IJREAT International Journal of Research in Engineering & Advanced Technology, Volume 1, Issue 1, March, 2013
ISSN: 2320 - 8791
www.ijreat.org

Elevating Formal Specification Extortion Using Quality Metrics to Appraise Code Excellence

S.Kishore Verma[1] , A.Senthamarai Selvan[2], S.Suresh[3]

[1,2,3] Assistant Professor, Department of Computer Science and Engineering
C.Abdul Hakeem College of Engineering, Melvisharam.

Abstract

To generate a high quality software applications, a strong emphasis on formal specification; especially during the later phases of software development is, necessary. Formal specifications plays vital temperament in program testing, optimization, refactoring, documentation, and, most importantly, debugging and repair. Nevertheless, they are difficult to write manually, and automatic mining techniques suffer from 90-99 % false positive rates. In this paper, we propose to augment a temporal-property miner by incorporating code quality metrics to address the problem. We evaluate 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. While used as a preprocessing step for an existing specification miner, our method identifies which input is most analytic 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. According to novel inference technique, our method has few false positives in practice under 89% (i.e. 63% when balancing precision and recall, 3% when focused on precision), whereas still finding useful specifications that find many bugs.

KeyWords:-*Specification mining, machine learning, software engineering, code metrics, program understanding.*

1. Introduction

Debugging, testing, maintaining, optimizing, refactoring, and documenting software, are time consuming, but remain critically important. Such maintenance is reported to consume up to 90 % of the total cost of software projects. A key maintenance concern is incomplete documentation. Faulty and pram behavior in deployed software costs up to $70 billion each year in the US [12]. Manual processes and especially automated tool support for finding and fixing errors in deployed software habitually entail formal specifications of correct program behavior (e.g., [6]); it is hard to renovate a coding error without a obvious view of what "correct" program behavior entails. Formal program specifications are hard for humans to construct, and wrong specifications are tricky for humans to debug and modify. Therefore, researchers have developed techniques to automatically infer specifications from program source code or execution traces [1], [2], [4], [6]. These techniques typically produce specifications in the form of finite state machines that depict legal sequences of program behaviors. Alas, these existing mining techniques are not enough precise in practice. A class of miners produces hefty but approximate specifications that must be corrected manually. As these hefty specifications are imprecise and tricky 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. Earlier research efforts have developed techniques for mining them automatically [4], [14]. Such techniques typically produce a hefty set of candidate specifications, often in a ranked list (e.g., [4]). A programmer must still evaluate this ranked list of candidate specifications to detach the true specifications from the false positive specifications. In this perception, a false positive is a candidate specification that does not portray required behavior: A program trace may violate such a "specification" and still be considered correct. A true specification illustrates behavior that may not be violated on any program trace or the program contains an error. Alas, techniques that produce this type of ranked list of smaller candidates suffer from prohibitively high false positives rates (90-99 %) [14], limiting their practical utility. This paper develops an automatic specification miner that balances true positives-as required behaviors-with false positives-non required behaviors.

```
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());
}
```

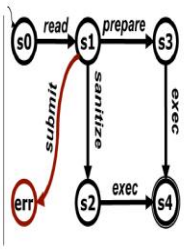

Fig. 1. Pseudocode and specification for a networked program that receives untrusted data via sockets. The `bad` method passes unsafe data to the database; `good` works correctly. Important `events` are italicized. The partial-correctness temporal safety specification shown on the right governs database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

1.1 Necessitate of Temporal Safety Specifications

A *partial-correctness temporal safety property* is a formal specification of an aspect of required or correct program behavior; they often portray 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 legitimate sequences of events. Fig. 1 shows source code and a specification relating to SQL injection attacks [8]. In this example, one possible 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 that encodes the specification that applies to its manipulation. The execution of program *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).Or else, the program violates the specification and contains an error. In this paper, we concentrate on the simplest and most common type of temporal specification: a two-state finite state machine, [7], [14]. A two-state specification state that an event *a* must always eventually be followed by event *b*. This corresponds to the regular expression $(ab)^*$ which we write $<a,b>$. We protest on this type of property because mining FSM specifications with more than two states is historically imprecise, and debugging such specifications manually is difficult.

