Specification Mining With Few False Positives

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Science)

by

Claire Le Goues

May 2009

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 and techniques that infer specifications automatically suffer from 90–99% false positive rates. Consequently, neither option is currently practiced 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. For example, we grant less credence to duplicate code, infrequently-tested code, and code that has been changed often or recently. By using additional information from the software engineering process, we are able to dramatically reduce this rate.

We evaluate our technique in two ways: as a preprocessing step for an existing specification miner and as part of a novel specification inference algorithm. Our technique identifies which traces are most indicative of program behavior, which allows off-the-shelf mining techniques to learn the same number of specifications using 60% of their original input. This results in many fewer false positives as compared to state of the art techniques, 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 combined with bug-finding software, 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.

Approval Sheet

This thesis is submitted in partial fulfillment of the

requirements for the degree of

Master of Science (Computer Science)

Claire Le Goues

This thesis has been read and approved by the Examining Committee:

Thesis Adviser

Committee Chair

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

May 2009

Contents

List of Figures

Chapter 1

Introduction

Writing the code is but one step in creating a software system. A software system's lifetime and budget is dominated by a number of processes that go well beyond code creation. Up to 90% of the cost of a software project is devoted to modifying existing code, detecting and correcting errors, and generally evolving a code base [52]. These activities are major parts of system maintenance [51]. The quality of a program's documentation strongly influences a programmer's ability to accurately evolve and modify existing code, because understanding correct software behavior is central to maintaining, changing, and correcting code. It is therefore unsurprising that up to 60% of maintenance time is spent studying existing software (e.g., [49, p.475], [50, p.35]). Incomplete documentation and incomplete understanding are thus key maintenance difficulties [19]. Beyond being expensive and time-consuming, these processes are critically important in the current software-deployment landscape: deployed programs with incorrect behavior cost billions of dollars each year [48]. Correctly modifying, maintaining, documenting and understanding code therefore remain important software engineering concerns.

Most of these important activities can be aided by the presence of *formal specifications* of one form or another. Formal specifications can help with program testing [7], optimization [43], refactoring [38], documentation [9], and repair [58]. In addition, there are many successful tools that can assist with bug detection and correction, such as SLAM [6], BLAST [35], MOPS [13], SPLINT [25], and ESP [18] projects. However, these tools typically depend on machine-readable specifications of program behavior (e.g., [45]).

"Specification" has different meanings to different practitioners. For example, a specification may describe legal sequences of function calls, such as is found in an API, or temporal properties to which the program must remain faithful such as might be suitable input to a model checker [6, 35, 13]. For instance, a program that `open()`s an important resource must always eventually `close()` it. On the other hand, a specification may consist of a prose document that outlines all of the correct and required behavior for a given piece of software. Alternatively, a specification may be written in a formal language such as Z or SPARK ADA [1, 12], and describe invariants that hold at various program points (such as function pre- or post- conditions or loop invariants). Such a specification can be used to prove various properties about a given program.

Very generally, a program specification is a *description of correct program behavior*. Automatic bug-finding tools currently require them, because it is difficult for a tool to tell when a program is behaving incorrectly without a sound description of what correct behavior entails. Put simply, without a precise definition of correct program behavior, these automatic and pseudo-automatic tools cannot detect or correct errors or deviations in program execution. In this thesis, we follow the terminology of Ammons *et al.* and use "specifications" to refer to formal, machine-readable, partial-correctness, temporal-safety properties of programs, and often specifically to those properties that describe the correct manipulation of an important resource or API. [5] We use “policy” and “specification” interchangeably.

Although low-level program annotations, such as those describing array bounds, are becoming more and more prevalent [17], the formal specifications we refer to are rare. Specifications are very difficult for humans to manually generate or debug and verify without tool support. Chen *et al.* studied the `setuid` API in Unix-like systems [14]. They developed a method for automatically and somewhat laboriously constructing a formal model for the API for the access control system in Unix. They discovered several key facts about describing such an API. First, if the goal is to provide a complete description of correct and incorrect behavior of a system, a human inspection of the system is largely insufficient. Second, once the formal model of the system had been completed, the authors discovered several security vulnerabilities in various Unix-like operating systems and notable omissions or inaccuracies in the documentation of the `setuid` interface. Relatedly, Ammons *et al.* studied the specifications emitted by an algorithmic tool, attempting to gauge their correctness by hand. This problem was so difficult that they developed a separate tool for the purpose of debugging specifications, claiming, “ . . . writing a correct specification is difficult, just as writing a correct program is difficult. Thus, just as we need methods for debugging programs, we need methods for debugging specifications.” [5] We can conclude that this type of formal specification is difficult for humans to construct, particularly unaided, and particularly post-facto, and that incorrect specifications are difficult for humans to debug and modify.

Specification mining projects attempt to address these problems by inferring specifications (both simple temporal properties and larger, more complete finite state machines) from program source code or execution traces [3, 4, 23, 29, 54, 62, 64]. Generally, these existing tools take program source code or traces that describe program behavior as input and output either a single, large specification that delineates acceptable program behavior or a list of smaller specifications that the miner believes are likely to apply to the program in question.

If a miner produces a list of candidate specifications, we can distinguish between *true* and *false* specifications. In this thesis, we follow the terminology of Weimer *et al.* [61] and say that a true specification describes required program behavior: if a program trace violates this specification, a human programmer or developer on the project would be correct in describing the behavior as an error. A false specification describes program behavior that is not required for correctness. That is, a program trace could violate a false specification and still be considered correct. A false candidate specification may describe legal program behavior, but does not describe *required* program behavior. For the purposes of this discussion, the distinction between true and false specifications is that the former describes *required* program behavior and the latter does not.

Modern specification mining techniques typically either produce imprecise specifications that do not describe the entirety of allowed or desired behavior [62, 5] or suffer from very high false positive rates of 90–99% [23, 61]. That is, a very large proportion of candidate specifications produced by these techniques are not true program specifications. Instead, they may describe patterns that appear frequently in the code, such as `printf` being commonly followed by `printf`, or correct but not required code usage, such as requiring that every `hasnext` call be followed by a `getnext` call. Moreover, some miners require that every inferred policy be inspected by a programmer for correctness. [5]. In any case, much of modern specification mining typically requires substantial programmer intervention, either to guide the specification inference itself [4, 3] or to separate valid learned specifications from invalid ones. Hence, current techniques are insufficiently precise for automated use or efficient industrial practice.

The task of specification mining can be compared to learning the rules of English grammar by reading essays written by high school students by miner looking at how words or punctuation are combined and extracting general rules about their usage. Similarly, specification miners examine how functions are called, and in what order, to determine general rules that apply across the program. A miner that looks at essays to determine the rules of English might take into consideration the grades the different papers received: papers with high marks are more likely to exhibit correct grammar than papers that received failing marks. As not all high school essays are of equivalent quality, not all code is equally likely to be correct. Current techniques in specification mining ignore the potential differences in code quality across a software project.

We propose to tackle the problem of false positives in specification mining by focusing on the essays of passing students and being 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 of equal quality. 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 well-tested, infrequently-changed, and commonly-executed traces.

One type of miner for temporal policies considers examples of program behavior described by sequences of legal events and attempts to extract finite state machines of varied complexity. The problem of mining temporal safety policies can be considered equivalent to learning a regular language based on finitely-many strings in the language. This problem is NP-complete in general [4], as it is impossible to learn regular languages in the limit [30, Theorem 1.8] based on finitely many examples. Consequently, existing miners use heuristics to decide what specifications are likely valid. Our algorithm is no different in that regard: we infer temporal safety properties of the form "b must follow a." Our principal insight is that our heuristics are based on information gleaned from the software engineering process.

The formal thesis statement of this work is:

> We can use measurements of the trustworthiness of source code to mine specifications with few false positives.

Essentially, we think that a major problem with current specification miners is that they do not effectively distinguish between good and bad code. We hypothesize that we can measure the "trustworthiness" of code. We further hypothesize that we can effectively use those measurements to mine useful specifications from static program traces with a lower rate of false positives than is found in other, similar miners.

We thus 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 thesis are:

- A set of source-level features related to software engineering processes that captures the trustworthiness of code for specification mining. We analyze the relative predictive power of each of these features and their relationship to one another, establishing that the features are largely independently useful metrics of code quality.

- Empirical evidence that our notions of trustworthy code serve as a reasonable basis for evaluating the trustworthiness of traces. We provide a characterization for such traces and show that off-the-shelf specification miners can learn the same specifications using only 60% of traces, as long as the traces conform to our notion of trustworthiness.

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

Chapter 2 describes temporal safety specifications and highlights their uses, and then provides a more complete overview of specification mining. Chapter 3 presents an example that motivates our insights about trustworthiness and specification mining. Chapter 4 describes our approach to specification mining, including the code trustworthiness metrics used (Section 4.1). Chapter 5 presents experiments supporting our claims and evaluating the effectiveness of our miner. We discuss related work in Chapter 6 and future work and conclusions in Chapter 7.

Chapter 2

Mining Temporal Safety Specifications

In this chapter, we present background on machine-readable two-state partial-correctness temporal safety specifications and how to mine them.

2.1 Temporal Safety Specifications

Specifications can come in many forms: English prose documents, first-order logic, or lower-level annotations such as array bounds. One particular type of specification takes the form of a machine-readable finite-state machine that encodes valid sequences of *events* relating to resources that a program manipulates during execution. Typically, an *event* is a function call that can take place as a program executes. 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 2.2 shows such an example specification for avoiding SQL injection attacks [44], based on the code in Figure 2.1.

Typically, each important resource, such as a lock, file handle, or socket, is tracked

```
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());
}
```

Figure 2.1: Pseudocode for an example internet service. The `bad` method passes untrusted data to the database; `good` works correctly. Important *events* are italicized.

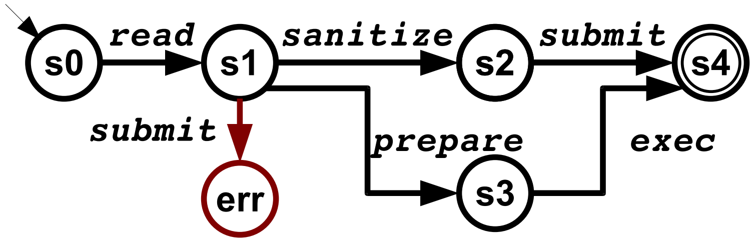

Figure 2.2: Example specification for Figure 1.

separately, with its own finite state machine [20]. At initialization, each finite state machine starts in its start state. Observed program events on a particular object can alter the state of the object's machine. A program *conforms* to a specification if and only if it terminates with all of its state machines in an accepting state. Otherwise, the program *violates* the specification, and there is be an error in either the source code, trace collection, or in the specification itself. Problem diagnosis typically requires programmer intervention.

This type of specification can describe properties such as locking [16], resource leaks [61], security [44], high-level invariants [27] and memory safety [36], and more specialized properties such as the correct handling of `setuid` [14] (and, more generally, other system calls) or asynchronous I/O request packets [6]. Moreover, these types of specifications can be used by many existing defect-finding tools (e.g., [6, 16, 18, 27]). Indeed, all existing bug-finders require an implicit or explicit notion of correct program behavior. These notions can be very broad, such as "a C program should not dereference a null pointer," or program specific, or may even come as implicit assumptions of the tool (such as with SPLINT [25], which checks a specific set of security properties). There are many particular defect-finding tools that can take as input the FSM specifications we are discussing here, such as [6, 16, 27].

