Testing Mined Specifications[*]

Mark Gabel
Department of Computer Science
The University of Texas at Dallas
mark.gabel@utdallas.edu

Zhendong Su
Department of Computer Science
University of California, Davis
su@ucdavis.edu

ABSTRACT

Specifications are necessary for nearly every software engineering task, but they are often missing or incomplete. "Specification mining" is a line of research promising to solve this problem through automated tools that infer specifications directly from existing programs. The standard practice is one of inductive learning: mining tools make observations about software and inductively generalize them into specifications. Inductive reasoning is unsound, however, and existing tools commonly grapple with the problem of inferring "false" specifications, which must be manually checked.

In this work, we introduce a new technique for automatically validating mined specifications that lessens this manual burden. Our technique is not based on heuristics; it rather uses a general, semantic definition of a "true" specification. We perform systematic, targeted program transformations to test a mined specification's *necessity for overall correctness*. If a "violating" program is correct, the specification is false. We have implemented our technique in a prototype tool that validates temporal properties of Java programs, and we demonstrate it to be effective through a large-scale case study on the DaCapo benchmarks.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

specification inference, reverse engineering, software tools

1. INTRODUCTION

Nearly all software engineering tasks require some form of a specification. Implementation, debugging, and testing, for example, all involve reconciling a software program's specified and actual behaviors. Documentation and source code comments are standard sources of specifications, but they are often incomplete, incorrect, or missing entirely. Worse yet, time-saving software tools—our research focus—require formal, machine-readable specifications, which are even rarer.

Research in *specification inference* [3] aims to solve this problem through tools that automatically reverse engineer specifications directly from programs. Although reasoning soundly about specifications from implementations is generally impossible, the intended behavior of a stable software project can be somewhat evident. For example, if calls to the function `Lock.lock()` often precede `Lock.unlock()`, it might be reasonable to suggest that this relationship is *necessary* and *required*. This example illustrates the essence of how specification "mining" tools work: they make *observations* about software and generalize them into *specifications*, albeit with a degree of uncertainty.

That "degree of uncertainty" is a central issue in specification inference. The fundamental cause of imprecision in specification inference is the standard *problem of induction*: generalizing from examples is unsound. A mining tool is easily deceived into reporting frequently-observed *coincidental* relationships as false specifications, *e.g.* "calls to `List.add()` must precede `List.isEmpty()`." The possibility of these "false positives" forces the user of the tool to *validate* each inferred specification manually.

Manual validation of mined specifications is an expensive and error-prone process. Unfortunately, it is also essential: truly sound reasoning about specifications *requires specifications*, a circular dilemma. Researchers have nonetheless tried to reduce this burden with several automated techniques for identifying and filtering likely-false specifications, including leveraging simple usage statistics [38], more advanced statistical models [24], and assorted heuristics [7]. All of these techniques are alike in that they operate on the belief that classes of true and/or false specifications are somehow "alike" and mechanically recognizable.

In our own continuing research, we have found existing specification validation techniques to be insufficient. The current "statistics and heuristics" approach does work adequately when applied to the examples frequently found in the research literature, such as specifications mined from language-level standard libraries. Specifications over Java's Iterator (`Iterator.hasNext()/next()`) and C's file descriptors (`open(fd)/close(fd)`) are recognizable examples. Our research focus, however, has been to scale specification mining "down" and "out" to *project-specific* and highly semantic properties—those perhaps most relevant to the majority of software tasks [27]—and existing techniques have left us at a "precision wall." In short, false positives dominated our early experimental results.

In this paper, we present a novel and effective automated technique for *validating* mined specifications. Our technique is not a

[*]This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, NSF SHF Grant No. 1117603, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

surface-level statistic or heuristic; it is instead based on a purely semantic definition of a "true" specification that is a result of a first-principles reexamination of the problem. We have formulated our approach around this key observation: *a true specification is one that is* necessary *for correctness*.

Our technique takes the following high-level approach. We start with a *program* and a (potentially false) *mined specification*, say, that a call to method `foo()` should always be followed by a call to `bar()`. We then perform a series of automated experiments, each of which involves 1) *transforming the program* to violate that specification (in this case, reordering and/or removing calls to `foo()` and `bar()`) and 2) evaluating *correctness* using testing. If we can generate a transformed program that both a) *violates the mined specification* and b) is *correct*, then we have shown the mined specification to be unnecessary for correctness, and thus false.

We call our technique Deductive Specification Inference, or DSI. The name "Deductive" reflects our goal of moving away from heuristics and statistics, but it is of course an idealization: as mentioned earlier, sound deductive reasoning about specifications from programs is impossible. That limitation is manifest here by the requirement that we precisely evaluate the *correctness* of a program, which, if possible, would *require* complete specifications—obviating the need to *mine* them! We have approximated this ideal with testing, and our implementation of our technique is thus (like any specification miner must be) imperfect. Nonetheless, DSI is effective in practice, as we show in a case study of several real Java programs.

This paper includes the following contributions:

1. A novel specification validation methodology, DSI, that avoids the use of statistics or heuristics.
2. An implementation of our method for a well-studied domain: temporal specifications over the sequences of method calls in a Java program.
3. A case study demonstrating our tool's effectiveness on several open source Java programs.
4. A detailed discussion of the nuances and limitations of our technique, including how these limitations speak to fundamental limitations of specification inference in general.

The next section (Section 2) provides an overview of our approach through a set of examples. Section 3 then presents in detail both the DSI methodology and a tool implementing DSI for the domain of temporal properties of method calls. Our experimental results and related discussions follow in Section 4. In Section 5 we discuss DSI in the context of related work, and in Section 6 we conclude with a discussion of future work.

2. OVERVIEW

In this section, we provide an overview of our general approach through a series of examples. We begin with an introduction to our target domain, temporal specifications, and continue with a presentation of our general technique as well as our implementation.

2.1 Temporal Properties and Their Inference

The examples in this section are drawn from the domain of *temporal specifications over program elements*. Here, "temporal" refers to the span of runtime execution and "program elements" refers to executable code. A temporal specification extends traditional state assertions ("variable `x` is always positive") with the notion of time ("once `x` is positive, `y` will eventually become positive as well").

One commonly studied class of temporal properties involves ordering restrictions on function calls. Functions are building blocks of software projects, and the order in which they are composed is both critical and subtle, especially in imperative and object-oriented systems with side effects. Common examples include locking disciplines, in which a specification might state "calls to methods `lock` and `unlock` on each `Lock` object strictly alternate at runtime" and resource usage rules, in which a partial specification might state "one should call `close` on a file descriptor soon after its final use."

Temporal properties are often much more domain-specific and obscure than these canonical "locking" and "resource" examples, and they are rarely fully documented. Researchers have recognized this problem and developed automated software tools capable of "mining" temporal properties directly from programs. The predominant models are forms of *inductive learning*. Many tools operate similarly in two high-level steps: 1) *observing* (at runtime or statically approximating) the behavior of a program and 2) *generalizing* that behavior into a specification.

