Quantify Program Evolution by Using Temporal Properties

[1]Enumula Karthik, [2]Dasi Sai Chand, [3]Kanjarla Devaki Nandan
[1,3]Ganapathy Engineering College Warangal
[2]Balaji Institute of Engineering and Sciences Warangal

Abstract
It is important that programmers and program maintainers understand important properties of the programs they modify and ensure that the changes they make do not alter essential properties in unintended ways. Manually documenting those properties, especially temporal ones that constrain the ordering of events, is difficult and rarely done in practice. We propose an automatic approach to inferring a target system's temporal properties based on analyzing its event traces. The core of our technique is a set of pre-defined property patterns among a few events. These patterns form a partial order in terms of their strictness. Our approach finds the strictest properties satisfied by a set of events based on the traces. 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 behaviour. 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 pre-processing 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 behaviour, which allows off-the-shelf mining techniques to learn the same number of specifications using 60% of their original input. 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.

Keywords
Temporal Properties, Specification Mining, Software Engineering, Code Metrics

I. Introduction

A common problem is ensuring that changes introduced in program maintenance do not change the program's behaviour in unexpected ways. In particular, changes should not alter important properties of the previous version of the program on which clients may rely. Unfortunately, most programs do not have specifications, so programmers modifying programs often do not know what those important properties are. We present an approach for largely automating the task of discovering important property differences. We focus on temporal properties, since they provide good opportunities for automatic inference and analysis, and because satisfying certain temporal properties is essential for the correctness of many programs, yet they are particularly hard for programmers to document and test manually.

A temporal property defines the sequence in which events take place. Temporal properties are especially important in concurrent programs in which threads interact through shared objects and messages. Writes from different threads to a shared object are mutually excluded using mechanisms such as locks to ensure that events are ordered consistently. While such properties are fundamental to program correctness, they are rarely documented or specified. Even when they are, it is extremely hard to assure them by inspection or testing due to the huge number of ways threads might interleave with each other.

In order to prove that a computer program is mature and free of bugs, and that Software Requirements Specifications (SRS), have been met is necessary to have a strategy that supports this process. The goal for any software project is to accomplish the above mentioned requirements, which means to achieve the best quality possible. Historically, the word "quality" has been adapted and has evolved together with the different technologies to which it has been applied. In the thirties, the metallurgical industry defined quality as a compliance to requirements; any deviation from such requirements meant loss of quality or limited trust in product quality. In the fifties, quality costs increased exponentially.

Therefore, specifications including tolerance (i.e., a deviation from perfection) were proposed. Inspections ensured that the product fell within a predefined tolerance. The goal of such inspections was to avoid corrections through the identification of product deviations from the original specification. Software development does not imply serial production costs, but is an intensive activity, and is reasonable to apply quality concepts in software development. It requires several specialists interaction and coordination during all development stages. In the following subsections, different perspectives of software quality are presented. The present project proposes a tool that allows realizing an automatic audit of source code. Its goal is to increase the quality of the source code checking if this fulfils with the specification. It also builds recommendations according to detected errors employing software metrics and finally advises the programmer how to improve his source code.

A. Software Quality

"Conformance to requirements" implies that requirements must be clearly stated such that they cannot be misunderstood. Then, in the development and production process, measurements are taken regularly to determine conformance to those requirements. The non conformances are regarded as defects—absence of quality. For example, one requirement (specification) for a certain radio may be that it must be able to receive certain frequencies more than 30 miles away from the source of broadcast. If the radio fails to do so, then it does not meet the quality requirements and should be rejected. The "fitness for use" definition takes customers' requirements and expectations into account, which involve whether the products or services fit their uses. Since different customers may use the products in different ways, it means that products must possess multiple elements of fitness for use.

B. Software Industry Requirements

Since software deals with man-made artifacts, is needed to view software development as an experimental science and build models of the artifacts and the processes by which they are manufactured.

