NEGWeb: Static Defect Detection via Searching Billions of Lines of Open Source Code *

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, USA
xie@csc.ncsu.edu

ABSTRACT

To find defects in programs, existing approaches mine programming rules as common patterns out of program source code and classify defects as violations of these mined programming rules. However, these existing approaches often cannot surface out many programming rules as common patterns because these approaches mine patterns from only one or a few project code bases. To better support static bug finding based on mining code, we develop a novel framework, called NEGWeb, for substantially expanding the mining scope to billions of lines of open source code based on a code search engine. NEGWeb detects violations related to neglected conditions around individual API calls. We evaluated NEGWeb to detect violations in local code bases or open source code bases. In our evaluation, we show that NEGWeb finds three real defects in Java code reported in the literature and also finds three previously unknown defects in a large-scale open source project called Columba (91,508 lines of Java code) that reuses 2225 APIs. We also report a high percentage of real rules among the top 25 reported patterns mined for five popular open source applications.

1. INTRODUCTION

To improve software quality, developers can write down programming rules and then apply static or runtime verification tools to detect software defects related to violations of these rules. But in practice, these programming rules often do not exist or even when they exist, they are written informally, not amenable to verification tools. To tackle the issue of lacking programming rules, various approaches have been developed in recent years to mine programming rules from program executions [3, 6, 18], program source code [1,2,4,5,10,12,13,15], or version histories [11,16]. Then these approaches detect likely defects as those violations of the mined programming rules. A common methodology adopted by these approaches is to discover common patterns (e.g., frequent occurrences of pairs or sequences of API calls) across a sufficiently large number of data points (e.g., code locations). Then these common patterns often reflect programming rules that should be obeyed when programmers write code using a similar set of rule elements such as API calls and condition checks. However, these existing approaches often cannot surface out many programming rules as common patterns because these approaches mine patterns from a small number of project code bases and there are often too few data points in these code bases to support the mining of desirable patterns. In other words, the number of data points to support a pattern related to a particular programming rule is often insufficient. This phenomenon is reflected on the empirical results reported by these exiting approaches: often a relatively small number of *real* programming rules mined from huge code bases.

A natural question to ask is how we can address this issue of lack of relevant data points in mining programming rules. Code Search Engines (CSEs) such as Google code search [7] and Koders [9] give us some hope. These CSEs can be used to assist programmers by providing relevant code examples with usages of the given query from a huge number of publicly accessible source code repositories.

To address the issue of lacking relevant data points in mining programming rules, we develop a novel framework, called NEGWeb, for static bug finding based on searching and mining billions of lines of open source code with the help of a code search engine such as Google code search [7]. Our new approach is the first bug finding approach with this scale and based on a Code Search Engine (CSE). Our previous work also developed the MAPO [17] and PARSEWeb [14] approaches for mining source files returned by a CSE but these previous approaches focus on mining API usage patterns to assist programmers to write API client code effectively. In contrast, our new NEGWeb framework focuses on static bug finding based on mining programming rules, which poses a different set of mining requirements. In static bug finding based on mining, one important challenge is to reduce false positives (e.g., reported warnings that do not indicate real defects or patterns that do not reflect real programming rules).

To address the issue of false positives in static bug finding, we develop NEGWeb to detect violations related to neglected conditions around individual API calls, in particular, (1) missing conditions that check the receiver or arguments of an API call *before* the API call or (2) missing conditions that check the return values or receiver of an API call *after* the API call. As shown by Chang et al. [4] in their

*This work is supported in part by NSF grant CNS-0720641 and Army Research Office grant W911NF-07-1-0431.

approach of revealing neglected conditions, neglected conditions are quite common among the defects in programs. Different from their approach, which focuses on mining one project code base, NEGWeb mines a much larger scope of code bases based on a CSE. In addition, NEGWeb's mining based on simple statistical analysis is more scalable than their approach based on frequent sub-graph mining with known scalability issues.

While enjoying the benefits provided by a CSE in terms of expanding the analysis scope, NEGWeb faces one challenge that existing approaches do not face: the code samples returned by a CSE are often partial and not compilable, because the CSE retrieves individual source files with usages of the given query API method, instead of entire projects. We develop several new heuristics along with previously developed heuristics [14] to tackle this challenge.

This paper makes the following main contributions:

- A novel framework for static bug finding based on a CSE. Our framework is the first bug finding approach that can deal with that large scale of open source code through a CSE.
- A set of rule templates for describing common neglected conditions around individual API calls. These templates allow us to exploit program dependencies among rule elements (e.g., condition checks and API calls) to reduce false positives.
- A technique for analyzing partial code samples through Abstract Syntax Trees (AST) and Directed Acyclic Graphs (DAG) and a set of heuristics for further reducing false positives.
- An Eclipse plugin implemented for the proposed framework and several evaluations to assess the effectiveness of the tool. In particular, NEGWeb confirms three real defects in Java code reported in the literature and also detects three previously unknown defects in a large-scale open source project that reuse 2225 APIs. We also report a high percentage of real rules among the top 25 reported patterns mined for five open source applications.

The rest of the paper is organized as follows. Section 2 explains the framework through an example. Section 3 describes key aspects of the framework. Section 4 discusses evaluation results. Section 5 discusses the threats to validity. Section 6 presents related work. Finally, Section 7 concludes.

2. EXAMPLE

We next use an example to describe how our NEGWeb framework mines condition patterns from billions of lines of open source code available on the web and uses the mined condition patterns to detect violations in either an input source application or in available open source projects on the web. We use the class `org.apache.bcel.verifier.Verifier` and its methods `doPass1`, `doPass2`, `doPass3a`, and `doPass3b` from the BCEL library as an illustrative example for explaining our framework. The `Verifier` class is mainly used for verifying generated class files.

