IEEE TRANSACTIONS ON SOFTWARE ENGINEERING VOL.38 NO.1 YEAR 2012

Measuring Code Quality to Improve Specification Mining

Claire Le Goues, Westley Weimer

Abstract—Formal specifications can help with program testing, optimization, refactoring, documentation, and, most importantly, debugging and repair. However, they are difficult to write manually, and automatic mining techniques suffer from 90–99% false positive rates. To address this problem, we propose to augment a temporal-property miner by incorporating code quality metrics. We measure code quality by extracting additional information from the software engineering process, and using information from code that is more likely to be correct as well as code that is less likely to be correct. When used as a preprocessing step for an existing specification miner, our technique identifies which input is most indicative of correct program behavior, which allows off-the-shelf techniques to learn the same number of specifications using only 45% of their original input. As a novel inference technique, our approach has few false positives in practice (63% when balancing precision and recall, 3% when focused on precision), while still finding useful specifications (e.g., those that find many bugs) on over 1.5 million lines of code.

Index Terms—specification mining, machine learning, software engineering, code metrics, program understanding

1 Introduction

Incorrect and buggy behavior in deployed software costs up to $70 billion each year in the US [46], [53]. Thus debugging, testing, maintaining, optimizing, refactoring, and documenting software, while time-consuming, remain critically important. Such maintenance is reported to consume up to 90% of the total cost of software projects [49]. A key maintenance concern is incomplete documentation [15]: up to 60% of maintenance time is spent studying existing software (e.g., [47]).Human processes and especially tool support for finding and fixing errors in deployed software often require formal *specifications* of correct program behavior (e.g., [43]); it is difficult to repair a coding error without a clear notion of what "correct" program behavior entails. Unfortunately, while low-level program annotations are becoming more and more prevalent [14], comprehensive formal specifications remain rare.

Many large, preexisting software projects are not yet formally specified [14]. Formal program specifications are difficult for humans to construct [11], and incorrect specifications are difficult for humans to debug and modify [4]. Accordingly, researchers have developed techniques to automatically infer specifications from program source code or execution traces [2], [3], [20], [25], [51], [60], [61]. These techniques typically produce specifications in the form of finite state machines that describe legal sequences of program behaviors.

Unfortunately, these existing mining techniques are insufficiently precise in practice. Some miners produce large but approximate specifications that must be corrected manually (e.g. [4]). As these large specifications are imprecise and difficult to debug, this article focuses on a second class of techniques that produce a larger set of smaller and more precise candidate specifications that may be easier to evaluate for correctness. These specifications typically take the form of two-state finite state machines that describe temporal properties, e.g. "if event a happens during program execution, event b must eventually happen during that execution." Two-state specifications are limited in their expressive power; comprehensive API specifications cannot always be expressed as a collection of smaller machines [25].

Despite this limitation, two-state machines are useful in both industrial and research practice [14], [31], and previous research efforts have developed techniques for mining them automatically [20], [59]. Such techniques typically produce a large set of *candidate* specifications, often in a ranked list (e.g. [20]). A programmer must still evaluate this ranked list of candidate specifications to separate the *true specifications* from the *false positive* specifications. In this context, a false positive is a candidate specification that does not describe required behavior: a program trace may violate such a "specification" and still be considered correct. A true specification describes behavior that may not be violated on any program trace, or the program contains an error. Unfortunately, techniques that produce this type of ranked list of smaller candidates suffer from prohibitively high false positives rates (90–99%) [59], limiting their practical utility.

This article develops an automatic specification miner that balances true positives – as required behaviors – with false positives – non-required behaviors. We claim that one important reason that previous miners have high false positive rates is that they falsely assume that all code is equally likely to be correct. For example, consider an execution trace through a recently mod-

• *Le Goues and Weimer are with the Department of Computer Science at The Universiy of Virginia, Charlottesville, VA 22904. E-mail: {legoues, weimer}@cs.virginia.edu*

ified, rarely-executed piece of code that was copied-and-pasted by an inexperienced developer. We believe that such a trace is a poor guide to correct behavior, especially when compared with a well-tested, stable, and commonly-executed piece of code. Patterns of specification adherence may also be useful to a miner: a candidate that is violated in the high quality code but adhered to in the low quality code is less likely to represent required behavior than one that is adhered to on the high quality code but violated in the low quality code. We assert that a combination of lightweight, automatically-collected quality metrics over source code can usefully provide *both* positive and negative feedback to a miner attempting to distinguish between true and false specification candidates. [1]

The main contributions of this article are:[2]

- We identify and describe lightweight, automatically-collected software features that approximate source code quality for the purpose of mining specifications, and we evaluate their relative predictive powers.
- We explain how to lift code quality metrics to metrics on traces, and empirically measure the utility of our lifted quality metrics when applied to previous static specification mining techniques. Existing off-the-shelf specification miners can learn just as many specifications using only the 44% highest-quality traces, refuting previous claims that more traces are neccessarily better for mining [57].
- We propose two novel specification mining techniques that use our automated quality metrics to learn temporal safety specifications while avoiding false positives. We compare our approaches to two previous approaches, evaluating on over 1.5 million lines of code. Our basic mining technique balances true and false specifications. It learns specifications that locate more safety-policy violations than previous miners (884 vs. 663) with a lower rate of false positives (63% vs. 89%). Our second technique focuses on precision. It obtains a 3% false positive rate (1 false candidate), an order-of-magnitude improvement on previous work. The specifications it finds identify 355 violations. To our knowledge, these are the first scalable specification miners that produce multiple two-state candidate specifications with false positive rates under 89%.

The rest of this paper is organized as follows. In Section 2, we describe temporal safety specifications and highlight uses, and give a brief overview of specification mining. Section 3 presents an example that motivates the insight formalized in our mining approach. Section 4 describes our approach to specification mining, including the quality metrics used. In Section 5, we present experiments supporting our claims and evaluating the effectiveness of our miner. We discuss related work in Section 6. We conclude in Section 7.

1. Enumerating all true specifications for a project is undecideable, and the number of possible true positives is usually unknown. We adopt the domain practice of using "all true specifications identified to date" as a proxy for "all true specifications" for a given program. Statistical techniques such as capture, recapture analyses do not apply because the underlying distribution of specifications in source code is unknown and cannot be assumed to be random.

2. Some of these points were previously presented [38], [59]; a detailed comparison between the previous work and this article is provided in Section 6.

2 Background

In this section, we present background on temporal safety specifications and how they may be mined automatically from source code.

2.1 Temporal Safety Specifications

A *partial-correctness temporal safety property* is a formal specification of an aspect of required or correct program behavior [37]; they often describe how to manipulate important program resources. We refer to such properties as "specifications" for the remainder of this article. Such specifications can be represented as a finite-state machine that encodes valid sequences of *events*. Figure 1 shows source code and a specification relating to SQL injection attacks [40]. In this example, one potential event involves reading untrusted data from the network, another sanitizes input data, and a third performs a database query. Typically, each important resource is tracked with a separate finite state machine [16] that encodes the specification that applies to its manipulation. A program execution *adheres* to a given specification if and only if it terminates with the corresponding state machine in an accepting state (where the machine starts in its start state at program initialization). Otherwise, the program *violates* the specification and contains an error.

This type of partial correctness specification is distinct from, and complementary to, full formal behavior specifications. They can be used to describe describe many important correctness properties, including resource management [59], locking [13], security [40], high-level invariants [23], memory safety [31], and more specialized properties such as the correct handling of `setuid` [11] or asynchronous I/O request packets [5]. Such specifications are used by almost all existing defect-finding tools (e.g., [5], [13], [14], [23]). Additionally, formal specifications are instrumental in program optimization [39], testing [6], refactoring [34], documentation [8], and repair [56].

In this article, we focus on the simplest and most common type of temporal specification: a two-state finite state machine [20], [38], [59]. A two-state specification states that an event a must always eventually be followed by event b. This corresponds to the regular expression $(ab)^*$, which we write $\langle a,b \rangle$. We focus on this type of property because mining FSM specifications with more than two states is historically imprecise, and debugging such specifications manually is difficult [4]. While two-state temporal properties are by definition more limited in their expressive power [25], [61], they can be used to

```
void bad(Socket s, Conn c) {
  string message = s.read();
  string query = "select * " +
    "from emp where name = " +
    message;
  c.submit(query);
  s.write("result = " +
    c.result());
}
```

```
void good(Socket s, Conn c) {
  string message = s.read();
  c.prepare("select * from "
    + " emp where name = ?",
    message);
  c.exec();
  s.write("result = " +
    c.result());
}
```

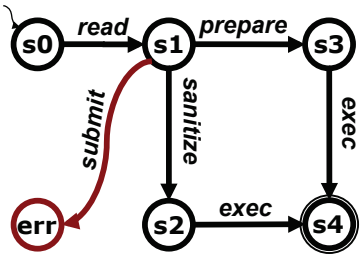

Fig. 1. Pseudocode and specification for an networked program that receives untrusted data via sockets. The **bad** method passes unsafe data to the database; **good** works correctly. Important ***events*** are italicized. The partial-correctness temporal safety specification shown on the right governs database interactions.

describe important properties such as those controlling resource allocation/deallocation or invariant restoration (examples include ⟨**open**,**close**⟩, ⟨**malloc**,**free**⟩, and ⟨**lock**,**unlock**⟩). These examples, and similar such specifications, are prevalent in practice [14].

