.

[78] "Phanomjs," https://github.com/moodysalem/java-phantomjs-wrapper.

[79] P. O. Meredith, D. Jin, F. Chen, and G. Rosu, "Efficient monitoring of parametric context-free patterns," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, 2008, pp. 148–157.

[80] P. O. Meredith and G. Rosu, "Efficient parametric runtime verification with deterministic string rewriting," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 70–80.

[81] J. W. Nimmer and M. D. Ernst, "Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java," in *RV*, 2001.

[82] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 213–224.

[83] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *PLDI*, 2003, p. 182–195. positives," in *TACAS*, 2009, pp. 292–306.

[85] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *ICSE*, 2014, p. 277–287.