Specification Mining With Few False Positives

Claire Le Goues and Westley Weimer[⋆]
{legoues,weimer}@virginia.edu
University of Virginia

Abstract. Formal specifications can help with program testing, optimization, refactoring, documentation, and, most importantly, debugging and repair. Unfortunately, formal specifications are difficult to write manually, while techniques that infer specifications automatically suffer from 90–99% false positive rates. Consequently, neither option is currently practical for most software development projects.

We present a novel technique that automatically infers partial correctness specifications with a very low false positive rate. We claim that existing specification miners yield false positives because they assign equal weight to all aspects of program behavior. By using additional information from the software engineering process, we are able to dramatically reduce this rate. For example, we grant less credence to duplicate code, infrequently-tested code, and code that exhibits high turnover in the version control system.

We evaluate our technique in two ways: as a preprocessing step for an existing specification miner and as part of novel specification inference algorithms. Our technique identifies which input is most indicative of program behavior, which allows off-the-shelf techniques to learn the same number of specifications using only 60% of their original input. Our inference approach has few false positives in practice, while still finding useful specifications on over 800,000 lines of code. When minimizing false alarms, we obtain a 5% false positive rate, an order-of-magnitude improvement over previous work. When used to find bugs, our mined specifications locate over 250 policy violations. To the best of our knowledge, this is the first specification miner with such a low false positive rate, and thus a low associated burden of manual inspection.

1 Introduction

Debugging, testing, maintaining, optimizing, refactoring, and documenting software are costly and time-consuming processes, yet they remain critically important: deployed programs with incorrect behavior cost billions of dollars and multiple lives each year [28]. Modifying existing code, correcting defects, and otherwise evolving software are major parts of maintenance [31], which is reported to consume up to 90% of the total cost of software projects [32]. Incomplete documentation is a key maintenance difficulty [12]: up to 60% of maintenance time is spent studying existing software (e.g., [29, p.475], [30, p.35]). Understanding

[⋆] This research was supported in part by National Science Foundation Grants CNS 0627523 and CNS 0716478, Air Force Office of Scientific Research grant FA9550-07-1-0532, and NASA grant NAS1-02117, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect their positions or policies.

correct software behavior is central to maintaining, changing, and correcting code. Human processes and especially tool support for these activities depend on formal specifications of proper program behavior (e.g., [26]). Unfortunately, while low-level program annotations are becoming more and more prevalent [11], formal specifications remain rare.

Formal specifications are difficult for humans to construct [9], and incorrect specifications are difficult for humans to debug and modify [3]. Specification mining projects attempt to address these problems by inferring specifications from program source code or execution traces [1, 2, 15, 17, 33, 37, 38]. Unfortunately, existing techniques typically produce imprecise specifications and suffer from false positive rates of 90–99% [36] — that is, a very large proportion of candidate specifications produced by these techniques are not true program specifications. Some miners require that every inferred policy be corrected manually [3].

Specification mining can be compared to learning the rules of English grammar by reading essays written by high school students; we propose to focus on the essays of passing students and be skeptical of the essays of failing students. We claim that existing miners have high false positive rates in large part because they treat all code equally, even though not all code is created equal. For example, consider an execution trace through a recently modified, rarely-executed piece of code that was copied-and-pasted by an inexperienced developer. We argue that such a trace is a poor guide to correct behavior when compared with a well-tested, infrequently-changed, and commonly-executed trace.

The problem of mining temporal safety policies is undecidable in general [2], as it is impossible to learn regular languages in the limit [18, Theorem 1.8] based on finitely many examples. Existing miners thus use heuristics to decide which specifications are likely true. Our algorithm is no different in that regard; we infer temporal safety properties of the form "b must follow a," using heuristics based on information gleaned from the software engineering process.

We propose a new automatic specification miner that uses artifacts from software engineering processes to capture the trustworthiness of its input traces. The main contributions of this paper are:

– A set of source-level features related to software engineering processes that capture the trustworthiness of code for specification mining. We analyze the relative predictive power of each of these features.
– Empirical evidence that our notions of trustworthy code serve as a basis for evaluating the trustworthiness of traces. We provide a characterization for such traces and show that off-the-shelf specification miners can learn just as many specifications using only 60% of traces.