Given an API such as `Verifier` and its methods, NEGWeb constructs a query with the class name and gathers relevant code samples from a CSE. An example code sample gathered from a CSE is shown in Figure 1. NEGWeb parses each code sample and transforms the sample into an

```
01:public static void verifyBCEL(String cName) {
02:  VerificationResult vr0, vr1, vr2, vr3;
03:  int mId = 0;
04:  Verifier verf = VerifierFactory.getVerifier(cName);
05:  if(verf != null) {
06:    vr0 = verf.doPass1();
07:    if(vr0 != VerificationResult.VR_OK)
08:       return;
09:    vr1 = verf.doPass2();
10:    if (vr1 == VerificationResult.VR_OK) {
11:       JavaClass jc = Repository.lookupClass(cName);
12:       for(mId=0; mId<jc.getMethods().length; mId++){
13:         vr2 = verf.doPass3a(mId);
14:         vr3 = verf.doPass3b(mId);
15:         if(Pass3aVerifier.do_verify(verf)) { ... }
16:  }    }   } }
```

Figure 1: Code sample gathered from a code search engine.

intermediate form, represented in the form of a Directed Acyclic Graph (DAG). NEGWeb uses *dominance* and *data-dependency* concepts, and gathers preceding and succeeding conditions around the nodes that include any of the methods such as `doPass1` or `doPass2`. NEGWeb identifies different condition patterns for `RECEIVER`, `ARGUMENT`, and `RETURN` objects around the given method. A few condition patterns identified from the example code sample are shown as below.

```
01: doPass1 RECEIVER NULLITY
02: doPass2 PRE_METHOD CONST_EQUAL doPass1
03: doPass1 RETURN CONST_EQUAL VerificationResult.VR_OK
04: doPass3a ARGUMENT GEN_EQUALITY
```

Each condition pattern consists of the method name, pattern type, condition type, and optional additional information separated by spaces. The condition pattern in Line 1 describes that before invoking the `doPass1` method, a `NULLITY` check must be done on the receiver variable of that method. An example of the pattern type `PRE_METHOD` shown in Line 2 describes that the method `doPass2` must be invoked only after the `doPass1` method. Line 3 describes that the return value of the `doPass1` method should be compared with the constant `VerificationResult.VR_OK`. Line 4 describes the `ARGUMENT` pattern type where the argument must be verified before invoking the `doPass3a` method.

NEGWeb mines the extracted condition patterns to compute frequent condition patterns, referred as mined patterns. NEGWeb applies these mined patterns on either the input source application or gathered code samples to detect violations. For example, consider the code sample below taken from existing open source projects with violations:

```
Verifier v = VerifierFactory.getVerifier(args[k]);
VerificationResult vr;
vr = v.doPass1();
vr = v.doPass2();
```

NEGWeb detects a violation from the preceding code sample as the sample violated the condition pattern "`doPass1 RETURN CONST_EQUAL VerificationResult.VR_OK`". Sometimes, the same violation can appear multiple times because the code sample can violate multiple condition patterns. For example, the preceding code sample also violates the condition pattern "`doPass2 PRE_METHOD CONST_EQUAL doPass1`".

3. FRAMEWORK

Our NEGWeb framework consists of seven major components: the application scanner, code search engine, code downloader, code analyzer, pattern extractor, pattern miner, and anamoly detector. Figure 2 shows an overview of all

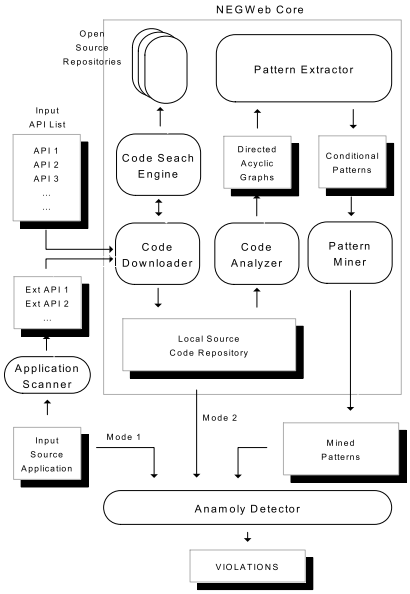

Figure 2: Overview of NEGWeb framework

components in NEGWeb. NEGWeb accepts an input source application or a set of interested APIs as input. When a source application is given as input, the application scanner identifies external classes and methods used by the application through package names of those classes. We use notation IC_x for each interested class, which can be an external class of a given source application or a class in the interested set of APIs. Each class IC_x includes a set of methods represented as $\{IM_{x1}, IM_{x2}, ..., IM_{xn}\}$. We use the same code sample shown in Figure 1 as an illustrative example to describe the process defined in our NEGWeb framework. We refer the `Verifier` class in the code sample as IC_x and methods `doPass1`, `doPass2`, `doPass3a`, and `doPass3b` as different IM_{xy} of the IC_x class.

3.1 Application Scanner

The application scanner accepts a source application as input and gathers the external classes and methods used by that application. A class is recognized as an external class if the package name of that class does not belong to the set of package names of the given application. The application scanner initially collects all packages and classes of the source application, and then uses the collected information to identify the external classes through their package names. The application scanner also gathers the set of methods referred by the source application for each such external class. The set of external classes and methods is provided as input to the code downloader.

3.2 Code Search Engine

Code Search Engines (CSE) such as Google [7] and Koders [9] are primarily used by programmers in searching for related samples from the available open source projects on the web. As CSEs can search billions of lines of source code available on the web, CSEs can serve as powerful resources of open source code. Due to the strength of these CSEs in searching for related samples, these CSEs can be exploited for other tasks such as detecting violations in the applications that reuse existing open source projects. Therefore, our framework uses CSEs to gather related samples for the given set of APIs.

Our framework uses Google code search (GCSE) [7] for collecting related samples because of two main reasons: (1) GCSE provides client libraries that can be used by other tools to interact with and (2) GCSE has public forums that provide good support. However, our framework is independent of GCSE and can leverage any other CSE to gather related samples.

3.3 Code Downloader

The code downloader accepts the IC_x set and their associated IM_{xy} sets as input, and interacts with CSE for searching and gathering related samples. To improve performance, the code downloader constructs only one query for each IC_x instead of different queries for each pair (IC_x, IM_{xy}). The gathered code samples are applicable to all IM_{xy} of the IC_x that is included in the query. An example code sample gathered from CSE for the query "`org.apache.bcel.verifier.Verifier`" related to the BCEL library is shown in Figure 1. The gathered code samples are stored in a repository and is referred as a local source code repository.

3.4 Code Analyzer

The code analyzer accepts the local source code repository as input and analyzes the code samples statically through Abstract Syntax Trees (AST) to construct DAGs. The code analyzer uses several type heuristics while analyzing these code samples as the code samples gathered through CSE are partial. The reason for the partial nature of these code samples is that the CSE extracts only the source files with usages of the given query instead of entire projects. These type heuristics help identify the object types in the code samples. For example, for the method `doPass1` in Statement 6, the type heuristics identify the return type as `VerificationResult` from the left-hand side of the assignment statement.

