Defect Forecasting by Specification Mining Using Code Quality Metrics

K. HEMANTH KUMAR, D. MURAHARI REDDY

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 percent 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 percent of their original input. As a novel inference technique, our approach has few false positives in practice (63 percent when balancing precision and recall, 3 percent 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.

I. 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 percent of the total cost of software projects [49]. A key maintenance concern is incomplete documentation [15]: Up to 60 percent 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 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 paper 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 percent) [59], limiting their practical utility.

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.

The main contributions of this paper are: 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 percent highest quality traces, refuting previous claims that more traces are necessarily 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 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 versus 663) with a lower rate of false positives (63 percent versus 89 percent). Our second technique focuses on precision. It obtains a 3 percent false positive rate (one 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 percent. 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.

II. 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 paper. Such specifications can be represented as a finite state machine that encodes valid sequences of events.

Fig. 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 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].

### Candidate Specification

| Specification | Correctness |
|---------------|-------------|
| Workspace.getReadAccess(), Workspace.doneReading() | Valid Specification |
| ASTPTRootNode.isConstant(), ASTPTRootNode.isEvaluated() | False Positive |

![Fig. 2. Two of the 665 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.](image)

In this paper, 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 $\delta \alpha \delta \beta$, which we write $ha; bi$. 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 describe important properties such as those controlling resource allocation/deallocation or invariant restoration (examples include hopen , close i, hmalloc , free i, and hlock,unlocki). 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 specifications remains difficult [25]. Given the large number of candidate ha; bi pairs generated by even a restricted set of program traces, determining which pairs constitute valid policies is nonobvious. Most pairs, even those that frequently occur together (such as hprint,printi or, more insidiously, hhasNext,getNexti), 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 percent of the time.

However, a candidate that is adhered to in 90 percent 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 2,808 positively ranked specifications generated by ECC were real: a 99.5 percent 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
nonerror 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 percent false positive rate.

However, this rate is still too high for automatic applications or industrial practice, as candidates must still be hand validated. This paper proposes a new specification mining approach that lowers this false positive rate.

III. 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 ha; bi, while traces that only contain a violate it. Fig. 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 hprint, printi, appear near one other on the z-ranked list of 655 candidates. 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 Fig. 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 Fig. 2. Traces that adhere to the true candidate have an average predicted frequency of 39 percent; those that violate it, only 4 percent.

By contrast, the false candidate’s adhering traces are predicted to be run less frequently than its violating traces (31 percent versus 58 percent)!

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 versus 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 versus 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 Fig. 2 follows that pattern.

The true specification’s adhering traces were last changed much earlier in the development process on average (revision 15,387 out of 29,324 total revisions) than the violating traces (revision 20,884). 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 antipattern: the adhering traces were last changed more recently than the violating ones (25,189 versus 19,238).

The candidates in Fig. 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 paper formalizes this notion and presents an empirical evaluation of its validity.
IV. 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 paper 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

- 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 (LOC). Methods in a class may refer 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 refer instance variables in common. LOC 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].

4.2 Mining Algorithm Details

Our mining algorithm extends our previous 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 t ranges over source code locations. The variable l represents a set of locations.
2. A set of quality metrics M1,...,Mk. Quality metrics may map either individual locations l to measurements, with M(t) ∈ Re (e.g., code churn) or entire traces to measurements, where M(t) ∈ Re (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 = {(a, b) | a should be followed by b}. We manually evaluate candidate specification validity.

Our algorithm first statically enumerates a finite set of intraprocedural traces in P. Because any nontrivial program contains an 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 ≤ k ≤ 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 nonexceptional path represents all nonexceptional paths through that method,
Finally, we consider all possible candidate specifications. For each $a$ and $b$ in $\Sigma$, we collect a number of features. Fig. 3 shows the set of features our miner uses to evaluate a candidate specification $ha; bi$.

$$N_a = \{t \mid a \in t \land \neg \text{Error}(t)\}$$

$$N_{ab} = \{t \mid a, b \in t \land \neg \text{Error}(t)\}$$

$$E_a = \{t \mid a \in t \land \text{Error}(t)\}$$

$$E_{ab} = \{t \mid a \in b \land \text{Error}(t)\}$$

$$z = \frac{SP_{ab}}{DF_{ab}}$$

$$SP_{ab} = 1 \text{ if } a \text{ and } b \text{ are in the same package, } 0 \text{ otherwise}$$

$$DF_{ab} = 1 \text{ if every value in } b \text{ also occurs in } a, 0 \text{ otherwise}$$

$$M_{ia} = M_i(t \mid a \in t)$$

$$M_{ia} = M_i(t \mid a, b \in t)$$

Fig. 3. Features used by our miner to evaluate a candidate specification $ha; bi$. $M_i$ is a quality metric lifted to sets of traces.

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 $\Sigma$; each event corresponds to a location $l$. We write $a \in t$ if event $a$ occurs in trace $t$ and $a, b \in t$ 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 $\text{Error}(t)$.

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

Finally, we consider all possible candidate specifications. For each $a$ and $b$ in $\Sigma$, we collect a number of features. Fig. 3 shows the set of features our miner uses to evaluate a candidate specification $ha; bi$. $N_a$ denotes the number of times $a$ occurs in a non-error trace. $Na$ denotes the number of times $a$ occurs in a normal trace, with or without $b$. We similarly write $E_a$ and $E_{ab}$ for these counts in error traces. $SP_{ab}$ is 1 when $a$ and $b$ are in the same package. $DF_{ab}$ is 1 when every value in $b$ also occurs in $a$, and 0 otherwise. $M_{ia}$ is the aggregate metric $M_i$ on the set of traces that contain $a$ followed by $b$. $M_{ia}$ is the aggregate metric $M_i$ on the set of traces that contain a followed by $b$ (resp. contain $a$). As we have multiple aggregation functions and metrics, $M_ia$ actually corresponds to over a dozen individual features. When combined with the aforementioned statistical measurements and frequency counts, each pair $ha; bi$ is described by over 30 total features $fi$. 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.