– A novel automatic mining technique that uses our trust-capturing features to learn temporal safety specifications with few false positives in practice. We evaluate it on over 800,000 lines of code and explicitly compare it to two previous approaches. Our basic mining technique learns specifications that locate more safety-policy violations than previous miners (740 vs. 426) while presenting far fewer false positive specifications (107 vs. 567). When focused on precision, our technique obtains a low 5% false positive rate, an order-of-

```
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 for an example internet service. The `bad` method passes untrusted data to the database; `good` works correctly. Important *events* are italicized.

magnitude improvement on previous work, while still finding specifications that locate 265 violations. To our knowledge, this is the first specification miner that produces multiple candidate specifications and has a false positive rate under 90%.

The rest of this paper is organized as follows. In Section 2 we describe temporal safety specifications and highlight uses. Section 3 gives a brief overview of specification mining. Section 4 describes our approach to specification mining, including the code trustworthiness metrics used (Section 4.1). In Section 5 we present experiments supporting our claims and evaluating the effectiveness of our miner. We discuss related work in Section 6 and conclude in Section 7.

2 Temporal Safety Specifications

A *partial-correctness temporal safety property* is a formal specification of an aspect of correct program behavior [23], typically describing how to manipulate certain important resources and interfaces. Specifications take the form of finite-state machines that encode valid sequences of *events* relating to those resources that occur during the program's exeuction. For example, one event may represent reading untrusted data over the network, another may represent sanitizing it, and a third may represent a database query. Figure 2 shows such a specification for SQL injection attacks [25], based on the code in Figure 1.

Typically, each important resource is tracked separately [13]. Each finite state machine starts in its start state. A program *conforms* to a specification if and only if it terminates with the corresponding state machine in an accepting state. Otherwise, the program *violates* the specification. Such specifications can describe properties such as locking [10], resource leaks [36], security [25], high-level invariants [16] and memory safety [19], and more specialized properties such as the correct handling of `setuid` [9] or asynchronous I/O request packets [5]. These partial correctness specifications are distinct from and complementary to full formal behavior specifications.

These types of specifications can be used by almost any existing defect-finding tool (e.g., [5, 10, 11, 16]); indeed, all bug-finders require implicit or explicit specifications. Formal specifications can also help with program testing [4], optimization [24], refactoring [21], documentation [6], and repair [34]. Formal specifications are rare in practice, but not due to a lack of possible uses.

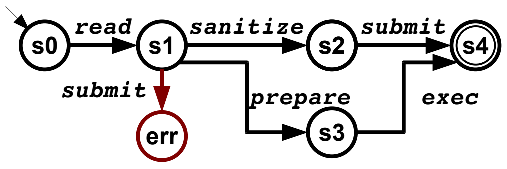

Fig. 2. Example specification for Figure 1.

3 Specification Mining

The goal of *specification mining* is to construct a formal specification using examples of program behavior [2]. Traces of program behavior can be collected from the source code (e.g., [15]) or from instrumented executions on indicative workloads (e.g., [37]). These traces usually take the form of a sequence of function calls. A specification miner examines such traces and produces one or more *candidate specifications*, which must then be verified by a human.

Existing specification miners fall into two categories. Some produce a single finite automaton policy with many states [1, 2, 37], while others produce many small automata [15, 17, 36, 38], typically of a fixed form. We focus on the latter, because large automata are much more difficult to verify or debug [3].

The simplest and most common type of temporal specification is a two-state finite state machine [15, 36]. Such two-state specifications require that event a must always be followed by event b, correspond to the regular expression $(ab)^*$, and are written $\langle a,b \rangle$. Examples include ⟨`open`,`close`⟩, ⟨`malloc`,`free`⟩, and ⟨`lock`,`unlock`⟩. Such specifications often describe resource allocation or the correct restoration of invariants, and are prevalent in practice. Even when attention is restricted to two-state specifications, mining remains difficult [17].

How a specification miner should decide what event pairs constitute a valid policy is non-obvious, especially in the face of red herrings such as ⟨`print`,`print`⟩, or even policy violations. Engler *et al.* note that programmer errors can be inferred by assuming the programmer is *usually* correct [15]. That is, common behavior implies correct behavior. Engler *et al.*'s `ECC` miner counts the number of times a and b appear together in order and the number of times that event a appears without event b, and uses the z-statistic to rank the likelihood that the correlation is deliberate on the part of the programmer. Their miner presents a ranked list of candidate specifications to the programmer for inspection. Without human guidance (e.g., without lists of important functions to focus on [15]), 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 [36].