The simplest and most common type of temporal specification is a two-state finite state machine [23, 61]. Such two-state specifications require that event a must always be followed by event b. This corresponds to the regular expression $(ab)^*$, and are written $\langle a,b \rangle$. Such specifications describe a particular aspect of correct program behavior [42], typically describing how to manipulate certain resources and interfaces. For example, a specification in this form can describe resource allocation or the correct restoration of invariants. Examples include ⟨`open`,`close`⟩, ⟨`malloc`,`free`⟩, and ⟨`lock`,`unlock`⟩. In order to be considered error-free, code described by such specifications must always adhere to them (unlike certain specifications which may describe allowable but not required behavior). These properties are prevalent in practice.

We focus in this work on these simple specifications, called *partial-correctness temporal safety properties*[1] , or temporal properties, for short. While the example in Figure 2.2 shows a multi-state finite state machine, we focus on these simpler, two-state properties. These properties are very similar to those expressed in the above example, in that they describe legal sequences of events on a given resource; they differ

[1] The use of the term correctness may be misleading in this context. We do not imply that programs that adhere to these specifications are necessarily *correct*. Many properties required for correct program behavior can not be expressed using this type of temporal property. Moreover, it is impossible to claim that a program is correct simply because it adheres to a specification. Rather, a program that does not violate such a property is known not to contain a very specific type of error, usually in resource handling.

only in that they are simpler and conform to a particular pattern. More complicated patterns are certainly possible, even in the field of specification mining [64].

We choose to focus on two-state temporal properties because they are simple but still sufficiently descriptive to be useful in the task of bug-finding [61], and because the problem of learning specifications, which reduces to learning a finite state machine for a regular language based on finitely many examples, is NP-complete [4]. Moreover, restricting attention to a single two-state pattern yields a very large potential state space: a program trace with n unique events has n^2 possible event pairs, and thus n^2 possible candidate specifications.

These partial correctness specifications are distinct from and complementary to full formal behavior specifications.

2.2 Specification Mining

Specification mining is the task of constructing a formal specification from examples of a program's behavior, or analysis of its source code [4], or both [62]. Such a task takes as input a set of *traces* of program behavior. A *trace* is a sequence of events with associated information, such as data values or similar context. Most commonly, traces consist of sequences of function calls. A specification miner examines such traces and produces zero or more *candidate specifications*, which must then be verified by a human.

Most existing specification miners for specifications describable by a finite state machine fall into two categories, distinguished by the type and size of the specifications they produce. Some produce a single, full-fledged finite automata policy with many states [3, 4, 62], such as we saw in Figure 2.2. Others produce many small automata [23, 29, 61, 64], typically of a fixed form.

We focus on the latter, because large automata are much more difficult to verify or debug [5], and previous work shows that two-state specifications are still useful in bug-finding [61]. Even when attention is restricted to two-state specifications, mining remains difficult [29].

Specification miners are highly dependent on their input traces. Traces of program behavior can either be statically generated — collected from the source code (e.g., [23]) — or acquired dynamically from instrumented executions on indicative workloads (e.g., [62]). Static traces can provide a complete characterization of program behavior. However, since there are potentially infinite possible paths through a program, only a subset of all static traces can be considered. Moreover, static traces provide an imprecise characterization of possible program behavior because exact program behavior is impossible to prove statically. Dynamic traces are potentially more accurate, because they only describe possible program behavior. However, dynamic traces are particularly sensitive to workload selection and may omit potentially relevant information by failing to characterize all possible types of program behavior. We focus attention on mining from statically generated traces.

A canonical example of this type of specification mining is described by Engler *et al.*, who note that programmer errors can be inferred by assuming the programmer is *usually* correct [23]. That is, common behavior implies correct behavior (uncommon behavior may represent a policy violation; this principle underlies modern intrusion detection systems e.g., [28]). Engler *et al.*'s `ECC` miner takes as input a set of program traces. Using these traces, the 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. The miner uses the z-statistic to rank the likelihood that the correlation is deliberate on the part of the programmer. Their miner presents a list of candidate specifications, ranked in order of their statistical likelihood. A human programmer must then evaluate the list to sort the true specifications from the erroneous ones. Erroneous specifications are candidates on the output list that do not describe required program behavior (that is, a program trace can violate an erroneous specification and still be considered error-free). We call such candidates *false positives*. Without human guidance (e.g., without lists of important functions to focus on [23]), 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 valide: a 99.5% false positive rate [61].

This example highlights a serious shortfall of previous mining techniques. Namely, it is not the case that common behavior always implies either correct or required behavior. There are two reasons for this. First, source code contains errors in the form of policy violations. The code does not provide an inviolable authority on what constitutes correct behavior. Second, some pairs of events very commonly appear together, but do not encode *required* program behavior. This class of candidates includes common red herrings such as ⟨`print`,`print`⟩, or ⟨`hasnext`,`getnext`⟩. A miner needs to distinguish correct from incorrect behavior and common from required behavior when evaluating specification candidates or else its output will be riddled with false positives.

Previous work observed that programmers often make mistakes in rarely-tested error handling code [61]. 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. Weimer *et al.* also included a software engineering consideration, restricting attention to specifications in which the events *a* and *b* came from the same package or library. They 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, because before being used, the candidate specifications must still be hand-validated. This thesis builds on this previous work by building a miner that operates on statically generated traces and outputs a candidate set of likely two-state temporal properties. We improve the process by incorporating a large amount of additional information from the software engineering process in the trace analysis.

Chapter 3

Motivating Example

In this chapter, we motivate our technique by presenting two candidate specifications and describing various differences between the code from which they were gathered. The purpose of this example is to establish that there may exist objective measurements of code quality that can help us distinguish between true and false positive candidate specifications. We do this by examining the code in which the candidate specification is followed and evaluating its trustworthiness. For the purposes of this example, we present one true specification and one false specification that a previously implemented miner presented to the programmer as specification candidates. We then analyze the traces through the code in which each specification appears. Our goal is to demonstrate there are measurements that we may perform on the code that could help us distinguish the valid specification from the invalid candidate.

The example specifications were mined by Engler's `ECC` technique [23] from Hibernate, an open source Java project that provides object persistence. This benchmark consists of 57,000 lines of code. We generated and analyzed a total of 12,894 traces from the source code by generating a maximum of 10 traces per method and using

```
Hibernate.cirrus.hibernate.SessionFactory beginTransaction()
Hibernate.cirrus.hibernate.Session close()
```

Figure 3.1: A valid specification for Hibernate

```
Hibernate.src.net.sf.hibernate.SessionFactory openSession()
java.util.Collection iterator()
```

Figure 3.2: A specification for Hibernate

symbolic execution to rule out infeasible paths.

Consider two candidate specifications mined by `ECC`. The first, a valid specification, is shown in Figure 3.1. This specification represents a portion of the finite state machine that describes appropriate usage of Hibernate's Session API. This API is central to the software project in question and is thus covered fairly extensively in the Hibernate documentation. This candidate is a valid specification in that, if a program trace violates it (beginning a transaction by opening a session and failing to close it), the trace contains an error. In contrast, consider the second candidate specification, presented by Engler's technique, shown in Figure 3.2. A cursory inspection by the programmer quickly identifies this specification as invalid: program traces may obviously violate this "specification" and still be considered correct by a

Metric	**Valid Specification**	**Invalid Specification**
Number of Traces	250	14
Average Path length	21	28
Max Path Frequency	85.4%	8.9%
Average Path Frequency	10.1%	0.07%
Max Path Readability	0.99440	0.00002
Average Path Density	20.35	56.63
Average Revision	1304	3812

Figure 3.3: Summary of selected features of the example specifications' code traces.

knowledgeable developer of the system.

These examples are particularly extreme, and as such they are easy to classify as valid and invalid, respectively, without additional knowledge of the program. However, a more detailed comparison of the specifications, and the code in which they appear, is also instructive. Figure 3.3 displays a summary of several such measurements. To find these numbers, we separated the complete set of traces that adhere to each of our candidate specifications. We call the set of traces that contain the true specification the “valid” traces and the set of traces that follow the false specification the “invalid traces”. We then calculated various metrics for each set of traces.

For example, the valid specification appears in 250 traces, while the invalid specification only appears in 14, out of a total of 12,894 traces generated for Hibernate. Moreover, every one of the “invalid” traces appears in testing code — implying, perhaps, that this code combination is not run frequently in the course of actually exercising the Hibernate API. We use a research tool [11] to statically predict the likely dynamic run-time frequency with which this path is exercised. This tool examines a path and uses a set of source-level features to predict which proportion of the time this path will likely be executed on an indicative workload – these numbers are percentages (of execution time) expressed between 0 and 100. The path frequency numbers for the traces under consideration support our intuition. The “valid” traces have a maximum statically predicted “path frequency” of 85% — these specifications are sometimes found on frequently run code. In contrast, the *maximum* statically predicted frequency for the “invalid” traces is less than 10%. No path on which the invalid specification is observed is predicted to run more than 10% of the time. We can conclude that the specification followed on frequently executed code is more likely to be correct, or even required.

A manual inspection of the invalid traces demonstrates that they are, generally, long and complicated. In fact, their average length is 28 events, almost three times the average length of traces in Hibernate overall. This suggests that the code in question is convoluted and not very readable. Previous work defines a "readability" metric that can describe sections of code. Readability numbers are based on a human study involving 120 participants and range from 0 to 1.0. A readability rank of 0 corresponds to "least readable", and 1 corresponds to "most readable" in the judgment of the human respondents. Buse *et al.* found a natural cutoff of approximately 0.6 that separates "readable" from "unreadable" code. This judgement correlates with bug density in source code. For further examples and explanations, see [10]. We hypothesize that unreadable code is similarly correlated with API policy violations, or that readable code is correlated with adherence to the API. This intuition plays out in this example when we measure the readability of the code from which our example specifications are mined. The *maximum* readability of any of the invalid paths is 0.0000232. This implies that the code on which our invalid specification is observed all exhibit very low predicted readability. The maximum readability of the valid traces is 0.962988, a dramatic counterpoint. These numbers support the claim that path readability may indeed correlate with the likelihood that an observed candidate specification is valid.

We can also measure how far along in the path-generation process we have to go to find a set of traces. We call this metric "path density", and it corresponds to how many acyclic traces will be statically generated in a breadth-first-traversal of a program, function, or type declaration before a given trace appears. Here, too, the candidate specifications show a marked difference: we need to emit many more traces before arriving at the "invalid" specification traces than we do to generate the "valid" traces. These measurements perhaps imply that these traces do not encompass the programmer's conception of the code's primary functionality.

Other artifacts from the software engineering process can contribute useful measurements to our analysis. For example, this project is stored and managed in an SVN code repository. *Code churn* measures how recently or frequently code has been changed. We can measure code churn by inspecting source control histories and determining when a change to line of code or a program trace was last committed, how many revisions have changed this particular program trace, etc. Previous research indicates that churn correlates strongly with errors or policy violations in code [47]. Measurements of churn on our valid and invalid policy traces adhere to this intuition. Our valid traces were last changed, on average, much earlier in the development process (revision 1304) than the invalid traces were (revision 3812). The valid traces have been stable for a longer period of time. Perhaps, as previous research indicates, the code that is more recently changed is less trustworthy or more prone to errors, and more likely to adhere to or contain true specifications.

