NEGWeb: Detecting Neglected Conditions via Mining Programming Rules from 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

Neglected conditions, also referred as missing paths, are known to be an important class of software defects. Revealing neglected conditions around individual API calls in an application requires the knowledge of programming rules that must be obeyed while reusing those APIs. To mine those implicit programming rules and hence to detect neglected conditions, we develop a novel framework, called NEGWeb, that substantially expands mining scope to billions of lines of open source code available on the web by leveraging a code search engine. We evaluated NEGWeb to detect violations of mined rules 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 541 classes and 2225 methods. We also report a high percentage of real rules among the top 25 reported patterns mined for APIs provided by five popular open source applications.

1. INTRODUCTION

Neglected conditions, also referred as missing paths, are known to be an important class of software defects. In particular, neglected conditions refer to (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. Neglected conditions are considered to be one of the primary reasons for many fatal issues such as *security* or *buffer overflow* vulnerabilities. A recent study conducted by Chang et al. [5] shows that 66% (109/167) of bug fixes applied in the Mozilla Firefox project are due to neglected conditions. These facts reinstate the significance of neglected conditions and a need for more research in this specific area.

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

Neglected conditions can be revealed either through inspection or by applying static or runtime verification tools. However, such processes require the knowledge of programming rules that must be obeyed while reusing APIs. In practice, these programming rules are often not available due to lack of documentation. Even when such documentation exists, it is often outdated [10]. To tackle the issue of lacking programming rules, various approaches have been developed in recent years to mine programming rules from program executions [4,7,19], program source code [1,2,5,6,11,13,14,16], or version histories [12,17]. In particular, a recent approach by Chang et al. [5] specifically addresses the problem of neglected conditions. Their approach applies frequent sub-graph mining on C source code to mine implicit condition rules and applies mined condition rules to detect violations. However, their approach is not scalable to large code bases as frequent sub-graph mining suffers from scalability issues. Moreover, their approach mines programming rules from a small number of project code bases and there are often too few relevant data points in these code bases to support the mining of desirable patterns.

Mining patterns from a small number of code bases may result in a low percentage of real programming rules and thereby a high percentage of false positives (e.g., reported warnings that do not indicate real defects or patterns that do not reflect real programming rules) among detected violations. For example, 87.8% of violations in the AspectJ application detected by the approach of Wasylkowski et al. [16] are identified as false positives. Furthermore, among top 15 violations detected by their approach, 9 of them are classified as false positives. The primary reason for a high percentage of false positives could be that their approach mines programming rules from a small number of code bases. These figures indicate that there is certainly a need for an approach that can mine patterns from a much larger number of code bases and yet scalable. To address the preceding issues, we develop a novel approach based on code searching, where we search for related code examples of each API in existing open source projects available on the web and mine those code examples to extract programming rules that must be obeyed in reusing that API. To mine gathered code examples, we use a simple statistical analysis that does not suffer from scalability issues and yet effective in analyzing and mining programming rules.

To reduce programmers' effort in using our approach, we define a few rule templates that capture programming rules that must be obeyed while reusing APIs. We conducted a preliminary investigation with the BCEL library[1] by inspecting the source code of the library to identify the percentage of programming rules that can be captured by our pre-defined rule templates. In our investigation, we found that our rule templates can capture up to 88% (441/497) of programming rules available around method calls made inside the BCEL library. This percentage is already quite high to indicate the wide scope of rules that can be potentially mined by our approach. Moreover, programmers are free to create new rule templates and our approach can easily be extended to handle those new templates.

In particular, our approach accepts an input source application and extracts APIs reused by the source application. Our approach interacts with a code search engine (CSE) such as Google code search [8] to gather related code examples for each API and mines gathered code examples to extract implicit programming rules that must be obeyed in reusing the API. Our approach creates programming rules as instances of defined rule templates and uses those programming rules to detect violations in the given source application. Our new approach is the first bug finding approach with this scale and based on a CSE.

While enjoying benefits provided by a CSE, our approach faces one new challenge that existing approaches do not face: the code examples returned by a CSE are often partial and not compilable, because CSE retrieves individual source files with usages of the given query API, instead of entire projects. Code examples are not compilable means that these code examples are syntactically correct but do not contain required information to resolve object types. We develop several new heuristics along with previously developed heuristics [15] to tackle this challenge.

Our previous work, on MAPO [18] and PARSEWeb [15], also developed approaches for mining source files returned by a CSE but these previous approaches focus on mining API usage patterns to assist programmers to write effective API client code. In contrast, our new NEGWeb framework developed in our approach 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.