To do this is needed to isolate and categorize the components of the discipline, define notations for representing these components, and specify the interrelationships among these components as they are manipulated. The components of the discipline consist of various processes (life cycle models, methods, techniques, tools), products (code components, requirements, designs, specifications, test plans), and other forms of experience (resource models, defect models, quality models, economic models). We need to build descriptive models of the discipline components to better understand: (1) the nature of the processes and products and their various characteristics, (2) the variations among them, (3) the weaknesses and strengths of both, and (4) mechanisms to predict and control them.

II. Motivated Example

In this chapter, we motivate our technique by presenting two candidate specifications and describing various differences between the code in which they appear. 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 looking at 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 analyse the traces through the code in which each specification appears. We hope to demonstrate there are measurements that we may perform on the code that could have helped us distinguish the valid specification from the invalid candidate. The example specifications were mined by Engler's ECC technique from Hibernate, an open source Java project that provides object persistence. This benchmark consists of 57,000 lines of code. We generated and analysed a total of 12,894 traces from the source code by generating a maximum of 10 traces per method and using symbolic execution to rule out infeasible paths.

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

Fig. 1: A Valid Specification for Hibernate

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

Fig. 2: An Specification for Hibernate

Consider two candidate specifications mined by ECC. The first, a valid specification, is shown in fig. 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 fig. 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 knowledgable developer of the system!

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

Fig. 3: Summary of Selected Features of the Example Specifications' Code Traces

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 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.

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

Fig. 4: The Single False Positive Presented by the Precise Miner

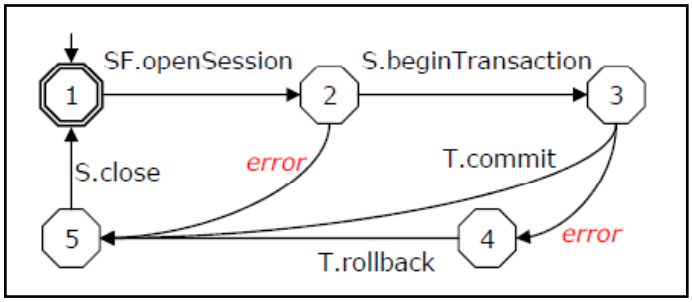

Fig. 5: A multi-state finite state machine describing the Hibernate Session API, taken from the Hibernate documentation. Our false positive corresponds to the S.beginTransaction and T.commit edges.

III. Temporal Safety Specifications

Specifications can come in many forms: English prose documents, first-order logic, or lower-level annotations like 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. Fig. 7 shows such an example specification for avoiding SQL injection attacks, based on the code in fig. 6.

Typically, each important resource, such as a lock, file handle, or socket, is tracked separately, with its own finite state machine. 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 resources' corresponding state machines in an accepting state. Otherwise, the program

violates the specification, and there may be an error in either the source code or in the specification itself. This typically requires programmer intervention to diagnose and solve the problem.

```
void bad(Socket s, Conn c) {
  string message = s.read();
  string query = "select * " +
    "from emp where name = " +
    message;
  c.submit(query);
  s.write("result = " +
    c.result());
}

void good(Socket s, Conn c) {
  string message = s.read();
  c.prepare("select * from "
    + " emp where name = ?",
    message);
  c.exec();
  s.write("result = " +
    c.result());
}
```

Fig. 6: Pseudocode for an example internet service. The bad method passes untrusted data to the database; good works correctly. Important events are italicized

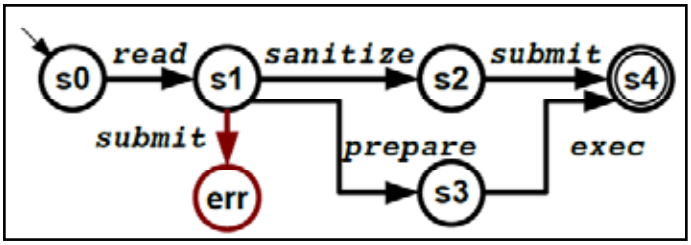

Fig. 7: Example Specification for fig. 6

The simplest and most common type of temporal specification is a two-state finite state machine. Such two-state specifications require that event a must always be followed by event b, correspond to the regular expression (ab)*, and are written <a,b>. Such specifications describe a particular aspect of correct program behaviour, 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.