1.2 Specification Mining Exploitation

Specification mining inquires to generate formal specifications of correct program behavior by scrutinizing actual program behavior. Program behavior is naturally portrayed in terms of sequences of function calls or other important events. Examples of program behavior may be composed statically from source code (e.g., [4]) or dynamically from instrumented executions on indicative workloads. A specification miner scrutinizes such traces and generates candidate specifications, which must be verified by a human programmer. Some miners produce a single finite automaton policy with many states [2]. Others produce many small automata of a fixed form [1], [8], [14]. As large automata are harder to verify or debug, we choose to focus on the second, as described above. Miners can also be led astray by strategy violations, as they seek to detect correct behavior from code that may be incorrect. The rest of this paper is structured as follows: We discuss related works in Section 2. Section 3 illustrates our approach to specification mining, including the quality metrics used. In Section 4, we present experiments supporting our claims and evaluating the effectiveness of our miner. We conclude in Section 5.

2. Related Works

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

2.1 Preceding Works in Specification Mining

Some of the research presented in this paper was previously presented [7], [14]. This work is intimately related to existing specification mining algorithms, of which there are a considerable number (see [14] for a survey). Our loom extends the ECC [4] and WN [14] 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 progress on the results of ECC by tapering the criteria used to pick candidate specifications (e.g., the candidate specification must render 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 sanctify both techniques in Section 3.2. 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

.

2.2 Preceding works in Software Quality Metrics

A complete review of software quality metrics is outside the scope of this paper; instead, we highlight several notable looms. Halstead proposed Software Science [5] (which did not prove accurate in practice [6]) to provide easily measurable, universal source code attributes. Function Point Analysis (FPA) 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 quantity of decision logic in a piece of software, and remains in industrial use to measure code quality and impose limits on complexity [10]. Chidamber and Kemerer proposed and evaluated six metrics (referred to as the CK metrics in this paper) to express the complexity of an object-oriented design [3]; these metrics appear to correlate with software quality, defined as "absence of defects." We go beyond than these metrics by examining additional software engineering artifacts to measure quality. Unlike FPA; our exertion 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 4.1.2). In recent times, Nagappan and Ball explored the correlation 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 [9]. This implies that more classy 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.

3. Our Loom

We present a new specification miner that works in three stages. First, it statically estimates the quality of source code fragments. Second, it elevates 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 enhancing the trace language to incorporate information from the software engineering process, we can appraise 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 probability that it is legal. Section 3.1 provides a detailed description of the set of features we have chosen to approximate the quality of code. Section 3.2 minutiae our mining algorithm.

3.1 Minutiae of Quality Metrics in our Loom

We characterize and evaluate two sets of metrics. The first set consists of seven metrics preferred to approximate code quality. Certainly, a major notion of this paper is that lightweight and imperfect metrics, when used in combination, can usefully approximate quality for the purposes of enhanced specification mining. Hence, we focus on selecting metrics that can be rapidly and automatically computed using generally available software artifacts, such as the source code or version control histories. The second set of metrics consists of beforehand proposed measures of code complexity. We use these primarily as baselines for our analysis of metric power in Section4 this evaluation may also be independently useful given their persistent use in practice [10].

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

Code churn. Earlier research has shown that frequently or recently modified code is more likely to contain errors [22].

Author rank. We infer 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.

Code clones. We infer that code that has been duplicated from a location may be more error prone because it has not necessarily been specialized to its new context (e.g., copy-paste code).

Code readability. Buse and Weimer developed a code metric trained on human perceptions of readability or understandability. 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. More readable code is less likely to contain errors.

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.

Path frequency. We speculate that common paths that are often executed by indicative workloads and test cases are more likely to be correct. We use a research tool that statically estimates the relative runtime frequency of a path through a method , normalized as a real number.

Path density. We speculate 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. Path density is expressed in whole numbers and can be normalized to the maximum number of enumerated paths (30/method, in our experiments).