In previous work, we observed that programmers often make mistakes in rarely-tested error handling code [36]. Tracking a single bit of information per trace — whether that trace corresponded to a program error or not — improved the mining accuracy dramatically, by an order of magnitude. We also included a software engineering consideration, restricting attention to specifications in which the events a and b came from the same package or library. We assumed that independent libraries, potentially written by separate developers, are unlikely to depend on each other for correctness at the API level. These insights reduced the number of candidates presented to the programmer by a large factor: on the same million lines of Java code, the `WN` miner generated only 649 candidate specifications, of which 69 were real, for an 89% false positive rate. However, this rate is still too high to be considered automatic; before being used, the candidate specifications must still be hand-validated.

4 Our Approach: Code Trustworthiness

We call code *trustworthy* if it is unlikely to exhibit API policy violations. Previous approaches have implicitly assumed that all execution traces are equally indicative of correct program behavior, that is, that all traces should be trusted equally. In this paper, we demonstrate that this assumption is incorrect. We present a specification miner that works in three stages:

1. Statically estimate the trustworthiness of each code fragment.
2. Lift that judgment to traces by considering the code visited along a trace.
3. Weight the contribution of each trace by its trustworthiness when counting event frequencies for specification mining.

We hypothesize that code is most trustworthy when it has been written by experienced programmers who are familiar with the project at hand, when it has been well-tested, and when it has been mindfully written (e.g., rather than copied-and-pasted). Previous work has found more errors in recently-changed code [27], unreadable code [7] and rarely-tested code [36]. Such information can be collected from a program's source code and version control history. Section 4.1 describes the set of features we have chosen to approximate the "trustworthiness" of code. Section 4.2 describes our mining algorithm in more detail.

4.1 Trustworthiness Metrics

Our goal is to automatically distinguish between code that is likely to adhere to program specifications and code that is not. Our specification miner thus uses a number of metrics to approximate the trustworthiness of code. We only consider metrics that can be computed automatically using commonly-available software engineering artifacts, such as the source code itself or version control information. In the interest of automation, we do not consider features that require manual annotation or human guidance. We use the following metrics:

Code Churn. Previous work has shown that frequently modified code is less likely to be correct [27]; changing the code to fix one defect often introduces another. We hypothesize that so-called churned code is less likely to adhere to specifications. Using version control information, we measure the time between the current revision and the last revision for each line of code in wall clock hours. Similarly, we measure the total number of revisions to each line.

Author Rank. We hypothesize that some developers have a better understanding of the implicit specifications for a project than others. A senior developer who has performed many edits may remember more of the program invariants than a developer recently added to the group. Source control repositories track the author of each change. The *rank* of an author is the proportion of all changes committed to the repository that were committed by that author. We measure the author rank of the last author to touch each line of code.

Copy-Paste Development. We hypothesize that duplicated code is more error-prone because it has not been specialized to its new context and because patches to the original may not have propagated to the duplicate. We further hypothesize that duplicated code does not represent an independent correctness argument on the part of the developer; if `printf` follows `iter` in 10 duplicated code fragments, it is not 10 times as likely that ⟨`iter`,`printf`⟩ is a real specification. We measure repetition using the open-source `PMD` toolkit's copy-paste detector, which is based on the Karp-Rabin string matching algorithm [20].

Code Readability. In previous work, we showed that more readable code is less likely to contain errors [7]. We hypothesize that more readable code is thus also more likely to adhere to specifications. We measure code readability using our software readability metric, which is based on textual source code features and agrees with human annotators [7].

Path Feasibility. Infeasible paths are an artifact of the static enumeration process; we claim that they do not encode programmer intentions. Previous work has argued that it is always helpful to have more traces, even incorrect ones [35]; our experiments suggest that quality is more important than quantity (see Section 5.2). Merely excluding infeasible paths confers some benefit. However, we further hypothesize that infeasible paths suggest pairs that are *not* specifications. If the programmer has made it impossible for b to follow a along a path, $\langle a,b \rangle$ is unlikely to be required. We measure the feasibility of a path using symbolic execution; a path is infeasible if an external theorem prover (in our case, Simplify) reports that its symbolic branch guards are inconsistent.