2.2 Validating Specifications

Figure 1 lists four examples of "potential" temporal specifications. They were synthesized from observations of real software projects, simplified excerpts of which are listed as well. Mining tools may report specifications like these for several reasons, including:

- The observed property is satisfied (or mostly so) by the observed program. This condition is often trivially true.
- The tool observes the property frequently, with examples occurring frequently at runtime or within the static source code. This encodes the belief that "common behavior is likely to be correct" [12].
- Assorted heuristics. For example, the property listed in Figure 1a involves a method named `execute`, which may match a "function name filter" that identifies naming patterns that have often been important in the past.

Ultimately, a specification mining tool takes an inductive leap, essentially "lifting" observations into specifications based on both observed evidence and prior beliefs.

"Potential" specifications may not be *true*, though, which is a natural consequence of inductive learning. When a programmer is presented with a mined specification, he or she must generally *validate* and/or debug it before it becomes useful. Approaches include:

- Code inspection. If the specification is not followed, would it lead to an obvious error?
- Reconciling with known requirements. Is the specification clearly (in)consistent with existing specifications?
- Consulting with experts and past software engineering data. Have the elements of this specification been involved in any prior issues?

Note the lack of a complete, algorithmic solution. This is precisely what makes specification inference difficult in practice and impossible in the limit. These validation techniques do follow a common theme, though: they involve using disparate sources of information to answer the following question as accurately as possible:

Given a potential specification φ, is φ necessary for my program's correct execution?

Our current work can be framed as a method for solving this problem systematically and automatically.

2.3 Automated Experimental Validation

Returning to the running examples, consider now the contrapositive of the "validation problem":

Does violating φ make my program incorrect?

```
CompilerTest test = ...
test.reset();
/* Set up 'prog' variable */
test.execute(prog, out, err);
```

a. "Call `CompilerTest.reset` at some point before calling `CompilerTest.execute`."

```
ResourceAttributes attr = ...
/* Other setup */
attr.setArchive(true);
attr.setSymbolicLink(false);
```

b. "`ResourceAttributes.setArchive` and `setSymbolicLink` must appear in sequence."

```
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
gen.loadThis();
/* ... other 'gen' invocations */
gen.invokeConstructor(type, constr);
```

c. "Call `GeneratorAdapter.loadThis` at some point before calling `GeneratorAdapter.invokeConstructor`."

```
SaveManager sm = this;
/* Other state restoration actions */
try {
  sm.restoreMarkers(resource, true, p);
  sm.restoreSyncInfo(resource, true, p);
} catch (Exception e) { /* Ignore */ }
```

d. "`SaveManager.restoreMarkers` and `restoreSyncInfo` must appear in sequence."

Figure 1: Four observed temporal properties and a selection of the Java source code that generated them.

```
CompilerTest test = ...
//test.reset();
/* Set up 'prog' variable */
test.execute(prog, out, err);
test.reset();
```

a. "Call `CompilerTest.reset` at some point before calling `CompilerTest.execute`."

```
ResourceAttributes attr = ...
/* Other setup */
//attr.setArchive(true);
attr.setSymbolicLink(false);
attr.setArchive(true);
```

b. "`ResourceAttributes.setArchive` and `setSymbolicLink` must appear in sequence."

```
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
//gen.loadThis();
/* ... other 'gen' invocations */
gen.invokeConstructor(type, constr);
gen.loadThis();
```

c. "Call `GeneratorAdapter.loadThis` at some point before calling `GeneratorAdapter.invokeConstructor`."

```
SaveManager sm = this;
/* Other state restoration actions */
try {
  //sm.restoreMarkers(resource, true, p);
  sm.restoreSyncInfo(resource, true, p);
  sm.restoreMarkers(resource, true, p);
} catch (Exception e) { /* Ignore */ }
```

d. "`SaveManager.restoreMarkers` and `restoreSyncInfo` must appear in sequence."

Figure 2: Transformed programs that should now be "wrong" if each specification is "real" or "necessary."

Phrasing the question this way suggests an experimental solution. Figure 2 reprises the potential properties listed in Figure 1, but the code excerpts have now been transformed. For the domain of temporal properties, we have a strong idea of what it means to "violate" a specification, and in each case the code has been "minimally" and straightforwardly modified to violate each property. If each of the potential specifications is *true*, then each program in Figure 2 should now be *wrong*.

The problem now reduces to judging each "experiment" as "correct" or "wrong." If we were able to judge any as being correct—despite being transformed—we could say with some certainty that the associated specification is unnecessary for correct execution and thus *false*. Similarly, if one of those programs were now incorrect, we would obtain evidence (but not proof) that the associated specification is necessary and *true*. Note the lack of the word "certainty" in the latter case: it is rife with subtlety and will be discussed in more detail throughout this paper.

Judging a program "correct" or "wrong" is generally impossible, of course, and to do so actually begs the question of a complete specification. However, correctness checking can often be *approximated* through testing and analysis, giving us the final component we need to automatically (but approximately) validate specifications. Our high-level technique is as follows:

1. Start with a proposed specification φ from a program P. For temporal function-call specifications, this might be of the form "calls to function `a` always precede calls to function `b`". The normal source of proposed specification will be a *specification mining tool*.
2. Create a suite of *experimental programs* around P and φ, a sort of "design space" populated with programs similar to P but violating φ. We accomplish this through automatic program transformations. Continuing the earlier example, this space may consist of the family of programs in which calls to `a` and `b` are reordered.
3. *Test* these experimental programs. If φ is found to be unnecessary for correctness, then φ is not a specification.

We call this process "Deductive Specification Inference," or DSI. This name reflects DSI's logical and experimental nature, but we emphasize again that truly sound deductive reasoning in this setting is impossible. Nonetheless, on our example properties this automated process is very revealing.

- The experiment in Figure 2a crashes early: `reset` does in fact set up the precondition for `execute` to run; the specification is true.
- Experiment 2b passes: the order in which these two fields are set is irrelevant.
- Experiment 2c fails, but *not* with an immediate crash: it ultimately causes operations much later in the test suite to fail.

`GeneratorAdapter` is a helper class within a Java bytecode library. Not following this temporal specification will actually result in the generation of bytecode that violates the Java Bytecode Specification, which is what ultimately causes the (much) later test failure.

- Experiment 2d passes, but perhaps surprisingly so: each operation contains a substantial amount of overlapping side effects. From a class-level perspective, though, the tests demonstrate that the observed ordering is irrelevant.

In theory, one could transform this specification *validation* procedure into a specification *inference* algorithm, as validation and inference are fundamentally the same problem. For temporal specifications, we could simply enumerate every possible ordering of function calls and systematically validate or invalidate each one. However, for efficiency and to better leverage advances in specification mining, we bootstrap the process with an inductive specification mining tool.

This section has provided an overview of our automated validation technique and how we implement it for the domain of temporal function-call properties. The following section presents our DSI methodology and implementation in full detail, including a discussion of the strengths and subtle limitations of the technique.