IV. Our Approach

We present a new specification miner that works in three stages. First, it statically estimates the quality of source code fragments. Second, it lifts those quality judgments to traces by considering all code visited along a trace. Finally, it weights each trace by its quality when counting event frequencies for specification mining. Code quality information may be gathered either from the source code itself or from related artifacts, such as version control history. By augmenting the trace language to include information from the software engineering process, we can evaluate the quality of every piece of information supporting a candidate specification (traces that adhere to a candidate as well as those that violate it and both high and low quality code) on which it is followed and more accurately evaluate the likelihood that it is valid. Section C provides a detailed description of the set of features we have chosen to approximate the quality of code; Section D details our mining algorithm.

A. Quality Metrics

We define and evaluate two sets of metrics. The first set consists of seven metrics chosen to approximate code quality. This list should not be taken as exhaustive, nor are the quality metrics intended to individually or perfectly measure quality. Indeed, a primary thesis of this article is that lightweight and imperfect metrics, when used in combination, can usefully approximate quality for the purposes of improved specification mining. Thus, we focus on selecting metrics that can be quickly and automatically computed using commonly-available software artifacts, such as the source code or version control histories. We looked particularly to previous work for code features that correlate with fault-proneness or observed faults. In the interest of automation, we exclude metrics that require manual annotation or any other form of human guidance.

The second set of metrics consists of previously proposed measures of code complexity. We use these primarily as baselines for our analysis of metric power; this evaluation may also be independently useful given their persistent use in practice. The metrics in the first set ("quality metrics") are:

1. Code Churn

Previous research has shown that frequently or recently modified code is more likely to contain errors, perhaps because changing code to fix one defect introduces another, or because code stability suggests tested correctness. We hypothesize that churned code is also less likely to adhere to specifications. We use version control repositories to record the time between the current revision and the last revision for each line of code in wall clock hours. We also track the total number of revisions to each line. Such metrics can be normalized or given as absolute ranges.

2. Author Rank

We hypothesize that the author of a piece of code influences its quality. A senior developer who is very familiar with the project and has performed many edits may be more familiar with the project's invariants than a less experienced developer. 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. We record the rank of the last author 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; others may assign more difficult and thus error-prone tasks to more senior developers), we note that it may be automatically collected from version control histories and is a proxy for expertise, which is otherwise challenging to approximate automatically.

3. Code Clones

We hypothesize that code that has been duplicated from an 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% of the time.

4. Code Readability

Buse et al. developed a code metric trained on human perceptions of readability or understandability. The metric uses textual source code features — such as number of characters, length of variable names, or number of comments — to predict how humans would judge the code's readability. 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. We therefore hypothesize that more readable code is also more likely to adhere to specifications. We use the research prototype developed by Buse et al. to measure the readability of source code.

5. Path Feasibility

Our specification mining technique operates on statically enumerated traces, which can be acquired without indicative workloads or program instrumentation. Infeasible paths are an unfortunate artifact of static trace enumeration, and we claim that they do not encode programmer intentions. 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. We prefer static paths for our purposes first because they are both easier to obtain and more complete than dynamic paths. In addition, we hypothesize that static paths combined with symbolic execution can provide additional useful information about behaviour the programmer believes should be impossible. /.'

6. Path Frequency

We theorize that 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. We use a research tool that statically estimates the relative runtime frequency of a path through a method, normalized as a real number.