Path Frequency. We theorize that paths that are frequently executed by indicative workloads and testcases are more likely to contain correct behavior. We use a research tool that can statically estimate the relative runtime frequency of a given path through a program [8] to measure path frequency. We measure relative runtime frequency with respect to the enclosing method.

Path Density. We hypothesize that a method with few static paths is likely to exhibit correct behavior and that a method with many paths is likely to exhibit incorrect behavior along at least one of them. We define "path density" as the number of traces it is possible to enumerate in each method and in each class. A low path density for traces containing the paired events ab and a high path density for traces that contain only a both make $\langle a,b \rangle$ a likely specification.

4.2 Mining Algorithm Details

Our mining algorithm extends our previous `WN` miner [36]. Formally, our miner takes as input:

1. The program source code P. The variable ℓ ranges over source code locations.
2. A set of trustworthiness metrics $M_1 \dots M_q$, with $M_i(\ell) \in \mathbb{R}$.
3. A set of important events Σ, typically 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 determine the validity of a particular candidate specification by manual inspection; we present experimental results in Section 5.

Our algorithm first statically enumerates traces through P. Since there are an infinite number of traces, we must choose a finite enumeration strategy. We consider each method m in P in turn. Using a breadth-first traversal, we enumerate the first k paths through m, assuming that branches can either be taken or not and that an invoked method can either terminate normally or raise any of its declared exceptions [36]. We pass through loops no more than once. This produces a set of traces T, where each trace t is a sequence of program locations ℓ. We write $a \in t$ if the event a occurs in trace t and $a \dots b \in t$ if the event a occurs and is followed by the event b in that trace. We also note whether or not a trace involves exceptional control flow; we write $\mathit{Error}(t)$ for this judgment [36].

Next, where necessary, our miner lifts trustworthiness metrics from locations to traces. Our lifting is parametric with respect to an aggregation function $A : \mathcal{P}(\mathbb{R}) \rightarrow \mathbb{R}$. We use the functions max, min, span and $\mathsf{average}$ in practice. We write M^A for a trustworthiness metric M lifted to work on traces: $M^A(t) = A(\{M(\ell) \mid \ell \in t\})$. We write $\mathcal{M}$ for 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*. We write N_{ab} for the number of times a is followed by b in a normal (non-error) trace. We write N_a for the number of times a occurs in a normal trace, with or without b. We similarly write E_{ab} and E_a for counts in error traces. We write $SP_{ab} = 1$ when a and b are in the same package (i.e., defined in the same library). We write $DF_{ab} = 1$ when a and b are connected by dataflow information: when every value and receiver object expression in b also occurs in a [36, Section 3.1].

In previous work we showed that both the `ECC` and `WN` miners can be expressed using this set of features [35]. The `ECC` miner returns $\langle a,b \rangle$ when a is followed by b in some traces but not in others: $N_a - N_{ab} + E_a - E_{ab} > 0$ and $N_{ab} + E_{ab} > 0$ and $DF_{ab} = 1$. The `WN` miner returns $\langle a,b \rangle$ when $E_{ab} > 0$ and $E_a - E_{ab} > 0$ and $DF_{ab} = SP_{ab} = 1$. Both of these miners encode arbitrary heuristic choices about which features are considered, the relative importance of various features, and which features must have high values.

We extend the set of features by adding the aggregate trustworthiness for each lifted metric M^A. We write $\mathcal{M}_{iab}$ (resp. $\mathcal{M}_{ia}$) for the aggregate metric values on the set of traces that contain a followed by b (resp. contain a). Figure 3 lists the

$$
\begin{array}{ll}
N_a & = |\{t \mid a \in t \ \wedge \ \neg \mathit{Error}(t)\}| \\
N_{ab} & = |\{t \mid a \ldots b \in t \ \wedge \ \neg \mathit{Error}(t)\}| \\
E_a & = |\{t \mid a \in t \ \wedge \ \mathit{Error}(t)\}| \\
E_{ab} & = |\{t \mid a \ldots b \in t \ \wedge \ \mathit{Error}(t)\}| \\
SP_{ab} & = 1 \text{ if } a \text{ and } b \text{ are in the same package, 0 otherwise} \\
DF_{ab} & = 1 \text{ if every value in } b \text{ also occurs in } a\text{, 0 otherwise} \\
\mathcal{M}_{i\,a} & = \mathcal{M}_i(\{t \mid a \in t\}) \qquad \text{where } \mathcal{M}_i \text{ is a lifted trustworthiness metric} \\
\mathcal{M}_{i\,ab} & = \mathcal{M}_i(\{t \mid a \ldots b \in t\}) \ \ \text{where } \mathcal{M}_i \text{ is a lifted trustworthiness metric}
\end{array}
$$

Fig. 3. Features used by our miner to evaluate a candidate specification $\langle a,b \rangle$.