3. TESTING SPECIFICATIONS

We have implemented DSI for the domain of *temporal function-call properties* of imperative and object-oriented systems. Our implementation uses automated program transformations to conduct its experiments, and it uses software testing to approximately evaluate correctness. We introduced the temporal function-call problem domain earlier (see Section 2.1) and continue here in more depth.

3.1 Temporal Function-Call Properties

We address a common class of specification: *ordering restrictions on function calls* within a software project. These specifications are common and error-prone, as they are not enforced by the type systems within standard compilers. When they are defined formally, however, advanced software tools can check them statically [5, 15] or at runtime [8], preventing and eliminating errors.

The formalism we use to represent specifications is *regular languages*. While the most general formalism for expressing these properties is some form of a temporal logic, many important properties can be expressed as simple regular languages. The earlier examples of "locking" and "resource disposal" are both regular: $(\texttt{lock}\ \texttt{unlock})^*$ and $(\texttt{read}^*\ \texttt{close})$, respectively. Each specification is quantified over a domain of possible "usage scenarios," which is a general way of capturing the notion that the properties only restrict *related* function calls, *e.g.* `lock` and `unlock` calls on the "same `Lock` object" or `read` and `close` calls on the "same file descriptor."

Our tool is implemented for programs written in the Java programming language. For simplicity we focus on temporal properties of function calls on a *single object*; that is, our domain of "scenarios" is the set of objects at runtime. Note that we use receiver objects solely as a convenient and reliable way of relating sets of method calls through data. No aspect of our implementation is fundamentally restricted to object-oriented systems.

Remark. A related concept is *typestate* [35], the notion that individual types have a high-level "state" that dictates when certain operations (*e.g.* method calls) are legal, *i.e.* when they do not violate an *internal* class invariant. Because we focus on single-object properties, it is tempting to view DSI narrowly as a form of "typestate inference." Our technique certainly *will* validate typestate properties (the two examples above fall into this category) but it is far more general. By using "overall system correctness" as an oracle rather than "obeys a preexisting local class invariant," our tool can (and frequently does) infer interesting domain-specific properties like that shown in Figure 1c: the class does not crash when used incorrectly—no *local* class invariant is violated—but it eventually causes a violation of a higher-level *system* specification.

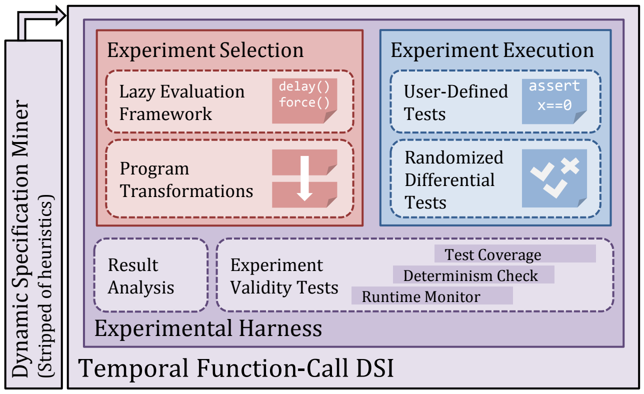

Figure 3: Implementation architecture.

3.2 Temporal Function-Call DSI

Our temporal function-call implementation of DSI takes as input:

1. a Java program, and
2. a mined temporal specification over a set of function calls.

It runs a series of automated experiments, returning as output:

1. "(likely) true specification", or
2. "(likely) false specification" (with supplemental details).

The high-level architecture appears in Figure 3. Execution occurs in two phases: *Preamble* and *Experiment*. The first involves constructing a set of "experiments" that aim to test the *necessity* of the mined specification. We implement this process using static program transformations that use a novel lazy evaluation framework for Java programs. The second phase is dynamic, running and interpreting these experiments with respect to a correctness oracle, which we approximate through testing.

Remark. Testing for "necessity for correctness" is superficially similar to testing for control flow or data flow *dependencies*, a heavily-studied subject. Lack of any control or data dependence is *sufficient* cause to invalidate a specification, but it is far from *necessary*. This issue is analogous to the distinction between *strong* and *weak* mutation testing [22].

Mining a set specifications to validate for a given program is straightforward. We make use of an existing dynamic inductive specification inference tool [17] that has had its heuristics disabled.

3.3 Phase I: Preamble

Our temporal properties describe ordering relationships between sets of function calls. We *test* a mined specification by transforming the input program, in various ways, so that all relevant function calls are reordered in a specification-violating way.

Transforming Java Programs Reordering function calls in real software projects is problematic. Highly local cases are simple: if two function calls appear on subsequent lines, their parameters tend to draw from the same variable scope, simplifying the actual transformation. In our experience, though, the source code of actual projects tends to form a complex and rigid "scaffolding" that is

```
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
gen.loadThis();
/* Other 'gen' invocations */
/* Possibly crossing procedure boundaries */
gen.invokeConstructor(type, constr);
```

a. Original source code.

```
Thunk t; // Global, known location
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
t = delay({gen.loadThis();});
/* Other 'gen' invocations */
/* Possibly crossing procedure boundaries */
gen.invokeConstructor(type, constr);
force(t)
```

b. Transformed source.

Figure 4: The essence of our transformation. Delaying the first invocation until the second has executed violates the mined specification.

difficult to modify—and the most important and subtle properties are likely to be those that are not confined to a pair of sequential lines of code.

We solve this problem by implementing a robust *lazy evaluation framework* for Java programs. Put simply, our framework lets us capture an arbitrary Java function call *and* its parameters in an executable *function object*, save it, and execute it later. It brings to Java the concept of *promises* in eager functional languages like Scheme. The entry point is analogous to the `delay()` primitive in Scheme, but slightly generalized: given an arbitrary sequence of (straight-line) Java bytecode, our framework 1) functionally abstracts it and 2) creates a closure with its (eagerly-bound) parameters, thus converting it into a *thunk*. This object may then be executed at any time, immediately or later, by an analogue of Scheme's `force()` primitive.

Lazy evaluation greatly simplifies the task of reordering function calls. Our transformation occurs at the bytecode level, but it maps conceptually well on to source code. An example of the essence of our transformation appears in Figure 4. The higher level operation in the figure is "delay the first function call until point p". This operation is the basis of all of our transformations, and the remaining questions are *when* and *where* to apply it.

Surveying Behavior Our selection technique relies on proactively-collected information about the runtime behavior of the program. Figure 5 provides an overview of the complete DSI process; the left pane depicts this Preamble phase. Before selecting experiments, we execute the program's test cases (the same tests that will be used during the Experiment phase) twice: once unmodified and once "instrumented," which collects a *property-related trace*. We then use this trace to generate our set of transformations. In addition, these "pilot runs" allow us to perform various sanity checks, including:

- Is the property actually exercised in these test cases? *[if not, then our "broken" programs will certainly be judged "correct," causing a false invalidation.]*
- Is the property satisfied by the program? *[if not, the currently-passing tests imply this is a trivially false specification]*
- Is the program's behavior deterministic enough to allow experimentation? *[if not, our experiment selection algorithm may fail.]*

The third check is not as stringent as it appears: we use a flexible form of execution indexing that tolerates a great amount of variation in program behavior.