These examples are instructive because they show that there are many features of code paths that seem to correlate with whether or not specifications found along those paths are likely to be true or false positives. This example shows several such metrics and motivates their use in the mining process. We can imagine collecting these measurements for all of our input traces and using them to guide a specification miner. Such a miner could look at the code that both adheres to and violates a candidate specification under consideration and use these metrics to inform its eventual judgement on its validity. However, they also motivate several further questions: how do we combine these features into a predictive model? Is average revision of last change more or less important than the maximum readability of the traces? Which other features extracted from software engineering artifacts contribute to the predictive power of such a model? How do the features correlate with one another, and are they equally predictive across different software projects? We will explore the application of these intuitions and an implementation of such a predictive model in the succeeding chapters.

Chapter 4

Our Approach: Code Trustworthiness

In the previous chapter, we presented an example to motivate our claim that there are many features of source code that can help us distinguish between true and false candidate specifications. These features can be derived directly from the source code, or can come from alternative software engineering artifacts. Essentially, however, we hypothesize that we can use these features to evaluate the likelihood that a given piece of code contains or adheres to project-specific temporal properties. We develop these intuitions in this chapter.

Previous approaches to specification mining have implicitly assumed that all execution traces are equally indicative of correct program behavior, that is, that all traces should be trusted equally. Thus, the appearance of a potential candidate specification in one trace is just as important as its absence in another trace when the frequency of a given pair is being calculated. However, this assumption does not always hold. Not all code is equally likely to be correct, and thus not all code is equally likely to adhere to program specifications. A particular execution trace may have been written by an inexperienced programmer who is unfamiliar with the underlying program API.

It may have been recently changed, and thus more likely to contain an error. It may not have been well-tested. Essentially, not all code is of the same quality, and we should consider this fact when calculating the probability that a specification pair is valid. In other words, not all event pairs found in traces should be weighted equally when determining candidate specifications.

We call code *trustworthy* if it is unlikely to exhibit API policy violations. Previous work hints at this idea by checking software engineering constraints about module packaging and error handling; we extend it here by introducing multiple notions of trust as well as by using non-binary notions of trust.

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 [47], unreadable code [10] and rarely-tested code [61]. Such information can be collected from a program's source code and version control history. By augmenting the trace language to include information from the software engineering process, we can evaluate the trustworthiness of every trace that supports or disputes the veracity of a candidate specification. We can thus more accurately evaluate the likelihood that it is valid.

We present a new 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.

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 automatically computed 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 [47]; 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. These measurements give us an idea both of how long it has been since the code was last changed as well as how "churned" the code is overall. We hypothesize that code with low churn is more likely to adhere to required temporal properties.

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 be more familiar with the code base and may remember more of the program's 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 to the repository that were committed by that author. We measure the rank of the last author to touch each line of code. This is not a standard measurement of author rank; we developed it in an effort to automatically measure author knowledge (without human annotation). We hypothesize that code modified by authors with a high author rank is more likely to adhere to required temporal properties.

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, and thus should not be treated as such in specification mining. For example, 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` tool-kit's copy-paste detector, which is based on the Karp-Rabin string matching algorithm [37]. We hypothesize that code that demonstrates a large amount of copy-paste development is less likely to adhere to required temporal properties.

Code Readability. We measure code readability using software readability metric, which is based on textual source code features and agrees with human annotators [10]. Buse *et al.* conducted a human study in which 120 subjects rated the "readability" of various code snippets, and then trained a linear model to predict readability based on various source-level features. Readability measures range from 0 to 1.0, with 1.0 meaning "most readable". A cutoff of approximately 0.6 separates unreadable from readable code. The work shows that more readable code is less likely to contain errors [10]. We hypothesize that more readable code is also more likely to adhere to required temporal properties.

Path Feasibility. Infeasible paths are an artifact of the static path enumeration process that can create input for specification miners. We claim that these infeasible (impossible) paths are unlikely to encode programmer intentions. Previous work has argued that it is always helpful to have more traces, even incorrect ones [59]; our experiments suggest that quality is more important than quantity (see Section 5.3). Merely excluding infeasible paths from miner input thus 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 [21]) reports that its symbolic branch guards are inconsistent. We hypothesize that feasible code paths are more likely to contain required temporal properties, and that infeasible paths are likely to contain false properties.

Path Frequency. We theorize that paths that are frequently executed by indicative workloads and testcases are more likely to contain correct behavior. This may be true for two reasons. First, common code paths are more likely to have been properly tested by developers, and thus are more likely to be correct. Second, the common case for program execution is likely to have received a large proportion of its programmer's attention, and is thus likely to have been mindfully (and correctly) written. We use a research tool that can statically estimate the relative run-time frequency of a given path through a program [11] to measure path frequency. This tool uses a set of source-level features to predict which proportion of the run-time in a method will follow a given path. The model is trained using indicative workloads on a number of benchmarks which are shown to likely generalize to other projects, such as those we examine here. Frequency numbers are percentages ranging from 0 to 100, normalized to the range $[0, 1]$. We measure relative run-time frequency with respect to the enclosing method. We hypothesize that code that exhibits high likely path frequency is more likely to contain required temporal properties.

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. A method containing many paths is likely to be more complex than a method containing fewer paths, and we predict that complexity and size may lead to programmer error. We define "path density" as the number of traces it is possible to enumerate in each method, class, and overall program. We hypothesize that 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.

We had several goals in developing this particular set of metrics. First, we sought out code measurements from previous work that seem to correlate with code correctness, such as readability, code churn, and measures of copy-paste development. This research seemed like a good place to start, following our hypothesis that code that is more or less likely to contain errors may also be more or less likely to adhere to specifications. We also sought to measure as many different easily-observable yet independent features of source code as possible, such as author rank, feasibility, and frequency. Our goals also included developing metrics that could be automatically collected to minimize programmer involvement and maximize the automatic applicability of our technique.

4.2 Mining Algorithm

Our mining algorithm extends the `WN` miner [61] and provides a method for incorporating our trustworthiness metrics into the process of mining specifications. Trustworthiness metrics may perhaps be extended to use in other miners in similar ways; for a discussion of the generalizability of our metrics, see Chapter 5.

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 $M_1 \ldots M_q$, 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 Chapter 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 [61]. 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 $a \ldots b \in t$ 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 [61].

Second, we collect a set of trustworthiness metrics that apply simply to program locations $\in \ell$, such as code churn and author rank.

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 data-flow information: when every value and receiver object expression in b also occurs in a [61, Section 3.1].

Previous work showed that both the `ECC` and `WN` miners can be expressed using this set of features [59]. 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. Both miners also calculate a z-statistic based on the number of traces (according to their respective definitions) to determine the likelihood that the event pair is likely to be correct, and use the z statistic to rank the output.

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

Figure 4.1: Features used by our miner to evaluate a candidate specification $\langle a,b \rangle$.

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 4.1 lists the 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), $\mathcal{M}_{ia}$ 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 if and only if $\sum_i \ c_i f_i < \mathit{cutoff}$. This involves a training stage to determine both the coefficients and the cutoff, described in detail in Chapter 5.

Chapter 5

Experiments

In this chapter, we describe our miner implementation in further detail and outline several experiments. These experiments evaluate the effectiveness of our new miner, provide a comparative analysis of our features over our entire dataset and on individual benchmarks, analyze the generalizability of the trustworthiness insight to other specification miners and explore the observed and potential limitations of our current technique.

5.1 Setup and Definitions

We evaluate our miner on the open-source Java benchmarks listed in Figure 5.1. We do not need source code *implementing* a particular interface to run our miner. Instead, we generate traces from the client code that *uses* that interface, as in [22, 4, 62, 64, 29]. However, we sometimes need to examine the library code that implements these interfaces to evaluate emitted candidate specifications and determine if they are true or false positives.

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	

Figure 5.1: Benchmarks used in our experiments.

We selected our benchmark programs to allow a direct comparison to previous work [29, 59, 61, 64]. To ease the collection of related trustworthiness metrics, we restricted attention to programs with CVS or SVN source-control repositories. For each program, we statically enumerated intra-procedural traces (up to 20 per method) and gathered the information required for the trustworthiness metrics described in Section 4.1. The most expensive operation was computing path feasibility, which required multiple calls to Simplify, an external theorem prover [21]. On a 3 GHz Intel Xeon machine, computing feasibility 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 911.982 seconds in total; this step only happens once. Collecting the other metrics for `mckoi-sql` is relatively inexpensive (e.g., 6 seconds for readability, 7 seconds for path frequency); more expensive operations, such as the collection of the revision history metrics, need only be computed once, and may be serialized to ease future mining runs. The actual mining process (i.e., considering the features for every pair of events in `mckoi-sql` against the cutoff) took 555 seconds.

In these experiments, a *false positive specification* is a candidate specification produced by a miner that a human does not judge to encode required program behavior.

A program can violate a false specification without necessarily containing an error; by contrast, there is necessarily an error whenever a program violates a true specification. Since specifications typically deal with the ordering of API functions, making this determination involves inspecting the source code implementing the API. The specification pair ⟨Socket.open,Socket.close⟩ is considered valid since the documentation and implementation suggest that important resources will be leaked if Sockets are not closed (and garbage collection with finalizers is insufficient [60]). On the other hand, ⟨Iterator.hasNext,Iterator.getNext⟩ is not a valid specification: while it is common to obtain the next element in from an Iterator after verifying that it is not empty, doing so is not required. Our manual determination of invalid specifications follows that of previous work [61] and conservatively errs on the side of annotating candidate specifications as false positives. The basic approach to evaluating a candidate specification is to manually inspect the code implementing the resources or libraries involved as well as the associated documentation.

We take this approach because one of the most common uses of specifications is automatic bug finding, and incorrect specifications often lead to undesired *false positive bug reports* [23]. Note however that false positive bug reports and false positive specifications are independent notions. For example, a false positive bug report can arise from an imprecise tool using a true specification. In our experiments, we count false positive specifications to measure miner precision and count the resulting true positive violations found using those specifications as one measure of specification utility.

5.2 Learning Cutoffs and Coefficients

First, we learn the coefficients and cutoff that determine which candidate specifications to output. This constitutes the *model* that we use to mine specifications from source code. In addition to mining specifications, we can analyze the model to evaluate the relative importance of each of our trustworthiness metrics in terms of predictive power. We can evaluate predictive power of a metric across all benchmarks as well as on individual projects to determine how much and why they can vary between projects. We can also use the model to attempt to correlate the metrics with one another. In sum, we can use our learned model to accurately characterize the importance of our metrics, the relationships between them, and their universality across benchmarks.

We use linear regression to find the coefficients for our miner. Linear regression takes a set of vectors as input. A vector consists of one numerical value for each feature in the model and a correct answer. Linear regression is a standard statistical technique that finds coefficients that best allows the features to predict the answers. 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 [59, 61] 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. This allows us to turn our linear model into a binary classifier. The objective function allows us to develop a model that exhibits certain desired properties. 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 true 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.