7. Path Density

We hypothesize that a method with more possible static paths is less likely to be correct because there are more corner cases and possibilities for error. We define "path density" as the number of traces it is possible to enumerate in each method, in each class, and over the entire project. 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 the maximum number of enumerated paths (30/method, in our experiments).Metrics in the second class ("complexity metrics") are:

8. Cyclomatic Complexity

McCabe defined cyclomatic complexity to quantify the decision logic in a piece of software. A method's complexity is defined as M = E − N + 2P, where E is the number of edges in the method's control flow graph, N is the number of nodes, and P is the number of connected components. 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. Despite this, Cyclomatic complexity remains in industrial use. We hypothesize that complexity will not helpfully contribute to our specification mining model.

9. CK Metrics

Chidamber and Kemerer proposed a suite of theoretically-grounded metrics to approximate the complexity of an object-oriented design. 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. The experiments in this article weight all methods equally (weight = 1).
- Depth of Inheritance Tree (DIT): maximal length from the class to the root of the type inheritance tree.
- Number of Children (NOC): number of classes that directly extend this class.
- Coupling Between Objects (CBO): number of other objects to which the class is coupled. Class A is coupled to Class B if one of them calls methods or references an instance variable defined in the other.
- Response for a Class (RFC): size of the response set, defined as the union of all methods defined by the class and all methods called by all methods in the class.
- Lack of Cohesion in Methods (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 0 otherwise.

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

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

Fig. 8: Features Used by our miner to evaluate a candidate specification <a,b> , Mi is a quality metric lifted to sets of traces.

D. Mining Algorithm Details

Our mining algorithm extends our previous WN miner, notably by including quality metrics. Our miner takes as input:

- The program source code P. The variable l ranges over source code locations. The variable l represents a set of locations.
- A set of quality metrics M1 . . . Mq. Quality metrics may map either individual locations _ to measurements, with Mi(l) R (e.g., code churn) or entire traces to measurements, where $Mi(l) \in R$ (e.g., path feasibility).

- A set of important events Σ, generally taken to be all of the function calls in P. We use the variables a, b, etc., to range over Σ.

Our miner produces as output a set of candidate specifications C = { <a,b> | a should be followed by b}. We manually evaluate candidate specification validity.

Our algorithm first statically enumerates a finite set of intra-procedural traces in P. Because any non-trivial program contains infinite number of traces, this process requires an enumeration strategy. We perform a breadth first traversal of paths for each method m in P. We emit the first k such paths, where k is specified by the programmer. Larger values of k provide more information to the mining analysis with a corresponding slowdown.

V. OpenSSL

Our second experiment considered recent versions of OpenSSL. The Secure Socket Layer (SSL) protocol provides secure communication over TCP/UDP using public key cryptography. We focus on the handshake protocol that performs authentication and establishes important cryptic parameters on both client and server sides before data are transmitted between them. OpenSSL, written in C, is a widely used open source implementation of SSL. In our experiments, we used our tool to automatically infer the temporal properties of implementation of the handshake protocol in multiple versions of OpenSSL. Chaki et. al used MAGIC, a C model checker that can automatically extract a model from a C program, to check OpenSSL's implementation of the handshake protocol.

A. SSL Handshake Protocol

Fig. 5 shows the client (left) and server (right) events in the SSL handshake protocol, which was derived from the SSL specification. The three boxes with dashed outlines contain internal states created in the OpenSSL implementation but not required in the SSL specification.

