International Journal of Engineering & Technology Research
Volume-1, Issue-1, July – September, 2013, pp. 89-97, © IASTER 2013
www.iaster.com, ISSN Online: 2347-4904, Print: 2347-8292

Improving the Code Quality by Specification Mining Using Classification and Prediction

Aiswaryadevi V J, Naveena S, Amitabh Wahi
Department of Information Technology
Bannari Amman Institute of Technology, Sathyamangalam, India

ABSTRACT

The process of testing involves various methods and techniques to improve the quality of the code. Testing is the process of executing a program with the intent of finding any errors. Formal specifications help with program testing, optimization, refactoring, documentation, debugging and repair. It is difficult to write manually, and automatic mining techniques suffer from 90-99 percent false positive rates. A temporal-property miner has been enabled to measure the quality of the code. Code churn, author rank, code clones and code readability are some of the main the quality metrics. In code clones, the detector is based on the randomized algorithms to solve the string-matching problem and some of its generalizations. The simhash algorithm represents the string's length by much shorter length, and achieves the efficiency. Accuracy of the quality metrics has been achieved.

Keywords: *Specification Mining, Software Engineering, Code Quality, Feature Extraction*

I. INTRODUCTION

Many large, pre-existing software projects are not yet formally specified. Formal program specifications are difficult for humans to construct, and incorrect specifications are difficult for humans to debug and modify. Accordingly, researchers have developed techniques to automatically infer specifications from program source code or execution traces. These techniques typically produce specifications in the form of finite state machines that describe legal sequences of program behaviours. Unfortunately, these existing mining techniques are insufficiently precise in practice. Some miners produce large but approximate specifications that must be corrected manually. As these large specifications are imprecise and difficult to debug, a larger set of smaller and more precise candidate specifications are produced that may be easier to evaluate for correctness. An automatic specification miner that balances true positives as required behaviours with false positives non-required behaviours is developed. The code clones are detected based on the randomized algorithms to solve the following string-matching problem and some of its generalizations. The algorithms represent strings of length much shorter strings called fingerprints, and achieve their efficiency by manipulating fingerprints instead of longer strings. The algorithms require a constant number of storage locations, and essentially run in real time. They are conceptually simple and easy to implement. This paper proposes the sim-hashing algorithm and naïve Bayesian algorithm to construct the miner and predict its efficiency. The efficiency of the miner can be analysed by using sim-hashing algorithm. In the next section, Section2 describes the related work, Section3 briefly describes about mining algorithm, classification algorithm been used.

II. BACKGROUND

An automatic specification miner balances the true positives as required behaviours with false positives as non-required behaviours. A true specification describes the behaviour that may not be violated on any program trace or the program contains an error. A candidate that is violated in the high-quality code but adhered to in the low-quality code is less likely to represent required behaviour than one that is adhered to on the high-quality code but violated in the low-quality code. Lightweight, automatically collected quality metrics over source code can usefully provide both positive and negative feedback to a miner attempting to distinguish between true and false specification candidates.[1]

The main contributions of this paper are[2]:

- Lightweight, automatically collected software features that approximate source code quality for the purpose of specification mining are evaluated for the predictive miners.
- Code quality metrics are lifted to metrics on traces and the utility of the lifted metrics are measured empirically.
- Two novel specification mining techniques use the automated quality metrics to learn the temporal safety specifications avoiding false positives.

A. Temporal Safety Specifications

A partial-correctness temporal safety property is a formal specification of an aspect of required or correct program behaviour. Those properties are referred to as “specifications”. Such specifications can be represented as finite state machine that encodes valid sequences of events. Fig.1 shows the source code and a specification relating to SQL injection attacks. In this example, one potential event involves reading untrusted data from the network, another sanitizes input data and a third performs the database query.

Typically each important resource is tracked with a separate finite state machine [6] that encodes the specification that applies to its manipulation. A program execution adheres to a given specification if and only if it terminates with the corresponding state machine in an accepting state. Otherwise, the program violates the specification and contains an error. This type of partial correctness-specification correctness is distinct from and complementary to, full formal behaviour specifications.

A two-state specification is one of the simplest and most common types of temporal specification. It states that an event a must always eventually be followed by event b. This corresponds to the regular expression (ab)*, which is written as <a,b>. Mining FSM specifications as shown in the Fig 1 with more than two-state specifications is historically imprecise and debugging such specifications manually is difficult. Two-state temporal properties are by definition more limited in their expressive power [2], [1].They can be used to describe important properties such as <**open,close**>, <**malloc,free**> and <**lock,unlock**>.