2.2 Specification Mining

Specification mining seeks to construct formal specifications of correct program behavior by analyzing actual program behavior. Program behavior is typically described in terms of sequences of function calls or other important events. Examples of program behavior may be collected statically from source code (e.g., [20]) or dynamically from instrumented executions on indicative workloads (e.g., [60]). A specification miner examines such *traces* and produces *candidate specifications*, which must be verified by a human programmer. Some miners produce a single finite automaton policy with many states [2], [3], [60]. Others produce many small automata of a fixed form [20], [25], [59], [61]. As large automata are more difficult to verify or debug [4], we choose to focus on the latter, as described above.

Mining even these simple two-state specification remains difficult [25]. Given the large number of candidate $\langle a,b \rangle$ pairs generated by even a restricted set of program traces, determining which pairs constitute valid policies is non-obvious. Most pairs, even those that frequently occur together (such as ⟨**print**,**print**⟩ or, more insidiously, ⟨**hasNext**,**getNext**⟩), do not represent required pairings: a program may legitimately call **hasNext** without ever calling **getNext**. Miners can also be led astray by policy violations, as they seek to discern correct behavior from code that may be incorrect.

Engler *et al.* note that programmer errors can be inferred by assuming that the programmer is *usually* correct [20]. In other words, common behavior implies correct behavior, while uncommon behavior may suggest a policy violation (a principle that similarly underlies modern intrusion detection, e.g. [24]). Intuitively, a candidate specification that is followed on 10 traces and violated on another 10 is unlikely to encode required behavior, since programmers rarely make mistakes a full 50% of the time. However, a candidate that is adhered to in 90% of relevant traces more likely represents required behavior. Engler *et al.*'s miner operates on this principle by counting the number of times a and b appear together and the number of times that a appears without b. It uses the z-statistic for comparing proportions to rank the likelihood that the correlation is deliberate, ultimately presenting a ranked list of candidate specifications for programmer review. Unfortunately, without additional human guidance, this technique is prone to a very high rate of false positives. On one million lines of Java code, only 13 of 2808 positively-ranked specifications generated by **ECC** were real: a 99.5% false positive rate [59].

We observed in previous work that programmers often make mistakes in error handling code [59]. We used an additional bit per trace — whether it passed through a **catch** block — when evaluating candidate pairs (i.e., event a must be followed by event b on at least one non-error trace and not followed by b on at least one error trace). Additionally, we required that the events a and b in a candidate pair come from the same package or library, assuming that independent libraries are unlikely to depend on one another for API-level correctness. These insights led to the **WN** mining algorithm, which improved mining accuracy by an order of magnitude. On the same million lines of Java code, **WN** generated only 649 candidate specifications, of which 69 were real, for an 89% false positive rate. However, this rate is still too high for automatic applications or industrial practice, as candidates must still be hand-validated. This article proposes a new specification mining approach that lowers this false positive rate.

3 Motivating Example

In this section, we motivate our quality-based mining technique by showing that we can use measurements of code quality to distinguish between a true and a false positive candidate specification.

Traces containing a and b in the appropriate order *adhere* to a candidate specification $\langle a,b \rangle$, while traces that only contain a *violate* it. Figure 2 shows two candidates, one true and one false, mined from **ptolemy2**, an open source Java project for design modeling by Engler *et al.*'s z-score technique [20], described in Section 2.2. Unfortunately, ranking candidates by their z-scores does not sufficiently distinguish them. The two candidates shown, as well as more obvious false positives such as ⟨**print**,**print**⟩, appear near one other on the z-ranked list of 655 candidates.

Candidate Specification	Correctness
⟨`Workspace.getReadAccess()`, `Workspace.doneReading()`⟩)	Valid Specification
⟨`ASTPtRootNode.isConstant()`, `ASTPtRootNode.isEvaluated()`⟩	False Positive

Fig. 2. Two of the 655 candidate specifications mined by `ECC` on `ptolemy2`. The first candidate is a true specification: `getReadAccess` must always eventually be followed by `doneReading` along all program paths. The second candidate is a false specification: a program path may include `isConstant` but not `isEvaluated` and still be considered correct.

We hypothesize that high quality code is a better guide to required behavior than low quality code, and thus should be given more weight when counting frequencies for mining. Taken from the other side, code that adheres to a true specification should generally be of higher quality than code that violates it (such violating code contains an error, by definition). Code traces adhering to or violating a false candidate should not differ dramatically in quality since, with respect to the spurious candidate, neither trace is more correct than the other. Bearing these patterns in mind, code quality can provide both positive and negative feedback to an automatic specification miner. We propose to identify quality metrics that distinguish between good and bad code, and thus the candidates presented in Figure 2. We briefly describe only a few metrics here for the purposes of illustration; Section 4.1 presents a complete list of metrics used.

Our previous work suggested that programmers make mistakes in error-handling code [59], perhaps because programmers do not reason properly about uncommon code paths (such as those through `catch` blocks). We surmise that a candidate that is adhered to on common paths but violated on uncommon paths is thus more likely a true specification, as the violations are more likely to be bugs. We use a research tool [10] that statically predicts the likelihood that a path will be executed when its enclosing method is called (its predicted *frequency*). We observe that the hypothesized pattern holds for the adhering and violating traces of the candidates in Figure 2. Traces that adhere to the true candidate have an average predicted frequency of 39%; those that violate it, only 4%. By contrast, the false candidate's adhering traces are predicted to be run less frequently than its violating traces (31% vs 58%)!

Other research presented a human-defined code *readability* metric; more readable code is correlated with fewer errors [9]. Reapplying the logic from above, we hypothesize that the true specification's adhering traces are more readable than its violating traces (containing an error), and that such a distinction might not hold for the false candidate; we use the research tool described in [9] to measure normalized code readability. The true specification's traces have quite different maximum readabilities: 0.98 for adhering traces vs. 0.59 for violating traces. By contrast, the false candidate's traces again follow the opposite pattern: adhering traces are less readable than violating traces (0.05 vs 0.31), suggesting that violations of the false positive candidate are not really errors.

Finally, previous research suggests that recently or frequently edited code is correlated with policy violations [45]; code that has been stable for a long time is more likely to be correct. A project's source control management logs admit measurement of *churn* along any given trace, and the code for the two candidates in Figure 2 follows that pattern. The true specification's adhering traces were last changed much earlier in the development process on average (revision 15387 out of 29324 total revisions) than the violating traces (revision 20884). The adhering traces have been stable for a longer period of time; they are more likely to be correct. The false candidate's traces again follow the anti-pattern: the adhering traces were last changed more recently than the violating ones (25189 vs. 19238).

The candidates in Figure 2 are difficult to distinguish by looking only at the proportion of traces on which they are followed or violated (i.e., their z-rank). However, the code from which they are mined is not of equivalent quality, and measurable features may follow patterns that can help us distinguish two otherwise very similar candidates. The rest of this article formalizes this notion and presents an empirical evaluation of its validity.

4 Our Approach

We present a new specification miner that works in three stages. First, it statically estimates the quality of source code fragments. Second, it lifts those quality judgments to traces by considering all code visited along a trace. Finally, it weights each trace by its quality when counting event frequencies for specification mining.

Code quality information may be gathered either from the source code itself or from related artifacts, such as version control history. By augmenting the trace language to include information from the software engineering process, we can evaluate the quality of every piece of information supporting a candidate specification (traces that adhere to a candidate as well as those that violate it and both high and low quality code) on which it is followed and more accurately evaluate the likelihood that it is valid. Section 4.1 provides a detailed description of the set of features we have chosen to approximate the quality of code; Section 4.2 details our mining algorithm.

4.1 Quality Metrics

We define and evaluate two sets of metrics. The first set consists of seven metrics chosen to approximate code *quality*. This list should not be taken as exhaustive, nor are the quality metrics intended to individually or perfectly measure quality. Indeed, a primary thesis of this article is that lightweight and imperfect metrics, when used in combination, can usefully approximate quality

for the purposes of improved specification mining. Thus, we focus on selecting metrics that can be quickly and automatically computed using commonly-available software artifacts, such as the source code or version control histories. We looked particularly to previous work for code features that correlate with fault-proneness or observed faults. In the interest of automation, we exclude metrics that require manual annotation or any other form of human guidance.

The second set of metrics consists of previously-proposed measures of code *complexity*. We use these primarily as baselines for our analysis of metric power in Section 5; this evaluation may also be independently useful given their persistent use in practice [48].

The metrics in the first set ("quality metrics") are:

Code Churn. Previous research has shown that frequently or recently modified code is more likely to contain errors [45], perhaps because changing code to fix one defect introduces another, or because code stability suggests tested correctness. We hypothesize that churned code is also less likely to adhere to specifications. We use version control repositories to record the time between the current revision and the last revision for each line of code in wall clock hours. We also track the total number of revisions to each line. Such metrics can be normalized or given as absolute ranges.

Author Rank. We hypothesize that the author of a piece of code influences its quality. A senior developer who is very familiar with the project and has performed many edits may be more familiar with the project's invariants than a less experienced developer. Source control histories track the author of each change. The *rank* of an author is defined as the percentage of all changes to the repository ever committed by that author. We record the rank of the last author to touch each line of code. While author rank may be led astray by certain methodologies (e.g., some projects may have a small set of committers that commit on behalf of more than one author [18]; others may assign more difficult and thus error-prone tasks to more senior developers), we note that it may be automatically collected from version control histories and is a proxy for expertise, which is otherwise challenging to approximate automatically.