Selecting Effective Experiments Consider the mined specification "method `foo` should always be called before method `bar`." There are many ways to violate this specification:

- Remove all calls of `foo` and `bar` from the program entirely.
- Randomly delete calls of `foo`/`bar` throughout the program.
- Reorder the calls by "delaying" all runtime invocations of `foo` until the program exits.
- Reorder the calls by "delaying" each invocation of `foo` by the *minimum* amount of time necessary to cause a violation.

These actions differ in the amount of change, or "disturbance," they cause on the target system, and the fourth, least intrusive option appears most sensible. There is an analogy here to traditional scientific experimentation and the issue of *control*. Here, we wish to answer the question, "is the stated *relationship* between `foo` and `bar` necessary for correctness?" A well-designed experimental answer should vary precisely *that relationship* and leave unchanged all other aspects of the program's execution. In realistic software projects, full, behavior-isolated "control" will be generally impossible: specifications overlap and program components communicate. Rather, as the examples demonstrate, it tends to be a matter of degree.

Selection Algorithm The preceding line of reasoning led us to an *experiment selection algorithm* that strives for completeness and control. We respect completeness by generating test programs that violate *each potential binding of the property at runtime*. We maintain control by doing so in as *minimally intrusive manner* as possible. That is, each time the elements of a mined specification are used at runtime, we delay the minimum *number* of function calls by the minimum *amount* of (execution) time necessary to violate the property.

Our experiment selection algorithm is listed as Algorithm 1. It takes as input 1) a mined specification and 2) the previously-described *trace* of the program's specification-related method calls. It returns an "instruction" for a minimal *experiment*, $(\mathsf{idx}, \mathsf{len})$, which can be interpreted as, "The minimally-intrusive way to violate the given specification is to `delay()` the idx*th* method call by len calls." This algorithm ensures minimality by taking a brute-force approach: it simply evaluates all possible transformations. In the worst case, this algorithm runs in time quadratic in the length of the trace; in practice, it is much closer to linear: the set of "trials" (line 2) is eagerly pruned (lines 10–18) and remains consistently small. Note that this algorithm is formulated in terms of a single "usage scenario," *e.g.* the calls surrounding a single file descriptor, and it returns a *single experiment*. In practice, we process multiple scenarios simultaneously and generate a *suite of experiments*.

Remark. The simple primitive "delay a call until time p" is powerful: for example, *multiple* functions can be simultaneously delayed and re-emitted in any order. In theory, this "delay primitive" is *not* powerful enough to "break" (*i.e.* transform to rejecting) every accepting string of any arbitrary (non-total) regular language. Fortunately, in practice, every valid trace of every regular *specification pattern* we have encountered in our work can be "broken" by delaying just a *single* event, albeit by varying amounts of time. We have performed a more thorough theoretical investigation of this problem, which we have omitted for brevity.

Other Implementation Notes One complication to our otherwise simple process is caused by the presence of explicit *return values* from the functions we "delay:" if a function is not evaluated, we cannot know what it will return. We solve this problem by implementing a model of what a *sensible programmer* would do in this situation, inspired by our principle of "minimally intrusive" experiments. We

Algorithm 1 Algorithm for selecting an *experiment* for a given usage scenario. Returns a minimally-disruptive perturbation that causes a violation of a mined specification.

```
Input: $P : (\Sigma = \{\mathsf{m1}, \mathsf{m2}, \ldots\}, S, s_0, \delta, F)$
          Mined regular specification over methods $\{\mathsf{m1}, \mathsf{m2}, \ldots\}$
       $T : \langle \mathsf{call} : (\mathsf{idx}, \mathsf{method}), \ldots \rangle$
          An indexed runtime trace of method calls for one ‘usage
          scenario,’ where $\mathsf{method} \in \{\mathsf{m1}, \mathsf{m2}, \ldots\}$
Output: $(\mathsf{idx}, \mathsf{len})$
          A minimal perturbation: delaying the call *at* trace index
          ‘idx’ *by* ‘len’ number of calls 1) violates $P$ and 2) minimizes
          ‘len’ over all such violating transformations.

 1: $s_{trace} \leftarrow s_0$                ▷ The state of the trace w.r.t. specification $P$
 2: $E \leftarrow \{\}$    ▷ Set of trial experiments: $\{(\mathsf{idx}, \mathsf{len}, s)\}$ If the call at ‘idx’
                       were to be ‘delay’ed by ‘len’ calls, $P$ would be in state $s$.
 3: $min \leftarrow$ NULL                       ▷ Minimum-amount perturbation
 4:
 5: for all $\mathsf{call} : (\mathsf{idx}, \mathsf{method})$ in $T$ do
 6:     for all $\mathsf{trial} : (\mathsf{idx}, \mathsf{len}, s)$ in $E$ do    ▷ trial.$s$ is the state of $P$
                                           had the call at trial.idx been delayed
 7:         $\mathsf{trial}.s \leftarrow \textsc{NextState}(P, \mathsf{trial}.s, \mathsf{call.method})$
 8:         $\mathsf{trial.len} \leftarrow \mathsf{trial.len} + 1$
 9:         $s_{forced} \leftarrow \textsc{NextState}(P, \mathsf{trial}.s, T[\mathsf{trial}.idx])$
                                     ▷ $s_{forced}$ is the state of $P$ if ‘forced’ now
10:         if $\textsc{IsRejecting}(s_{forced})$ and $\textsc{IsSink}(s_{forced})$ then
11:             if $min =$ NULL or $\mathsf{trial.len} < min.\mathsf{len}$ then
12:                 $min \leftarrow (\mathsf{trial.idx}, \mathsf{trial.len})$
13:             end if
14:             $E \leftarrow E \setminus \mathsf{trial}$
15:         else if $min \neq$ NULL and $\mathsf{trial.len} \geq min.\mathsf{len}$ then
16:             $E \leftarrow E \setminus \mathsf{trial}$
17:         end if  ▷ Perf. optimization: track minimum and prune early
18:     end for
19:     $E \leftarrow E \cup (\mathsf{call.idx}, 0, s_{trace})$              ▷ Create a new trial
20:     $s_{trace} \leftarrow \textsc{NextState}(P, s_{trace}, \mathsf{call.method})$
21: end for
22: for all $\mathsf{trial} : (\mathsf{idx}, \mathsf{len}, s)$ in $E$ do       ▷ Tabulate remaining trials
23:     if $min =$ NULL or $\mathsf{trial.len} < min.\mathsf{len}$ then
24:         $min \leftarrow (\mathsf{trial.idx}, \mathsf{trial.len})$
25:     end if
26: end for
27: return $min$
```

implement a simple type analysis that allows us to replace the return value of a delayed call with the value of the "nearest-defined local variable of the appropriate type" (or the language-defined default value if one is not found). This general definition automatically captures many intuitive actions, including reusing the return value from a previous call (among others).