The constructed DAG consists of two kinds of nodes: control and non-control. Control nodes, referred as CT, represent the control-flow statements such as *if*, *while*, and *for*, which control the flow of the program execution. Non-control nodes, referred as NT, represent statements such as method-invocations or type casts. For example, Statement 5 in the code sample is a control node and Statement 6 is a non-control node. While encountering a control node, say CT_i, the code analyzer also identifies all variables, say $\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node. Each variable V_j is associated with its corresponding condition type T_j. Therefore, a control node CT_i includes a set of pairs $\{(V_1, T_1), (V_2, T_2), .., (V_n, T_n)\}$. For example, the control node CT_5 (suffix indicates the statement id) includes the {(`verf`, `NULLITY`)} pair. The possible values for T_j are shown in Table 1. The table also shows the description and additional information associated with each T_j. The additional information represents operators, constants, or method-invocations associated with each pair (V_j, T_j). For example, the control node CT_{10} includes the pair {(`vr1`,`CONST_EQUAL`)} and the additional information associated with this pair includes the constant `VerificationResult.VR_OK`.

In general, the variable V_j can participate in the conditional expression either directly or indirectly. A direct participation refers to the scenario where the conditional check is conducted directly on the V_j such as the `NULLITY` condition type shown in Table 1. An indirect participation refers to the

Table 1: Values of condition types, T_j associated with variable, V_j in a conditional expression.

Condition Type (T_j)	Description	Additional Info
NULLITY	direct null check. Eg: if(V_j != null) { .. }	operator involved, say !=
M_NULLITY	indirect null check. Eg: if($M_k(V_j)$ == null) { .. }	operator involved, say == method-invocation, M_k
BOOLEAN	if the variable type is boolean. Eg: if(V_j) { .. }	
M_BOOLEAN	indirect boolean check. Eg: if($M_k(V_j)$) { .. }	method-invocation, say M_k
CONST_EQUAL	if the variable is compared with a constant. Eg: if(V_j == SUCCESS)	operator involved, say == constant value, say SUCCESS
M_CONST_EQUAL	indirect constant equality check. Eg: if($M_k(V_j)$ == FAILURE)	operator involved, say == constant value, say FAILURE method-invocation, say M_k
RETVAL_EQUAL	if the variable is compared with the return value of a method-invocation. Eg: if($V_j < M_l$)	operator involved, say < method-invocation, say M_l
M_RETVAL_EQUAL	if the variable is compared indirectly with the return value of a method-invocation. Eg: if($M_k(V_j) > M_l$)	operator involved, say > method-invocation, say M_k method-invocation, say M_l
INSTANCE_CHECK	if the conditional check involves `instanceof` operator Eg: if(V_j instanceof Integer)	type-name, say Integer
GEN_EQUALITY	if the conditional check does not fall in preceding types. Eg: if($V_j < V_a$) { ... }	operator involved, say < other expression, say V_a

scenario where V_j is an argument of a method-invocation, say M_k, that is involved in the conditional expression. For example, the condition type `M_NULLITY` refers to the preceding scenario. To distinguish the direct and indirect participations, the condition types of indirect participation are prefixed with "M_".

The constant values shown in Table 1 can be literals or constant variables. The code analyzer identifies constant variables such as `SUCCESS` or `FAILURE` by using a heuristic based on general guidelines of the Java programming language. The guidelines describe that variable names with all capital letters can be treated as constants.

While constructing the DAG, the code analyzer identifies nodes in the graph that contain IM_{xy} and marks those nodes as APINodes, referred as AN_i. The constructed DAG can contain one or more AN_i nodes and this DAG serves as a Control Flow Graph (CFG) for the pattern extractor that identifies the condition patterns around each IM_{xy}. In the example code sample, the code analyzer identifies Statements 6, 9, 13, and 14 as APINodes.

3.5 Pattern Extractor

The pattern extractor accepts the constructed DAG as input from the code analyzer and performs forward and backward traversal over each AN_i node to identify condition patterns around each IM_{xy}. In particular, the pattern extractor uses the concept of *dominance* with a blend of *control-flow* and *data-flow* dependencies. The condition patterns identified by the pattern extractor can be classified into two major categories: preceding and succeeding patterns. We next describe how the pattern extractor identifies these preceding and succeeding condition patterns.

3.5.1 Preceding Condition Patterns

The pattern extractor extracts preceding condition patterns by using the concept of dominance. The definition of dominance concept is given below:

Dominance:*A node N dominates another node M in a control flow graph (represented as N dom M) if every path from the starting node of the CFG to M includes N.*

Initially, the pattern extractor identifies the dominant CT_i nodes for each AN_k node. For example, the pattern extractor identifies that CT_5 dominates AP_6. The pattern extractor computes an intersection between the variable set associated with the CT_i node, say $\{V_1, V_2, ..., V_n\}$, and the receiver or argument variables of the AN_k node, say $\{REC_k, ARG_{k1}, ..., ARG_{k2}\}$. If the intersection $\{V_1, V_2, ..., V_n\} \cap \{REC_k, ARG_{k1}, ..., ARG_{k2}\} \neq \Phi$, the pattern extractor checks whether the AN_k node is data-dependent on the CT_i node. The data-dependency check ensures that the variable involved in the CT_i node is not re-defined in the path between CT_i and AN_k nodes. If the AN_k node is data-dependent on the CT_i node, the pattern extractor extracts the associated condition pattern. The associated condition pattern for nodes CT_5 and AN_6 in the example code sample is "`doPass1 RECEIVER NULLITY`", which indicates that a `NULLITY` check must be done on the the receiver variable of the method `doPass1`. The general condition pattern format extracted by the pattern extractor includes IM_{xy}, pattern type, and condition type (T_j). However, for some condition patterns, a fourth element that represents other method-invocations is included in the pattern format.

The pattern extractor extracts three types of preceding condition patterns: `RECEIVER`, `ARGUMENT`, and `PRE_METHOD`. We next describe the details of each pattern type.

RECEIVER: represents condition patterns on the receiver variable preceding the call site of IM_{xy}. For example, the receiver condition pattern associated with CT_5 and AN_6 for the method `doPass1` is "`doPass1 RECEIVER NULLITY`".

ARGUMENT: represents condition patterns on the argument variable preceding the call site of IM_{xy}. For example, the argument condition pattern associated with CT_{12} and AN_{13} for the method `doPass3a` is "`doPass3a ARGUMENT GEN_EQUALITY`".