Code Clones. We hypothesize that code that has been duplicated from an another location may be more error-prone because it has not necessarily been specialized to its new context (e.g., copy-paste code), and because patches to the original code may not have propagated to the duplicate. Research has shown that cloned code is changed consistently a mere 45–55% of the time [36]. While not all code cloning is harmful [32], perhaps because common code clones may be more comprehensively tested, we further hypothesize that duplicated code does not represent an independent correctness argument: if `print` follows `hasNext` in 20 duplicated code fragments, it is not necessarily 20 times as likely that ⟨`hasNext`,`print`⟩ is a true specification. As it is impossible to automatically and retroactively distinguish between coincidental and deliberate code clones, we approximate this metric using clone detection techniques. We use the open-source `PMD` toolkit's clone detector to track likely copy-paste repetition. The detector is based on the Karp-Rabin string matching algorithm [33]. We express the measure of code cloing for a given code fragment as the product of the length of the code segment and the number of times it has been copied.

Code Readability. Buse *et al.* developed a code metric trained on human perceptions of readability or understandability [9]. The metric uses textual source code features — such as number of characters, length of variable names, or number of comments — to predict how humans would judge the code's readability. Readability is defined on a scale from 0 to 1, inclusive, with 1 describing code that is highly readable. More readable code is less likely to contain errors. We therefore hypothesize that more readable code is also more likely to adhere to specifications. We use the research prototype developed by Buse *et al.* to measure the readability of source code [9].

Path Feasibility. Our specification mining technique operates on statically enumerated traces, which can be acquired without indicative workloads or program instrumentation. Infeasible paths are an unfortunate artifact of static trace enumeration, and we claim that they do not encode programmer intentions. Merely discounting provably infeasible paths may confer some benefit to the mining process. However, infeasible paths may suggest pairs that are *not* specifications: a programmer may have made it impossible for b to follow a along a path, suggesting that $\langle a,b \rangle$ is not required behavior. We prefer static paths for our purposes first because they are both easier to obtain and more complete than dynamic paths. In addition, we hypothesize that static paths combined with symbolic execution can provide additional useful information about behavior the programmer believes should be impossible. We measure the feasibility of a path using symbolic execution; a path is infeasible if a theorem prover reports that its symbolic branch guards are inconsistent. Path feasibility is expressed as one of $\{0, 0.5, 1\}$; 0 denotes an infeasible path, 1 a required path, and 0.5 a path that may or may not be feasible or required.

Path Frequency. We theorize that common paths that are frequently executed by indicative workloads and test cases are more likely to be correct. First, the programmer may reason more thoroughly about the "common case", and second, highly-tested code is less likely to contain errors. We use a research tool that statically estimates the relative runtime frequency of a path through a method [10], normalized as a real number.

Path Density. We hypothesize that a method with more possible static paths is less likely to be correct because there are more corner cases and possibilities for error. We define "path density" as the number of traces it is possible to enumerate in each method, in each class, and over the entire project. A low path density for traces

containing paired events ab and a high path density for traces that contain only a suggest that $\langle a,b \rangle$ is a likely specification. Path density is expressed in whole numbers and can be normalized to the maximum number of enumerated paths (30/method, in our experiments).

Metrics in the second class ("complexity metrics") are:

Cyclomatic Complexity. McCabe defined *cyclomatic complexity* [44] to quantify the decision logic in a piece of software. A method's complexity is defined as $M = E - N + 2P$, where E is the number of edges in the method's control flow graph, N is the number of nodes, and P is the number of connected components. There is no theoretical upper bound on the complexity of a method. The complexity of an intraprocedural trace is the complexity of its enclosing method. Previous work suggests that Cyclomatic complexity correlates strongly with the length of a function and does not correlate well with errors in code [22], [50]. Despite this, Cyclomatic complexity remains in industrial use [48]. We hypothesize that complexity will not helpfully contribute to our specification mining model.

CK Metrics. Chidamber and Kemerer proposed a suite of theoretically-grounded metrics to approximate the complexity of an object-oriented design [12]. The following six metrics apply to a particular class (i.e., a set of methods and instance variables):

- **Weighted Methods per Class (WMC)**: number of methods in a class, weighted by a user-specified complexity metric. Common weights selected in practice are 1, the method length, or the method's Cyclomatic complexity. The experiments in this article weight all methods equally (weight = 1).
- **Depth of Inheritance Tree (DIT)**: maximal length from the class to the root of the type inheritance tree.
- **Number of Children (NOC)**: number of classes that directly extend this class.
- **Coupling Between Objects (CBO)**: number of other objects to which the class is coupled. Class A is coupled to Class B if one of them calls methods or references an instance variable defined in the other.
- **Response for a Class (RFC)**: size of the response set, defined as the union of all methods defined by the class and all methods called by all methods in the class.
- **Lack of Cohesion in Methods (LOCM)**: Methods in a class may reference instance variables in that class. P is the set of methods in a class that share in common at least one instance variable with at least one other class method. Q is the set of methods that do not reference instance variables in common. LOCM is $|P - Q|$ if $|P - Q| > 0$ and 0 otherwise.

The CK metrics are also sometimes used in industry to measure design or system complexity. Research on their utility has yielded mixed results — studies have correlated subsets of the metrics with fault-proneness, though they do not tend to agree on which subsets are predictive [7], [52], [54].

N_a	=	$\lvert\{t \mid a \in t \ \wedge \ \neg Error(t)\}\rvert$
N_{ab}	=	$\lvert\{t \mid a \ldots b \in t \ \wedge \ \neg Error(t)\}\rvert$
E_a	=	$\lvert\{t \mid a \in t \ \wedge \ Error(t)\}\rvert$
E_{ab}	=	$\lvert\{t \mid a \ldots b \in t \ \wedge \ Error(t)\}\rvert$
z	=	ECC z-score
SP_{ab}	=	1 if a and b are in the same package, 0 otherwise
DF_{ab}	=	1 if every value in b also occurs in a, 0 otherwise
$\mathcal{M}_{ia}$	=	$\mathcal{M}_i(\{t \mid a \in t\})$
$\mathcal{M}_{iab}$	=	$\mathcal{M}_i(\{t \mid a \ldots b \in t\})$

Fig. 3. Features used by our miner to evaluate a candidate specification $\langle a,b \rangle$. $\mathcal{M}_i$ is a quality metric lifted to sets of traces.

4.2 Mining Algorithm Details

Our mining algorithm extends our previous **WN** miner [38], [59], notably by including quality metrics from Section 4.1. Our miner takes as input:

1) The program source code P. The variable ℓ ranges over source code locations. The variable l represents a set of locations.
2) A set of quality metrics $M_1 \ldots M_q$. Quality metrics may map either individual locations ℓ to measurements, with $M_i(\ell) \in \mathbb{R}$ (e.g., code churn) or entire traces to measurements, where $M_i(l) \in \mathbb{R}$ (e.g., path feasibility).
3) A set of important events Σ, generally taken to be all of the function calls in P. We use the variables a, b, etc., to range over Σ.

Our miner produces as output a set of candidate specifications $C = \{ \ \langle a,b \rangle \mid a$ should be followed by $b\}$. We manually evaluate candidate specification validity.

Our algorithm first statically enumerates a finite set of intra-procedural traces in P. Because any non-trivial program contains infinite number of traces, this process requires an enumeration strategy. We perform a breadth-first traversal of paths for each method m in P. We emit the first k such paths, where k is specified by the programmer. Larger values of k provide more information to the mining analysis with a corresponding slowdown. Experimentally, we find that very large k provide diminishing returns in the tradeoff between correctness and time/space. Typical values are $10 \leq k \leq 30$. To gather information about loops and exceptions while ensuring termination, we pass through loops no more than once, and assume that branches can be either taken or not and that an invoked method can either terminate normally or raise any of its declared exceptions. Thus, a path through a loop represents all paths that take the loop at least once, a non-exceptional path represents all non-exceptional paths through that method, etc. This approach is consistent with other researchers' path enumeration strategies, including those used by some of our metric-collection techniques [10]. We find that the level of detail provided by this strategy is adequate for our purposes, but note

that it is possible to collect additional detail, such as by increasing the number of loop iterations.

This process produces a set of traces T. A trace t is a sequence of events over Σ; each event corresponds to a location ℓ. We write $a \in t$ if event a occurs in trace t and $a \dots b \in t$ if event a occurs and is followed by event b in that trace. We note whether a trace involves exceptional control flow; this judgment is written $Error(t)$.

Next, our miner lifts quality metrics from individual locations to traces, where necessary. This lifting is parametric with respect to an aggregation function $A : \mathcal{P}(\mathbb{R}) \to \mathbb{R}$. We use the functions max, min, span and average to summarize quality information over a set of locations l. M^A denotes a quality metric M lifted to traces: $M^A(t) = A(\{M(\ell) \mid \ell \in t\})$ (metrics that operate over sets of locations do not need to be aggregated; $M^A(t) = M(l)$ where l is the set of locations in t). $\mathcal{M}$ denotes the metric lifted again to work on sets of traces: $\mathcal{M}(T) = A(\{M^A(t) \mid t \in T\})$.