Multithreaded programs caused complications as well. Avoiding any single-threaded assumptions handled most issues, but our early experiments revealed several fundamental challenges. For one, we may "move" a function call to a program point at which an important lock is no longer held. To solve this issue, our lazy analysis framework makes note of the locks held during a runtime invocation of `delay()` and optionally attempts to reacquire them, if necessary, when the thunk is `force()`ed. In another case, a delayed call was indirectly responsible for some event that, if omitted, would cause a second call to block indefinitely, creating a deadlock. In this case, we instituted a global "inactive timeout" on our experiments: if the subject program makes no forward progress after a period of time, we forcibly terminate the program.

The execution of these initial runs, tests, and experiment selection algorithms form the entirety of the Preamble phase. At its conclusion, we have produced a set of transformed experimental programs that are ready to be evaluated.

3.4 Phase II: Experiment

The Experiment phase is conceptually simple: we run each experiment in the suite of transformed programs and interpret the results.

Testing as an Oracle Our approximation of a correctness oracle is *testing*. This portion of our tool is pluggable to allow the use of user-defined tests and test oracles. In addition, we also provide a default implementation based on randomized regression testing. We first run the unmodified, assumed-correct program on random inputs. We record the input/output behavior on these inputs as a *behavioral profile*, which then becomes our test oracle. This process is similar to Differential Testing [14], which uses automatically generated tests to test modified software for regressions.

Analysis of Results At a high-level, the only important output is the success or failure of each individual test: failures suggest the mined specification is *valid*; successes suggest it is *false*. However, exactly *how* a test executes and fails can be useful knowledge. Regardless of the property—the precise relationship it defines or the number of functions it references—each experiment reduces to delaying a single function call until a second call completes (albeit with other calls possibly executing in between). This simplicity allows us to analyze a finite set of cases that may arise during execution; these cases are depicted in the right pane of Figure 5.

- *Normal:* This is the standard, unmodified case for reference. Function f_1 executes, followed by f_2, which is followed by normal execution.
- *Stage 0:* f_1 has been delayed, but execution failed before reaching f_2. The experiment failed at a very fundamental level: we could not violate this property using our standard program transformations. Interpretation of this case could fall in either direction: the tests did fail, but the experiment was not actually "conducted."
- *Stage 1:* Execution fails while executing f_2. This is indicative of a real specification, but the circumstances also suggest additional information: f_1, the delayed call, appears to directly or indirectly establish f_2's *precondition*.
- *Stage 2:* Execution fails while forcing the execution of f_1. Once again, this is evidence of a real specification, but it also reveals that f_2 puts the program in a state in which f_1 cannot safely execute. An example might include f_1 involving the "use" of a resource and f_2 "closing" it.
- *Stage 3:* The experiment fully completes and execution continues as in the normal case. If the tests pass, then we have strong evidence that the specification is *false*: obeying it appears to be *unnecessary for correctness*. If the tests fail, then we have *potentially* revealed what may be a particularly *subtle and important* true specification—one that, if violated, *silently* puts the system in an undefined error state.

3.5 Validity and Limitations of Results

In the ideal, we have a fully representative universe of perfectly controlled experiments executed under fully exhaustive tests, and DSI always classifies valid and false specifications correctly. This ideal is impossible both in theory and in practice, imposing several limitations on the technique—limitations that also apply to specification inference in general.

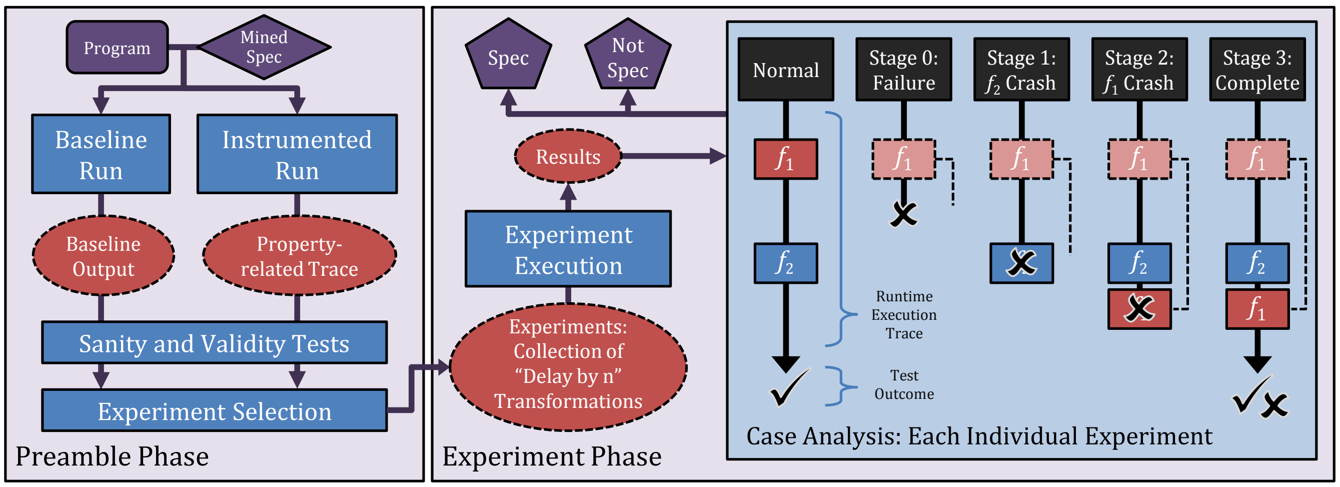

Figure 5: The complete DSI process for temporal function-call properties.

False Validations DSI can misjudge false specifications as true. Two main issues surround DSI's false validations: *false causal inference* and *overgeneralization*. Both are ultimately a result of problems with DSI's experiments.

False causal inference arises when DSI attempts to violate a specification, but that violation causes unintended, non-specification-related side effects. As we discussed earlier, this issue relates to the matter of experimental *control*, and perfectly controlled automated experimentation within complex, interconnected software systems is impossible in the limit. We have, however, sought to mitigate this problem with our notion of minimally-intrusive transformations, and, to a lesser extent, our choice of domain. Temporal specifications are a natural fit for DSI: they specify the *ordering* of method calls, a separable concern that can be modified fairly independently of the methods themselves.

Overspecification occurs due to a lack of sufficient examples of a specification's usage in a code base. Consider a hypothetical specification: "`List.size()` must be called before calling `List.get(int)`." It is arguably false: although it reflects a sensible bounds checking practice, there are plenty of use cases where it need not be followed. If those alternate use cases are not reflected in the program, though, DSI will only generate *failing* experiments and judge it valid. This class of false inference is naturally more prevalent when DSI evaluates more "intended-general" specifications, such as those over standard libraries. There is an interesting nuance here, however: many apparently-overspecified false specifications may in fact be valid. If every `List` in one specific project does always have the possibility of being empty, then DSI has classified a *true, project-specific refinement* of a more general specification.