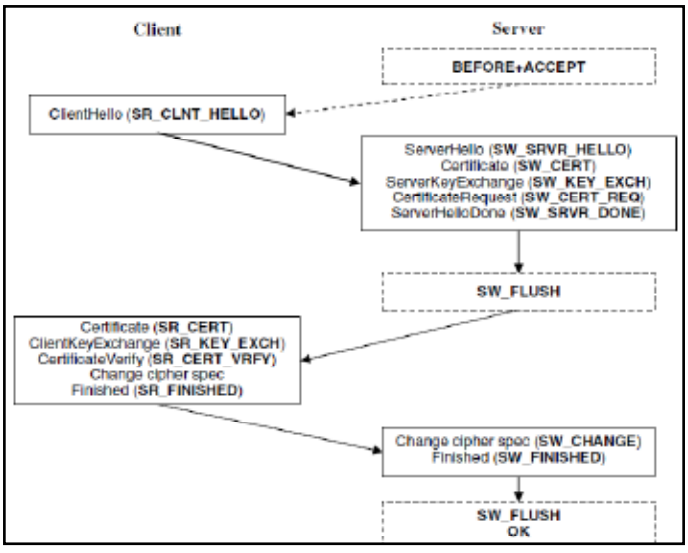

Fig. 9: SSL Handshake Protocol States

The remaining boxes contain sequences of states corresponding to messages required by the SSL handshake protocol. The server state in the OpenSSL implementation corresponding to the beginning of either receiving the corresponding message from the client or sending the message to the client for each message is shown in the parentheses. For example, if the server is in the BEFORE+ACCEPT state and receives a ClientHello message from a client, the server enters the SR_CLNT_HELLO state. Receiving the change cipher spec message is not part of the handshake process, so that message has no corresponding state change. It does have a corresponding state for sending (SW_CHANGE) in the implementation that is monitored in our experiment.

The handshake begins when the server receives a ClientHello message from a client. Then the server sends out five messages consecutively (corresponding to the states SW_SRVR_HELLO, SW_CERT, SW_KEY_EXCH, SW_CERT_REQ and SW_SRVR_DONE). Next the server enters the SR_CERT state in which it tries to read certificate from the client (whether the client sends its certificate or not depends on if the server requires one in its certificate request message). Then the server reads consecutively four messages from the client (certificate, key exchange, certificate verify, and finished). If no error occurs, the server sends out its ChangeCipherSpec message and wraps up the handshake by sending its Finished message.

As shown in the dash lined box, the server implemented several additional internal states which we also monitored. First, the server always initializes its state to BEFORE+ACCEPT at the beginning of the handshake. After sending each batch of messages, the server flushes the socket by entering the SW_FLUSH state. In the end, OK is another internal state indicating that the server cleans things up in its side and is ready for transmitting data with client. A typical event trace is shown in fig. 6.

In the OpenSSL server implementation, the handshake process is encapsulated in a method. A server starts the handshake process by calling the ssl3_accept method which implements the protocol state machine as an infinite loop that checks the current protocol state, sends or receives messages, and advances the state accordingly.

BEFORE+ACCEPT→SR_CLNT_HELLO→
SW_SRVR_HELLO→SW_CERT→
SW_KEY_EXCH→SW_CERT_REQ→
SW_SRVR_DONE→SR_CERT→
SR_KEY_EXCH→SR_CERT_VRFY→
SR_FINISHED_SW_CHANGE→
SW_FINISHED→OK

Fig. 10: A Server Event Trace of Normal Handshake Process

B. Testing process

Our experiment considered only properties of the server implementation, but in order to generate test data we needed to execute it with sample clients. We are particularly interested in analyzing the server's behaviour when the client does not follow the protocol correctly, since this is often a source of errors. The OpenSSL client implementation starts the handshake process by calling the ssl3_connect method, which implements the protocol state machine in a similar fashion as ssl3_accept. We modified ssl3_connect so that after every state it may either behave correctly or enter some randomly selected state. For example, suppose the current state is A, after finishing task X, the state should be changed to B. In our modified version, the state would correctly transition to B with 95% probability, but with 5% probability would transition into a randomly selected different state instead.

C. Faulty Clients Without Errors Generated

Next we consider the set of traces corresponding to faulty clients that (surprisingly) did not generate any error event on either the server or client. These traces recorded behavior from clients that jumped to a random state at some point during their execution, but did not lead to either the client or server reporting an error or failing to complete the handshake process.