Finally, we consider all possible candidate specifications. For each a and b in Σ, we collect a number of *features*. Figure 3 shows the set of features our miner uses to evaluate a candidate specification $\langle a,b \rangle$. N_{ab} denotes the number of times b follows a in a non-error trace. N_a denotes the number of times a occurs in a normal trace, with or without b. We similarly write E_{ab} and E_a for these counts in error traces. $SP_{ab} = 1$ when a and b are in the same package. $DF_{ab} = 1$ when dataflow relates a and b: when every value and receiver object expression in b also occurs in a [59, Section 3.1]. z is the statistic for comparing proportions used by the **ECC** miner to rank candidate specifications. The set of features further includes the aggregate quality for each lifted metric M^A. We write $\mathcal{M}_{iab}$ (respectively $\mathcal{M}_{ia}$) for the aggregate metric values on the set of traces that contain a followed by b (resp. contain a). As we have multiple aggregation functions and metrics, $\mathcal{M}_{ia}$ actually corresponds to over a dozen individual features.

When combined with the aforementioned statistical measurements and frequency counts, each pair $\langle a,b \rangle$ is described by over 30 total features f_i. We avoid asserting an *a priori* relationship between these features and whether a pair represents a true specification. Instead, we will build a classifier that examines a candidate specification and, based on learned a linear combination of its feature values, determine whether it should be emitted as a candidate specification. A training stage, detailed in Section 5, is required to learn an appropriate classifier relating features to specification likelihood.

5 Experiments

In this section, we empirically evaluate our approach. We begin by explaining how we build a model relating code quality metrics to the likelihood that a candidate is a true specification, and how this model can be used as a specification miner. We use this model to evaluate several research questions:

Program Version	LOC	Description
hibernate2 2.0b4	57k	Object persistence
axion 1.0m2	65k	Database
hsqldb 1.7.1	71k	Database
freecol 0.4.0	73k	Game
cayenne 1.0b4	86k	Object persistence
jboss 3.0.6	107k	Middleware
mckoi-sql 1.0.2	118k	Database
tvbrowser 2.2.5	130k	TV Guide
jedit 4.0	140k	Text editor
jasperreports 1.2.0	153k	Dynamic content
jfreechart 1.0.13	316k	Data reporting
ptolemy2 3.0.2	362k	Design modeling
Total	1.5M	

Fig. 4. Open-source Java benchmark set.

1) Our first set of experiments evaluates the predictive power and statistical independence of the code quality metrics.
2) Our second experiment provides evidence that our metrics improve existing techniques for automatic specification mining.
3) Our final experiment measures the efficacy of our new specification miner in terms of mined specification utility and false positive rate, using previous techniques as a baseline.

We perform our evaluation on the 12 open-source Java benchmarks shown in Figure 4. Several of these programs allow a direct comparison with previously published results [25], [38], [57], [59], [61]: **hibernate**, **axion**, **hsqldb**, **cayenne**, **jboss**, **mckoi-sql** and **ptolemy2**. We do not need source code *implementing* a particular interface; instead, we generate traces from the client code that *uses* that interface (as in [3], [19], [25], [61]); we thus mine both specifications specific to a particular benchmark as well as library-level API requirements. We restrict attention to programs with CVS or SVN source-control repositories, since such information is necessary for certain metrics. For the purposes of consistent data collection, we use the **cvssuck** utility to convert CVS repositories to SVN repositories. We used the **blame** command in SVN to collect author information, and **info** to collect churn information. We statically enumerated up to 30 traces per method per benchmark.

Our technique is relatively efficient. The most expensive operation is computing path feasibility, as it requires multiple calls to a theorem prover (we use **simplify** [17]). Computing feasibility on the **mckoi-sql**, our median benchmark, took 25 seconds on a 3 GHz Intel Xeon machine. Enumerating all static traces for **mckoi-sql**, with a maximum of 30 traces per method, took 912 seconds in total; this happens once per program. Collecting the other metrics for **mckoi-sql** is relatively inexpensive (e.g., 6 seconds for readability, 7 seconds for path frequency). The actual mining process (i.e., considering the features for every pair of events in **mckoi-sql** against the cutoff) took 555 seconds. The total time for our technique was about 30 minutes per 100,000 lines of

code.

5.1 Learning a Model

First, we construct a linear model that, given a set of features associated with a potential $\langle a,b \rangle$ pair, determines whether it should be output as a candidate specification. We use linear regression to learn the coefficients c_i and a $cutoff$, such that our miner outputs $\langle a,b \rangle$ as a candidate specification iff $\sum_i \ c_i f_i > cutoff$. In other words, a specification is emitted if the linear combination of its features weighted by the coefficients exceeds the cutoff. A notable weakness of linear regression is that we do not know *a priori* if the proposed features are related linearly; it is possible for the features to have a non-parametric relationship. Accordingly, we add the log, absolute value, square, and square root of each feature vector to the testing and training sets.[3]

Linear regression requires annotated answers (in our case, a set of known-valid and known-invalid specifications). Our training set consists of valid and invalid specifications mined and described in previous work [57], [59] and manually annotated specifications from the new benchmarks. We used the source code of a and b, surrounding comments, source code in which a and b was either adhered to or violated, and related documentation (such as the `Hibernate` APIT documentation) to evaluate whether a candidate specification represented a true or false positive. A potential threat to the validity of the model is over-fitting by testing and training on the same data. We must therefore verify that our miner is not biased with respect to our training data. We mitigate this threat with 10-fold cross validation [35]. We randomly partition the data into 10 sets of equal size. We test on each set in turn, training on the other nine; in this way we never test and train on the same data. Bias is suspected if the average results of cross-validation (over many random partitionings) are different from the original results. The difference was less than 0.01% for our miner, indicating little or no bias.[4]

We evaluate potential models with *recall* and *precision*. Recall measures the probability that a given real specification is returned by the algorithm, expressed as the fraction of real specifications returned out of all known real specifications. Precision measures the probability that a returned candidate specification is a true specification, expressed as the fraction of candidate specifications that are true positives. A high recall indicates that the miner is doing useful work (i.e., returning real specifications), but without a corresponding high precision, real specifications drown in a sea of false positives. An information retrieval task can trivially maximize either precision or recall be returning nothing (all returned elements are true positives) or everything (all true positives are returned along with everything else). Accordingly, information retrieval tasks may measure the harmonic mean of precision and recall, known as the *f-measure*. Given the set of coefficients, we perform a linear search to find a cutoff that maximizes one of these functions. Our *normal miner* maximizes f-measure; our *precise miner* maximizes precision (yielding very few false positives). We do not build a miner to maximize recall because our dominant concern is reducing false positives.

3. A logistic model, which fits features to a probability range between 0 and 1, would also naturally fit our task. However, such a model still requires a training phase to produce a binary classifier. As the same work is required to employ either type of model, we prefer a linear model because it admits straightforward statistical analyses and, in practice, is sufficiently accurate.

4. We perform cross validation in lieu of partitioning our dataset because it allows us to evaluate on more benchmarks, lines of code, and specifications, supporting the generality of our technique, while still establishing that the technique is not biased by over-fitting.

Metric	F	p
Code Churn	44.2	<0.0001
Path Frequency	26.4	<0.0001
Readability	19.9	<0.0001
Author Rank	17.4	<0.0001
Path Feasibility	11.9	0.0006
Path Density	9.3	0.0379
Code Clones	8.2	0.0039
Cyclomatic Complexity	1.0	0.2498
CK Metric RFC	21.1	<0.0001
CK Metric DIT	9.1	0.0026
CK Metric NOC	7.3	0.0067
CK Metric WMC	7.4	0.0064
CK Metric CBO	6.9	0.0085
CK Metric LOCM	5.1	0.0235
Exceptional Path	31.8	<0.0001
One Error	28.2	<0.0001
Same Package	2.9	0.0890
Dataflow	1.9	0.1679

Fig. 5. Analysis of variance of features in our model. The F column displays a feature's F-ratio: the square of the variance in the model explained by a feature over the variance not explained. The p column shows the probability that the feature does not affect the miner. F values approaching 1 indicate lack of predictive power; $p \leq 0.05$ indicates statistical significance. The bottom four features are present in the `WN` miner [59].

5.2 Predictive Power of Quality Metrics

In this section, we evaluate the coefficients of the linear model to understand the overall predictive power of each of our proposed quality metrics, compare the utility of the metrics on different benchmarks and qualitatively analyze observed differences, and establish the independence of many of the quality metrics.

5.2.1 Quality Metrics Across All Benchmarks

Our first experiment **evaluates the relative importance of our quality metrics**. We perform a per-feature analysis of variance on the linear model; the results are shown in Figure 5. All of the quality metrics defined in Section 4.1 except Cyclomatic complexity had a significant main effect ($p \leq 0.05$). The code churn metric, encoding how frequently and recently a line of code has been changed in the source control repository, was our most important

feature. Path feasibility is of moderate predictive power; it is, to our knowledge, the only feature that had been previously investigated in the context of mining [2].

The author rank metric is significantly predictive in this analysis, overturning previous observations [38] that it has little predictive power. These experiments involve a much larger benchmark set (1.5 M vs. 0.8 M LOC). In addition, we enumerate 30 traces per method; in previous work we enumerated 10. These differences appear to account for the change: on these benchmarks, author rank increases in importance by 50% for every 10 additional traces per method generated between 10 and 30. The previous set of benchmarks may have been insufficiently varied, and the previous set of traces insufficiently deep, resulting in an imprecise model.