Either one of these measurements may be trivially maximized: returning all possible candidate specifications will definitionally include all valid specifications, yielding a recall of 100%. Returning no specifications excludes all invalid specifications, similarly yielding 100% precision.

Information retrieval provides another metric for evaluating the effectiveness of a given model: the F-measure. The general formula for F-measure for a given non-negative real β is shown in equation 5.1:

$$F_\beta = \frac{(1+\beta^2) * (\texttt{precision} * \texttt{recall})}{\beta^2 * (\texttt{precision} + \texttt{recall})} \tag{5.1}$$

β controls how much either precision or recall is weighted in this combined metric. $\beta = 1$ gives us the *harmonic mean* of precision and recall, which reasonably mitigates the problem of trivially maximizing either individual metric when evaluating potential models/coefficients. This metric is preferable because it allows us to build a model that maximizes both precision and recall while avoiding the trivial model.

We choose to build two different miners, focusing on different information retrieval metrics: a “normal miner” maximizes the harmonic mean, achieving a balance of true and false positives, while a “precise miner”, with very few false positives, maximizes just precision. We do not build a miner that attempts to maximize recall, for several reasons. First, it is not obvious how one defines recall for the task of specification mining. For the purposes of these experiments, we define all possible true positive specifications as the union of all specifications found by all miners on our benchmark sets over all of our experiments; however, this definition is imperfect, and omits multi-state specifications and API specifications that are not relevant to the client code in question. Moreover, constructing a miner that attempts to maximize recall requires a human annotator to evaluate an enormous number of candidate specifications. Note that a method for trivially maximizing recall in this context is to return all possible candidate specifications for a given benchmark – such a set would, by definition, include as many true positive specifications as could be defined using all two-pair combinations of events in the program under consideration. Over our entire benchmark set of input traces, there are 18268 unique events, yielding 333,719,824 total possible 2-state candidate specifications. Assuming optimistically that each specification requires 10 seconds of human time to validate, such a set of candidate specifications would take more than 100 human years to evaluate, which is outside the scope of our experimental means. Finally, one of our primary motivations is to construct a specification miner suitable for automatic applications, that is, one in which the false positive rate is low but the specifications are still useful. Accordingly, recall is a less suitable evaluation metric than the other two.

5.2.1 Trustworthiness Metrics

Our first experiment **evaluates the relative importance of our trustworthiness metrics**. Figure 5.2 shows a per-feature analysis of variance (ANOVA), over all of the training data. 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

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
One Error	21.2	0.0000
Same Package	14.2	0.0002
Exceptional	10.8	0.0000
Dataflow	7.6	0.0058

Figure 5.2: Analysis of variance of various trustworthiness metrics. The first seven metrics entries describe our new features; the final four metrics were explored in previous work. F measures the predictive power of a feature, or how useful it is in evaluating whether a specification is true or false. A number near 1.0 indicates that the metric is not predictive. A high F value indicates a powerful, predictive feature. p represents the standard statistical significance test – it evaluates the probability that a feature is not predictive in the model Thus, a smaller value of p denotes that the metric is more likely to be predictive. A p value below 0.05 indicates a statistically significant feature (all features except Author Rank were significant).

does not affect the model. The p column shows the probability that the feature does not affect the miner. A smaller value for p means that the feature is likely to affect the model; $p \leq 0.05$ denotes a significant result.

All of our trustworthiness features except Author Rank had a significant main effect ($p \leq 0.05$). 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. This is probably related to our formulation of author rank. A better measure for author rank or author ability may be more useful than the one we present here; this remains an open area of research for future work. The Frequency metric, encoding our static prediction of how often the path would be executed at run-time [11], was the most important feature: commonly-run paths do not demonstrate erroneous behavior.

5.2.2 Comparison With Features Proposed In Previous Work

Previous work examined the relationship between error traces and specification false positive rates [61]. Weimer *et al.* used several criteria to select candidate pairs: every event b in an event pair must occur at least once in exception cleanup code, there must be at least one error trace with a but without b, both events must be declared in the same package, and every value and receiver object expression in b must also be in a.

We included these features in our model to determine their predictive power. The results are shown in the second half of Figure 5.2. These features also affect the model with varying strength: the “exceptional data flow” condition affects the model most strongly, and the “one error” condition has the least significant effect. These features are not as predictive as our most predictive trustworthiness metric (Frequency), but tend to fall between feasibility and path density in terms of effect on the model.

Other features have been investigated as useful heuristics to help distinguish between true and false candidate specifications, such as name similarities (the edit distance between the two events) or the reachability of one event from another on a call graph [64]. However, the only other feature from our list of proposed trustworthy metrics to have been previously investigated, to our knowledge, is feasibility [3]. All of our new trustworthiness features were more important than this feature.

5.2.3 Trustworthiness Metrics Between Benchmarks

In the first experiment, we analyzed the relative importance of the trustworthiness metrics on data collected from all seven benchmarks. However, we further wish to determine whether and how these metrics vary in importance between the different benchmarks. Such an analysis can provide insight into how much the development culture on a particular project influences which metrics are important and helps us analyze the universality of our metrics.

To conduct this experiment, we built specification mining models for each of our benchmarks using the same technique as for the entire benchmark set. We then performed an analysis of variance on each model. The results of these individual analyses are shown in figure Figure 5.3. The average and standard deviations of F and p for each individual metric are shown at the bottom of the table. Notable outliers are highlighted.

These results illuminate interesting differences between the different benchmarks.

	Frequency		Copy-Paste		Code Churn		Density		Readability		Feasibility		Author Rank	
Program	F	p	F	p	F	p	F	p	F	p	F	p	F	p
hibernate2	10.9	0.0010	81.8	0.0000	7.4	0.0066	8.1	0.0044	3.7	0.0553	4.9	0.0275	5.0	0.0257
axion	22.8	0.0010	2.3	0.1316	5.5	0.1879	22.8	0.0000	4.2	0.0393	1.0	0.3110	10.1	0.0015
hsqldb	16.3	0.0000	2.0	0.1571	3.3	0.0700	7.7	0.0054	7.0	0.0082	5.4	0.0249	0.8	0.3724
cayenne	11.1	0.0008	8.5	0.0035	13.1	0.0003	9.5	0.0021	21.2	0.0000	2.2	0.1363	5.4	0.0205
jboss	7.1	0.0077	11.8	0.0006	10.3	0.0013	21.3	0.0000	5.6	0.0182	6.6	0.0010	4.7	0.0294
mckoi-sql	4.9	0.0266	0.6	0.4241	3.9	0.0476	14.5	0.0001	6.8	0.0092	4.9	0.0270	2.8	0.0943
ptolemy2	11.1	0.0009	8.8	0.0037	8.7	0.0031	8.2	0.0043	4.0	0.0461	7.3	0.0070	8.4	0.0037
Average	12.0	0.0054	16.5	0.1029	7.5	0.0453	13.2	0.0002	7.5	0.0367	4.6	0.0764	5.3	0.0782
Std Dev	5.95	0.0097	29.1	0.1569	3.5	0.0685	6.5	0.0024	6.2	0.0314	2.3	0.1130	3.2	0.1334

Figure 5.3: Analysis of variance for each individual benchmark. To find these numbers we build a specification mining model for each program, ignoring the data from the others. The predictive power of each metric is measured separately for each benchmark program. F measures the predictive power of a feature, or how useful it is in evaluating whether a specification for a given benchmark is true or false. A number near 1.0 indicates that the metric is not predictive; a larger number means the metric is more predictive. p represents the standard statistical significance test – it evaluates the probability that a feature is not predictive in the model. Thus, a smaller value of p indicates that the metric is more likely to be predictive. A p value below 0.05 indicates a statistically significant feature. Outlier cells are highlighted in gray.

While we can develop informed hypotheses as to the nature of these distinctions, further study is necessary to more fully understand the situations under which the different metrics work well or poorly. Despite this, we consider each metric in turn to explore the obvious distinctions that arise between benchmarks and attempt to reasonably explain them:

1. **Frequency:** Frequency has very strong predictive power on the `axion` benchmark and comparably weaker predictive power for `mckoi-sql`. An explanation for this behavior may lie in the fact that `axion` ships with a very comprehensive test suite. Unit tests and similar testing code is likely to demonstrate correct usage of particular (important) functionality. While there exists testing code that exercises an API incorrectly or is otherwise improperly implemented, it is reasonable to assume that, generally, unit testing code is relatively straightforward and follows implicit specifications describing the code under test. Moreover, the `axion` testing code itself is relatively simple: the test classes typically consist of no more than 100 lines of code. Axion may thus be particularly susceptible to the definition of path frequency: the frequency measurement of a given path is the statically predicted frequency with which a path through a method will be executed dynamically, at an intra-class level. Paths through testing classes are therefore likely to be correct and also likely to be executed with high frequency. This provides a likely explanation for the strong predictive power of the frequency metric on the axion benchmark.

   It is not obvious why frequency provides comparatively poorer predictive power on the `mckoi-sql` benchmark. For a large program, where we might expect more average behavior (as we see on `cayenne`, `jboss`, or `ptolemy`, `mckoi-sql` is an outlier on a large number of metrics. We leave an exploration for why this program is so different from the others to later work.

2. **Copy-Paste:** Two notable outliers on the predictive power of the copy-paste code metric are `mckoi-sql` and `hibernate`. This probably relates to with the amount of code that has been cut-and-pasted in the `hibernate` benchmark relative to its size as compared to the other benchmarks. When considered as a percentage of total lines of code, 0.8% of hibernate's code is marked as copied by the PMD toolkit. This figure is twice as high as that of the benchmark with the next highest percentage, `jboss` (which is also the benchmark where the copy-paste metric has the second-highest measure of predictive power). `mckoi-sql` has the lowest percentage of code that has been copied and pasted, at 0.03%.

   This relationship is not necessarily strictly linear for the benchmarks with intermediate values of F for this metric. However, a general trend does appear to hold.

3. **Code Churn:** Code churn is comparatively less predictive on the `hsqldb` and `mckoi-sql` benchmarks than it is on the other five. This may be related to the number of total revisions in the repositories for these benchmarks as compared to the others. The `hsqldb` 1.7.1 release contained 72 revisions in the svn repository, while `mckoi-sql` had undergone 594 (relatively few compared to its size). Comparatively, `axion` had undergone 1178 revisions, `cayenne` 1959 (quite a few, given the size of the program), `hibernate` 2080, `jboss` 13378, and `ptolemy` 29323. The amount of source control information available appears to affect the usefulness of the code churn metric on a given software project.

4. **Density:** Density has a surprisingly strong predictive power on the axion and jboss benchmarks. Observations we made earlier suggest a possible explanation for this behavior. The axion source code contains a large number of test cases which display generally correct (API-consistent) behavior and consist of very simple code. We hypothesized that frequency has strong predictive power on axion because paths through test cases receive high frequency numbers and are also very likely to correctly adhere to specifications. The linear model generated for axion indicates that there is a positive relationship between total path density and likelihood that a specification candidate is valid. The actual order in which traces are generated therefore matters, because this particular way of calculating path density simply tracks how many traces were generated before this trace in the trace enumeration process. The axion code base is organized in such a way that traces through the regular source code are enumerated first, and make up the first 53% of all traces listed in the trace input. The remaining 47% of traces come from the unit testing code. Thus, if we extend our hypothesis that testing code is likely to be correct, the traces generated later are more likely to be correct, and total density logically correlates strongly with specification validity.