False Invalidations DSI can also misclassify true specifications as false. The main issue here is conceptually simpler than the previous cases: software testing is naturally incomplete, so it is possible for the "spec-violating" experimental program to be misjudged as correct. We avoid the more egregious cases of this problem in practice with the Preamble phase's sanity checks that ensure the test cases exercise the property-related behavior. In other words, we run all experiments along the paths of known test cases, giving us a form of "partial completeness." Note, though, that these checks do *not* guarantee that the test suite (especially its oracle) is perfect. In any case, DSI provides a very concrete counterexample that is likely to be much easier to inspect than a specification on its own: a *test case* in the form of a similar *alternate program* that both 1) violates the mined specification and 2) is apparently correct.

The Inability to Violate a Specification Occasionally, DSI is unable to violate a given mined specification. This anomaly is surprisingly useful in practice. We believe our tool's ability to "break" specifications is comparable in power to a human programmer: it lacks human ingenuity, but it does have access to the unique and powerful lazy analysis primitives. If DSI cannot violate a proposed specification in any reasonable way, that specification is likely to be unimportant *even if technically true*.

An example will clarify this point. Our implementation works by reordering function calls. Consider the following code snippet:

```
public String getResult() {
   return this.calc.compute();
}
```

A simple inductive inference tool may observe that `getResult` and `compute` always execute in sequence at runtime and may present the relationship as a specification. Note, though, that the structure of the program prevents any sensible violation. A programmer would quickly dismiss this specification as false for the very same reason that DSI would fail to (even begin to) prove it true: neither DSI nor a programmer could possibly violate it.

This specific problem of "false specifications caused by one function calling another" has been addressed in the literature through purpose-built heuristics, such as the "control-flow artifact filter" in Yang *et al.*'s Perracotta tool [38]. DSI's natural "failure to violate" elegantly generalizes and handles this and many other cases of "spurious specifications" *without the need for heuristics*.

Our implemented tool is robust, scalable, and general. In the following section, we present the results of a case study of our tool on real, widely-used Java projects.

4. CASE STUDY

This section presents the results of a case study of our DSI tool for Java programs. We have sought to answer the following research questions:

1. Is our tool robust? Does it function on complex, real-world software?
2. What are the characteristics of DSI-validated specifications of set of real-world Java programs?
3. What are the practical strengths and limitations of our tool, and how do they translate to the DSI methodology as a whole?

We continue with a discussion of our experimental setup, which is followed by a presentation and an analysis of our results.

4.1 Experimental Setup

Our test subjects are a set of Java programs drawn from the DaCapo benchmark suite (version 9.12-Bach).[1] DaCapo differs from traditional "microbenchmarks" in that it is formed from over a million lines of code of real, widely-used applications, creating an uncommonly realistic workload.

DSI operates on mined specifications. As we noted earlier, we modified an inductive specification learning tool [17] to function without heuristics. We configured it to find all instances of *simple sequences*: pairs of method calls that appear to be obeying a sequential ordering restriction between them. For brevity, we omit the implementation and experimental details of this process.

Remark. The "sequence" template is simple, almost to the point of being *simplistic*. In practice, it captures a surprisingly broad amount of specification behavior. Our previous work on the Javert tool [18] has demonstrated that almost all temporal function-call specifications can be decomposed into fundamental pieces, simple sequences and small loops. In addition, note that our DSI tool is not inherently limited to simple patterns, both in theory and in the current implementation: it can work with any regular specification pattern over any number of distinct functions.

We ran all experiments in parallel on several 64-bit Linux servers, each configured with with Intel processors (Xeon and Core 2) and the 64-bit Oracle Java Virtual Machine, Server Edition, version 1.6.0_25.

4.2 Results

A summary of our results appears in Figure 6. Our tool individually analyzed 7,848 mined specifications, systematically judging each as a *likely specification* or *likely non-specification*. In each case our tool performed robustly, both validating and invalidating specifications within large, complex software projects.

Performance Performance was acceptable: the majority of specifications were analyzed in under two minutes. Our task is embarrassingly parallel as well, a fact we utilized fully in our study by using several compute servers. Notable performance exceptions were the Jython and Eclipse benchmarks. Jython and Eclipse contain many mined specifications whose violation hinders termination, forcing our system to often wait until a conservative timeout had expired (five minutes) before proceeding. In practice, this timeout can be reduced to a value more appropriate to a particular project.

Likely Non-Specifications Figure 6 lists detailed results of our tool's experiments on the 7,848 input specifications. The first group of four columns describes the mined specifications our tool judged as *likely non-specifications*.

The first column ("Could Not Violate") counts specifications that could not violate *in any way*, and as discussed earlier, were judged to be "unimportant." Many of these cases were a result of the "control-flow artifact" specifications described earlier (one function directly or indirectly calling another), but there were several other cases as well that were naturally captured by our tool's "cannot-violate implies non-importance" principle. For example, one case involved the static type system preventing us from moving an object's constructor call after its first true method call, the more abstract principle here being "ordering restrictions involving constructors cannot be violated by programmers." This eliminated the need to write a "constructor ordering relationships are false" heuristic.

The second two columns correspond to "Stages" of execution described earlier and depicted in Figure 5. In the second column ("Stage 0"), every experiment resulted in a program crash soon after we "delayed" the first function call. Stage 0 results generally fall into two categories:

1. The fact appears to be fully "enforced" at runtime, either through assertions or other checks, and is thus impossible to violate on tested code.

2. There is a more specific and relevant specification we should be analyzing instead. For example, if functions `a`, `b`, and `c` all execute in sequence at runtime and a crucial relationship exists between `a` and `b`, we will be unable to run a successful experiment involving `a` and `c`. Our heuristic-less specification miner forces us to test *every* possible mined specification, so we would *eventually* properly classify the crucial `a`/`b` specification. The general practical implication is that DSI can more readily experiment on smaller, more manageable pieces (the *transitive reductions*) of larger specifications.

The third column lists those mined specifications whose experiments all fully completed, violating the property, but continued to be judged correct by the relevant tests. The experiments can be inspected by a programmer and serve as a form of "certificate" demonstrating that the *lack of the specification's necessity*—our primary goal. Note the sheer volume of this category: 55% of the mined specifications (over 4,000) were invalidated, every one of which had the potential to be called a "true specification" by an inductive learning tool and to waste a programmer's time. We note also that the standard *frequency heuristic* does *not* appear to properly classify these "shown unnecessary" specifications. In our results, significant portions of both rarely- and frequently-executed specifications were invalidated.

In response to Research Question 1, we are encouraged by the fact that this level of interference and experimentation was possible in large and complex software projects: our initial hypothesis was that almost *any* runtime perturbation of function ordering would result in a fairly immediate crash.