B. Specification Mining

Specification mining seeks to construct formal specifications of correct program behaviour by analysing actual program behaviour. Program behaviour is typically described in terms of sequences of function calls or other important events. Examples of program behaviour may be collected statically from source code or dynamically from instrumented executions on indicative workloads. A specification miner examines such traces and produces candidate specifications, which must be verified by a human programmer.

Mining even these two simple two-state specifications remains difficult [5]. Given the large number of candidate <a,b> pairs generated by even a restricted set of program traces, determining which pairs constitute valid policies is non-obvious. Most pairs, even those that frequently occur together (such as **<print,print>** or more insidiously, <**hasNext,getNext**>), do not represent required pairings: A program may legitimately call **hasNext** without ever calling **getNext.**

Engler et al. note that programmer errors can be inferred by assuming that the programmer is usually correct [2]. In other words, common behaviour implies correct behaviour, while uncommon behaviour may suggest a policy violation (a principle that similarly underlies modern intrusion detection.

III. GENERAL TERMS AND DEFINITIONS

Code quality information can be gathered either from the source code itself or from related artefacts, such as version control history. By augmenting the trace language to include information from the software engineering process, the quality of every piece of information supporting a candidate specification is evaluated. Section A describes the quality metrics used in the software engineering process whereas Section B describes the mining algorithm used.

A. Quality Metrics

Two sets of metrics are evaluated. The first set consists of seven metrics chosen to approximate code quality. The second set of metrics consists of previously proposed measures of code complexity. The metrics in the first set (“quality metrics”) are:

Code Churn: Previous research has shown that frequently or recently modified code is more likely to contain errors [5], perhaps because changing code to fix one defect introduces another, or because code stability suggests tested correctness. The churned code is also likely to adhere to specifications. Version control repositories are used to record the time between the current version and the last version for each line of code in wall clock hours. The total number of revisions to each line is also tracked. Such metrics can be normalized or given as absolute ranges.

Author Rank: The hypothesis used here is that the author of a piece of code influences its quality. A senior developer who is familiar with the project and has performed many edits may be more familiar with the project’s invariants than a less experienced developer. Source control histories track the author of each change. The rank of an author is defined as the percentage of all changes to the repository ever committed by that author. Rank of the last author is recorded to touch each line of code. While author rank may be led astray by certain methodologies (e.g., some projects may have a small set of committers that commit on behalf of more than one author [8]; others may assign more difficult and thus error-prone tasks to more senior developers).

Code Clones: The hypothesis used here is that, the code that has been duplicated from another location may be more error prone because it has not necessarily been specialized to its new context (e.g., copy-paste code) and because patches to the original code may not have propagated to the duplicate. Research has shown that cloned code is changed consistently a mere 45-55 percent of the time [6]. While not at all code cloning is harmful [2], perhaps because common code clones may be more comprehensively tested, since that duplicated code does not represent an independent correctness argument: If **print** follows **hasnext** in 20 duplicated code fragments, it is not necessarily 20 times as likely that <**hasNext**,**print**> **is a true specification.**

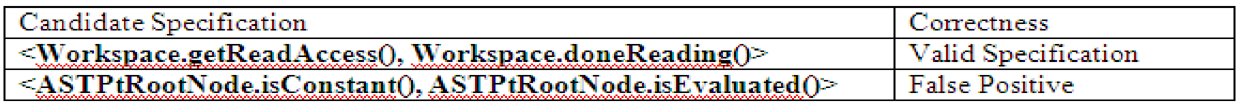
Code readability: Buse and Weimer developed a code metric trained on human perceptions of readability or understandability [9]. The metric uses textual source code features-such as number of characters, length of variable, names, or number of comments-to predict how humans would judge the code's readability.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

Path feasibility: Our specification mining technique operates on statically enumerated traces, which can be acquired without indicative workloads or program instrumentation. Infeasible paths are an unfortunate artefact of static trace enumeration and it can be claimed that they do not encode programmer intentions. Merely discounting provably infeasible paths may confer some benefit to the mining process. However, infeasible paths may suggest pairs that are not specifications: A programmer may have made it impossible for b to follow a along a path, suggesting that <a,b> is not required behaviour. Static paths are preferred first because they are both easier to obtain and more complete than dynamic paths. In addition, it is hypothesized that static paths

Path frequency: Common paths that are frequently executed by indicative workloads and test cases are more likely to be correct. First, the programmer may reason more thoroughly about the "common case" and second, highly tested code is less likely to contain errors. combined with symbolic execution can provide additional useful information about behaviour; the programmer believes to be impossible. The feasibility of a path is measured using symbolic execution; a path is infeasible if a theorem proper reports that its symbolic branch guards are inconsistent. Path feasibility is expressed as one of {0, 0.5, 1}; o denotes an infeasible path. 1 a required path and 0.5 a path, that may or may not be feasible or required.