   Similar logic seems to explain the jboss outlier, which is also strongly influenced by total path density. While jboss does not contain as many obvious unit tests as axion does, it does contain many paths through type declarations containing the word “Debug” and, as with axion, these traces incidentally appear late in the trace generation process.

   Thus, these outliers are very much an implementation accident, suggesting that total path density is very sensitive to the order in which we process input files. Method density and type declaration density (which counts how many traces have been output for a given method or type declaration) do not display this level of variation or sensitivity across benchmarks.

5. **Readability:** We explored a number of aspects of program readability to attempt to explain why it is very predictive on `cayenne` and much less so on `hibernate`. We looked at the readability of all traces in a program as well as those traces that contain the actual predicted specifications. However, we could not find a reasonable explanation for the discrepancy. The only obvious statistical difference between the benchmarks is the range of readable traces in a given function - `hibernate` contains functions with very large numbers of readable traces (> 300), while `cayenne` has a much lower range (< 75 in its function with the most readable traces). However, this may be related to the density of the code in each of these benchmarks. In any case, its relationship to the predictive power of readability in each of the benchmarks is unclear. Further study is necessary to get to the root of this discrepancy.

6. **Feasibility:** On this metric, the `axion` and `ptolemy` projects are ouliers: on `axion`, feasibility has little-to-no likely predictive power, whereas it is comparatively very predictive on `ptolemy`. Note that the number of conditionals in a program is relevant here: a path cannot be marked infeasible by a symbolic execution engine if there are no conditional guards along the path. Programs with more conditional guards along its paths would therefore likely contain more paths that the symbolic execution could mark infeasible, perhaps increasing the usefulness of the feasibility metric in evaluating candidate specifications. As a

heuristic, we can count the number of times `if` or `while` appear in the source code for our projects (this is obviously not an ideal metric, but it is sufficiently illustrative for our purposes). `ptolemy` has approximately seven times as many conditional guards as axion, which correlates well with this metric's relative predictive power between these benchmarks. This ratio of guards to feasibility predictive power holds for all benchmarks except for `mckoi-sql`, which contains fewer conditionals than we might expect based on this hypothesis.

This size and age of the `ptolemy` benchmark may also influence feasibility's predictive power: an older, larger project may contain more old, irrelevant code and invalid paths that have developed over the development process.

7. **Author Rank:** Again, `axion` is a large outlier on this metric. Author rank measures the proportion of changes to the code base as a whole where authored by the author of a given line/trace. This formulation of rank attempts to estimate an author's potential familiarity with the code base as a proxy for the author's likelihood to write correct code.

   `axion`'s testing code, which appears to strongly influence our predictive model in other ways (see above), also seems to influence the importance of author rank. All of the testing code was written by one author, who also seems to have worked on a variety of other portions of the source code. This author likely wrote correct code, as suggested by the earlier analysis of the testing code, and so his rank is the most likely factor in the high predictive power of author rank on the `axion` benchmark.

   The other benchmarks vary in predictive power, but are generally lower, as we might expect both from the definition of rank and based on the ANOVA results across the entire benchmark set.

Benchmark size appears to influence the uniformity of our results. `axion`, one of our smaller benchmarks, is an outlier on several of our metrics, which seems to be related to the proportion of its source that consists entirely of simple, correct test cases. The last and largest benchmark is also interesting, however. `ptolemy` is our largest benchmark by far, consisting of 362k lines of code. It is also the benchmark that displays the most average behavior. None of the metrics display outlier values on this benchmark. This fact is encouraging, because we would hope that the largest, most well-established programs are the most general and thus the least likely to contain anomalous behavior for a given predictive model. Put differently, the largest projects are hopefully the most average program for a given set of measurements. We find this is true for our trustworthiness metrics, in that, with the exception of `mckoi-sql`, the larger the programs grow (the table above is ordered by benchmark size), the fewer outliers in metric predictive power appear. `mckoi-sql` is anomalous in several ways: the different features are not especially uniform, it is an outlier on several of them, and several of our explanations of benchmark variation do not seem to apply. The reasons for this are not obvious, and deserve further study

Two metrics, however, were highly predictive on all benchmarks: statically predicted path frequency and code density. These metrics are perhaps more universal than the others because they are independent of the code development methodology on any given software project. Where repository commit histories may be influenced by coding practices or standards, and can thus potentially be ascribed to prescriptions that guide developer behavior, frequency and density are metrics related entirely to the traces themselves. They do not suggest normative standards of code

Metric	Churn	Rank	Copy-Paste	Feasibility	Density	Frequency	Readability
Churn	1	0.64	0.07	0.02	0.11	0.00	-0.13
Rank	-	1	0.10	0.04	0.02	0.02	-0.08
Copy-Paste	-	-	1	0.00	-0.05	0.07	0.08
Feasibility	-	-	-	1	0.01	0.06	0.00
Density	-	-	-	-	1	-0.14	0.00
Frequency	-	-	-	-	-	1	0.01
Readability	-	-	-	-	-	-	1

Figure 5.4: Pearson correlation coefficients between the different metrics measured across all benchmarks. A number in a location signifies the correlation between the metrics listed in the row and column of that location. ±1 indicates perfect correlation, while -0.3 to 0.3 indicates small to no correlation. All metrics except Code Churn and Author Rank are effectively uncorrelated.

trustworthiness: they are purely descriptive metrics, largely impossible to influence with programmer behavior. However, as we saw above, they may be sensitive to trace generation implementation.

These observations provide insight into the nature of our metrics, their strengths and weaknesses, and their variability between software projects. We are encouraged by their relative uniformity on our largest benchmark. However, further study is necessary to scientifically verify our observations as well as to better characterize the taxonomy along which these metrics may fall, their usefulness as applied to a specific software project, and their utility as normative (vs. descriptive) descriptors of code quality.

5.2.4 Correlation Between Trustworthiness Metrics

To analyze the relationship between our metrics themselves, we collected the metrics across the entire benchmark set. We then performed pair-wise Pearson correlation calculations between all possible pairs of metrics. Figure 5.4 shows the results of these computations. The number in a location in the table corresponds to the Pearson correlation coefficient between the metrics along the given row or column; metrics have a perfect correlation (of 1) with themselves. A correlation may be either positive (both variables increase together) or negative (when one variable increases, the other decreases). The correlation coefficient may range between -1.0 and 1.0, where ±1.0 indicates that the two variables analyzed are equivalent, potentially modulo a scalar value. Highly correlated metrics are likely measuring the same thing, and so we would like our metrics to be as uncorrelated as possible. This will support our claim that they provide independent measurements of code quality.

The interpretation of the magnitude of the correlation between variables is an open question, heavily dependent on the nature of the problem (a correlation of 0.9 may be very strong in a social science application, but quite low in a scientific setting with highly precise measurements). Despite this, a common heuristic interpretation holds that correlations between 0.0 and 0.3 (or 0.0 and -0.3) are very small. According to this heuristic, the table shows clearly that all metric pairs except one are uncorrelated. The exception is Author Rank, which correlates with relative strength with Code Churn. This makes sense given how we measure Author Rank: an author's rank is simply the proportion of changes to the code base she has committed. We can observe several other intuitively reasonable effects if we consider the pairs of metrics with smaller correlations. For example, churn seems to negatively influence readability: the more that code has been changed, the less readable it becomes. Churn also correlates slightly with density: code with more static paths is churned more. Similarly, higher-density paths have a slightly smaller statically predicted execution frequency. There is a small correlation between author rank and proportion of code that has been cut-and-pasted; perhaps programmers who commit more to a code base are more familiar

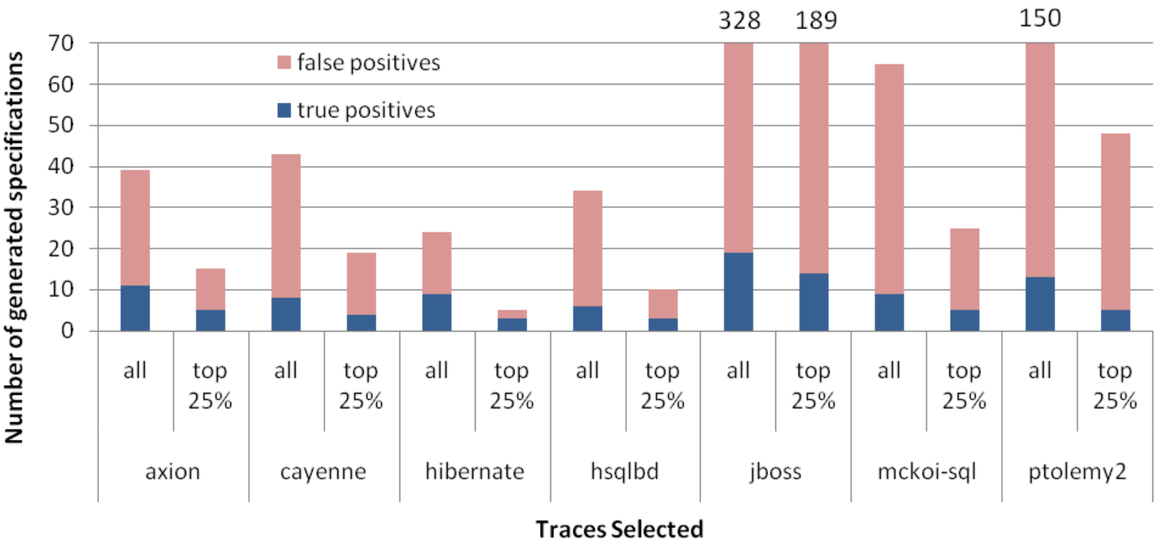

Figure 5.5: The true and false positives mined by the WN miner from previous work on various input sets. The total height of the bar represents the number of candidate specifications returned to the user for inspection.

with portions of the code that may be pasted elsewhere. However, again, these effects are very small.

We can therefore conclude that all of our significant metrics are distinct, non-overlapping features that describe independent aspects of code quality.

5.3 Trust Matters for Trace Quality

In our second experiment, we **demonstrate that our trustworthiness metrics improve existing techniques for automatic specification mining**. We do this for two reasons. First, previous work claimed that more input is necessarily better for specification mining [59]. That is, miners tend to find more true specifications the more traces they receive as input. We hypothesize that this is not the case and that,

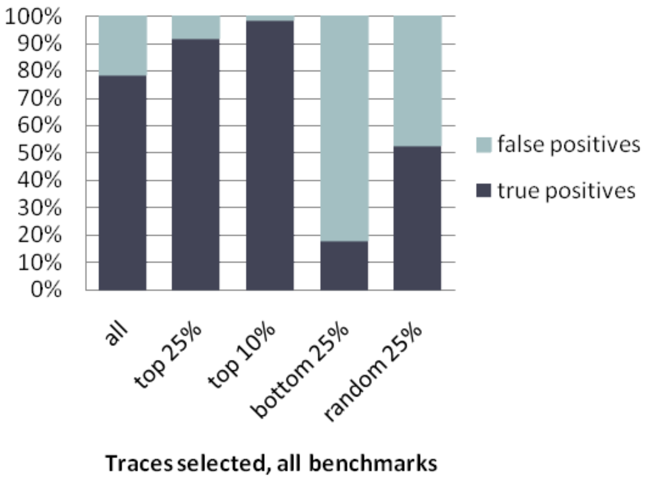

Figure 5.6: The false positive rate of the off-the-shelf `WN` miner on various input sets, as a percentage. The dark and light split of the bar shows the true/false positive rates of the candidate specifications returned to the user.

instead, miners can find the same specifications on less input, given that the input is *trustworthy*. In other words, we claim that the input to a specification miner has a significant effect on its false positive rate. We further claim that we can lower the false positive rate by using only trustworthy traces. Second, we wish to establish that our claims generalize beyond our specific mining implementation: trustworthy code is useful even when used with other, off-the-shelf, unmodified miners.