We also evaluated the predictive power of traditional complexity metrics: Cyclomatic complexity and the six CK metrics for object-oriented design complexity (recall that we weight all methods equally for the purposes of the WMC metric). Our analysis of variance shows that Cyclomatic complexity has no significant effect on the model, and is not predictive for whether code conforms to specifications for correct behavior. This is consistent with previous research suggesting that Cyclomatic complexity is not predictive for code faults [22], [50]. The six CK metrics, however, vary in predictive power, though all have a significant main effect. With the exception of response for a class (RFC), which is meant to approximate the interconnectedness of the design, the CK metrics are less predictive the other proposed quality metrics. Thes results suggest that the CK metrics may indeed capture an element of code quality or complexity, though they vary in their ability to do so.

In our previous work that examined the relationship between error traces and specification false positive rates [59], we used used several criteria to select candidate pairs: every event b in an event pair must occur at least once in exception cleanup code ("Exceptional Path"), there must be at least one error trace with a but without b ("One Error"), both events must be declared in the same package ("Same Package"), and every value and receiver object expression in b must also be in a ("Dataflow"). We included these features in our model to determine their predictive power. The results are shown in the lower section of Figure 5. The "Exceptional Path" and "One Error" conditions affect the model quite strongly, while the "Same Package" and "Dataflow" conditions are less significant. They are not as predictive as Code Churn, our most predictive metric.

5.2.2 Quality Metrics Between Benchmarks

The previous experiment analyzed the relative importance of the quality metrics across all twelve benchmarks. This section **qualitatively explores factors that affect the predictiveness of our features** by comparing the predictive power of the metrics on each benchmark. We present a non-exhaustive set of observations about benchmark features that appear to relate to metric power. This analysis provides insight into factors that may affect the metrics' predictive power, such as a particular development methodology. The purpose of this discussion is to explore the circumstances under which the proposed technique is useful, and under which it may be misled.

To explore this area, we built individual specification mining models for each of our benchmarks using the technique described in Section 5.1, and perform analyses of variance on each model, noting outliers. We omit the full results in the interest of brevity. Several interesting patterns emerge, however.

First, benchmark size appears to influence the uniformity of our results. **`axion`**, one of our smaller benchmarks, is an outlier on several of our metrics. **`ptolemy`**, our largest benchmark, displays the most uniform behavior, displaying no outlier behavior with any metric. This is encouraging: we expect that the largest, most well-established programs are the most general and thus the least likely to contain anomalous behavior for a predictive model, a trend that generally holds here.

The presence of explicit testing code (e.g., **`JUnit`** unit tests) appears to influence the relative predictive strength of **Frequency**, **Author Rank**, and **Path Density**. Unit testing code tends to be straightforward (in our benchmarks, unit test classes are shorter than 100 lines each, on average), follows implicit specifications, and is designed to check correct program behavior, and is therefore likely to probe and assert that correct behavior. Paths through testing classes are therefore both likely to be correct and also to be executed with high frequency relative to their enclosing methods, A notable outlier on these three metrics is **`axion`**, which ships with a comprehensive unit test suite that comprises a full 48% of the codebase. **Frequency** is more predictive on **`axion`** than several of the other metrics ($F = 22.8, p = 0.001$), as is **Author Rank** ($F = 10.1$, $p = 0.015$), likely because the entire test suite appears to have been written by one author. This author likely wrote correct code, and thus that one programmer's rank is the most likely factor in the high predictive power of author rank on the **`axion`** benchmark. **Path density** is strongly predictive on the **`axion`** ($F = 22.8$, $p < 0.0001$) and **`jboss`** ($F = 21.3, p < 0.0001$) benchmarks. **`jboss`** also ships with a large number of **`JUnit`** unit tests. As unit testing methods are very simple, in general, they also have low density, and thus low density is predictive of specification validity.

The predictive power of **code clones** appears related to the amount of code that is marked as copied in a benchmark: in the limit, if a project contains no duplicate code, the metric is uniformly zero and has no predictive power. The metric is very strongly predictive on **`hibernate`** ($F = 81.8, p < 0.0001$). The PMD toolkit marks 0.8% of **`hibernate`**'s code as cut-and-paste. This figure is twice as high as that of the benchmark with the next highest percentage, **`jboss`** (on which the metric is also strongly predictive). This metric is not predictive on **`mckoi-sql`** ($F = 0.6, p = 0.424$), which has the lowest percentage of copied and posted code (0.03%).

The predictive power of **code churn** appears similarly correlated with the total number of revisions on a given benchmark — that is, it improves with the granularity of code churn judgments and cannot help for projects that contain only one revision. The **hsqldb** 1.7.1 release contained 72 revisions in the svn repository, while **mckoi-sql** had undergone 594 (relatively few compared to its size); on these benchmarks, this metric is not very predictive ($F = 3.3$, $p = 0.070$ and $F = 3.9$, $p = 0.047$, respectively). Comparatively, **tvbrowser** had undergone 5706 revisions, **jfreechart** 2056, and **cayenne** 1961. Correspondingly, **tvbrowser**, and **jfreechart** had F values of 22.0 and 33.4, respectively, with p values < 0.0001.

The number of conditionals in a program is relevant to **feasibility's** predictive power: a path cannot be marked infeasible by static analysis if there are no conditional guards along it. Programs with more conditional guards potentially contain more paths that can be marked infeasible. To gain additional insight, we counted the number of `if` or `while` constructs appearing in our benchmarks' source code. For **ptolemy**, on which feasibility is comparatively highly predictive ($F = 7.3$, $p = 0.007$) contains approximately seven times as many conditional guards as **axion**, on which the metric is not predictive ($F = 1.0$, $p = 0.311$). This ratio of guards to feasibility predictive power holds for all benchmarks except for **mckoi-sql**, which contains fewer conditionals than this hypothesis predicts. Note that `if` and `while` also increase CFG edges and connected components, which are key parts of Cyclomatic complexity and some CK metrics, but feasibility is much more predictive. We hypothesize that, because an external theorem prover reasons about conflicting information, feasibility contains richer semantic information. Number of edges, connected components, and Cyclomatic complexity all correlate strongly with method length; they indicate only which paths contain many branch points. They do not speak to which paths are required or impossible, information which brings more to bear on whether or not a candidate event pair represents required behavior (as hypothesized in Section 4.1, event pairs that must follow one another may be more likely to represent required behavior; those that cannot follow one another on a given path are more likely to not represent required behavior).

We conclude by observing that statically predicted path frequency and code churn are highly predictive on all benchmarks. These metrics may apply more universally because they are independent of local developer choice (cf. readability). Both code churn and path frequency implicitly take advantage of previous testing and validation work done by human developers: code that has not been churned recently is presumably behaving correctly on the test suite or in deployment, and path frequency similarly points to code which is likely to be frequently executed on indicative workloads. Code that is well-tested, and thus conforms to specifications, is likely to have low churn values and high frequency values. Conversely, metrics such as author rank or code clones are most useful in certain corner cases: not all development organizations will have relatively novice programmers or a plethora of duplicate code.

These observations provide insight into the nature of our metrics, their strengths and weaknesses, and their variability between software projects.

5.2.3 Correlation Between Quality Metrics

The final part of our first experiment **provides empirical evidence that our quality metrics are distinct.** Figure 6 shows the results of performing pair-wise Pearson correlation calculations between all metric pairs across the entire benchmark set. A correlation coefficient may range between -1.0 and 1.0, where ± 1.0 indicates that the two variables analyzed are linearly equivalent. A common heuristic for interpreting the magnitude of a correlation holds that correlations between 0.0 and ± 0.2 are very small [26].[5] All correlations have $p < 0.0001$ and are considered significant. According to this heuristic, most statistically significant quality metric pairs are uncorrelated. Code churn and code clones are very slightly correlated (0.24); however, it is logical that code cloning will increase with repository age and size. Readability and Cyclomatic complexity are weakly correlated (0.38); both are known to correlate with path length [9], [44].

Several of the CK metrics do correlate with one another. CBO correlates weakly with RFC (0.33), and more strongly with LOCM (0.56); RFC correlates strongly with WMC (0.77). The CK metrics were designed together to approximate the complexity of an object-oriented design using easily-identifiable features of its type definitions. Most of the metrics, with the exception of DIT (which does not correlate with the others) are defined in terms of number of methods called, defined, or used in the class; some interdependence is to be expected. WMC and RFC are the only complexity metrics that correlate notably with a quality metric: density (as expected, since density is defined both in terms of the number of paths through a method as well as the number of paths through a class, which is influenced by the number of methods defined in the class). The stronger correlations between the CK metrics themselves and between the CK metrics and the quality metrics may partially explain their lesser utility when actually applied to specification mining, an issue explored in greater detail in Section 5.4.

A potential threat to the validity of our hypothesis is that the proposed metrics may be correlated. The results shown in Figure 6 mitigate this threat by suggesting that the proposed quality metrics are not linear combinations of one another. To corroborate these results, we performed a Principal Components Analysis (PCA) on the quality metrics. A PCA can indicate the number of components in a set of features that contribute to the overall variance in the system. Given the 7 features in the set of quality metrics, the PCA revealed that a

5. Some work suggests that ± 0.3 signifies no correlation; in general, cutoffs for interpreting correlations are heuristics.