set of features considered by our miner when evaluating a candidate specification $\langle a,b \rangle$. Since we have multiple aggregation functions and metrics (see Section 4.1), $\mathcal{M}_{i\,a}$ actually corresponds to over a dozen individual features.

We also include a number of statistical features, fractions and percentages related to the main frequency counts $N_a \ldots E_{ab}$, such as the z-statistic used by ECC to rank candidate specifications; we thus use over 30 total features f_i for each pair $\langle a,b \rangle$. Rather than asserting an *a priori* relationship between these features that candidate specifications must adhere to, we use linear regression to learn a set of coefficients c_i and a cutoff *cutoff*, such that our miner outputs $\langle a,b \rangle$ as a candidate specification iff $\sum_i \ c_i f_i < \mathit{cutoff}$. This involves a training stage to determine both the coefficients and the cutoff, described in detail in Section 5.

5 Experiments

We evaluate our miner on several open-source Java benchmarks, shown in Figure 4. We selected these programs to allow a direct comparison to previous work [17, 35, 36, 38]. We restricted attention to programs with CVS or SVN source-control repositories. For each program, we statically enumerated traces (up to a limit of 20 per method) and gathered the required information for the trustworthiness metrics described in Section 4.1. We do not need source code *implementing* a particular interface; instead, we generate traces from the client code that *uses* that interface (as in [2, 14, 17, 38]). One expensive operation was computing path feasibility, which required multiple calls to Simplify, an external theorem prover. On a 3 GHz Intel

Program Version	LOC	Description
`hibernate2 2.0b4`	57k	Object persistence
`axion 1.0m2`	65k	Database
`hsqldb 1.7.1`	71k	Database
`cayenne 1.0b4`	86k	Object persistence
`jboss 3.0.6`	107k	Middleware
`mckoi-sql 1.0.2`	118k	Database
`ptolemy2 3.0.2`	362k	Design modeling
Total	866k	

Fig. 4. Benchmarks used in our experiments.

Xeon machine, computing it on the `mckoi-sql` (our second-largest) benchmark took 25 seconds. Enumerating all static traces for `mckoi-sql`, with a maximum of 20 traces per method, took 912 seconds in total; this happens once per program. Collecting the other metrics for `hsqldb` 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 Trustworthiness Metrics: Learning Cutoffs and Coefficients

First, we learn the coefficients and cutoff that determine which candidate specifications to output, and thus the relative importance of our trustworthiness metrics. We use *recall* and *precision* to evaluate potential coefficients. Recall is the number of real specifications returned out of all possible real specifications, or the probability that a real specification is returned by the algorithm. Precision is the fraction of candidate specifications that are not false positives. A high recall indicates that the miner is doing useful work (i.e., returning real specifications), but without a corresponding high precision, those real specifications will be drowned in a sea of false positives. We claim that current false positive rates are too high for existing techniques to be of practical use.

We use linear regression to find the coefficients for our miner. Linear regression requires annotated answers (i.e., a set of known-valid and known-invalid specifications). We use the valid and invalid specifications mined and described in previous work [35, 36] as a training set. Given the set of linear regression coefficients, we perform a linear search of possible cutoffs and choose the one that maximizes an objective function. That objective function can be the harmonic mean of precision and recall (our *normal miner*) or just precision (which yields a *precise miner* with very few false positives).

Our first experiment **evaluates the relative importance of our trustworthiness metrics**. A per-feature analysis of variance, over all of the training data, is shown in Figure 5. The F column denotes the F-ratio, or the square of the variance explained by the feature over the variance not explained. It is near 1 if the feature does not affect the model. The p column shows the probability that the feature does not affect the miner.