For each of our benchmarks, we run the unmodified `WN` miner [61] on multiple input trace sets. For generality, we restrict attention to feasible traces, since miners such as `JIST` already disregard infeasible paths [3].

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 5.2). We use this trustworthiness measure to sort the traces from most to least trustworthy, and then use portions of this sorted list as input.

We explore the impact of trustworthy traces on false positive rates by passing various proportions of trustworthy input to the `WN` miner. Figure 5.5 shows the raw results when only the 25% most trustworthy traces are used; Figure 5.6 shows the results normalized to 100% for a clearer view of the actual false positive rate. On the baseline set, `WN` has 683 false positives, or 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. The effect on false positive rates is even stronger when we consider the top 10% of traces (though again, we find fewer true positives than we do on the entire input set)..

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.

First, we can conclude that the trustworthiness concept can generalize to other specification miners beyond our own. Its usefulness is not an accident of our implementation. Second, we disprove the claims of previous work: more input is not necessarily better for specification mining, so long as the input is trustworthy. These results have several additional potential applications. 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 [59]: focus should be placed on sharing information from trustworthy traces. Our trustworthiness metrics could generally be used as a reprocessing step to improve any static trace-based specification miner (e.g., [23, 29, 61, 64]). However, they can be even more useful when directly incorporated into a mining algorithm.

5.4 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 overfitting by testing and training on the same data. We train our miner using labeled training data. We then test the model developed in this testing phase and evaluate the miner by hand-annotating and inspecting the candidate output. A potential problem with this approach is that our results might not generalize if the model is moved to evaluate a software project that we did not use in training. Fortunately, statistics offers a method to evaluate this risk for a given task. We use 10-fold cross validation to mitigate this threat [40]. 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 partitions) 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. Finally, recall that our trustworthiness metrics are defined independently from the programs in the benchmark sets, and consists of numerical values and ranges that are either objective or are developed in previous work on different input sets. Therefore, it is safe to conclude that our results are significant and will generalize to other programs outside our training set.

5.4.1 Mining Results

Figure 5.4 shows the results of applying our new specification miner from Chapter 4 to the benchmarks in Figure 5.1. 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 report this number because it represents a reasonable metric for specification utility. A miner that produces trivial specifications is not especially useful. Unfortunately, it is difficult to measure the quality of a specification with respect to refactoring or documenting code, and so we use the number of violations found as a heuristic for the utility of the true specifications mined (we do not report the number of violations found for false positive specifications). Finally, we include published results for the `WN` [61] and `ECC` [23] miners for comparison.

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

Figure 5.7: Comparative specification mining results on 800kLOC. “Specs” indicates valid specifications, “False” indicates false positive candidate 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 as points of comparison because of their comparatively low false positive rates. Other methods produce even more candidates. The Perracotta miner can produce multi-state specifications that are more complicated than those presented here. However, they do explore the generation of two-state properties. On jboss, the Perracotta miner produces 490 candidate two-state properties of this variety, which the authors say "is too many to reasonably inspect by hand." [64] Gabel and Su report mining over 13,000 candidates from hibernate [29]. By contrast, 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. 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. In our experience, 150 or so candidate specifications present a reasonable challenge for manual inspection, particularly when the potential payoff is high. More than a couple of hundred candidates becomes arduous.

We also trained a "precise miner" that seeks to maximize precision and reduce the false positive rate as much as possible. This "precise miner" finds fewer valid specifications total, but its 5% false positive rate approaches levels required for practical automatic use. It finds 30% as many specifications as WN, and 60% of the violations, but each candidate inspected yields over 12 violations on average. Moreover, users are often unwilling to wade through voluminous tool output [23, 36].

Both miners far outperform the ECC miner in terms of true specifications, false positive rates, and violations found.

The fact that our miners find more useful specifications than previous work may be related to how they select specifications for consideration. The WN miner only considers specifications that are followed at least once in general and violated at least once along an exceptional path. This implements the intuition that programmers make mistakes in exception-handling or clean-up code. The notion of a “bad path” is important to the WN miner’s selection of candidate specifications: the specification must be followed on at least one “good” path and violated on at least one “bad” path in order to be considered for ranking. Similarly, our miner considers both “good” and “bad” paths when evaluating potential specifications: a specification is likely to be valid if it is often followed on trustworthy (good) paths and violated on untrustworthy (bad) paths. Our criterion for specification consideration is broader. There are many more untrustworthy paths than there are exceptional paths. The WN miner by definition restricts potential specifications to a small subset of possibilities, and thus may limit itself to finding specifications that can only detect a certain type of error. By contrast, our trustworthiness-metric-based miner admits a broader range of specifications and definition of “bad” code. This may explain why it finds more violations per specification. However, a study of the types of violations found by a specification and of specification utility in general remains the focus of potential future work.

5.4.2 Miner Limitations

Our technique displays some limitations in terms of the specifications it can and cannot accurately mine.

```
Hibernate.cirrus.hibernate.Session beginTransaction()
Hibernate.cirrus.hibernate.Transaction commit()
```

Figure 5.8: The single false positive presented by the precise miner

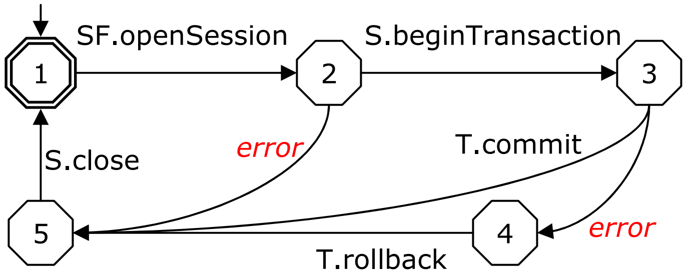

Figure 5.9: A multi-state finite state machine describing the Hibernate Session API, taken from the Hibernate documentation. Our false positive corresponds to the S.beginTransaction and T.commit edges.

First, it is worth discussing the false positives that remain, even when we optimize for precision. The precise miner performs without false positives in all cases but one. On the Hibernate benchmark, the miner yields 1 false positive. This anomaly is worth examining in further detail. The candidate specification in question is shown in Figure 5.8. At first glance, this candidate is not obviously a false positive. It represents a common code pattern that is documented as part of the Hibernate API [61], shown as a finite state machine (FSM) in Figure 5.9. The true-positive example in Chapter 3 (⟨`beginTransaction`,`close`⟩) is adapted from this FSM. In fact, the code pattern in Figure 5.8 is *almost* the required behavior. Intuitively, it represents the *common* behavior in code that uses this API (except in error cases). While the behavior encoded by this specification is correct, it is not *required*: it is possible for a path to call `openSession` and not eventually call `commit` and still be error-free if, for example, it calls `rollback` instead. The only required behavior from this FSM that can be encoded as the type of temporal property we are mining is ⟨`beginTransaction`,`close`⟩, because any path that calls `beginTransaction` must always eventually call `close` if it is to be considered error-free. This is not true of `beginTransaction` and `commit`.

Because this behavior is probably the common case for usage of this API, there are no traces on which this false specification is followed on which the true specification is not, and thus the trustworthiness measurements are almost identical for the two sets of traces.

This example is interesting for several reasons. First, it implies that further study is needed to help distinguish between extremely common behavior and required behavior, and that trustworthiness, while a very useful metric for distinguishing between the two, cannot always solve the problem. However, it also suggests that our assumptions of two-state specifications and the utility of trust measurements are usually reasonable. The miner mistook this candidate specification because of the level of overlap between this code pattern and the actual required behavior. None of the other benchmark APIs were mistakenly handled in this manner. This implies that our model for specifications as they appear in the code is fairly sound in practice.

Despite this anomaly, the false positive rate of our "precise miner" is very low. With a 5% false positive rate, and more useful specifications (in terms of bug-finding) than those of previous work, we claim that our precise miner might be reasonable in both interactive and automatic settings.

There are further specifications and classes of specifications that our trustworthiness-miner technique will necessarily miss.

Most obviously, specifications that are more complicated than the two-state properties we consider here will be excluded from our analysis. One imagines that the trustworthiness metrics may extend to mining more complicated patterns. However, an investigation of this hypothesis remains future work.

Trustworthiness may not be able to find all possible two-state specifications. Our practice of mining on client code leaves our technique vulnerable to situations in which the client code does not make use of patterns controlled by specifications – if a program never opens a socket (or even if a program closes a socket without ever opening it), ⟨`open`,`close`⟩ will not appear in our miner's candidate set. Certain patterns of usage may be allowable for a given API, but a client making use of the API may have stricter requirements and use the API in a way that treats sequences of calls as specifications even if they are not truly required behavior according to the library documentation. Specification mining from client traces is sensitive to patterns of use in the client code.

Trustworthiness metrics may be fooled by development methodologies or code evolution. A well-established, well-tested and stable code base may be dramatically reshuffled if an underlying library API is changed and displays new and different specifications to the client code. The previously stable code may change substantially and quickly, and programmers may need to rapidly code around changed usage patterns while maintaining previous functionality. This could yield code with high churn and lower readability even though it has only been changed in order to maintain correctness. Very stable code that has not been changed will appear more trustworthy to our miner, potentially causing us to mine the old, incorrect specifications for the API in question. Alternatively, very stable code in an infrequently used (in practice) portion of a project may appear very stable to our algorithm but contain just as many bugs and incorrect usage patterns as a portion of the code that is still evolving, further confusing our trustworthiness approach. Finally, coding practices may fool our metrics or at least render them less useful, such as a refactoring pass over a project that renders everything more or equally readable, or a large number of unit tests that confuse the path density and frequency metrics. The impact of these potential development situations on our metrics and their usefulness in specification mining remains an area of future investigation. However, they represent intuitively reasonable areas in which our metrics would exclude true specifications in the interest of favoring trustworthy code in the specification mining process.

5.5 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 [29, 59, 61, 64], 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 5.4 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.