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
$\mathcal{M}_{ia}$	=	$\mathcal{M}_i(\{t \mid a \in t\})$
$\mathcal{M}_{iab}$	=	$\mathcal{M}_i(\{t \mid a \ldots b \in t\})$

Fig2 Features used by our miner to evaluate a candidate specification<*a*,*b*>.M_i is quality metric lifted to sets of traces

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

Cyclomatic complexity. McCabe defined cyclomatic complexity 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.

CK metrics. Chidamber and Kemerer proposed a suite of hypothetically grounded metrics to approximate the complexity of an object-oriented design [3]. The following six metrics apply to a particular class (i.e., a set of methods and instance variables):1.Weighted methods per class (WMC).2. Depth of inheritance tree (DIT).3. Number of children (NOC). 4.Coupling between objects (CBO).5.

Response for a class (RFC). 6.Lack of cohesion in methods (LOCM).

3.2 Minutiae of our Mining Algorithm

Our mining algorithm extends our preceding WN miner [7],[14], notably by including quality metrics from Section 3.1. Our miner takes as input:

1. The program source code P. The variable *l* varies 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 either individual locations *l* to measurements, with $M_i(l) \in \mathbb{R}$ (e.g., code churn) or intact traces to measurements, where $M_i(l) \in \mathbb{R}$ (e.g., path feasibility).
3. A set of important eventsΣ, generally taken to be all of the function calls in P. We use the variables *a*, *b*, etc., to range over $\sum$.

Our miner generates as output a set of candidate specifications C = {<*a*, *b*>| *a* should be followed by b}. We manually evaluate candidate specification legality. Our algorithm first statically enumerates a finite set of intra procedural 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 $10 \leq k \leq 30$. 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, 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. 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 $\sum$; 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 *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: $\wp(\mathbb{R}) \rightarrow \mathbb{R}$. 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: $M^A(t) = A(\{M(l) | l \in t\})$ ⟶ 1
(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*).*M* denotes the metric lifted again to work on sets of traces(T)= $A(\{M^A(t) | t \in T\})$. Finally, we consider all possible candidate specifications. For each *a* and *b* in $\sum$, we collect a number of features. Fig. 2 shows the set of features our miner uses to evaluate a candidate specification <*a*, *b*>. 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} =1when dataflow relates *a* and **b**: when every value and receiver object expression in b also occurs in a [14, 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 M_{iab} (resp.$M_{i\ a}$) 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, M_{ia} actually corresponds to over a dozen individual features.

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

Tab.1. Analysis of variance of features in our model. The bottom four features are present in the WN miner [31].

When combined with the aforementioned statistical measurements and frequency counts, each pair <a, b> 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 4, is

required to learn an appropriate classifier relating features to specification likelihood.

4. Experiments

In this section, we empirically evaluate our approach. We begin by considering the constructed linear 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.

4.1 Prognostic Control 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.

marks arouBenchnd Quality Metrics

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 Tab.1. All of the quality metrics defined in Section 3.1, 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 [7] that it has little predictive power. These experiments involve a much larger benchmark set (1.5 M versus 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 percent 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 [11]. The six CK metrics, however, vary in predictive power, though all have a significant main effect. With the exception of response for a class, which is meant to approximate the interconnectedness of the design, the CK metrics are less predictive than the other proposed quality metrics. These 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 [14], we 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 Fig. 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.

4.2 Rendering of Quality 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 [14] 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 3.1, with coefficients based on their relative predictive power for specification mining (the F column in Tab.1); 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.

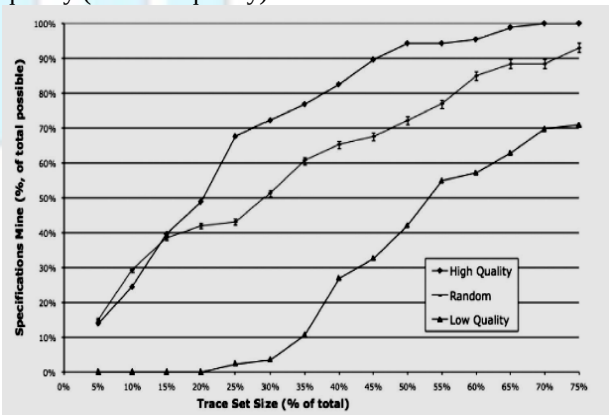

Fig. 3. Performance of the WN specification miner on subsets of the total trace set. The y-axis shows the percentage of the possible true

specifications mined. The false positive rate on high quality (85 percent) is lower than on random (89 percent).