Path density: A hypothesizes used here is that a method with more possible static paths is less likely to be correct because there are more corner cases and possibilities for error. Path density is defined as the number of traces possible to enumerate in each method, in each class and over the entire project. A low path density for traces containing paired events ab and a high path density for traces that contain only a suggest that <a,b> is a likely specification. Path density is expressed in whole numbers and can be normalized to maximum number of enumerated paths.

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



Candidate Specification	Correctness
<Workspace.getReadAccess(), Workspace.doneReading()>	Valid Specification
<ASTPtRootNode.isConstant(), ASTPtRootNode.isEvaluated()>	False Positive

Cyclomatic Complexity: McCabe defined cyclomatic complexity [44] to quantify the decision logic in a piece of software. A method's complexity is defined as M=E-N+2P, where E is the number of edges in the method's control flow graph, N is the number of nodes and P is the number of connected components. There is no theoretical upper bound on the complexity of a method. The complexity of an intra-procedural trace is the complexity of its enclosing method. Previous work suggests that Cyclomatic complexity correlates strongly with the length of a function and does not correlate well with errors in code [7]. Despite this, Cyclomatic complexity remains in industrial use. Another hypothesize followed is that complexity will not helpfully contribute to our specification mining model.

CK metrics: Chidamber and Kemerer proposed s suite of theoretically grounded metrics to approximate the complexity of an object-oriented design [2]. The following six metrics apply to a particular class (i.e., a set of methods and instance variables):

- **Weighted methods per class (WMC).** Number of methods in a class, weighted by a user-specified complexity metric. Common weights selected in practice are 1, the method length, or the method's Cyclomatic complexity.
- **Depth of inheritance tree (DIT).** Maximal length from the class to the root of the type inheritance tree.
- **Number of children (NOC).** Number of classes that directly extends this class.
- **Coupling between objects (CBO).** Number of other objects to which the class is coupled. Class A is coupled to Class B if one of them calls methods or references an instance variable defined in the other.
- **Response for a class (RFC)**. Size of the response set, defined as the union of all methods defined by the class and all methods called by all methods in the class.
- **Lack of cohesion in methods (LOCM).** Methods in a class may reference instance variables in that class. P is the set of methods in a class that share in common at least one instance variable with at least one other class method. Q is the set of methods that do not reference instance variables in common. LOCM is $|P – Q|$ if $|P - Q| > 0$ and o otherwise.

The CK metrics are also sometimes used in industry to measure design or system complexity. Research on their utility has yielded mixed results-studies have correlated subsets of the metrics with fault proneness, though they do not tend to agree on which subsets are predictive [7], [2], [4].

B. Mining Algorithm Details

The two mining algorithm extends the previous WN miner [8,9], notably by including quality metrics from section III A. The two mining algorithm is constructed and compared with their efficiency. The miner takes the input as follows:

1. The program source code P. The variable ℓ ranges over source code locations. The variable l represents a set of locations.
2. A set of quality metrics M1……Mq. Quality metrics may map either individual locations ℓ to measurements, with Mi(ℓ) € IR (e.g., code churn) or entire traces to measurements, where Mi(l) € IR (e.g., path feasibility).
3. A set of important events $\sum$, generally taken to be all of the function calls in P. The variables a, b, etc., are used to range over $\sum$.

The miner produces as output a set of candidate specifications C = {<a,b> | a should be followed by b}.Validity of the candidate specification are evaluated manually.

The Rabin-karp algorithm first statically enumerates a finite set of intra-procedural traces in P. Because any non-trivial program contains an infinite number of traces, that requires an enumeration strategy. A breadth-first traversal of paths for each method m in P is performed. The first k paths are emitted, where k is specified by the programmer. Larger values of k provide more information to the mining analysis with a corresponding slowdown. Typical values are $10 \leq k \leq 30$. To gather information about loops and exceptions while ensuring termination, it is passed through loops no more than once. Thus a path through a loop represents all paths that take the loop at least once; a non-exceptional path represents all non-exceptional paths through that method, etc. This process produces a set of traces T. A trace t is a sequence of events over ∑; each event corresponds to a location ℓ.

The miner lifts the quality metrics from individual locations to traces, where necessary. This lifting is parametric with respect to an aggregation function A: P (IR) $\rightarrow$ IR. The functions like min, max, span and average are used to summarize quality information over a set of locations *l*. The functions and terminologies used are shown in the Fig 3.