PRE_METHOD: represents condition patterns on other method-invocations of the same receiver variable preceding the call site of IM_{xy}. For example, the `PRE_METHOD` condition pattern associated with CT_7 and AN_9 for the method `doPass2` is "`doPass2 PRE_METHOD CONST_EQUAL doPass1`". This pattern indicates that before invoking the `doPass2` method, the method `doPass1` must be invoked and a condition check must be performed on the return value of the method `doPass1`.

3.5.2 Succeeding Condition Patterns

The pattern extractor extracts the succeeding condition patterns by using the concept of post-dominance. The pat-

tern extractor identifies the post-dominant CT_i nodes for each AN_k node and checks whether the identified CT_i node is data-dependent on the AN_k node. The format of succeeding condition patterns is the same as the preceding condition patterns. The pattern extractor extracts three types of succeeding condition patterns: `RETURN`, `SUC_RECEIVER`, and `SUC_METHOD`. We present the details of each succeeding pattern type below.

RETURN: represents condition patterns on the return variable succeeding the call site of IM_{xy}. For example, the return condition pattern associated with AN_6 and CT_7 for the method `doPass1` is "`doPass1 RETURN CONST_EQUAL`".

SUC_RECEIVER: represents condition patterns on the receiver variable succeeding the call site of IM_{xy}. For example, the succeeding receiver condition pattern associated with AN_{14} and CT_{15} for the method `doPass3b` is "`doPass3b SUC_RECEIVER M_BOOLEAN do_verify`", which indicates that the invocation of the `doPass3b` method must be followed by a boolean check on the receiver variable. The condition type `M_BOOLEAN` indicates that the receiver variable participates indirectly as an argument of another method-invocation such as `do_verify`.

SUC_METHOD: represents condition patterns on other method-invocations of the same receiver variable succeeding the call site of IM_{xy}. For example, the `SUC_METHOD` condition pattern for the method `doPass1` associated with AN_6 and CT_{10} is "`doPass1 SUC_METHOD CONST_EQUAL doPass2`".

In general, the `SUC_METHOD` condition pattern type is useful for methods such as `Iterator.next()` and `Iterator.hasNext()` that are often used in loops, where one of the methods such as `hasNext` is involved in the conditional expression of the loop. If not, this condition pattern can result in many false positives. Therefore, in the NEGWeb framework, we limited this condition pattern type to those APIs such as Java Util APIs, which are often suggested to be used in loops.

3.5.3 Empty Condition Patterns

Sometimes the call sites of IM_{xy} do not have any preceding or succeeding condition patterns. To store the number of call sites that do not have condition patterns, the pattern extractor associates an attribute called *Empty Condition Pattern* (`ECP`). The ECP_{xy} attribute is used by the pattern miner while computing the support values for each condition pattern of IM_{xy}.

3.6 Pattern Miner

The pattern miner accepts a set of extracted condition patterns, say CP_i, for each IM_{xy} and mines frequent condition patterns among that set. The primary objective of the pattern miner is to reduce the number of false positives while computing the mined patterns. Each condition pattern is associated with a frequency attribute, referred as CPF_i, which gives the number of occurrences of that condition pattern among all call sites of IM_{xy}. Initially, the pattern miner computes support, referred as CPS_i, for each condition pattern. The formula used for computing CPS_i is given as below:

$$CPS_i = CPF_i \ / \ (\sum_{j=1}^{N} CPF_j + ECP_{xy})$$

where N is the number of CP_i for the IM_{xy}.

The rationale behind using ECP_{xy} in computing CPS_i is to identify the actual support of a condition pattern among all call sites of IM_{xy}. The consideration of ECP_{xy} can help reduce the number of false positives among the mined condition patterns. The pattern miner also computes the support for ECP_{xy}, referred as $ECPS_{xy}$, using the formula given as below:

$$ECPS_{xy} = ECP_{xy} \ / \ (\sum_{j=1}^{N} CPF_j + ECP_{xy})$$

The pattern miner uses two threshold values Upper Threshold (UT) and Lower Threshold (LT) for computing frequent condition patterns and for classifying the mined patterns in to three confidence levels: `HIGH`, `AVERAGE`, and `LOW`. Initially, the pattern miner compares the value of $ECPS_{xy}$ and UT. If the value of $ECPS_{xy}$ is greater than UT or the support values of all condition patterns are less than LT, the pattern miner ignores all extracted condition patterns. The pattern miner checks the condition patterns with support greater than UT and identifies them as mined patterns with confidence `HIGH`. Instead, if the support values of all condition patterns are near the maximum support value, which will be less than UT, the pattern miner identifies all condition patterns as mined patterns with confidence level `HIGH`. The rationale behind this assumption is that a group of condition patterns can often appear together in code samples. For example, consider that two `RETURN` condition patters, say `SUCCESS` and `FAILURE`, appeared together in 10 code samples. The computed support values for these two condition patterns will be 0.5 each, resulting in a low confidence level. However, these condition patterns appeared in all code samples and should have a high confidence level. We use the value of $ECPS_{xy}$ to classify non-high condition patterns into confidence levels `AVERAGE` and `LOW`. We used values 0.75 for UT and 0.1 for LT. These values are based on our empirical experience with different subjects.

3.7 Anamoly Detector

The anamoly detector operates in two modes. In Mode 1, the anamoly detector uses mined heuristics to detect violations in the source application. In Mode 2, the anamoly detector uses mined heuristics to detect violations in the gathered code samples. In particular, the anamoly detector re-extracts the condition patterns for each IM_{xy}, and checks whether the newly extracted condition patterns contain the mined patterns. Any missing mined patterns are reported as violations. For each detected violation, the anamoly detector assigns a confidence level, which is the same as the confidence level of the associated condition pattern.

Additionally, the anamoly detector uses two heuristics to reduce the number of false positives and to sort the detected violations based on their importance.

Anamoly Heuristic 1:*A violation detected for succeeding condition patterns can be ignored if the corresponding variable is a part of the return statement of the enclosing method declaration or an argument of another method-invocation.*

This heuristic is based on our experience with different subjects where an expected conditional check often appears at the call site of the enclosing method declaration.

Anamoly Heuristic 2:*A violation detected for a call site with no conditional checks around can be given higher preference than violations detected for other call sites with a few conditional checks around.*

The rationale behind this heuristic is that call sites with no conditional checks can have a higher chance of being a defect than call sites with a few conditional checks around. The anamoly detector sorts the detected violations based on attributes confidence level, support of the related condition pattern, and the favorable number of code samples.