All features except Author Rank had a significant main effect ($p \leq 0.05$). The Frequency metric, encoding our static prediction of how often the path would be executed at run-time [8], was our most important feature: commonly-run (and thus well-tested) paths do not demonstrate erroneous behavior. All of our new trustworthiness features were more important to mining than feasibility, which is, to our knowledge, the only one that had been previously investigated [1]. We were surprised to discover that our formulation of author rank had no effect on the model: whether the last person to touch a line of code was a frequent contributor to the project is not related to whether traces adhered to specifications.

5.2 Trust Matters for Trace Quality

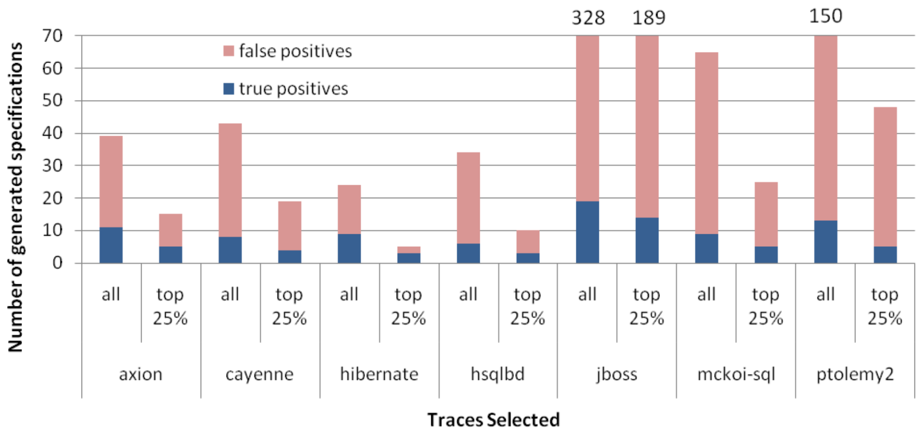

Fig. 6. The false positive rate of the off-the-shelf WN miner on various input sets. The total height of the bar represents the number of candidate specifications returned to the user for inspection.

In our second experiment, we **demonstrate that our trustworthiness metrics improve existing techniques for automatic specification mining**. For each of our benchmarks, we run the unmodified WN miner [36] on multiple input trace sets. For generality, we restrict attention to feasible traces, since miners such as JIST already disregard infeasible paths [1].

We compare WN's performance on a baseline set of feasible static traces to its performance on trustworthy subsets of those traces. For this experiment we define the trustworthiness of a trace to be 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).

Metric	F	p
Frequency	32.3	0.0000
Copy-Paste	12.4	0.0004
Code Churn	10.2	0.0014
Density	10.4	0.0013
Readability	9.4	0.0021
Feasibility	4.1	0.0423
Author Rank	1.0	0.3284

Fig. 5. Analysis of variance.

On the entire baseline set, WN miner produces 75 real specifications. Averaged over all the benchmarks, WN finds the same specifications using only the top 60% most trustworthy traces: 40% of the traces can be dispensed with while preserving true positive counts. As a point of comparison, when a random 40% of the traces are discarded, we find only 56 true specifications in total, with a 4% higher rate of false positives.

We also explore the impact of trustworthy traces on false positive rates by passing various proportions of trustworthy input to the `WN` miner. Figure 6 shows the results when only the 25% most trustworthy traces are used. On the baseline set, `WN` has 683 false positives: a false positive rate of 90%. When restricted to the 25% most trustworthy traces, `WN` produces 39 real specifications and 306 false positives: a false positive rate of 89%. Notably, we find over one-half of the specifications with only one-fourth of the input, without sacrificing the false positive rate. Beyond halving the raw false positive rate, and thus human effort required to validate the results, this is useful if the smaller output set contains particularly helpful specifications, which we investigate next. As a lower bound, only two true specifications can be mined from the 25% *least* trustworthy traces.

Any static specification mining technique involves a particular trace enumeration strategy; trace generation is often a bottleneck. Rather than enumerating a certain number of traces per method, we claim that trustworthy traces should be pursued and untrustworthy traces should be skipped. These results also have implications for multi-party techniques to mine specifications collaboratively by sharing trace information [35]. Focus should be placed on sharing information from trustworthy traces. Our trustworthiness metrics could generally be used as a preprocessing step to improve any static trace-based specification miner (e.g., [15, 17, 36, 38]). However, they can be even more useful when directly incorporated into a mining algorithm.