The Naïve Bayesian algorithm is used to classify the metrics and traces and predict the accuracy value.

For comparison, Simhash algorithm is used to implement the mining technique using hash functions. Simhash has been used successfully in different areas of research, such as text retrieval, web mining and so on [8,9]. Simhash is a state of art fingerprint based data similarity measure for matching the text with hash patterns and retrieves those metrics. SimCad is a tool which uses the simhash algorithm and works. The Code clones metric is mainly focused on comparison with the mining techniques because the Simhash algorithm is proven efficient when used in detecting code clones. Fig 4 represents the percentage of accuracy obtained through Rabin-Karp versus percentage of accuracy obtained through Sim-Hash.

A multi-level indexing scheme is also used to organize the per-processed code base on which the code detection algorithm is applied. The indexing scheme speeds up the potential search and allows the approach to be scalable by maintaining the indices in persistent storage (e.g., a database).Sim hash algorithm works with three phases: pre-processing, detection and output generation.

1. Pre-Processing

The pre-processing step sets up the environment and organizes the data over which the detection algorithm is applied. There are four sub-steps in pre-processing as shown in the Fig.5. The first two sub-steps: fragment extraction with pretty printing and source transformation/normalization. The remaining two sub-steps: simhash generation and indexing are discussed in detail below.

- **Simhash generation**
  Fingerprinting is a well-known approach in data processing that maps an arbitrarily large data item to a much shorter bit sequence that uniquely identifies the original data. In this approach, each code block will be transformed into an n-bit fingerprint, the simhash value of that block.

- **Multi-level indexing**
  A two level indexing scheme has been introduced to organize the simhash data generated. The goal of data organization is to speed up the neighbour search query in string matching. The indexing is first done by the size of the code fragment in terms of lines of code and then by the number of 1-bits in the binary representation of the

ISSN (O) 2347- 4904
(P) 2347-8292

simhash fingerprint as shown in the Fig 4. Thus, each of the first level indices points to a list of second level indices points to a list of second level indices. Each of the second level indexes in turn points to a list of simhash values having the same number of 1-bits in its binary form.

The Naïve Bayesian algorithm is used to classify and predict the accuracy of the metrics mined.

IV. RESULTS

Quality metrics mined by the Rabin-Karp algorithm for the input code base is tabulated as below:

Table 1.Quality Metrics mined by Rabin-Karp algorithm

Features Extracted	**Metrics collected for ecj37**
Code Churn	1.7838616714697406
Code Clones	6.0
Author Rank	87.0
Readability	24271.0
WMC	0.0
DIT	0.0
NOC	0.0
CBO	0.0
RFC	0.0
LCOM	0.0
Coupling	0.0
NPM	0.0
Path feasibility	6562.0
Path frequency	7458.001
Path densitiy	6125.0

Quality metrics mined by the SimHash algorithm for the input code base is tabulated as below:

Table 2.Quality Metrics mined by SimHash Algorithm

Features Extracted	**Metrics collected for ecj37**
Code Churn	1.7838616714697406
Code Clones	17.0
Author Rank	87.0
Readability	24271.0
WMC	0.0
DIT	0.0
NOC	0.0
CBO	0.0
RFC	0.0
LCOM	0.0
Coupling	0.0
NPM	0.0
Path feasibility	6562.0
Path frequency	7458.001
Path densitiy	6125.0

ISSN (O) 2347- 4904
(P) 2347-8292

Accuracy obtained by the Naive Bayesian classification is brought under 89 percent in case of Rabin-karp mined specifications and below 96 percent in the case of SimHash.

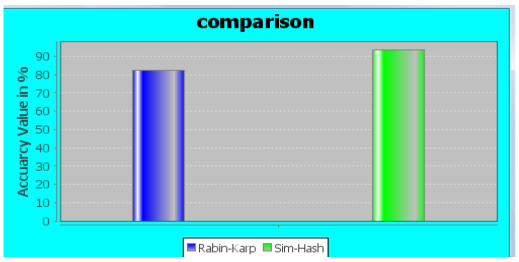


Figure 2. Accuracy Comparison

Accuracy values against each of the input code base obtained is plotted as bar chart in the Fig 2.The accuracy values are tabulated as well in the table 3.