Table 2: Condition patterns mined by NEGWeb and their violations.

Application	Input Application		CSE		Categories of first 25 patterns			Time	#
	#Classes	#Methods	#Samples	#Patterns	#Rules	#Usage Patterns	#False Positivies	(in min.)	Violations
Java Util APIs	19	144	49858	64	20	5	0	7.98	15234
BCEL	357	2691	9697	322	13	8	4	1.04	776
Hibernate	1233	11452	32486	542	21	2	2	5.17	850
Java Servlet APIs	19	89	16628	53	18	0	7	2.92	505
Java Transaction APIs	7	37	5555	15	12	2	1	0.80	421

4. EVALUATION

We conducted four different evaluations on NEGWeb to show that NEGWeb can effectively mine real rules from related code samples gathered through a CSE, and can be effective in identifying real defects. In the first evaluation, we used two applications and three API libraries to mine condition patterns and manually confirmed the extracted top 25 condition patterns through the available documentation and source code of the applications. For generality, we refer all subjects as applications. In the second evaluation, we applied the mined condition patterns in a novel way to detect violations in available open source applications. As NEGWeb mainly targets at neglected conditions that help increase the robustness of applications, we manually confirmed the top 50 violations as defects or other categories through inspection. In the third evaluation, we verified whether NEGWeb can confirm known Java defects reported in the literature by earlier related approaches. In the fourth evaluation, we conducted a case study with a large-scale application called Columba. The details of subjects and results of our evaluation are available at `http://ase.csc.ncsu.edu/negweb/`.

4.1 Open Source Applications

In this section, we describe condition patterns mined by NEGWeb for two open source applications and three API libraries that vary in size and purpose. We also apply the mined condition patterns onto the gathered code samples to detect violations in available open source projects.

4.1.1 Condition Patterns

The characteristics such as the number of classes and methods of the five applications used for mining condition patterns are shown in Columns "Classes" and "Methods" of Table 2. The Java Util package includes the collections framework and other popular utilities used by many different applications. The BCEL library, developed by Apache, is mainly used to analyze, create, and manipulate Java class files. The Hibernate framework abstracts relational databases into an object-oriented methodology. Java servlets and Java Transactions are industry standards for developing multitier server-side Java applications. The common reason for selecting these applications is the presence of condition patterns as described by their associated documentations that can help confirm the real patterns. We feed the set of classes and methods of each application as input to NEGWeb.

Column "Samples" of Table 2 shows the number of code samples gathered and analyzed for each application from the code search engine. For example, NEGWeb gathered and analyzed 49,858 code samples for Java Util packages. The number of condition patterns mined for each application is shown in Column "Patterns" of Table 2. We manually analyzed the first 25 patterns of each application and classified them into three categories: rules, usage patterns, and false positives. Rules describe the properties that must be satisfied for using an API, whereas usage patterns are common ways of using an API. We used the available on-line documentations, JML specifications[1], or the source code of the application for classifying the mined condition patterns into these three categories. As shown in Table 2, most of the mined patterns are classified as rules and a few are classified as false positives. The number of false positives is a little more for Java Servlet, because of one common pattern that appeared among these false positives. The common pattern is "`ServletRequest INSTANCE_CHECK HttpServletRequest`", which describes that an instance check has to be performed with the `ServletRequest` class before invoking its methods. Although this pattern is a common usage, the available specification of Java Servlet does not confirm this pattern as a real rule. The primary reason for the lesser number of false positives in other subjects is due to the large number of analyzed data points gathered through CSE.

Table 3: Classification of mined patterns of Java Util package.

Pattern type	#Total	#Rule	#UP	#FP
`RECEIVER`	5	3	0	2
`ARGUMENT`	6	5	0	1
`PRE_METHOD`	12	11	0	1
`RETURN`	19	12	7	0
`SUC_METHOD`	22	5	10	7
`SUC_RECEIVER`	0	0	0	0
SUM	**64**	**36**	**17**	**11**

UP: usage pattern, FP: false positive

We manually classified all condition patterns of the Java Util package. The primary reason for selecting the Java Util package for manual analysis is the availability of JML specification that can help confirm the mined patterns. The classified categories of all patterns for the Java Util APIs are shown in the last row of Table 3. Among all mined patterns, the real rules constitute 56.25% (36/64) and usage patterns constitute 26.56% (17/64). A non-negligible percentage of 17.18% (11/64) is classified as false positives. However, the number of false positives in the top 25 rules shown in Table 2 is zero. This evaluation shows the effectiveness of our mining heuristics that surface out real rules by ranking false positives below. Table 3 further shows the classification of the mined condition patterns based on the pattern types of NEGWeb. NEGWeb is effective in extracting and mining real rules for pattern types `PRE_METHOD` and `RETURN`, which are usually the main sources of neglected conditions. The pattern type `SUC_METHOD` has the largest number of false positives.

We next describe the mined patterns for the `Matcher` class of Java Util pacakges. NEGWeb identified 10 patterns for this class, which is an engine that performs matching operations on a character sequence by interpreting a given regular expression. For each mined pattern, we show the method

[1] `http://www.eecs.ucf.edu/~leavens/JML/`

name, pattern type, condition type, and additional information. The additional information is optional and is shown for pattern types such as PRE_METHOD and SUC_METHOD.

```
find, RETURN, BOOLEAN,
start, PRE_METHOD, BOOLEAN, find
end, PRE_METHOD, BOOLEAN, find
group, PRE_METHOD, BOOLEAN, find
start, SUC_METHOD, BOOLEAN, find
find, SUC_METHOD, BOOLEAN, group
group, SUC_METHOD, BOOLEAN, find
```

The preceding mined patterns indicate that a BOOLEAN check must be performed on the return value of the find method and this find method must precede and succeed methods start, end, and group. These mined patterns indicate that the methods start, end, and group are used inside a loop with a boolean check on the find method.

NEGWeb also mined undocumented condition patterns in the input applications. We confirmed these condition patterns by inspecting the source code and the comments written in source files. For example, in the BCEL library, the method copy of the class Instruction cannot be used for its child class Select. Therefore, before using the copy method, a condition check on the receiver variable must be performed. The associated mined condition pattern of NEGWeb is "copy RECEIVER INSTANCE_CHECK Select". The pattern describes that before invoking the copy method of Instruction class, an instance check must be performed on the receiver variable with the sub-class Select. Similarly, while using the methods doPass3a and doPass3b of the class Verifier, the caller must make sure that the index value passed as a parameter should be within the range of the number of methods in the corresponding class. The reason is that these methods operate on the Vector class that throws IndexOutOfBoundsException when the parameter value is not within the required range.