5.6 Experimental Summary

This experimental section supported several key claims. First, we built a model to relate trustworthiness metrics in code to valid specifications. We then performed an analysis of variance over the entire benchmark set to provide a comparative analysis of the importance of the different metrics. We learned that statically predicted path frequency is very predictive but that our formulation of author rank is not; our model implies a need for a better definition of author rank in future work. We performed similar ANOVAs on models built for each individual benchmark and explored several reasons for variation in a metric's predictive power. This gave some insight into what drives the predictive power of a metric and clarified how the features work in practice. Next, we analyzed potential correlations between our features and found that our significant features were statistically independent of one another (the exception to this is author rank, which is not significant over the entire benchmark set). For our second set of experiments, we partitioned an input set of program traces by trustworthiness and gave different sets of input to an existing miner. This experiment established that we can eliminate at least 40% of untrustworthy statically generated trace input to existing miners and still find the same specifications. We also showed that smaller sets of trustworthy input can find fewer specifications, but with much lower rates of false positives, than the entire input set; these smaller sets also find more specifications than random sets of equivalent sizes. This set of experiments disproved of previous work that claimed that more input is necessarily better for specification mining: less input is just as good, if not better, so long as the input is trustworthy. These experiments also showed that the idea of trustworthiness generalizes to other specification miners and the metrics' usefulness are not an accident of our implementation. Finally, we evaluated the performance of our new miner. We found that our normal miner outperforms previous work by 20%, and finds more useful specifications with fewer false positives than previous input. Our precise miner found fewer true positives, but reduced the false positive rate by an order of magnitude and found a large number of potential violations when compared to previous work. In fact, this miner found only one false positive on the entire input set. Finally, we explored some limitations of our technique as well as threats to validity. While trustworthiness is a useful technique for mining two-state specifications from source code, plenty of work remains to be done; we explore some of these ideas in Chapter 7.

Chapter 6

Related Work

In this chapter, we describe related work. Our work falls between two main research areas: specification mining and software engineering quality metrics.

6.1 Previous Work in Specification Mining

Our work is most closely related to existing specification mining algorithms (see [61] for a survey). The ECC [23] and WN [61] algorithms form the basis of our own. ECC operates on statically enumerated traces and consider pairs of events that occur together on at least one trace, operating under the assumption that the programmer is usually correct (and behavior that deviates from the norm is likely incorrect). They use the z-statistic to rank the candidate specifications in the order of their likelihood [41]. However, this technique is prone to a high rate of false positives, although this concern was not a primary focus of their work. WN improved on their results by narrowing the criteria used to select candidate specifications and considering various source code- and software engineering-based features: candidate event pairs are required to have a data-flow dependence between them, should help find at least one error, should come from the same package, and, most notably, should exhibit a violation along at least one *exceptional path*. These insights improved on the true and false positive rates of `ECC` by an order of magnitude. Our work uses many of the same techniques for the gathering of event pairs and incorporates features used by these two miners in the model-development process. We also validate some of the assumptions underlying the work in the `WN` miner. We also use the specifications mined by these previously published techniques to train our linear regression.

Whaley *et al.* mine interface specifications in two ways: statically and dynamically [62]. The `WML_static` 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 miner uses multiple finite state machine sub-models to model a class's interface, and develops the sub-models using the statically generated traces as training input. More specifically, the miner models values of fields that conditionally raises exceptions along some path. If a function will raise an exception when a field has a certain value, and another function will set that field to that value, the event pair is prohibited in the final policy. The static miner outputs the most permissive policy based on these restrictions. In contrast to their work, our approach learns policies based on common client usage and not on a direct object-field encoding of typestate: as a result, we can potentially learn policies that are based on other resource implementation strategies. However, we do not learn the same type of specifications.

The `WML_dynamic` [62] miner examines dynamic traces and also produces a permissive multi-state specification that permits all observed behavior. Because of this, it is very sensitive to correct input selection. We attempt to incorporate similar insights in the construction of our miner by using the static path frequency prediction metric, and we, too, find that incorporating information about the dynamic behavior of programs can improve the accuracy of mining. However, some of our features are sensitive to the input and the nature of the software projects in question.

The JIST [3] miner refines the WML_static approach and uses techniques from software model checking to rule out infeasible paths. The JIST miner attempts to develop the most permissive interface for any given class. The tool takes as user input a class and a particular undesirable exception; the final policy describes the class and the most permissive API that, when followed, does not allow the exception to occur. They treat the potential API as a regular expression and the task of learning the interface as a game involving asking a teacher (the program traces) questions about what is in the regular language in question. In this way, JIST uses the statically generated paths to successively refine the policy. We also use symbolic execution to generate our paths, and similarly find it beneficial to the construction of a miner. In addition, JIST works on a per-class basis, operates on library implementation code, and requires that the programmer choose the class under consideration. By contrast, our approach works on client code, does not require manual guidance to pinpoint interesting classes, but only generates two-state policies and may consider more infeasible paths.

The Perracotta [64] miner is a dynamic analysis tool that mines multiple candidate specifications that match a given template (the two-state specification form used in this paper is one such template). Perracotta defines a hierarchy of property templates that capture commonly-occurring program behavior. Their dynamic analysis can compute an approximate policy that is later checked by a verification tool such as a model checker; it has been used on systems code including Windows kernel APIs and Unix filesystem implementations, and has successfully inferred large 24-state policies.

By contrast, our work is limited to mining simpler two-state candidate specifications. Gabel and Su [29] 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. This suggests that our work needs to be further extended to capture behavior that cannot be expressed by simply composing the temporal properties we mine. However, while our temporal properties cannot express all possible program specifications, we do find them useful in identifying potential errors in code.

The `Strauss` tool [4] form "scenarios" using events from dynamic traces of program behavior. It requires a number of programmer inputs, such as a seed event and the maximum scenario size. The tool extracts automata from scenarios based using an out-of-the-box probabilistic finite state machine learner. The learner outputs a single policy, which the tool then "cores" by dropping transitions with low probability weights. The final output is a single specification. In later work, they show that debugging even these cored specifications can be difficult, and so provide `Cable`, a new tool to assist in the process [5].

Shoham *et al.* [54] mine by using abstract interpretation where the abstract values are specifications. Their technique is inter-procedural and uses alias analysis to avoid exploring false paths. They offer a new algorithm to summarize abstract traces to effectively rule out incorrect behavior: they learn automata-based specification abstractions and then merge similar usage patterns. Their work takes a complementary approach to obtaining precision in a static specification miner: instead of focusing on software engineering metrics related to how the code was created, they use program analyses to analyze the program precisely.

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. In contrast, we explicitly seek to minimize the amount of necessary user input and strive for almost total automation of the mining process. This is the motivation behind minimizing our false positive rate as much as possible. However, our initial results suggest that human guidance could be incorporated in our process to great effect: a human might annotate which portions of the source code constitute the test suite, for example. Unlike `Strauss`, `WML_dynamic`, `Perracotta`, `JIST`, and Shoham *et al.*, we produce multiple candidate specifications rather than a single specification. Even though they may encode important program behavior, complex specifications can be difficult to debug and verify [5] Unlike `Perracotta` or Gabel and Su, we cannot mine more complicated templates, such as three state specifications. Like `ECC`, `WN`, `Perracotta`, and Gabel and Su, our miner is scalable.

The `Daikon` system [24] is an example of a dynamic invariant-detection system. In `Daikon`, the program under consideration is instrumented so that at various program points, the validity of potential algebraic relations on program variables (e.g., `x = y`, `y <= z` 3+, etc.) is checked and recorded. The program is run on an indicative workload, and any relations that are true on all runs are considered as candidate program invariants. The invariants learned by `Daikon` are not directly comparable to the temporal-safety specifications considered here, and in general the Daikon approach is complementary. Our approach does not require indicative workloads or program transformations, but it will never find useful invariants such as `y <= z` 3+. Such invariants are quite useful as pre- and post-conditions for full formal program verification.

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 two-state candidates has a false positive rate under 90%. As an example, Gabel and Su's approximate algorithm produces 13,608 candidate specifications on Hibernate alone; their strict algorithm produces more [29]. In contrast, our precise miner presents a 5% false positive rate that still finds over 250 violations.

6.2 Previous Work in Software Quality Metrics

There is a large amount of previous research devoted to automatically analyzing source code's quality[1]. We seek to provide a broad overview here of both long-established measures as well as more recent work.

Perhaps the most well-known software metric is the McCabe complexity metric [46], which measures the complexity of a piece of software or a software module using graph-theoretic approaches. The metric seeks to measure the amount of decision logic in a given piece of software. McCabe defines M, the *cyclomatic complexity* of a structured program, as:

$$M = E - N + 2P \tag{6.1}$$

where E is the number of edges in the program's control flow graph, N is the number of nodes in the graph, and P is the number of connected components. M is an upper bound for the number of test cases necessary to achieve complete coverage, and a lower bound for the number of total possible paths through the control flow

[1] Industry has adopted some of this work to try to impose minimum standards for code quality, to mixed effect

graph. This metric has been adopted by some industrial practitioners to measure code quality and impose limits on code complexity; the Eclipse IDE will notably calculate it for you, in the same way that many word processors will calculate the Flesh-Kincaid metric for text readability. It can be used, at least, to suggest areas of code that may need additional programmer attention.

This metric focuses on the graph structure of the program; our metrics are limited to source-level features or features that can be extracted from other software engineering artifacts such as the source control repository. We do not focus on the graph structure of the program. However, future experiments might explore the usefulness of cyclomatic complexity as a trustworthiness metric and compare it to the other metrics explored here.

Halstead *et al.* developed *Software Science*, wherein he attempted to develop fixed rules and measurements for software based on easily measurable, universal attributes of source code [33]. He defined the following measures:

$$n_1 = \text{number of distinct operators in a program} \tag{6.2}$$

$$n_2 = \text{number of distinct operands in a program} \tag{6.3}$$

$$N_1 = \text{number of operator occurrences} \tag{6.4}$$

$$N_2 = \text{number of operand occurrences} \tag{6.5}$$

Based on these primitive measures, he developed a number of derivative measures measures. The following list is a selection of these measures:

$$\texttt{Vocabulary}(n) \quad = \quad n_1 + n_2 \tag{6.6}$$

$$\texttt{Length}(N) \quad = N_1 + N_2 = \quad N * \texttt{log}_2(n_1 + n_2) \tag{6.7}$$

$$\texttt{Volume}(V) \quad = N * \texttt{log}_2(n) = \quad N * \texttt{log}_2(n_1 + n_2) \tag{6.8}$$

$$\texttt{Level}(L) \quad V^*/V = \quad N * \texttt{log}_2(n_1 + n_2) \tag{6.9}$$

$$\texttt{Effort}(E) \quad = \quad V/L \tag{6.10}$$

$$\texttt{Faults}(B) \quad = \quad V/S^* \tag{6.11}$$

$$\texttt{Time}(T) \quad = \quad E/S \tag{6.12}$$

Vocabulary and Length are the closest measures to our traditional notion of code length of lines of code. Volume represents the number of bits required to encode the program in an alphabet using one character per operand/operator. Level measures the level of abstraction of the procedure, characterized as the inverse of the difficulty of coding an algorithm or program. Effort is meant to measure the number of elementary mental discriminations, or psychological moments (S) required to code the program. A moment of psychological time is defined by psychologist John Stroud as a discrete burst of mental activity taking place in the brain [32]. Faults attempts to predict the number of faults in a program, and Time approximates the amount of time needed to code a given program, module, or algorithm.

These measures, while intuitively appealing, ultimately did not prove to hold in practice [34]. These metrics use a small class of measures (operators and operands) to attempt to predict a larger number of code features (Effort, Faults, Time, etc). Our metrics differ from these in that they are derived from software engineering artifacts in addition to the source code and they are attempting to predict one particular feature: whether or not a candidate specification is a true or false positive. In addition, our model does not assume an *a priori* combination of features; we use linear regression to determine the best combination of our metrics.