As noted in the previous section, it is possible for DSI to mistakenly invalidate a specification. If a transformation successfully violates a specification, it is possible for it to "infect" the state of a system in such a way that existing tests do not detect. We selected a limited random sample of 25 of these invalidated specifications and manually analyzed their usage in each project's source code. *None* of them appeared to be improperly classified: the ordering exhibited in the source code is apparently coincidental.

Likely Specifications To answer RQ2, we analyze the set of likely specifications in more depth. The second group of columns in Figure 6 contains counts of the mined specifications our tool judged to be *likely true*. Each column corresponds to a "stage" of execution described earlier and depicted in Figure 5. The first two columns of this group correspond to fairly standard and local "typestate" specifications:

1. Specifications whose experiments all end in Stage 1 exhibit a *precondition* relationship: the first function call establishes (a part of) the precondition of the second, and delaying it causes the appropriate crash.

2. Specifications whose experiments all end in Stage 2 exhibit a *state transition* relationship: the first function call was legal in its original context, but it becomes illegal after the second

[1] A small minority of the standard benchmarks were omitted *solely* due to technical problems with their execution under instrumentation. We have no reason to believe our results will significantly differ once they are included.

		Likely Non-Specifications				Likely Specifications			
Benchmark	**Mean Time/ Spec. (s)**	Could Not Violate	Stage 0: Failure	Stage 3: Complete	Total	Stage 1: f_2 Crash	Stage 2: f_1 Crash	Stage 3: Complete	Total
avrora	47.1	16	91	281	388	18	2	52	72
batik	69.4	190	543	1171	1904	27	5	127	159
eclipse	225.9	357	297	627	1281	25	45	75	145
h2	102.8	19	96	325	440	18	2	37	57
jython	936.4	135	129	200	464	10	8	11	29
luindex	35.1	8	125	256	389	10	32	32	74
lusearch	91.4	16	55	73	144	3	6	14	23
pmd	79.1	116	152	593	861	12	1	37	50
sunflow	34.1	4	79	64	147	13	0	36	49
xalan	76.0	102	278	713	1093	18	6	55	79
Total		963	1845	4303	7111	154	107	476	737

Figure 6: Detailed results of our case study. Each entry is a count of the number of mined specifications our tool judged to fall into the given column's category. Times are given in terms of single-CPU time per property, which is independent of our parallel execution of the larger collection of experiments.

function executes (*e.g.* attempting to use a resource after it has been closed).

Most interesting are the specifications whose experiments all fully completed (Stage 3). In these cases, each experiment *silently* corrupted the program state and caused the tests to fail at a later—sometimes much later—time. A small, randomly sampled collection of these specifications follows.

- In Lucene, delaying a "commit" operation on an index until after it is closed corrupts the index but causes no overt failure.
- In H2, a connection information object will silently return a bogus password hash from a connection information if read before initialization is complete.
- Various XML parsers in the projects require handlers to be set before parsing begins and will not warn the user if none has been set (our sample included two of these).
- In Sunflow, the `SunflowAPI` object's initialization is idiomatic. Moving any part of it until after a call to `render` causes incorrect output but no obvious crash. Two other examples in our sample followed a similar "idiomatic initialization" pattern.

We are continuing to analyze these results in depth, but thus far this case study has suggested that our DSI tool is general, robust, and effective.

4.3 Discussion

To further answer Research Questions 2 and 3, we analyze our results in further detail.

Confidence in Results DSI is systematic and appears to yield strong, algorithmically-decided results. However, as we discussed in Section 3.5, our methodology is subject to a number of potential threats that may lead to incorrect classification. We have not yet manually validated all $8,000$ classified specifications, but we are nonetheless confident in the quality of the majority of our results.

One notable exception is the class of specifications (weakly) classified as false during "Stage 0: Failure" (Column 2 of "Likely Non-Specifications in Figure 6). Recall that these are specifications for which we `delay` the first call, but the program crashes soon after, suggesting that they are difficult to meaningfully violate. In the previous section, we did classify these "failed experiments" into two fairly benign general categories, but we believe that with continued exploration we may encounter cases for which a *human programmer* could have conducted a successful experiment. If this occurs, however, we intend to use those examples to improve DSI's transformations.

On the other hand, we are generally confident in DSI's systematic *invalidations*: to date, we have observed no false classifications caused by insufficiently powerful tests. We are also optimistic about our continued evaluation of the "true" specifications. Our continuing manual validation of these specifications has turned up no false positives thus far.

Performance and Usefulness Obscured by the "average time per specification" performance statistic is the fact that this case study consumed two full weeks of CPU time. Our prototype is certainly computationally expensive: for each mined specification, it generates a family of transformed programs and runs an entire test suite several times. We note that DSI is currently implemented straightforwardly and that we expect many significant optimizations to be possible, including, for example, performing multiple transformations at once or taking advantage of a capture/replay framework. We also emphasize that our results form a rich dataset on which simpler *approximations* may be trained, similar to Le Goues and Weimer's [24] lightweight statistical model for classifying mined specifications.

5. RELATED WORK

This section discusses our automated specification validation technique in terms of several lines of related work.

Inductive Specification Inference Inductive specification inference techniques have been developed for a variety of domains. Kremenek *et al.* have presented a framework [23] based on probabilistic inference that reflects the essence of the inductive process: leveraging beliefs about software to infer general specifications from specific examples of programs. Several targeted techniques infer temporal specifications, the subject of our own implementation of DSI. Both dynamic [3, 9, 18, 25, 38] and static [34] techniques follow the same general approach: they observe temporal relationships in programs and inductively elevate them to specifications. Other successful application domains of inductive specification inference are assertions over program state [13, 20], determinism specifications for concurrent programs [6], and function contracts [30]. Less formal approaches include lightweight "programming rules" [7, 26], which have been particularly effective at revealing programming mistakes.

These techniques all confront one central issue: precision. Generalization is unsound, and the inductive leap from one program to a specification about an entire class of programs is essentially an educated guess. This issue is not merely theoretical: a recent study [29] has shown that one third of inductively-learned code contracts are incorrect or irrelevant (and in our experience, the temporal property domain can be worse). As a result, specification inference research tends to contain an empirical component evaluating precision, and specific techniques have also been proposed to help programmers debug [4] and filter [24] erroneous mined specifications.

DSI provides a new way of approaching the specification validation problem. Rather than *speculating* about specifications, our technique allows one to test them *experimentally* and *automatically*. As a standalone automated validation procedure, it treats classification as a separate problem and could thus be used to improve any of these existing inductive techniques.

Validating Mined Specifications with Testing In recent work developed concurrently with our own, Nguyen and Khoo [28] use mutation testing to validate one class of temporal specification: precondition relationships. If a function call `foo()` appears to be a precondition for calling `bar()`, their technique generates a test that *deletes* the call to `foo()` and simply replaces its return value with a *randomly-generated value*. Their technique then classifies the specification as "significant" if and only if `bar()`, specifically, then crashes with an exception.