Column "Time" in Table 2 presents the amount of time taken by NEGWeb for mining patterns from the gathered code samples for each application. The amount of processing time depends on the number of samples gathered for the corresponding application. For example, NEGWeb took 7.98 minutes for mining patterns from 49,858 code samples gathered for Java Util packages. All experiments are conducted on a machine with 3.0GHz Xeon processor and 4GB RAM.

4.1.2 Defects in Open Source World

As NEGWeb mines patterns by gathering related code samples from available open source applications through a CSE, we applied the mined patterns in a novel way to detect violations in the gathered code samples themselves. Given a set of interested APIs, NEGWeb can detect the set of open source applications that violate properties of the given APIs. This feature of NEGWeb could be quite useful while dealing with APIs such as security APIs to check whether there are any security holes in applications used on the web. We used applications of Table 2 to detect violations in their gathered code samples. The results of our evaluation are shown in Column "Violations" in Table 2.

Given the large number of violations that we have detected, we manually analyzed violations of the mined rule (from the top 25 mined patterns of each application) that is used to detect the largest number of violations. We classified the detected violations into five categories: defect, code smell, wrapper, hint, and false positive. The violation categories of code smell and hint are inspired by the approach of Wasylkowski et al. [15]. A code smell indicates that something may go wrong, whereas a hint helps increase the readability of the program. We introduced the category of wrappers to indicate that the mined patterns are spread across several methods of a wrapper class. For example, a user-defined class such as UIterator abstracts the functionality of the Iterator by providing different methods such as next() and hasNext(). In this scenario, the patterns mined for the Iterator class are applicable to the wrapper class UIterator as well. These wrappers can be useful

The API chosen from each application and the classification categories for the first 50 are shown in Table 4. Columns "Pattern Type" and "Support" give the type of the condition pattern such as RECEIVER and support value of the condition pattern, respectively. Column "Open Source" gives the total number of open source projects that contain those detected violations. The total number of violations for each API is shown in Column "Total". The manual classification results of the first 50 violations are shown in Columns "Defect", "CS", "WP", "Hint", and "FP". Except for BCEL, the number of false positives is quite low for APIs of other applications. The reason for a high number of false positives in BCEL is due to limitations in the current NEGWeb implementation such as not handling conditional expressions in assignment statements, which we plan to address in near future work without difficulty.

We next describe details of defects detected in open source applications for Java Servlet APIs. The API used from this application is "ServletConfig,getInitParameter(String)" and the condition pattern used is "getInitParameter RETURN NULLITY", which indicates that a nullity check must be performed on the return value of the getInitParameter method. We confirmed this pattern from the JTA specification, which describes that this method can return null, if the parameter does not exist. However, 31 open source projects violated this mined condition pattern. Among the top 50 violations, 38 violations are classified as defects in our inspection. We found some more interesting facts during this evaluation. We use the below code sample that is collected from the existing open source projects to describe these facts.

```
...String jspCP = config.getInitParameter("jspCP");
if (jspCP != null) {...}
this.javaEncoding = config.getInitParameter("javaEncoding");
```

In the preceding code sample, the getInitParameter method is used two times. However, the NULLITY check on the return value is done only once, and is ignored during the second invocation. As the getInitParameter method can return null, the absence of NULLITY check can cause NullPointerException. We also found that the same piece of code that has violations is used in different applications. For example, we found a similar violated code in open source projects tomcat, fisheye, and jboss for the getInitParameter method. As programmers often tend to copy related code from existing applications, the violations can also propagate from applications to applications. Our results show the number of neglected conditions that exist in the available open source applications and the necessity for an approach such as NEGWeb.

4.2 Real defects from the literature

We evaluated NEGWeb to check whether it can detect known defects described in the literature. We picked two

Table 4: Classification of violations detected in open source world.

Application	Input API (Class,Method)	Pattern Type	Support	#Open Source	# Total	Categories of first 50 violations				
						#Defect	#CS	#WP	#Hint	#FP
Java Util APIs	Matcher,group	PRE_METHOD	0.810	7	21	0	12	2	3	4
BCEL	Type,equals	RETURN	0.967	2	7	0	0	0	1	6
Hibernate	Filter,setParameterList	RECEIVER	0.875	1	4	4	0	0	0	0
Java Servlet APIs	ServletConfig,getInitParameter	RETURN	0.577	31	50	38	2	0	8	2
Java Transaction APIs	TransactionManager,getTransaction	RETURN	0.283	40	230	22	3	0	19	6

CS: code smell, WP: wrapper, FP: false positive

defects in the AspectJ application detected by JADET[2] [15] and two defects in Java SSE Library and Joeq detected by DIDUCE [8]. We next describe details of these defects and explain the evaluation results with NEGWeb.

4.2.1 Defects Detected by JADET

In the AspectJ application, JADET detected two defects related to loops that are incorrectly executed at most once. We show the code sample taken from JADET as below:

```
private boolean verifyNIAP (...) {...
    Iterator iter = ...;
    while(iter.hasNext()) {
        ... = iter.next();         ...;
        return verifyNIAP(...);     } }
```

As shown in the preceding code sample, the `return` statement in the `while` loop causes the method to return without iterating all elements in the `Iterator`. NEGWeb confirmed this defect with a support value of 0.895 as the code sample violated the pattern "next `SUC_METHOD BOOLEAN` hasNext" of the class `Iterator`. NEGWeb confirmed the other defect reported by JADET that is also related to a similar scenario.

4.2.2 Defects Detected by DIDUCE

We collected two defects reported by DIDUCE in applications Java SSE and Joeq. The defect in the Java SSE library is related to not handling the return value of the `read` method of the class `InputStream`. The method `read` returns the number of bytes that are actually read; programmers often forget to check whether the number of read bytes is equal to the expected number of bytes to be read. NEGWeb confirmed this defect with a support value of 0.708.