Albrecht developed another technique for measuring the size of an information system called function point analysis (FPA) [2]. The general goal of this technique is to measure the function value delivered to a user or customer that is independent of the language or technology used. To measure productivity, one defines a product and a cost, where the product is the functional value that a piece of software delivers to the user. The functional user requirements are categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. The number of each of these objects are weighted by their relative value to the customer, and their weighted sum is adjusted according to several factors. The functional size of a program can be used as input to several project and organizational decisions, such as determining an application's development or maintenance budget, the productivity of a software team, the software size or complexity, or necessary testing effort, where the number of necessary test cases is equal to the number of function points in a piece of software.

FPA differs from our work in several ways. It seeks to measure code utility from the point of view of value delivered to the user; our work does not consider usefulness of code. Function points measure productivity, which is distinct from trustworthiness. While this type of analysis measures features beyond the simple source code features from software science, it does not look at external software engineering artifacts such as source control in the same way our work does. However, like cyclomatic complexity, the number of function points in a piece of code might provide informative input to our trustworthiness model. However, it does require more programmer intervention than we consider ideal.

Chidamber and Kemerer proposed a metrics suite for object oriented design to help manage the process of object-oriented design and program development [15]. They proposed and evaluated six design metrics: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC), and lack of cohesion among methods (LCOM). These metrics describe program complexity, like the others do, and do not depend on additional software engineering artifacts. Basili *et al.* found that all but LCOM correlated with defects in a class [8]. Other researchers found similar relationships between these OO complexity metrics and software quality [56, 57], where quality is defined as "absence of defects." As with the other metrics suites, we may be able to incorporate these measures of code complexity into our trustworthiness model to either evaluate the usefulness of these metrics or generally improve the accuracy of our model.

More recent work has looked at similar issues to those we consider here. Nagappan and Ball analyzed the relationship between software dependences, code churn, and post-release failures in the Windows Server 2003 operating system [47]. They found that a combination of dependences and relative churn between dependent code modules provided strong predictive power for post-release failures. Their conception of code churn code is more sophisticated than ours in that they define *relative* code churn, or the amount of churn in one module as compared to another, dependent module, and show it to be far more predictive of future errors than *absolute* churn. We use *absolute* churn as a metric in our work. This work suggests that more sophisticated measures of churn might be more predictive; future experimentation might evaluate this hypothesis. We also ignore the relationships between source code modules, though an application of this idea to trustworthiness is less obvious. Fenton *et al.* explored the use of several metrics for fault prediction in code, and found that size and complexity do not relate to fault density, and they found that popular complexity metrics do not seem to correlate to fault density [26]. Graves *et al.* similarly attempt to predict errors in code by mining source control histories [31].

Like our work, these studies use features independent of the source code - module dependencies and source control - to make predictions about code. Unlike our work, they define quality as "absence of defects"; we define it as "adherence to specifications of correct behavior." The similarity of these definitions suggest that our use of detected errors as a proxy for specification utility may be valid in practice. Generally, this related work provides support for our claim that there is a relationship between code churn, complexity metrics, and code quality. This work also suggest several additional metrics, notably of code complexity, that may be of use to our trustworthiness model. No other work, to our knowledge, combines as many distinct measures of code trustworthiness from such a varied array of software artifacts. However, the related research suggests that our approach is reasonable.

Chapter 7

Future Work and Conclusions

In this chapter, we lay out possible avenues for future research and summarize our work with concluding remarks.

7.1 Future Work

This work suggest multiple avenues for extension and further research. First, we could explore different measurements for code trustworthiness or compare our own to established metrics, such as those outlined in Chapter 6. Another natural next step is determining how well the trustworthiness-based model for specification mining extends to larger specifications and more complicated patterns. The basic `(ab)*` template we mined here is useful in practice, but really only scratches the surface of the more complicated specifications written and used by practitioners and error-finding tools. Second, we might examine our heuristic for specification utility. “Number of violations found” may represent a reasonable characterization of utility for our purposes, but a more complete treatment of the issue might examine fault severity or explore other measures of specification utility, such as usefulness in documentation or code refactoring. Third, we focused on creating as automatic a miner as possible, with the hopes of mitigating the need for programmer intervention. This may be helpful for automatic applications. However, there are many other potential uses for specifications. We might study the influence a programmer's intervention has on the mining process to either guide mining or provide insight as to important measures of code trustworthiness. Our comparative results suggest that testing code may be particularly informative to mining. We could evaluate what effect "knowing" (either through programmer annotation or inference) which portion of the source code constitute testcases has on the accuracy of specification mining. This work also raises the question of how automatic mining compares to specification generation by an informed expert with access to the source code. This suggests interesting comparative experiments in which we compare the output of our miner and the output of a programmer, and perhaps explore how helpful the output of our tool is when used by an informed expert developing more comprehensive specifications or trying to understand the code.

Our trustworthiness metrics may have applications outside the mining of simple two-state specifications. Future work may explore the ways in which the idea of trustworthy code can be brought to bear on other problems. For example, we have spent time exploring the use of genetic program to automatically repair programs. Code trustworthiness may offer useful guidance to a genetic algorithm that tries to repair a program by randomly selecting portions of the code to change. Untrustworthy code may be more likely to require changes, and trustworthy code might be more likely to contain reasonable fixes. We might try to correlated trustworthy code with other interesting features, such as error density, and adapt the idea for use in a bug-finding tool. Finally, the metrics themselves bear comparison to other accepted and novel measures of code quality, particularly those currently in use in industry.

7.2 Conclusions

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 some existing automatic specification miners have 90–99% false positive rates. We wished to create a better specification mining technique that finds useful specifications with a lower rate of false positives.

Our primary claim in this work is that not all parts of a program are equally indicative of correct program behavior. We believe that a major problem with previous specification miners is that they count the contribution of all code equally when calculating the probability that an event pair is a true specification. Instead, we believe that code should be weighted by the likelihood that it is correct or that it conforms to program APIs.

We encode this intuition using trustworthiness metrics such as predicted execution frequency, measurements of copy-paste code, code churn, software readability or path feasibility. We use these metrics to build a specification miner by learning a linear model using previously published results. Statistical analysis on this model yields several interesting conclusions. First, these metrics are generally independent code measurements and thus each independently contribute to the overall model. The exception to this claim is Author Rank: our formulation of this metric does not substantively effect the overall model, and shows a moderate correlation with the code churn metric. However, the model showed some variability across the different benchmarks, and there are software projects for which the rank of an author is a useful predictive measurement for whether code adheres to correct specifications. Size and structure of a program's source code also seems to affect the model. For example, our metrics were more consistent on the largest benchmark, `ptolemy`, while the size and layout of the test suite affected the utility of various metrics on the `axion` benchmark.

We further show that these metrics can be used to improve the performance of existing trace-based miners by focusing on trustworthy traces. If we use the metrics to sort the traces and use an off-the-shelf miner on the most trustworthy of the input, we can dramatically lower the false positive rate and still find useful specifications. Very good true positive results can be found by simply passing the top 25% of traces to an existing miner, which may have important implications in collaborative specification mining. We finally concluded that equivalent results can be obtained using only 60% of the input, invalidating the claims of previous work that more input is necessarily better for specification mining.

We used 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 to programmers. 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 (and thus potential errors). 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 an important first step towards utility in an automated setting.

Bibliography

[1] ISO/IEC 13568:2002. *Information technology – Z formal specification notation – Syntax, type system and semantics*. ISO, Geneva, Switzerland.

[2] Allan J. Albrecht. Measuring application development productivity. In I. B. M. Press, editor, *IBM Application Development Symposium*, pages 83–92, October 1979.

[3] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Principles of Programming Languages*, 2005.

[4] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.

[5] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. Debugging temporal specifications with concept analysis. In *Programming Language Design and Implementation*, pages 182–195, 2003.

[6] 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, April 2006.

[7] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Formal Methods for Components and Objects*, pages 1–22, 2004.

[8] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[9] Raymond P. L. Buse and Westley Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[10] Raymond P. L. Buse and Westley Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.

[11] Raymond P. L. Buse and Westley Weimer. The road not taken: Estimating path execution frequency statically. Technical report, University of Virginia, 2008.

[12] Rod Chapman and Peter Amey. Spark 95 – the spade ada 95 kernel (excluding ravenspark). Technical report, Bath, UK, 2008.

[13] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium*, 2004.

[14] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[15] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[16] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 762–765, 2000.

[17] Manuvir Das. Formal specifications on industrial-strength code-from myth to reality. In *Computer-Aided Verification*, page 1, 2006.

[18] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[19] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[20] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*, pages 59–69, 2001.

[21] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[22] Dawson Engler, Ben Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[23] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating System Principles*, pages 57–72, 2001.

[24] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[25] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[26] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[27] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Language Design and Implementation*, pages 234–245, 2002.

[28] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, page 120, 1996.

[29] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *International Conference on Software Engineering*, pages 51–60, 2008.

[30] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[31] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[32] Maurice Halstead and Rudolf Bayer. Algorithm dynamics. In *ACM'73: Proceedings of the annual conference*, pages 126–135, New York, NY, USA, 1973. ACM.

[33] MH Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[34] Peter G. Hamer and Gillian D. Frewin. M.h. halstead's software science - a critical examination. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 197–206, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[35] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[36] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA Companion*, pages 132–136, 2004.

[37] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

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

[39] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *Symposium on Theory of Computing*, pages 273–282, 1994.

[40] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[41] Ted Kremenek and Dawson R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium*, pages 295–315, 2003.

[42] O. Kupferman and R. Lampert. On the construction of finite automata for safety properties. In *Automated Technology for Verification and Analysis*, volume 4218, pages 110–124, 2006.

[43] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. *SIGPLAN Not.*, 40(1):364–377, 2005.

[44] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *USENIX Security Symposium*, pages 271–286, August 2005.

[45] Donna Malayeri and Jonathan Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, pages 200–220, 2006.

[46] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[47] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement*, pages 364–373, 2007.

[48] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, May 2002.

[49] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[50] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* John Wiley & Sons, Inc., 1996.

[51] C. V. Ramamoothy and W-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[52] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices.* 2003.

[53] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.

[54] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *International Symposium on Software Testing and Analysis*, pages 174–184, 2007.

[55] Murray Stokely, Sagar Chaki, and Joël Ouaknine. Parallel assignments in software model checking. *Electr. Notes Theor. Comput. Sci.*, 157(1):77–94, 2006.

[56] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[57] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen. An empirical study on object-oriented metrics. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 242, Washington, DC, USA, 1999. IEEE Computer Society.

[58] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[59] Westley Weimer and Nina Mishra. Privately finding specifications. *IEEE Trans. Software Eng.*, 34(1):21–32, 2008.

[60] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 419–431, 2004.

[61] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, 2005.

[62] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis*, 2002.

[63] Yichen Xie and Alexander Aiken. Saturn: A SAT-based tool for bug detection. In *Computer Aided Verification*, pages 139–143, 2005.

[64] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *International Conference on Software Engineering*, pages 282–291, 2006.