I. Appendix

Algorithm 1 DSI usage in our preliminary study

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

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

1:
2: **procedure** runDSI(P, T, T_{map}) ▷ Main procedure
3: results ← {} ▷ Set of Tuples ($\mathsf{S}^f_{test}, \mathsf{S}^t_{test}$)
4: **for all** ℓ **in** {all, class, method} **do**
5: **for all** test **in** tests(ℓ, T) **do** ▷ tests(ℓ, T): level ℓ tests
6: $\mathsf{S}^f_{test}, \mathsf{S}^t_{test}$ ← validate(test, $\mathsf{T}_{map}[test]$, P)
7: results ← results ∪ ($\mathsf{S}^f_{test}, \mathsf{S}^t_{test}$)
8: **return** S^f, S^t
9:
10: **procedure** validate(test, $\mathsf{T}_{map}[test]$, P) ▷ DSI procedure
11: S^f ← {}, S^t ← {} S^u ← {} ▷ S^f: spurious specs, S^t: likely specs, S^u: specs that DSI could not invalidate
12: **for all** s **in** $\mathsf{T}_{map}[test]$ **do**
13: out1 ← run(P, test) ▷ out* ∈ $\{pass, fail, crash\}$
14: out2, trace1 ← instrRun(P, test) ▷ 1^{st} instrumented run
15: out3, trace2 ← instrRun(P, test) ▷ 2^{nd} instrumented run
16: **if** sanityCheck(s, out1, out2, out3, trace1, trace2) **then**
17: exps ← experiments(s, trace) ▷ Algorithm 1 in [1]
18: **else**
19: $S^u \leftarrow S^u \cup \{s\}$; *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: delay call at trace index 'e.idx' by 'e.len' calls
25: out4, stage ← runExp(P, I, e)
26: **if** stage == 3 ∧ out4 == pass **then**
27: $S^f \leftarrow S^f \cup \{s\}$; *continue*
28: **if** stage == 0 **then** $S^u \leftarrow S^u \cup \{s\}$; *continue*
29: **else** $S^t \leftarrow S^t \cup \{s\}$
30:
31: **procedure** sanityCheck($s, o1, o2, o3, t1, t2$)
32: **if** o1 ∈ $\{fail, crash\}$ **then return** false ▷ Base run crash
33: **if** o2 ∈ $\{fail, crash\}$ **then return** false ▷ Instrument fail
34: **if** o2 ≠ o3 **then return** false ▷ Nondeterministic runs
35: **if** s $\not\models$ t1) ∨ s $\not\models$ t2 **then return** false ▷ Traces not in s
36: **return** true

Algorithm 2 DSI usage in Temari

Inputs: P: a program, T: the whole test suite for P,
S: a two-letter spec

Outputs: "buggy" if S is likely buggy, and
"unknown" otherwise.

procedure runDSI(P, T, S) ▷ Run DSI with unit tests
out1 ← run(P, T) ▷ out* ∈ {pass, fail, crash}
out2, trace1 ← InstrRun(P, T) ▷ First instrumented run
out3, trace2 ← InstrRun(P, T) ▷ Second instrumented run
if sanityCheck(S, out1, out2, out3, trace1, trace2) **then**
exps ← experiments(S, trace) ▷ Algorithm 1 in [1]
else
return unknown
return runDSIExperiments(S, P, T, exps)

procedure runDSIExperiments(S, P, T, exps)
V ← {} ▷ Collect the set of DSI verdicts
for all e : (idx, len) **in** exps **do**
▷ Violate S; delay call at trace index 'e.idx' by 'e.len' calls
out4, stage ← runExp(P, T, e) ▷ mutate P, does T pass?
if out4 == OK **then** V ← V ∪ {buggy}
if stage ∈ {0, 1, 2, 3} **then** V ← V ∪ {unknown}
if V == {buggy} **then**
return buggy
else
return unknown

Algorithm 1 shows how we run DSI in our preliminary study; it extends DSI to also use assertions and to use multiple test runs. Algorithm 1 takes a program P, a set of tests T, and a map from each test to the specs that were mined from that test (T_{map}). The outputs are sets of likely buggy (S^f) and likely true (S^t) mined specs. Note that in Algorithm 1 we run tests across all granularity levels: test methods, test classes, and the whole test suite. The reason is that it is not clear a priori what granularity level one should use for mining and validation. But, we only use tests that mine a spec to validate that spec.

The entry point to Algorithm 1 is `runDSI` (lines 2–8). For all tests at the test method, test class, and test suite granularity levels, `runDSI` calls `validate` (lines 10–21) to check whether each spec that is mined from that test is likely buggy or likely true. For each mined spec 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 the trace obtained from running a test is consistent with the spec, `s`, being validated. If a sanity check fails, DSI terminates and the outcome on `s` is `unknown`—`validate` internally tracks `unknown` specs in a set, S^u, that can be exposed for debugging. The sanity checks run each test thrice (lines 13–15) and compare the outcomes (lines 16, 31–36). The first run (line 16) executes P on test and records whether the program crashes. The second and third runs (line 14 and line 15) execute a version of P that is instrumented to collect a new execution trace for the test; both runs record the test result (`pass`, `fail`, or `crash`) and the trace. Procedure `sanityCheck` (lines 31–36) takes these, and returns `false` if a sanity check failed and `true` otherwise.

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

Experiment Selection. DSI creates *experiments* that mutate P to violate the spec 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 [1]). It takes a trace and a spec, and it produces a set of experiments. Each experiment is a pair $\langle \mathsf{idx}, \mathsf{len} \rangle$, where delaying the invocation of `a()` at idx by len steps in the trace is minimally disruptive. We highlight three important aspects of experiment selection:

(1) Return values. Method `a()` may return a value which intervening code between `a()` and `b()`, inclusive, may rely on. To reduce crashes due to missing return values of the delayed method, DSI fills in the nearest value of a variable of the same type. If no such variable exists, then DSI uses a default value. Replacing return values needs improvement in DSI (Section V-B).

(2) Locks. Experiments may require delaying `a()` and invoking it at a later location, after needed locks were released. DSI records such locks and tries to reacquire them before invoking `a()`. Runs that deadlock are terminated after a timeout.

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

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

Stage 0: After delaying `a()`, before calling `b()`—cannot run experiment and `s` cannot be validated; add `s` to S^u (line 28).

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

Stage 2: After `b()`, but before `a()` is called–`b()` puts the program in a state in which `a()` cannot be run.

Stage 3: After `b()` and `a()` run in that order—violating `s` breaks P.

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

Algorithm 2 shows how we use DSI in TEMARI. The sanityCheck procedure is the same as in Algorithm 1, but there is no validate procedure involved. Different from Algorithm 1, Algorithm 2 outputs whether the spec is buggy or not. So, if the spec is buggy in all DSI experiments, then runDSI will output `buggy`; otherwise it outputs `unknown`. That is, for TEMARI, knowledge of whether a constraint is likely true is not useful for finding bugs in the spec.

REFERENCES

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

[2] 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.