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 percent highest quality traces: 55 percent 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 [13], reducing the costs of trace gathering by factor of 2 is significant. As a point of comparison, when a random 55 percent of the traces are discarded, we find only 58 true specifications in total (67 percent of the total possible set), with a 3 percent 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 percent of the traces (the "High-Quality" traces), the bottom N percent of the traces (the "Low-Quality" traces), and a random N percent 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. Fig. 3 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.

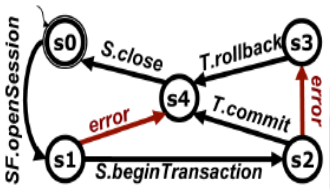

Fig. 4. A finite state machine describing the Hibernate Session API

To highlight one point on the graph: On 25 percent of the input, the high-quality traces yield 65 percent of the total possible mined specifications. By contrast, the random traces yield less than half, at 43 percent, and the low-quality traces, only 2 percent (only two true specifications!). By the time the top-quality traces have yielded all possible true specifications, the random traces have found 88 percent, and the low-quality traces, 63 percent.

4.3 Specification Mining on Quality-Based

A leave-one-out analysis shows that including the CK metrics in the model raises both the true and false positive rates. 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. 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 [14] and ECC [4] mining techniques. The normal miner finds useful specifications with a low false positive rate. It improves on the false positive rate of WN by 26 percent, while still finding 72 percent of the same specifications. It finds four 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 seven per valid specification. The precise miner produces only one false positive, on Hibernate: <S.beginTransaction, T.commit>. Fig. 4 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 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. 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 percent false positive rate approaches levels required for automatic use. Despite the one false positive and the fact that it finds 34 percent as many specifications as WN, the precise miner still finds 53 percent 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.

5. Conclusion

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 loom 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 method can also be used alone: To our acquaintance, this is the first miner of two-state temporal properties to maintain a false positive rate under 89 percent. We believe that our method 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] Claire Le Goues and Westley Weimer "Measuring Code Quality to Improve Specification Mining" IEEE Transactions on Software Engineering, Vol. 38, No. 1, January/FEBRUARY 2012
[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," Proc. ACM SIGPLANSIGACT Symp. Principles of Programming Languages, 2005.
[3] 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.
[4] D.R. Engler, D.Y. Chen, and A. Chou, "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," Proc. Symp. Operating System Principles, pp. 57-72, 2001
[5] M. Halstead, Elements of Software Science. Elsevier, 1977.
[6] P.G. Hamer and G.D. Frewin, "M.H. Halstead's Software Science -A Critical Examination," Proc. Int'l Conf. Software Eng., pp. 197-206, 1982.
[7] C. Le Goues and W. Weimer, "Specification Mining with Few False Positives," Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems, pp. 292-306, 2009.
[8] V.B. Livshits and M.S. Lam, "Finding Security Errors in Java Programs with Static Analysis," Proc. USENIX Security Symp.,pp. 271-286, Aug. 2005.
[9] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," Proc. Int'l Symp. Empirical Software Eng. and Measurement,pp. 364-373, 2007.
[10] J.C. Sanchez, L. Williams, and E.M. Maximilien, "On the Sustained Use of a Test-Driven Development Practice at IBM," Proc. AGILE,pp. 5-14, Aug. 2007.
[11] M. Shepperd, "A Critique of Cyclomatic Complexity as a Software Metric," Software Eng. J., vol. 3, no. 2, pp. 30-36, 1988.
[12] J. Sutherland, "Business Objects in Corporate Information Systems," ACM Computing Surveys, vol. 27, no. 2, pp. 274-276,1995.
[13] W. Weimer and N. Mishra, "Privately Finding Specifications,"IEEE Trans. Software Eng., vol. 34, no. 1, pp. 21-32, Jan./Feb. 2008.
[14] W. Weimer and G.C. Necula, "Mining Temporal Specifications for Error Detection," Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems, pp. 461-476, 2005.