DSI is a much more general technique that targets the entire class of regular temporal properties, of which preconditions are one special case. It also implements function reordering, a much more complex and nuanced form of experimentation than their simple deletion of function calls (which can be done in our tool as well). Nguyen and Khoo's work also focuses only on "local" specifications: a tested API must crash immediately with an API-related exception to be judged significant. DSI tests significance with a more general notion of system-level correctness, which fully encompasses both these "local preconditions" and a much larger class of subtle, system-level specifications. In addition, Nguyen and Khoo do not discuss the important threats and limitations we describe in Section 3.5.

Fraser and Zeller use an idea similar to DSI in recent work on generalizing unit tests [16]. We aim to invalidate *mined specifications*; they aim to invalidate *inferred assertions for unit tests*, and both techniques use mutation as the underlying technique. Apart from the technical differences arising from the differing problem domains, DSI's general execution somewhat differs as well. Put plainly, we invalidate specifications when a known-violating program does not appear to be broken. They invalidate assertions when a known-broken program does not appear to violate the assertion.

Non-Inductive Specification Inference Some specification inference techniques are non-inductive. Work on extracting *component interfaces* [2, 21, 36] is superficially similar to inductive specification inference techniques, but the processes are more well-defined and involve no inductive generalization. These techniques require low-level specifications as input (or they are extracted from explicit assertions in the code). They then solve the formal problem of extracting a sound, higher-level "model" of component usage that avoids violating any of the given low-level specifications. In essence, DSI takes this process and lifts it to entire programs. In place of a model of a specific component, we define a set of specification-violating program transformations, and in place of low-level specifications, we use system tests.

Combining Testing with Specification Inference Our implementation of DSI leverages testing to enhance specification inference, the general idea of which was originally proposed by Xie and Notkin [37]. Dallmeier *et al.*'s Tautoko tool [10] also uses testing in a similar way. Tautoko's problem setup is similar to that of the "component interface" tools described earlier: generate a component "model" that avoids errors. In Tautoko's case, the component is a Java class that is assumed to crash or otherwise raise an error if used incorrectly. Tautoko starts with an inductively inferred model, but it then enhances it in a feedback loop by generating targeted, exploratory tests. There is a parallel here to DSI: we start with an inductively learned specification and essentially "generate tests" to validate it.

Testing "Necessity" with Experiments Our implementation of DSI uses experimentation to infer properties of programs. Ruthruff *et al.* [33] describe a general conceptual framework, Experimental Program Analysis, for conducting experiments within programs; our methodology can be seen as an instance of this framework. An idea similar to "testing for necessity" has been used by Renieris *et al.* in their study of *elided conditionals* [31]. In their work, the authors generate experiments that test the whether or not the outcome of a conditional statement (*i.e.* a branch) affects the outcome of a test, much as we generate experiments that test the necessity of proposed specifications.

Automatic parallelization tools make use of *Commutativity Analysis* [1, 32], which evaluates the necessity of a given ordering of program statements. This is similar to how we test the "necessity" of a temporal ordering constraint. Commutativity analysis uses a far stricter criterion than DSI does in practice: that various orderings produce semantically *identical* results. In our implementation of DSI, the fact that two functions commute is only one of several reasons a specification might be invalidated, and the fact that two functions do *not* commute does not imply a given ordering is necessary to correctness.

Our work is also similar to Mutation Testing [11, 19]. In mutation testing, modified programs ("mutants") are used to experimentally test the fault-finding ability of *test suites*. In our work, we use transformed programs that are similar to standard mutants to experimentally test the validity of potential *specifications*.

6. FUTURE WORK AND CONCLUSION

This paper has presented DSI, a new methodology for automatically validating specifications mined from programs. DSI's novelty lies in its ability to systematically test a specification for *necessity for correctness*. We have implemented DSI for the domain of temporal function-call properties of Java programs. Our implementation creates *experiments* through fully automated program transformations, and it evaluates them with traditional software testing. In a case study, we demonstrated that our tool is effective on real-world programs.

Our most immediate future work involves implementing DSI for other domains to further demonstrate the strengths of the method, and our early results in this area have been promising. We are also interested in exploring other aspects of the definition of importance of a specification. We presently define "important" conservatively: if a programmer can feasibly violate a specification it is important. Collecting more data on what it means for a specification to be "important" might allow us to synthesize other effective and *testable* definitions like this "ability to violate" concept introduced here. Finally, we are exploring the possibility of using our DSI-validated directly with other software tools—without intervention by a human programmer.

References

[1] F. Aleen and N. Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See The Big Picture. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '09, 2009.

[2] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[4] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging Temporal Specifications with Concept Analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, 2003.

[5] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, 2001.

[6] J. Burnim and K. Sen. DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, 2010.

[7] R.-Y. Chang, A. Podgurski, and J. Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In *Proceedings of ISSTA '07*, 2007.

[8] F. Chen and G. Rosu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of OOPSLA '07*, 2007.

[9] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining Object Behavior with ADABU. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, 2006.

[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating Test Cases for Specification Mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, 2010.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4), 1978.

[12] 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 '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proceedings of ICSE*, 2000.

[14] R. B. Evans and A. Savoia. Differential Testing: A New Approach to Change Detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007.

[15] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[16] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, 2011.

[17] M. Gabel and Z. Su. Symbolic Mining of Temporal Specifications. In *Proceedings of ICSE '08*, 2008.

[18] M. Gabel and Z. Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of SIGSOFT '08/FSE-16*, 2008.

[19] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Trans. Softw. Eng.*, 3(4), 1977.

[20] J. Henkel and A. Diwan. Discovering Algebraic Specifications from Java Classes. In *ECOOP 2003 - 17th European Conference on Object-Oriented Programming*, 2003.

[21] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive Interfaces. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[22] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng.*, 8(4), July 1982.

[23] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From Uncertainty to Belief: Inferring the Specification Within. In *OSDI'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[24] C. Le Goues and W. Weimer. Specification Mining with Few False Positives. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[25] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

[26] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of ESEC/FSE-13*, 2005.

[27] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, 2006.

[28] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *Proceedings of the 13rd International Conference on Formal Engineering Methods*, ICFEM '11, 2011.

[29] N. Polikarpova, I. Ciupa, and B. Meyer. A Comparative Study of Programmer-Written and Automatically Inferred Contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, 2009.

[30] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static Specification Inference Using Predicate Mining. In *Proceedings of PLDI*, 2007.

[31] M. Renieris, S. Chan-Tin, and S. P. Reiss. Elided Conditionals. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[32] M. C. Rinard and P. C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6), 1997.

[33] J. R. Ruthruff, S. Elbaum, and G. Rothermel. Experimental program analysis: a new program analysis paradigm. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, 2006.

[34] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static Specification Mining Using Automata-based Abstractions. In *Proceedings of ISSTA*, 2007.

[35] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

[36] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-oriented Component Interfaces. In *Proceedings of ISSTA '02*, 2002.

[37] T. Xie and D. Notkin. Mutually Enhancing Test Generation and Specification Inference. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing*. 2004.

[38] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of ICSE*, 2006.