Metric	Rank	Code Clones	Feasibility	Density	Frequency	Readability	Complexity	WMC	DIT	NOC	CBO	RFC	LOCM
Churn	0.02	0.24	-0.01	0.01	0.02	0.02	-0.01	0.02	-0.01	0.01	-0.03	0.02	-0.01
Rank	1.00	0.04	-0.02	0.00	-0.02	0.01	0.02	-0.01	0.12	-0.01	0.08	0.08	0.05
Code Clones	-	1.00	0.01	0.01	0.01	0.02	0.00	0.04	0.01	0.05	-0.02	0.02	-0.01
Feasibility	-	-	1.00	0.00	-0.03	0.00	0.05	0.03	0.03	0.01	0.06	0.05	0.02
Density	-	-	-	1.00	0.05	0.01	0.03	0.48	0.06	-0.02	0.15	0.57	0.05
Frequency	-	-	-	-	1.00	0.13	0.15	0.12	-0.01	0.12	0.08	-0.04	0.06
Readability	-	-	-	-	-	1.00	0.38	0.23	0.17	0.03	0.12	0.18	0.25
Complexity	-	-	-	-	-	-	1.00	0.10	0.01	0.09	0.07	0.04	0.07
WMC	-	-	-	-	-	-	-	1.00	-0.04	0.09	0.18	0.77	0.21
DIT	-	-	-	-	-	-	-	-	1.00	-0.01	0.15	0.20	-0.02
NOC	-	-	-	-	-	-	-	-	-	1.00	0.02	-0.04	0.01
CBO	-	-	-	-	-	-	-	-	-	-	1.00	0.33	0.56
RFC	-	-	-	-	-	-	-	-	-	-	-	1.00	0.12

Fig. 6. Pearson correlation coefficients r between the different metrics measured across all benchmarks. $|r| \leq 0.2$ is considered low-to-no correlation, $0.2 < |r| \leq 0.5$ is considered a weak correlation [26]. $p < 0.0001$ for all correlations.

combination of 6 is necessary to account for 99% of the overall data variance. This result is consistent with our correlation calculations above. Taken together, the analyses support our claim first that the metrics we propose describe independent aspects of code quality, and second, that, with one exception, the quality metrics do not strongly correlate with the complexity metrics. Further study is required to more comprehensively investigate the relationship between the metrics, and their relationship to quality in other applications.

5.3 Quality Matters for Specification Mining

Our second experiment **presents empirical evidence that our quality metrics improve an existing technique for automatic specification mining**. For each of our benchmarks, we run the unmodified **WN** miner [59] on multiple input trace sets of varying levels of quality. The quality of a trace is defined as a linear combination of the metrics from Section 4.1, with coefficients based on their relative predictive power for specification mining (the F column in Figure 5); we use this measurement to sort the input trace set from highest- to lowest- quality. We compare **WN**'s performance on random baseline sets of static traces to its performance on high quality (and low quality) subsets of those traces. For generality, we restrict attention to feasible traces, since other miners such as **JIST** already disregard infeasible paths [2].

In total on all of the benchmarks, **WN** miner produces 86 real specifications. On average, **WN** finds all of the same specifications using only the top 45% highest quality traces: 55% of the traces can be dispensed with while preserving true positive counts. Since static trace enumeration can be expensive and traces are difficult to obtain [57], reducing the costs of trace gathering by a factor of two is significant. As a point of comparison, when a random 55% of the traces are discarded, we find only 58 true specifications in total (67% of the total possible set), with a 3% higher rate of false positives.

We next explore the impact that the quality of a trace set has on mining success by passing proportions of the total input set (all traces from all benchmarks) to the **WN** miner. We perform mining on the top N% of the traces (the "High Quality" traces), the bottom N% of the traces (the "Low Quality" traces), and a random N% of the traces. For the "Random" results, we presented the average of five runs on different random subsets; error bars denote one standard deviation in either direction.

Figure 7 presents the results of this experiment by showing the percentage of the total specification set mined by **WN** at each trace set size for sets of high, random, and low quality traces. We conclude that trace quality has a strong impact on the success of the miner. First, the higher-quality traces allow the miner to find more specifications on smaller input sets than do the randomly selected traces; the low-quality traces consistently yield far fewer true specifications. To highlight one point on the graph: on 25% of the input, the high-quality traces yield 65% of the total possible mined specifications. By contrast, the random traces yield less than half, at 43%, and the low-quality traces, only 2% (only 2 true specifications!). By the time the top-quality traces have yielded all possible true-specifications, the random traces have found 88%, and the low quality traces, 63%.

This trend holds at all data points except at 10% of the input, where the random subsets yielded a very slightly higher percentage of total specifications found. However, the difference in the number of specifications mined is very low: only 3 additional true specifications are found on the random subset. More importantly, the difference in the false positive rate high quality versus the random traces is most marked on smaller subsets: at both 5 and 10%, the miner has a false positive rate of 83% on high quality traces and 89% on random traces. Although false

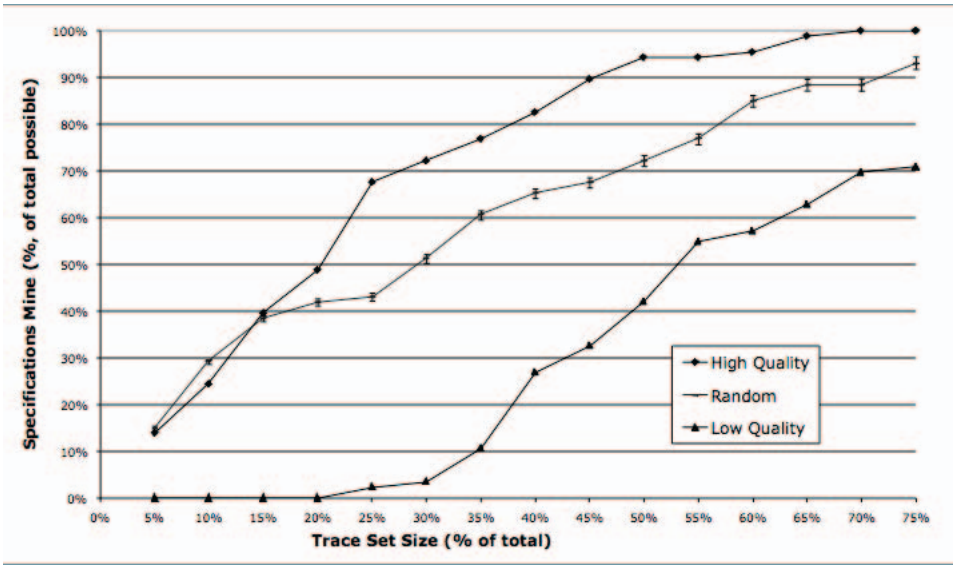

Fig. 7. Performance of the WN specification miner on subsets of the total trace set. "Random" points are an average of five randomly selected subsets. "High quality" quality are the top N% of the traces when sorted by quality. The y-axis shows the percentage of the possible true specifications mined. The false positive rate on high quality (85%) is lower than on random (89%).

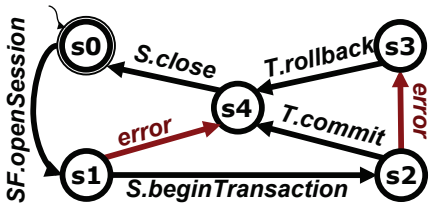

Fig. 9. A finite state machine describing the `Hibernate Session` API, from the `Hibernate` documentation.

positive rate is not shown in Figure 7, the miner finds a lower rate of false positives on the high-quality traces than on the random traces 85% vs. 89%, on average.

Trace generation is often a bottleneck for static specification mining techniques. We claim that, in general, high quality traces should be pursued and low quality traces should be skipped. Our quality metrics could therefore improve any static trace-based specification miner (e.g., [20], [25], [61]). These results also have implications for multi-party techniques to mine specifications collaboratively by sharing trace information [57]: focus should be placed on sharing information from high quality traces. Further, this experiment suggests that our notion of code quality generalize beyond our particular mining application/implementation.

5.4 Quality-Based Specification Mining

Our main experiment **measures the efficacy of our new specification miner**. The miner uses features from previous miners and the quality metrics proposed in Section 4.1; it excludes complexity metrics. We omit Cyclomatic complexity because it is not predictive in the linear model (Section 5.2). A leave-one-out analysis shows the including the CK metrics in the model raises both the true and false positive rate. As our goal is useful specifications with few false positives, we omit features, even those that are predictive for true positives, that increase the false positive rate substantially.

Figure 8 shows the results of applying the new miners to the benchmarks in Figure 4. For each benchmark, we report the number of true and false positive candidates returned (determined by manual verification). Recall the normal miner minimizes both false positives and negatives, while our precise miner minimizes false positives. For comparison, we also show results of the `WN` [59] and `ECC` [20] mining techniques. These miners were chosen for comparison because of their comparatively low false positive rates; other methods produce even more candidates. On `jboss`, the `Perracotta` miner produces 490 candidate two-state properties, which the authors say "is too many to reasonably inspect by hand." [61] Gabel and Su report mining over 13,000 candidates from `hibernate` [25]. Finally, as a heuristic for measuring mined specification utility, we report the number of distinct methods that violate the valid mined specifications (i.e., the number of potential policy violations found by a bug-finding tool using that specification). Each method is counted only once per specification, even if multiple paths through that method violate it. See [58, pp.423–425] for a survey of the bugs found in these benchmarks.