The second defect in the Joeq application is related to not checking the return value of the method `put` of the class `Hashtable`. When an object is inserted into the `Hashtable` through the `put` method, the method either returns an existing associated object with that key value or returns `null`. NEGWeb could not confirm this defect as the support for the extracted pattern is low. Among 774 related code samples gathered from the code search engine, NEGWeb detected that only 5 code samples have the `NULLITY` check on their return value. However, this defect mainly depends on the semantic logic of the application rather than the usage commonality of the API. Therefore, reporting violations based on these kinds of patterns can result in a large number of false positives. We want to emphasize that the motivation of NEGWeb is mainly to mine the most common condition patterns that can cause potential defects and to reduce the number of false positives among the detected violations.

[2]JADET reported three defects in the paper. One defect with BCEL APIs is not related to neglected conditions and does not fall into the scope of our current approach.

Table 5: Evaluation results of Columba case study.

SNo	Rank	Supp.	PC	# Total	Categories of violations				
					#Defect	#CS	#WP	#Hint	#FP
1	1	0.944	Rule	2				2	
2	2	0.939	Rule	2		1		1	
3	3	0.929	Rule	13	2		2	9	
4	4	0.917	UP	13			2	11	
5	5	0.875	Rule	1		1			
6	6	0.869	Rule	3				3	
7	7	0.861	Rule	2			2		
8	8	0.850	Rule	2			1	1	
9	8	0.850	Rule	1	1				
10	8	0.850	Rule	1		1			
11	8	0.850	FP	1					1
12	8	0.850	Rule	1					**1**
13	8	0.850	Rule	1		1			
14	9	0.833	FP	4					4
15	10	0.800	FP	2					2
16	11	0.786	Rule	1			1		
17	12	0.782	Rule	4				4	
18	13	0.771	UP	6				6	
19	14	0.762	Rule	1					**1**
20	15	0.756	Rule	1			1		
21	16	0.750	Rule	1		1			
22	16	0.750	Rule	1		1			
23	17	0.727	FP	4					4
24	18	0.722	Rule	1					**1**
25	19	0.710	Rule	1		1			

4.3 Case Study: Columba

Columba 1.4[3] is an open source email client application written in Java. Columba provides a user-friendly graphical interface and is suitable for internationalization support. The Columba application includes 1165 classes and 6894 methods that contain a total of 91,508 lines of Java code. We used NEGWeb to mine rules and detect violations in the Columba application. NEGWeb identified the external classes and methods used by Columba and mined condition patterns for those classes by interacting with CSE. NEGWeb gathered and analyzed 309,757 code samples for mining these condition patterns. NEGWeb applied the mined condition patterns to detect violations in Columba. NEGWeb mined 559 condition patterns related to the external classes and methods used by Columba. Among these 559 condition patterns, 370 condition patterns were used to detect 1647 violations. Each mined condition pattern can be used to detect multiple violations at different locations of the source code. We manually analyzed violations of the first 25 mined patterns. As each mined pattern can detect multiple violations at different locations of source code, these 25 patterns detected 70 violations. We classified these 70 violations into different violation categories using the same classification criteria described in Section 4.1.2.

The results of our evaluation are shown in Table 5. Each row in the table represents a mined pattern. Columns "Supp."

[3]`http://sourceforge.net/projects/columba/`

and "PC" give the support and manually assigned category of each pattern, respectively. Column "Total" gives the total number of violations detected by that pattern. Among the first 25 mined patterns, 19 patterns are classified as real rules, 2 are classified as usage patterns, and 4 are classified as false positives. The results also show that the top 10 patterns do not have any false positives. As each pattern type can be used to detect multiple violations of different categories, depending on the usage in the source code, we show how many of the total violations of each pattern fall into different violation categories. In total, there are 70 violations among which, 3 are defects, 7 are code smells, 9 are wrappers, 37 are hints, and 14 are false positives. All three defects among 70 violations are detected with the top 10 mined patterns. In general, a false-positive pattern leads to a false-positive violation. For example, Patterns 11 and 14 are false positives that caused 5 violations of the false-positive category. However, the additional 3 false positives highlighted in the table are due to limitations in the current NEGWeb implementation such as not handling conditional expressions in assignment statements. We plan to address these limitations in near future work and these limitations can be addressed without any difficulty. The largest number of hints are detected with Pattern 4, which is "`next SUC_METHOD BOOLEAN hasNext`" of the `Iterator` class. Although this mined pattern is a real rule, lists are used for storing a single object instead of multiple objects in Columba. Therefore, we classified these violations as hints that can help improve the maintainability and readability of code.

We next describe defects detected by NEGWeb in Columba. We confirmed these defects through inspecting the source code of the associated class and call sites of the violated methods. The first defect is in the method `removeDoubleEntries` of the `MessageBuilderHelper` class. We show the code sample of that method as below:

```
private static String removeDoubleEntries(String input) {
  Pattern sP = Pattern.compile("s*(<[^s<>]+>)s*");
  ArrayList entries = new ArrayList();
  Matcher matcher = sP.matcher(input);
  while (matcher.find()) {
     entries.add(matcher.group(1)); }
  Iterator it = entries.iterator(); ...
  String last = (String) it.next(); ... }
```

The method `removeDoubleEntries` tries to identify character sequences that match with a regular expression. If the given input string does not match with the regular expression "`s*(<[s<>]+>)s*`", no elements will be added to the `entries` list. The preceding code sample violated the mined pattern "`next PRE_METHOD BOOLEAN hasNext`", which describes that `hasNext` must be invoked before calling `next` of the `Iterator` class. In the code sample, the first element from the `it` variable is retrieved without checking whether there are any elements in the list through `hasNext`. Moreover, the retrieved variable is type casted to a string. This defect can cause `NullPointerException` in the described scenario. Although the current method is `private`, the other public caller methods of the current class do not handle any exceptions, resulting in the propagation of the exception to their call sites. The second defect is also related to a similar scenario that violated the mined pattern "`next PRE_METHOD BOOLEAN hasNext`".

The third defect is related to the JPIM[4] library used by the Columba application. Columba invokes the method `unmarshallContacts` of the class `ContactUnmarshaller` that is defined by the JPIM library. NEGWeb identified the pattern "`unmarshallContacts RETURN NULLITY`", which indicates that a `null` check must be performed on the return value of the `unmarshallContacts` method. As this pattern is an undocumented one, we confirmed this pattern by inspecting the source code of the JPIM library. In Columba, this method is invoked in the class `VCardParser` and no `NULLITY` check on the return variable was done. The absence of the `NULLITY` check can cause a `NullPointerException`.