5.3 Trustworthy Specification Mining

For our main experiment, we **measure the efficacy of our new specification miner** on all input trace sets. We must first verify that our miner is not biased with respect to our training data. A potential threat to the validity of our results is over-fitting by testing and training on the same data. We use 10-fold cross validation to mitigate this threat [22]. 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. If the average results of cross-validation (over many random partitionings) are different from the original results, it may indicate bias. For our experiment, the difference was less than 0.01%, indicating little or no bias.

Figure 7 shows the results of applying our new specification miner from Section 4 to the benchmarks in Figure 4. For each benchmark, we report the number of candidate specifications returned, broken down into valid specifications and false positives (determined by manual verification of the results). We also report the number of distinct methods that violated the valid mined specifications (i.e., the number of policy violations found by using that specification with a bug-finding tool). Each method is counted only once per specification, even if multiple paths through that method violate it. We reprint published results for the `WN` [36] and `ECC` [15] miners for comparison. Recall that our normal miner minimizes both false positives and false negatives, while our precise miner only minimizes false positives (see Section 5.1).

Program	Normal Miner			Precise Miner			`WN`			`ECC`		
	Specs	False	Bugs	Specs	False	Bugs	Specs	False	Bugs	Specs	False	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
`jboss`	14	75 = 84%	255	2	0 = 0%	12	11	103 = 90%	94	2	442 = 99%	4
`cayenne`	5	7 = 58%	45	3	0 = 0%	23	5	30 = 86%	18	3	308 = 99%	8
`mckoi-sql`	7	10 = 59%	20	2	0 = 0%	7	19	137 = 88%	69	2	344 = 99%	5
`ptolemy`	6	1 = 14%	44	3	0 = 0%	13	9	183 = 95%	72	3	653 = 99%	12
Total	49	107 = 69%	740	20	1 = 5%	265	68	567 = 89%	426	13	2508 = 99%	50

Fig. 7. Comparative mining results on 800kLOC. "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.

The `WN` and `ECC` 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." [38] Gabel and Su report mining over 13,000 candidates from `hibernate` [17]. Our precise miner produces six – one is a false positive, and the other five find over 150 violations.

Our normal miner finds important specifications with a low false positive rate. It improves on the false positive rate of `WN` by 20%. Moreover, the specifications that it finds generally find more violations than those found by `WN`: 740 violations, or 15 per valid specification, compared to `WN`'s 426, or 7 per valid specification. However, the end-user inspects both valid and invalid specifications. Each candidate specification from our miner helps to find 4 violations on average; for `WN`, less than 1 violation is found on average per candidate inspected.

This precise miner finds fewer valid specifications, but its 5% false positive rate approaches levels required for automatic use. It finds 30% as many specifications as `WN`, but 60% of the violations: each candidate inspected yields over 12 violations on average. Users are often unwilling to wade through voluminous tool output [15, 19]; with a 5% false positive rate, and more useful specifications than those of previous work, we claim that our precise miner might be reasonable in both interactive and automatic settings.

5.4 Threats to Validity

Although our two miners outperform existing approaches in terms of bugs found and false positives avoided, our results may not generalize to industrial practice. The benchmarks used in this project may not be representative of other projects. We chose the benchmarks to be directly comparable with previous work [17, 35, 36, 38], and note that the domains represented are more indicative of server and back-end computing than of client code. A second threat is over-fitting. We use cross-validation in Section 5.3 to demonstrate that our results are not biased by over-fitting. A third threat lies in our manual validation of the output: our human annotation process may mislabel candidate specifications. To mitigate this threat we re-checked a fraction of our judgments at random and used the source code of a and b to evaluate $\langle a,b \rangle$. A final threat lies in our use of "bugs found" as a proxy for specification utility: while our mined specifications find more policy violations, they may not be as useful for tasks such as documenting or refactoring. We leave an investigation of specification utility for future work.

6 Related Work

Our work is most closely related to existing specification mining algorithms (see [36] for a survey). The `ECC` [15] and `WN` [36] algorithms are formalized in detail in Section 4.2. The `WML_static` [37] miner examines library source code, assumes that typestate is explicitly captured by object fields and thrown exceptions, and produces a single multi-state specification. The `WML_dynamic` [37] miner examines dynamic traces and produces a permissive multi-state specification that describes all observed behavior. The `JIST` [1] miner refines the `WML_static` approach and uses techniques from software model checking to rule out infeasible paths. The `Perracotta` [38] miner mines multiple candidate specifications that match a given template (e.g., the two-state specification form used in this paper is one such template). Gabel and Su [17] extend `Perracotta` using BDDs, show that two-state mining is NP-complete, and show that some specifications cannot be mined by composing multiple two-state specifications. The `Strauss` tool [2] uses probabilistic finite state machine learning to learn a single specification from traces. Shoham *et al.* [33] mine by using abstract interpretation where the abstract values are specifications.