The normal miner finds useful specifications with a low false positive rate. It improves on the false positive rate of `WN` by 26%, while still finding 72% of the same specifications. It finds 4 times as many true specifications as `ECC`. Moreover, the specifications that it finds find more violations on average than those found by `WN`: 884 violations, or 13 per valid specification, compared to `WN`'s 426, or 7 per valid specification.

The precise miner produces only one false positive, on `Hibernate`: ⟨`S.beginTransaction`, `T.commit`⟩. Figure 9 shows the relevant API. The candidate behavior is not required because one can legally call `T.rollback` instead of `T.commit`. However, there are no traces on which the

Program	Normal Miner				Precise Miner				**WN**				**ECC**			
		False				False				False				False		
	Specs	#	%	Bugs	Specs	#	%	Bugs	Specs	#	%	Bugs	Specs	#	%	Bugs
hibernate	7	8	53%	279	5	1	17%	153	9	42	82%	93	3	421	99%	21
axion	7	5	42%	71	4	0	0%	52	8	17	68%	45	0	96	100%	0
hsqldb	3	1	25%	36	1	0	0%	5	7	55	89%	35	0	244	100%	0
freecol	3	0	0%	13	3	0	0%	13	4	10	71%	17	0	316	100%	0
cayenne	5	7	58%	45	3	0	0%	23	5	30	86%	18	3	308	99%	8
jboss	14	75	84%	255	2	0	0%	12	11	103	90%	94	2	442	99%	4
mckoi-sql	7	10	59%	20	2	0	0%	7	19	137	88%	69	2	344	99%	5
tvbrowser	7	4	36%	44	4	0	0%	19	10	68	87%	99	2	412	99%	2
jedit	2	3	60%	14	0	0	0%	0	5	84	94%	57	1	155	99%	3
jasperreports	4	2	33%	19	3	0	0%	14	4	17	81%	20	0	642	100%	0
jfreechart	2	0	0%	44	2	0	0%	44	2	5	71%	44	1	419	99%	4
ptolemy	6	1	14%	44	3	0	0%	13	9	183	95%	72	3	653	99%	12
Total	67	116	63%	884	32	1	3%	355	93	751	89%	663	17	4452	99%	59

Fig. 8. Comparative mining results on 1.5M LOC. "Specs"indicates valid specifications, "False" indicates false positive specifications. "Bugs" totals, for each valid specification found, the number of distinct methods that violate it. The two left headings give results for our Normal Miner and our Precise Miner; **WN** and **ECC** are previous algorithms.

false candidate is followed on which the true specification is not, and very few on which the false candidate is violated while the true candidate is not. Our technique therefore cannot distinguish between the two sets of traces, because their quality measurements are nearly identical. This example suggests that further study is needed to help distinguish between extremely common and required behavior. However, we are encouraged by the fact that none of the other APIs demonstrated such behavior, and believe that this implies that our model for specification form and behavior is reasonable in practice.

The precise miner finds fewer valid specifications than either the normal miner or the **WN** miner (it finds almost twice as many true specifications as the **ECC** technique), but its 3% false positive rate approaches levels required for automatic use. Despite the one false positive and the fact that it finds 34% as many specifications as **WN**, the precise miner still finds 53% of the violations: each candidate inspected yields 11 violations on average. This suggests that the candidates found by the precise miner are among the most useful. Users are often unwilling to wade through voluminous tool output [20], [31]; with a 3% false positive rate, and more useful specifications, we claim that our precise miner might be reasonable in both interactive and automatic settings.

5.5 Threats to Validity

There are several threats to the validity of our results. First, they may not generalize to the programs built in industrial practice because our benchmark set may not be representative, representing a threat to external validity. We believe that the addition of approximately 650k LOC of benchmarks compared to previous work mitigates this threat. Moreover, the additional benchmarks are taken from a variety of areas, ranging from dynamic content generation (**jedit**) to a TV guide (**tvbrowser**). We feel that the size and breadth of our benchmarks mitigates this threat significantly.

The first threat to construct validity is over-fitting of our model to the training data. We use cross-validation in Section 5.4 to demonstrate that our results are not biased by over-fitting. A second such threat lies in our manual validation of output candidate specifications: our human annotation process may mislabel the output used for both training and testing. We mitigated this threat by using the source code that both defined and made use of a and b, and related documentation and comments, as available, to evaluate $\langle a,b\rangle$. Specifications were annotated by more than one researcher over the course of development. We also re-checked a fraction of our judgments at random. In addition to spot-checking specification validity over the course of our experiments, we performed a systematic check by re-evaluating 120 randomly selected annotated specifications. This set included both true and false positive specifications. Our re-evaluation identified one falsely-labeled candidates, an error rate of less than 1%.

A final threat lies in our use of "bugs found" as a proxy for specification utility. First, we do not validate the veracity of each reported violation and, given the imprecision of static trace generation, we cannot be certain that every reported violation represents a true error. Moreover, while our mined specifications find more policy violations than those returned by previous techniques, they may not be as useful for tasks such as documenting or refactoring. However, this highlights the fact that the errors identified by a given specification are not the only measurement of its potential utility. Even if this measure of success is imprecise at best, specifications remain important components of the software engineer-

ing process, and our focus on a low false positive rate is an important first step towards industrial practicality of mining approaches. We leave further investigation of specification utility for future work.

6 Related Work

Our work is most related to the two distinct fields of specification mining and software quality metrics.

6.1 Previous Work in Specification Mining

Some of the research presented in this paper were previously presented [38], [59]. This article expands on those papers by providing:

- A motivating example that compares similar candidate specifications to highlight our insights.
- Additional benchmarks, bringing the size of the benchmark set from 866K LOC to 1.6 million LOC.
- A more complex statistical analysis of the predictive power of the metrics. We qualitatively and quantitatively explore factors affecting predictive power, and situate our work in software metric research.
- A comparison of the utility of these metrics to classic complexity metrics (Cyclomatic complexity [44] and the CK-metrics for object-oriented design complexity [12]) and other previously-proposed metrics in the context of specification mining [59].
- A more detailed study of the effects of trace quality on an existing mining technique.

This work is closely related to existing specification mining algorithms, of which there are a considerable number (see [59] for a survey). Our approach extends the **ECC** [20] and **WN** [59] techniques. Both mine two-state temporal properties (referred to as specifications in this article) from static program traces, and use heuristics and statistical measures to filter true from false positives. **WN** improves on the results of **ECC** by narrowing the criteria used to select candidate specifications (e.g., the candidate specification must expose a violation along at least one exceptional path) and by considering additional source code and software engineering features (e.g., whether the events are defined in the same library or package). We formalize both techniques in Section 4.2. We also use some of the same benchmarks in our evaluation to allow explicit comparison, and we incorporate the features used by the previous miners into our own.

Whaley *et al.* propose a static miner [60] that produces a single multi-state specification for library code. The miner constructs a permissive policy that disallows $\langle a,b \rangle$ if function b raises an undersirable exception when an object field is set to a value that function a sets. The same work proposes a dynamic miner that produces a permissive multi-state specification describing all observed behavior in a set of dynamic traces. The **JIST** [2] miner refines Whaley *et al.*'s static approach by using techniques from software model checking to rule out infeasible paths. **Perracotta** [61] mines multiple candidate specifications that match a given FSM template. Gabel and Su [25] extend **Perracotta** using BDDs, and show both that two-state mining is NP-complete, and some specifications cannot be created by composing two-state specifications. **Strauss** [3] uses probabilistic finite state machine learning to learn a single permissive specification from traces. **GK-tail** is a technique for learning a finite state machine with edge constraints, called extended finite state machine (EFSM) specifications [42]. EFSMs describe legal sequences of program events subject to invariants on data values, such as might be learned by Daikon [21]. Lo *et al.* use learned temporal properties, such as those mined in this article, to *steer* the learning of finite state machine behavior models [41]. Shoham *et al.* [51] mine by using abstract interpretation, where the abstract values are specifications.

Unlike the static miner in Whaley *et al.*, **JIST**, **Strauss** and Shoham *et al.*, we do not require that the user provide important parts of the specification, such as the exceptions of interest. Unlike **Strauss**, the Whaley *et al.* dynamic miner, **JIST**, **GK-tail**, Lo *et al.*, and Shoham *et al.*, we produce multiple candidate specifications rather than a single specification; complex specifications are difficult to debug and verify [4]. Unlike **Perracotta** or Gabel and Su, we cannot mine more complicated templates (e.g., FSMs with three states), though this is not intrinsic to our quality-metric-based approach. Like **ECC**, **WN**, Gabel and Su, and others, our miner is scalable. We do construct more complicated models from mined temporal properties like Lo *et al.*, however, our miner is tolerant of buggy input traces. We also evalute the learned models in terms of externally-verified property correctness (instead of whether the learned model accepts all input traces, a common alternative definition of recall). Notably, we evaluate on precision, which we feel is important to the eventual adoption of automatic mining techniques in industrial practice.