The first code smell is in the method `createPages` of the `cDocument` class. We show the code sample of the `cDocument` class as below:

```
public class cDocument { ...
  boolean uptodate = false;
  List objects;
  public void print() {
    if (!uptodate) {createPages();}  ... }
  public void appendPO(cPO obj){
    objects.add(obj);
    uptodate = false;}
  private void createPages(){ ...
    Enumeration objEnum = ...;
    nextObj = (cPO) objEnum.nextElement();  ...}
```

As shown in the preceding code sample, the first element of the `Enumeration` is retrieved and is type casted without checking whether there are any elements through the method `hasMoreElements`. Although the method `createPages` cannot be invoked from outside, invocation of the method `print` prior to invoking `appendPO` causes a `NullPointerException`. Although this violation can be a defect, we classified this violation as a code smell because the current call sites of these two methods invoke the methods in the proper order such as `appendPO` followed by `print`.

4.4 Summary of the Evaluation

The major advantage of NEGWeb compared to existing approaches [1, 2, 4, 10, 15] is that NEGWeb can identify real rules due to the large number of data points used for mining the condition patterns. The number of false-positive patterns mined by NEGWeb is relatively low compared to other existing approaches as shown in our first evaluation. NEGWeb confirmed three out of four defects that are reported in the literature and are related to neglected conditions. NEGWeb detected three new defects in the Columba application, which is a popular email client. We showed that NEGWeb can identify defects in existing open source applications when a set of interested APIs is given as input. This feature could be quite useful for APIs such as security APIs. We also showed that NEGWeb can deal with the scalability issues in processing the large number of related code samples gathered from a CSE.

5. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs and CSE used are representative of true practice. The current subjects range from small-scale applications such as Java Servlets to large-scale applications such as BCEL, Hibernate, and Columba. We used only one CSE, i.e., Google code search, which is a well-known CSE. These threats could be reduced by more experiments on wider types of subjects and by using other CSEs in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our

[4] `http://jpim.cvs.sourceforge.net/jpim/`

NEGWeb prototype might cause such effects. There can be errors in our inspection of source code for confirming defects. To reduce these threats, we inspected the available specifications and also call sites in source code.

6. RELATED WORK

The most related work to our NEGWeb approach is the approach developed by Chang et al. [4] that applies frequent subgraph mining on C code to mine implicit condition rules and to detect neglected conditions. Both NEGWeb and their approach target at the same type of defects: neglected conditions. NEGWeb significantly differs from Chang et al.'s approach in three main aspects. First, their approach is limited on a much smaller scale of code repositories (in fact, only one project code base) than NEGWeb, which exploits a CSE to search for billions of lines of code. Second, the scalability of their approach is heavily limited by its underlying graph mining algorithms, which are known to suffer from scalability issues, whereas NEGWeb uses simple statistics to surface out condition rules, being much more scalable. Third, their approach reports "few apparent violations of rules" in the code base being analyzed, whereas NEGWeb detected not only known bugs detected by other existing approaches but also previously unknown bugs.

PR-Miner developed by Li and Zhou [10] uses frequent itemset mining to extract implicit programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann [11] uses association rule mining to extract simple rules from software revision histories for Java code and detect bugs related to rule violations. PR-Miner or DynaMine may suffer from issues of high false positives as their rule elements are not necessarily associated with program dependencies. In addition, NEGWeb targets at a much larger scale of code bases than PR-Miner or DynaMine.

Williams and Hollingsworth [16] incorporates an API call return value checker for C code, which checks that a value returned by an API call is tested before being used. This type of return-value testing before use falls into a subset of the types of rules being mined by NEGWeb. Different from their tool, NEGWeb does not require or rely on version histories, which may not include the types of bug fixing (required by their tool) related to the rules being mined. Acharya et al. [2] developed a tool to mine interface details (such as an API call's return values on success or failure and error flags) from model-checker traces for C code, and then generate interface robustness properties for bug finding. Similar to Williams and Hollingsworth [16], Acharya et al.'s tool mines only a subset of neglected conditions (e.g., return-value testing before use) mined by NEGWeb. In addition, as shown by Acharya et al. [2], only the interface details of 22 out of 60 POSIX API functions can be successfully mined by their tool, whereas NEGWeb exploits a CSE to alleviate the issue by collecting relevant API call usages from the web.

Engler et al. [5] proposed a general approach for finding bugs in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Their approach allows users to define rule templates. NEGWeb follows a similar methodology to find bugs. However, beyond the general rule templates proposed in their approach, NEGWeb's rule templates are more specific to detecting neglected conditions around API calls and NEGWeb incorporates various heuristics to help reduce false positives.

7. CONCLUSION

We developed a framework, called NEGWeb, that tries to address the issue of lacking relevant data points faced by the existing static defect finding approaches that mine programming rules from source code of one or a few project code bases. NEGWeb tries to address the preceding issue by leveraging a code search engine, which can search for related code samples in billions of lines of available open source code. NEGWeb detects violations related to neglected conditions around individual API calls by mining condition patterns from the gathered code samples. We evaluated our framework with five open source projects and confirmed the top 25 mined condition patterns. We confirmed three known Java defects in the literature and found three new defects in a large-scale application called Columba. We also detected defects in existing open source applications that reuse a given API. In future work, we plan to extend NEGWeb framework to the C programming language.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *Proc. ESEC/FSE*, September 2007.

[2] M. Acharya, T. Xie, and J. Xu. Mining Interface Specifications for Generating Checkable Robustness Properties. In *Proc. ISSRE*, pages 311–320, November 2006.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. POPL*, pages 4–16, 2002.

[4] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proc. ISSTA*, pages 163–173, 2007.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. SOSP*, pages 57–72, 2001.

[6] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[7] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[8] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Pro. ICSE*, pages 291–301, 2002.

[9] The Koders source code search engine, 2005. `http://www.koders.com`.

[10] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Codes. In *Proc. FSE*, pages 306–315, 2005.

[11] V. B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proc. ESEC/FSE*, pages 296–305, 2005.

[12] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-Sensitive Inference of Function Precedence Protocols. In *Proc. ICSE*, pages 240–250, 2007.

[13] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proc. ISSTA*, pages 174–184, 2007.

[14] S. Thummalapenta and T. Xie. PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, November 2007.

[15] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *Proc. ESEC/FSE*, September 2007.

[16] C. C. Williams and J. K. Hollingsworth. Recovering system specific rules from software repositories. In *Proc. MSR*, pages 1–5, 2005.

[17] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. of MSR*, pages 54–57, 2006.

[18] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proc. ICSE*, pages 282–291, 2006.