V. 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:

- Our first set of experiments evaluates the predictive power and statistical independence of the code quality metrics.
- Our second experiment provides evidence that our metrics improve existing techniques for automatic specification mining.
- 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 Fig. 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], and [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.

| 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   | 80k | 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 |
| freecart 1.0.13 | 316k| Data reporting |
| ptolemy2 3.0.2  | 362k| Design modeling |

Total: 1.5M
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 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 ha; bi pair, determines whether it should be output as a candidate specification. We use linear regression to learn the coefficients ci and a cutoff, such that our miner outputs ha; bi as a candidate specification iff \( Pi \cdot cifi > \text{cutoff} \). In other words, a specification is emitted if the linear combination of its weights 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 nonparametric 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 were either adhered to or violated, and related documentation (such as the Hibernate API) to evaluate whether a candidate specification represented a true or false positive. A potential threat to the validity of the model is overfitting 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 percent 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 Section 5.4 to demonstrate that our results are not biased by overfitting.

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 ha; bi. Specifications were annotated by more than one researcher over the course of development. We also rechecked 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 reevaluating 120 randomly selected annotated specifications. This set included both true and false positive specifications. Our reevaluation identified one falsely labeled candidate, an error rate of less than 1 percent. 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 measure of its potential utility. Even if this measure of success is imprecise at best, specifications remain important components of the software engineering process, and our focus on a low false positive rate is an important first step toward industrial practicality of mining approaches. We leave further investigation of specification utility for future work.

VI. 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 was previously presented [38], [59]. This paper 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 866,000 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.

https://doi.org/10.46564/ijraet.2020.v08i07.001
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 paper) 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 incorporate the features used by the previous miners into our own.

Whaley et al. propose a static miner [60] that produces a single multistate specification for library code. The miner constructs a permissive policy that disallows ha; bi if function b raises an undesirable 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 multistate 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 paper, 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 evaluate 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 percent.

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 percent and a precise miner with a rate of 3 percent.

VII. 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 percent 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 percent (versus the 89 percent rate of previous work). When focused on precision, our technique obtains a 3 percent 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 percent. 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 toward 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,” Proc. IBM Application Development Symp., pp. 83-92, 1979.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, “Synthesis of Interface Specifications for Java Classes,” Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, 2005.

[3] G. Ammons, R. Bodik, and J.R. Larus, “Mining Specifications,” Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, pp. 4-16, 2002.

[4] G. Ammons, D. Mandelin, R. Bodik, and J.R. Larus, “Debugging Temporal Specifications with Concept Analysis,” Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 182-195, 2003.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusk, S.K. Rajamani, and A. Ustuner, “Thorough Static Analysis of Device Drivers,” EuroSys European Conf. Computer Systems, pp. 103-122, 2006. Proc. ACM SIGOPS/

[6] T. Ball, “A Theory of Predicate-Complete Test Coverage and Generation,” Proc. Int’l Symp. Formal Methods for Components and Objects, pp. 1-22, 2004.

[7] V.R. Basili, L.C. Briand, and W.L. Melo, “A Validation of Object- Oriented Design Metrics as Quality Indicators,” IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 1996.

[8] R.P.L. Buse and W. Weimer, “Automatic Documentation Inference for Exceptions,” Proc. Int’l Symp. Software Testing and Analysis, pp. 273-282, 2008.

[9] R.P.L. Buse and W. Weimer, “A Metric for Software Read- ability,” Proc. Int’l Symp. Software Testing and Analysis, pp. 121-130, 2008.

[10] R.P.L. Buse and W. Weimer, “The Road Not Taken: Estimating Path Execution Frequency Statical,” Proc. Int’l Conf. Software Eng., pp. 144-154, 2009.

[11] H. Chen, D. Wagner, and D. Dean, “Setuid Demystified,” Proc. USENIX Security Symp., pp. 171-190, 2002.

[12] S.R. Chidamber and C.F. Kemerer, “A Metrics Suite for Object Oriented Design,” IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June 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,” Proc. Int’l Conf. Software Eng., pp. 762-765, 2000.

[14] M. Das, “Formal Specifications on Industrial-Strength Code— FromMyth to Reality,” Proc. Int’l Conf. Computer-Aided Verification, p. 1, 2006.

[15] S.C.B. de Souza, N. Anquetil, and K.M. de Oliveira, “A Study of the Documentation Essential to Software Maintenance,” Proc. Ann. Int’l Conf. Design of Comm.: Documenting and Designing for Pervasive Information, pp. 68-75, 2005.

[16] R. DeLine and M. Faßndrich, “Enforcing High-Level Protocols in Low-Level Software,” Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 59-69, 2001.

[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?” Proc. Working Conf. Reverse Eng., pp. 11-20, 2009.

[19] D. Engler, B. Chefl, A. Chou, and S. Hallem, “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions,” Proc. Conf. Symp. Operating System Design and Implementation, 2000.

[20] D.R. Engler, D.Y. Chen, and A. Chou, “Bugs as Inconsistent Behavior: A General Approach to Inerring Errors in Systems Code,” Proc. Symp. Operating System Principles, pp. 57-72, 2001.

[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,” Science of Computer Programming, vol. 69, nos. 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. Software.

https://doi.org/10.46564/ijraet.2020.v08i07.001