Again all six versions agreed on the two event chains shown in Figure 11, though the number of such traces varies a little bit. These two chains closely follow the SSL specification about the normal handshake behavior of a server implementation. However, there are a few key distinctions between the patterns in fig. 10 and fig. 11.

SR_CLNT_HELLO→SW_SRVR_HELLO→
SW_CERT→SW_KEY_EXCH→
SW_CERT_REQ→SW_SRVR_DONE→SR_CERT
BEFORE+ACCEPT→SR_KEY_EXCH→
SR_CERT_VRFY→SR_FINISHED→
SW_CHANGE→SW_FINISHED→OK

Fig. 11: Inferred Alternating Chains for Nonerror Faulty Clients

We found that two Alternating properties that are present in fig. 6 do not appear in fig. 7. Instead, those event pairs had weaker patterns. First, SR_CERT and SR_KEY_EXCH satisfied the MultiCause pattern. Second, BEFORE+ACCEPT and SR_CLNT_HELLO satisfied the MultiEffect pattern. Fig. 12 shows a trace that violated our expected Alternating properties. The key distinction between this trace and the traces produced using correctly behaving clients is that the eight-event sequence appears twice. Fig. 9 shows the corresponding client events. The faulty client falsely changed its state to renegotiate (an internal state in the implementation of the client, not shown in fig. 9, since it is not part of the normal handshake process) instead of sending certificate (i.e. CW_CERT) after reading the five messages from server (ServerHello, Certificate, ServerKeyExchange, CertificateRequest, and ServerHelloDone). Then, the client started the handshake again by sending the client hello message which caused the server to repeat the hello stage of the handshake again. Although the handshake returned to normal and ended successfully after both parties repeated the hello stage twice, we found that the handshake can still be successful no matter how many times the hello stage is repeated. If a client always changes its state to renegotiate after receiving the server done message, the server and the client will enter an infinite loop.

BEFORE+ACCEPT, OK+ACCEPT,
(SR_CLNT_HELLO, SW_SRVR_HELLO, SW_CERT,
SW_KEY_EXCH, SW_CERT_REQ, SW_SRVR_DONE,
SW_FLUSH,SR_CERT,)2
SR_KEY_EXCH, SR_CERT_VRFY, SR_FINISHED,
SW_CHANGE, SW_FINISHED, SW_FLUSH, OK

Fig. 12: Trace Generated with a Faulty Client

BEFORE+CONNECT, OK+CONNECT,
CW_CLNT_HELLO, CR_SRVR_HELLO,CR_CERT,
CR_KEY_EXCH, CR_CERT_REQ,CR_SRVR_DONE,
RENEGOTIATE,
BEFORE, CONNECT, BEFORE+CONNECT,
OK+CONNECT, CW_CLNT_HELLO,
CR_SRVR_HELLO, CR_CERT, CR_KEY_EXCH,
CR_CERT_REQ, CR_SRVR_DONE,
CW_KEY_EXCH, CW_CHANGE, CW_FINISHED,
CW_FLUSH, CR_FINISHED, OK

Fig. 13: Client Trace Corresponding to the Server Trace Shown in fig. 7.

D. Faulty Client With Other Types of Error

All servers agreed on the temporal properties inferred for traces within this category. This did not lead us to detect any interesting problems, but did confirm that the server versions handled misbehaving clients consistently.

VI. Conclusion

This tool presents a new approach to automatically evaluate and provide recommendations for programmers in order to improve the source code quality, and consequently, the software product itself. We presented a prototype tool that automatically infers temporal properties of programs by analyzing test execution traces, and argued that such a tool can be a useful asset in reliable program evolution. Our experimental results demonstrate that our approach is able to automatically determine important temporal properties and identify differences that reveal interesting properties of programs.