Table 3.Accuracy comparison

Code Base	RABIN-KARP	SIM-HASH
Ecj37	83.90915395976744	93.21452672190284
Packge	82.86230789063268	92.07847438367307
src	82.25780680118972	92.67139731547512
Ecj36	82.61276741204576	93.58769862547894

F-Measure Analysis

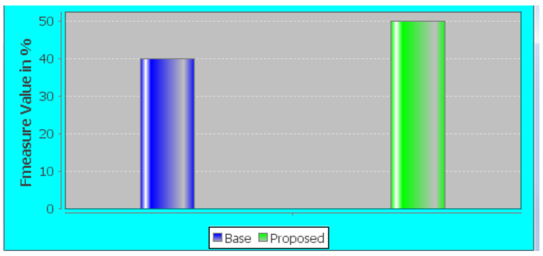


Figure A3.13 shows the F-Score measure of Rabin-Karp algorithm versus F-Score measure of Sim-Hash algorithm.

Figure A3.13 F-Score analysis

True Positive cases and false positive cases are compared using the graph represented by the figure 3.

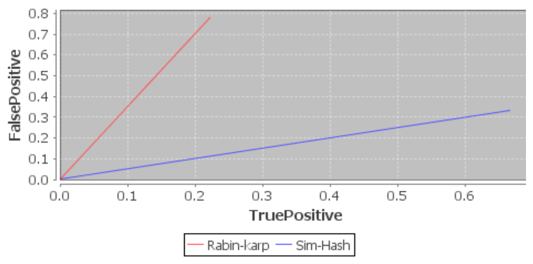


Figure 3. True Positive Vs False Positive

F-Score Analysis

Depending on the F-score analysis, the false positive rate is minimized to be fewer than 3 percent when focused on recall measure and it is maintained fewer than 58 percent when focused on the precision. When focused on the precision measure, it is observed that 60 percent of the false positive rate is obtained. It has been noted that about 20 percent of the false positive rate is reduced.

Table 7.2.5 F-Score analysis

Input Package	Precision	Recall	Fscore	FP rate
Src	0.68	0.40	0.92	0.006
Package	0.70	0.65	0.50	0.002

Table 7.2.5 shows the relation between recall rate and F-score rate of each input packages. When the recall rate increases, F-score rate decreases and hence the false positive rate is reduced further.

V. CONCLUSIONS

Measuring the code quality is becoming an important part of the software productivity. Program verification is used to measure the quality of code patterns in the source code package. As automatic program verification tools become more prevalent, specifications become the limiting factor in the program verification efforts. Candidate policies that describe the real or common program behaviour are obtained. The information about the exceptional paths of the specification traces are mined using the novel miner. Well known complexity metrics are evaluated and obtained by the specification mining. Fifteen different metrics are used to compute the quality metrics of the candidate specifications.

The metrics that are computed and obtained include predicted execution frequency, code clone detection measure, code churn readability and path feasibility. The lower rate of false-candidate specifications is achieved by increasing the performance of temporal trace based miners. The algorithm matched the coding standards with the code patterns in the source lines of code and achieved accuracy in the range of 89-94 percent. Accuracy of the traces is obtained for the performance analysis. The miner maintains a false positive rate under 89 percent. Both the algorithms are compared and rated by the accuracy values obtained.

REFERENCES

[1] Claire Le Goues, Westley Weimer., "Measuring Code quality to improve specification Mining" IEEE transactions on Software Engineering 2012, Vol 38.,pp 175-190.

[2] Claire Le Goues, Westley Weimer., *"Specification Mining with few false Positives"*,Lecture Notes in Tools and Algorithms for the Construction and Analysis of Systems Computer Science Volume, Springer 1989, 2009, Volume 5505, pp 292-306.

[3] Westley Weimer, George C. Necula., "Mining Temporal specification for error detection" Lecture Notes in Tools and Algorithms for the Construction and Analysis of Systems Computer Science Volume, Springer 1989, 2009, Volume 3440, pp 461-476.

[4] G. Ammons, R. Bod´ık, and J. Larus. Mining specifications. In Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 4–16.

[5] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer.Experimental assessment of random testing for object-oriented software. In ISSTA '07: Proceedings of the 2007 International symposium on Software testing and analysis, pp 84–94, New York, NY, USA, 2007.

[6] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International symposium on Software testing and analysis, pages 218–228, New York, NY, USA, 2002.

[7] Subramaniyam.R., "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects"IEEE transactions on software engineering 2003,Vol 29,pp 297-310.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27(2):1–25,Feb. 2001. A previous version appeared in ICSE '99, Proceedings of the 21st International Conference on Software Engineering, pp 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. ACM Transactions Software Engineering Methodology, 17(2):1–34, 2008*FLEXChip Signal Processor (MC68175/D)*, Motorola, 1996.