Unlike `WML_static`, `JIST`, `Strauss` and Shoham *et al.*, we do not require that important parts of the specification, such as the classes of interest, be given in advance by the user. Unlike `Strauss`, `WML_dynamic`, `JIST`, and Shoham *et al.*, we produce multiple candidate specifications rather than a single specification; complex specifications are difficult to debug and verify [3]. Unlike `Perracotta` or Gabel and Su, we cannot mine more complicated templates, such as three-state specifications. Like `ECC`, `WN`, and Gabel and Su, our miner is scalable. The primary difference between our miner and previous miners is that we use software engineering information, encoded as trustworthiness metrics, to weight input traces and thus obtain low false positive rates. To our knowledge, no published miner that produces multiple candidates has a false positive rate under 90%; we present one with a 5% false positive rate that still finds over 250 violations.

7 Conclusion

Formal specifications have myriad uses, from testing and optimizing, to refactoring and documenting, to debugging and repair. Formal specifications are difficult to produce manually, and existing specification miners typically have 90–99% false positive rates. We claim that not all parts of a program are equally indicative of correct behavior. We encode this intuition using trustworthiness metrics such as predicted execution frequency, measurements of copy-paste code, code churn, software readability or path feasibility. These metrics can be used to improve the performance of existing trace-based miners by focusing on trustworthy traces: equivalent results can be obtained using only 60% of the input. We also use our metrics to create a new specification miner and compare it to two previous approaches on over 800,000 lines of code. Our basic miner learns specifications that locate hundreds more bugs than previous miners while presenting hundreds fewer false positive candidates. When focused on precision, our technique obtains a low 5% false positive rate, an order-of-magnitude improvement on previous work, while still finding specifications that locate hundreds of violations. To our knowledge, among specification miners that produce multiple candidate specifications, this is the first to maintain a false positive rate under 90%. We believe it to be a first step towards utility in an automated setting.

References

1. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.
2. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
3. G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Programming Language Design and Implementation*, pages 182–195, 2003.
4. T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO*, pages 1–22, 2004.
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 *European Systems Conference*, pages 103–122, Apr. 2006.
6. R. P. L. Buse and W. Weimer. Automatic documentation inference for exceptions. In *ISSTA*, pages 273–282, 2008.
7. R. P. L. Buse and W. Weimer. A metric for software readability. In *ISSTA*, pages 121–130, 2008.
8. R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically, 2009.
9. H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.
10. 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*, pages 762–765, 2000.
11. M. Das. Formal specifications on industrial-strength code-from myth to reality. In *CAV*, page 1, 2006.
12. S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *SIGDOC*, pages 68–75, 2005.
13. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

14. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.
15. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
16. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
17. M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, pages 51–60, 2008.
18. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
19. D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, pages 132–136, 2004.
20. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
21. Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin. Automated support for program refactoring using invariants. *ICSM*, pages 736–743, 2001.
22. R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *IJCAI*, 14(2):1137–1145, 1995.
23. O. Kupferman and R. Lampert. On the construction of finite automata for safety properties. In *ATVA*, volume 4218, pages 110–124, 2006.
24. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. *SIGPLAN Not.*, 40(1):364–377, 2005.
25. V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *USENIX Security Symposium*, pages 271–286, Aug. 2005.
26. D. Malayeri and J. Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, pages 200–220, 2006.
27. N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*, pages 364–373, 2007.
28. National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, May 2002.
29. S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
30. T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.
31. C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.
32. R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. 2003.
33. S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, pages 174–184, 2007.
34. W. Weimer. Patches as better bug reports. In *GPCE*, pages 181–190, 2006.
35. W. Weimer and N. Mishra. Privately finding specifications. *IEEE Trans. Software Eng.*, 34(1):21–32, 2008.
36. W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, pages 461–476, 2005.
37. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002.
38. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, pages 282–291, 2006.