The primary difference between our miner and previous static temporal property miners is that we use code quality metrics to weight input traces with a goal of low false positive rates. To our knowledge, no published miner that produces multiple two-state candidates has a false positive rate under 89%. We present two mining prototypes that identify potentially useful specifications (in terms of the number of identified potential violations): a normal miner with a rate of 63%, and a precise miner with a rate of 3%.

6.2 Previous Work in Software Quality Metrics

A full survey of software quality metrics is outside the scope of this article; instead, we highlight several notable approaches. Halstead *et al.* proposed *Software Science* [29] (which did not prove accurate in practice [30]), to provide easily measurable, universal source code attributes. Function Point Analysis (FPA) [1] estimates value delivered to a customer, which can help approximate, for example, an application's budget, the productivity of a software team, the software size or complexity, or

the amount of testing necessary. Cyclomatic complexity estimates the amount of decision logic in a piece of software, and remains in industrial use to measure code quality and impose limits on complexity [48]. Chidamber and Kemerer proposed and evaluated six metrics (referred to as the CK metrics in this article) to describe the complexity of an object oriented design [12]; these metrics appear to correlate with software quality, defined as "absence of defects." Several researchers have explored this correlation [7], [52], [54], and others have used the object-oriented metrics or design patterns to predict software faults [28], [55].

We go farther than these metrics by examining additional software engineering artifacts to measure quality. Unlike FPA, our work does not consider usefulness of code. Unlike Software Science, our model does not assume an *a priori* combination of features. However, we evaluate the utility of both Cyclomatic complexity and of the CK metrics in our model (see Section 5.2.1). We determined that Cyclomatic complexity is not a useful measure of code quality as applied to specification mining. This corroborates previous research [22], [50] that found that certain popular complexity metrics, Cyclomatic complexity in particular, do not correlate with fault density. The CK metrics do have some predictive strength in a linear model relating quality to specification likelihood, but we find that including them in a mining model increases false as well as true positives.

More recently, Nagappan and Ball analyzed the relationship between software dependences, code churn (roughly, the amount that code has been modified as measured by source control logs), and post-release failures in the Windows Server 2003 operating system [45]. They show that *relative* code churn, or the amount of churn in one module as compared to a dependent module, is more predictive of errors than *absolute* churn (which we use here). This suggests that more sophisticated measures of churn might be more predictive in our model. Graves *et al.* similarly attempt to predict errors in code by mining source control histories [27].

Like our work, these studies use features independent of the source code to make predictions. Unlike our work, they define quality as "absence of defects", instead of "adherence to specifications of correct behavior." This suggests that our use of detected errors as a proxy for specification utility may be valid. The previous work supports our claim that there is a relationship between code churn, complexity, and quality.

7 Conclusion

Formal specifications have a variety of applications, including testing, maintenance, optimization, refactoring, documentation, and program repair. However, such specifications are difficult for human programmers to produce and verify manually, and existing automatic specification miners that discover two-state temporal properties have prohibitively high false positive rates. An important problem with these techniques is that they treat all parts of a program as equally indicative of correct behavior. We instead measure code quality to distinguish between true and false candidate specifications. Our metrics include predicted execution frequency, code clone detection, code churn, readability and path feasibility, among others. We also evaluate well-known complexity metrics when used in specification mining.

Our approach improves the performance of existing trace-based miners by focusing on high-quality traces. Compared to previous work, we obtain equivalent results using only 45% of the input and with a slightly, but consistently, lower rate of false positives. Our technique can also be used alone: we propose two new specification miners and compare them to two previous approaches. Our basic miner learns more specifications and identifies hundreds more violations than previous miners while presenting hundreds fewer false positive candidates, with a false positive rate of 63% (versus the 89% rate of previous work). When focused on precision, our technique obtains a 3% false positive rate, an order-of-magnitude improvement on previous work, and finds specifications that locate hundreds of violations. To our knowledge, this is the first miner of two-state temporal properties to maintain a false positive rate under 89%.

A combination of independent, imperfect code quality metrics may prove useful to other automatic static analyses that look at source code to draw conclusions about code or predict faults. We believe that our technique is an important first step towards real-world utility of automated specification mining, as well as to the increased use of quality metrics in other analyses.

References

[1] A. J. Albrecht, "Measuring application development productivity," in *IBM Application Development Symposium*, 1979, pp. 83–92.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in *POPL*, 2005.

[3] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *Principles of Programming Languages*, 2002, pp. 4–16.

[4] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Programming Language Design and Implementation*, 2003, pp. 182–195.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *EuroSys*, 2006, pp. 103–122.

[6] T. Ball, "A theory of predicate-complete test coverage and generation," in *FMCO*, 2004, pp. 1–22.

[7] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[8] R. P. L. Buse and W. Weimer, "Automatic documentation inference for exceptions," in *ISSTA*, 2008, pp. 273–282.

[9] ——, "A metric for software readability," in *International Symposium on Software Testing and Analysis*, 2008, pp. 121–130.

[10] ——, "The road not taken: Estimating path execution frequency statically," in *ICSE*, 2009, pp. 144–154.

[11] H. Chen, D. Wagner, and D. Dean, "Setuid demystified," in *USENIX Security Symposium*, 2002, pp. 171–190.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from Java source code," in *ICSE*, 2000, pp. 762–765.

[14] M. Das, "Formal specifications on industrial-strength code — from myth to reality," in *Computer-Aided Verification*, 2006, p. 1.

[15] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *SIGDOC*, 2005, pp. 68–75.

[16] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," in *PLDI*, 2001, pp. 59–69.

[17] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[18] M. Di Penta and D. M. German, "Who are source code contributors and how do they change?" in *Working Conference on Reverse Engineering*. IEEE Computer Society, 2009, pp. 11–20.

[19] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI*, 2000.

[20] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as inconsistent behavior: A general approach to inferring errors in systems code," in *Symposium on Operating System Principles*, 2001, pp. 57–72.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[22] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, 2000.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Programming Language Design and Implementation*, 2002, pp. 234–245.

[24] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *IEEE Symposium on Security and Privacy*, 1996, p. 120.

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

[26] L. L. Giventer, *Statistical Analysis in Public Administration*. Jones and Bartlett, 2007.

[27] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.

[28] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[29] M. Halstead, *Elements of Software Science*. New York: Elsevier, 1977.

[30] P. G. Hamer and G. D. Frewin, "M.H. Halstead's Software Science - a critical examination," in *ICSE*, 1982, pp. 197–206.

[31] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA Companion*, 2004, pp. 132–136.

[32] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" considered harmful," in *WCRE*, 2006, pp. 19–28.

[33] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[34] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," *International Conference on Software Maintenance*, pp. 736–743, 2001.

[35] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," *IJCAI*, pp. 1137–1145, 1995.

[36] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *WCRE*. IEEE Computer Society, 2007, pp. 170–178.

[37] O. Kupferman and R. Lampert, "On the construction of finite automata for safety properties," in *ATVA*, 2006, pp. 110–124.

[38] C. Le Goues and W. Weimer, "Specification mining with few false positives." in *TACAS*, 2009, pp. 292–306.

[39] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," *SIGPLAN Not.*, vol. 40, no. 1, pp. 364–377, 2005.

[40] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *USENIX Security Symposium*, Aug. 2005, pp. 271–286.

[41] D. Lo, L. Mariani, and M. Pezzè, "Automatic Steering of Behavioral Model Inference," in *FSE*. ACM, 2009, pp. 345–354.

[42] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *ICSE*, 2008, pp. 501–510.

[43] D. Malayeri and J. Aldrich, "Practical exception specifications." in *Advanced Topics in Exception Handling Techniques*, 2006, pp. 200–220.

[44] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[45] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM*, 2007, pp. 364–373.

[46] National Institute of Standards and Technology, "The economic impacts of inadequate infrastructure for software testing," Tech. Rep. 02-3, May 2002.

[47] S. L. Pfleeger, *Software Engineering: Theory and Practice*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[48] J. C. Sanchez, L. Williams, and E. M. Maximilien, "On the Sustained Use of a Test-Driven Development Practice at IBM," in *Agile 2007*. IEEE Computer Society, August 2007, pp. 5–14.

[49] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*, 2003.

[50] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Softw. Eng. J.*, vol. 3, no. 2, pp. 30–36, 1988.

[51] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *International Symposium on Software Testing and Analysis*, 2007, pp. 174–184.

[52] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

[53] J. Sutherland, "Business objects in corporate information systems," *ACM Comput. Surv.*, vol. 27, no. 2, pp. 274–276, 1995.

[54] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *METRICS*, 1999, p. 242.

[55] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Trans. Softw. Eng.*, vol. 30, no. 12, pp. 904–917, 2004.

[56] W. Weimer, "Patches as better bug reports," in *Generative Programming and Component Engineering*, 2006, pp. 181–190.

[57] W. Weimer and N. Mishra, "Privately finding specifications," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 21–32, 2008.

[58] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes." in *OOPSLA*, 2004, pp. 419–431.

[59] ——, "Mining temporal specifications for error detection," in *TACAS*, 2005, pp. 461–476.

[60] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA*, 2002.

[61] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *International Conference on Software Engineering*, 2006, pp. 282–291.

Claire Le Goues received the BA degree in computer science from Harvard University and the MS degree from the University of Virginia, where she is currently a graduate student. Her main research interests lie in combining static and dynamic analyses to prevent, locate, and repair errors in programs.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an assistant professor at the University of Virginia. His main research interests include static and dynamic analyses to improve software quality and fix defects.