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 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%. This research establish the foundations in order to define new quality criteria for the evaluation of software and extending the source code metrics value through computational intelligence.

References

[1] Claire Le Goues, Westley Weimer "Measuring Code Quality to Improve Specification Mining" IEEE Transactions On Software Engineering Vol.38 No.1 Year 2012.

[2] Jinlin Yang, David Evans" Automatically Inferring Temporal Properties for Program Evolution" IEEE Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04) 1071-9458/04

[3] Claire Le Goues"Specification Mining With Few False Positives" May 2009.

[4] Valentin Dallmeier ,Nikolai Knopp, Christoph Mallon, Gordon Fraser Member, IEEE, Sebastian Hack, Andreas Zeller Member, IEEE "Automatically Generating Test Cases for Specification Mining".

[5] Franco Madou, Martín Agüero, Gabriela Esperón, Daniela López De Luise"Software for Improving Source Code Quality" World Academy of Science, Engineering and Technology 59 2011.

[6] A. J. Albrecht, "Measuring application development productivity," in IBM Application Development Symposium, 1979, pp. 83–92.

[7] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in POPL, 2005.

[8] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in Principles of Programming Languages, 2002, pp. 4–16.

[9] G. Ammons, D. Mandelin, R. Bod´ık, and J. R. Larus, "Debugging temporal specifications with concept analysis," in Programming Language Design and Implementation, 2003, pp. 182–195.
[10] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. Mc-Garvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in EuroSys, 2006, pp. 103–122.
[11] T. Ball, "A theory of predicate-complete test coverage and generation," in FMCO, 2004, pp. 1–22.
[12] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of objectoriented design metrics as quality indicators," IEEE Trans. Softw.Eng., vol. 22, no. 10, pp. 751–761, 1996.
[13] R. P. L. Buse andW.Weimer, "Automatic documentation inference for exceptions," in ISSTA, 2008, pp. 273–282.
[14] P. G. Hamer and G. D. Frewin, "M.H. Halstead's Software Science - a critical examination," in ICSE, 1982, pp. 197–206.
[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in OOPSLA Companion, 2004, pp. 132–136.
[16] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" considered harmful," in WCRE, 2006, pp. 19–28.
[17] R. M. Karp and M. O. Rabin, "Efficient randomized patternmatching algorithms," IBM J. Res. Dev., vol. 31, no. 2, pp. 249–260, 1987.
[18] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," International Conference on Software Maintenance, pp. 736–743, 2001.
[19] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," IJCAI, pp. 1137–1145, 1995.
[20] J. Krinke, "A study of consistent and inconsistent changes to code clones," in WCRE. IEEE Computer Society, 2007, pp. 170–178.
[21] O. Kupferman and R. Lampert, "On the construction of finite automata for safety properties," in ATVA, 2006, pp. 110–124.
[22] C. Le Goues and W. Weimer, "Specification mining with few false positives." in TACAS, 2009, pp. 292–306.
[23] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," SIGPLAN Not., vol. 40, no. 1, pp. 364–377, 2005.
[24] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in USENIX Security Symposium, Aug. 2005, pp. 271–286

KARTHIK ENUMULA received his B.TECH degree in CSE from GANAPATHY ENGINEERING COLLEGE WARANGAL (DT) in 2012. At present, He is engaged in "A HYBRID ALGORITHM OF BACKWARD HASHING AND AUTOMATION TRACKING FOR VIRUS SCANNING".

SAI CHAND DASI received his B.TECH degree in CSE from BALAJI INSTITUTE OF ENGINEERING AND SCIENCES WARANGAL (DT) in 2012. At present, He is pursuing masters in CSE at Staffordshire University United Kingdom.

Devaki Nandan Kanjarla received his B.TECH degree in CSE from Ganapathy Engineering College Warangal (DT) in 2012. At present, He is engaged in "A Hybrid Algorithm of Backward Hashing And Automation Tracking for Virus Scanning".