This paper makes the following main contributions:

- A novel framework for finding neglected conditions 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. In our preliminary investigation, we found that our rule templates can capture up to 88% of programming rules available around method calls made inside the BCEL library.
- A set of heuristics for analyzing partial code samples and a scalable mining algorithm that computes frequent programming rules. Our mining algorithm includes several heuristics that attempt to reduce false positives among mined programming rules.
- 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 reuses 541 classes and 2225 methods. We also report a high percentage of real rules among the top 25 reported patterns mined for APIs provided by five open source applications.

[1]http://jakarta.apache.org/bcel/

```
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.

The rest of the paper is organized as follows. Section 2 describes rule templates defined by our approach. Section 3 explains the framework through an example. Section 4 describes key aspects of the framework. Section 5 discusses evaluation results. Section 6 discusses our limitations and future work. Section 7 discusses threats to validity. Section 8 presents related work. Finally, Section 9 concludes.

2. RULE TEMPLATES

We next present the rule templates defined by our approach for capturing programming rules around individual API calls (for generality, we refer methods in an API as individual API calls). We use the code example shown in Figure 1 as an illustrative example for describing our rule templates.

In general, a method invocation in Java consists of four elements: the receiver object, method name, arguments, and return object. For example, the method invocation in Statement 13 of the code example has the receiver object `verf`, argument `mId`, and the return object `vr2`. The condition checks for a method invocation can appear before the call site (say, preceding conditions), or can appear after the call site (say, succeeding conditions). More specifically, we are interested in condition checks on the receiver object and arguments before a call site and condition checks on the receiver object and return object after the call site. To capture such condition checks as programming rules, we define a rule template and a programming rule as follows:

Definition 1: *A rule template is a five-tuple (T_{type}, O_{type}, MI, AI, POS), where*

T_{type}: template type
O_{type}: object type such as a receiver or return
MI: method invocation
AI: optional additional information
POS: location with respect to a call site

Definition 2: *A programming rule is an instance of a rule template.*

The element T_{type} (template type) of a rule template describes the type of the condition checks in the programming rule, whereas the element O_{type} (object type) represents the participating object of the individual API call such as the receiver, argument, or return object. The element MI (method invocation) stores the individual API call. The element AI is optional additional information associated with each programming rule. The information stored in the AI element is dependent on T_{type}. As we capture both preceding and succeeding programming rules, the element POS describes the location of the programming rule with respect to the call site of an individual API call. We next describe each element in detail.

Table 1 shows several possible values for the T_{type} element of the rule template. For each template type, we present the name, description with an example, and additional information (AI) that is associated with the programming rule. The additional information is dependent on the T_{type} element. For example, the template type `direct-null-check` captures programming rules such as `null` checks done on the receiver, arguments, or return objects. In the description of each template type, we use notation `var` to denote a receiver, argument, or return object of the individual API call. The additional information for the template type `direct-null-check` stores the operator involved in the conditional expression such as "==" or "!=". Column "Coverage" shows the number of rules that belong to each template type divided by the total number of rules that we found in our preliminary investigation with BCEL. All template types described in the table include 88% of the rules we found during our investigation.

Our template types are broadly classified into two different categories: direct and indirect. The rationale behind these two categories is that a condition check can be performed directly on the variable or can be done indirectly through another method invocation, say MI, where the current variable is an argument of that method invocation. The template type `indirect-null-check` in the table is an example for the indirect template type. For indirect template types, the optional additional information also stores the associated method invocation. This gathered additional information for each programming rule is later used while detecting violations.

Apart from template types shown in the table, we define two more template types that are specific to the receiver object of a method invocation.

must-happen-before. this template type represents condition checks on other method invocations of the same receiver object preceding the call site of an individual API call. For example, Statements 6 to 9 in the code example describe 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`. The corresponding programming rule can be captured by this template type as "`direct-const-check` on `return` of `doPass1` `must-happen-before doPass2`".

must-happen-after. this template type captures condition checks on other method invocations of the same receiver object succeeding the call site of an individual API call. In general, this template 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 template type can result in many false positives based on our empirical investigation. Therefore, in the NEGWeb framework, we limit this template type to those APIs (such as Java Util APIs) that are often suggested to be used in loops.

Definition 2 describes programming rules as instances of a rule template. However, to provide better readability, we present programming rules in a textual form in this paper. For example, the programming rule "`direct-null-check` on `receiver before doPass1`" indicates that a `null` check should be done on the receiver object of `doPass1` before its call site. In this instance, T_{type} is `direct-null-check`, O_{type} is `receiver`, MI is `doPass1`, and POS is `before`.

3. EXAMPLE

We next use an example to describe how our NEGWeb framework mines programming rules from code examples gathered from a CSE and uses mined rules to detect violations in an input source application. We use the class `org.apache.bcel.verifier.Verifier` and its methods `doPass1`, `doPass2`, `doPass3a`, and `doPass3b` from the BCEL library for explaining our framework. The `Verifier` class is used for verifying generated class files.

Initially, NEGWeb constructs a query with the class name and gathers related code examples from a CSE. A code example gathered from a CSE is shown in Figure 1. NEGWeb parses each method declaration of the code example and builds a control flow graph. NEGWeb uses *dominance* and *data-dependency* concepts, and gathers preceding and succeeding condition checks on receiver, argument, or return objects around the nodes that include any of the methods such as `doPass1` or `doPass2`. NEGWeb creates rule candidates that describe the captured condition checks. A few rule candidates extracted from the code example are shown below:

```
01: direct-null-check on receiver before doPass1
02: direct-const-check on return of
      doPass1 must-happen-before doPass2
03: direct-const-check on return after doPass1
      with VerificationResult.VR_OK
04: direct-expr-check on argument1 before doPass3a
```

The rule candidate in Line 1 describes that before invoking the `doPass1` method, a `null` check must be done on the receiver variable of that method. An example of the template type `must-happen-before` 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 that the argument must be verified before invoking the `doPass3a` method.

NEGWeb mines extracted rule candidates to compute frequent rules, referred as mined programming rules. NEGWeb applies these mined programming rules on the input source application to detect violations. For example, consider the code sample below with a potential violation.

```
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 rule candidate in Line 3. Sometimes, the same violation can appear multiple times because a code sample can violate multiple programming rules. For example, the preceding code sample also violates the rule candidate in Line 2. This example motivates the idea of exploiting a CSE for gathering related code examples and applying analysis techniques to capture implicit

Table 1: Possible T_{type} values of a rule template and associated additional information

Name	Description	Additional Info	Coverage
direct-null-check	direct null check. e.g., if(var != null) { .. }	operator involved, e.g., !=	97/497
indirect-null-check	if variable is an argument of a method invocation. e.g., if(MI(var) == null) { .. }	operator involved, e.g., == method invocation, MI	2/497
direct-boolean-check	if the variable type is boolean. e.g., if(var) { .. }		65/497
indirect-boolean-check	indirect boolean check. e.g., if(MI(var)) { .. }	method invocation, e.g., MI	25/497
direct-const-check	if the variable is compared with a constant. e.g., if(var == SUCCESS)	operator involved, e.g., == constant value, e.g., SUCCESS	110/497
indirect-const-check	indirect constant equality check. e.g., if(MI(var) == FAILURE)	operator involved, e.g., == constant value, e.g., FAILURE method invocation, e.g., MI	1/497
direct-retval-check	if the variable is compared with the return value of a method invocation. e.g., if(var < $MI_{other}()$)	operator involved, e.g., < method invocation, e.g., $MI_{other}()$	0/497
indirect-retval-check	if the variable is compared indirectly with the return value of a method invocation. e.g., if(MI(var) > $MI_{other}()$)	operator involved, e.g., > method invocation, e.g., MI method invocation, e.g., $MI_{other}()$	0/497
instance-check	if the conditional check involves `instanceof` operator e.g., if(var instanceof Integer)	type-name, e.g., Integer	106/497
direct-expr-check	if the variable is compared with an expression such as another variable. e.g., if(var < expr) { ... }	operator involved, e.g., < other expression, e.g., expr	35/497

programming rules for detecting violations in a given input application. However, there are many other issues that are not obvious in this illustrative example, as described next. (1) How do we analyze partial code examples gathered from a CSE? (2) How do we exploit program dependencies in the source code while extracting rule candidates? (3) How do we mine extracted rule candidates to make our approach scalable to large code bases? We address these issues in the subsequent section, where we present key aspects of our NEGWeb framework.

4. FRAMEWORK

Our NEGWeb framework consists of seven major components: the application scanner, code search engine, code downloader, code analyzer, rule-candidate extractor, rule-candidate miner, and anomaly detector. Figure 2 shows an overview of all components in NEGWeb. The application scanner accepts a source application as input and gathers the external classes and methods reused by that application. The code downloader interacts with a CSE to gather related code examples of each external class. The code analyzer analyzes these gathered code examples statically and builds a control flow graph for each method declaration in gathered code examples. The rule-candidate extractor exploits the control flow graph and extracts rule candidates around call sites of each external method invocation. The rule-candidate miner mines extracted rule candidates and identifies programming rules, which are further used by the anomaly detector to detect violations in the given application. We next present the details of each component.

4.1 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 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.

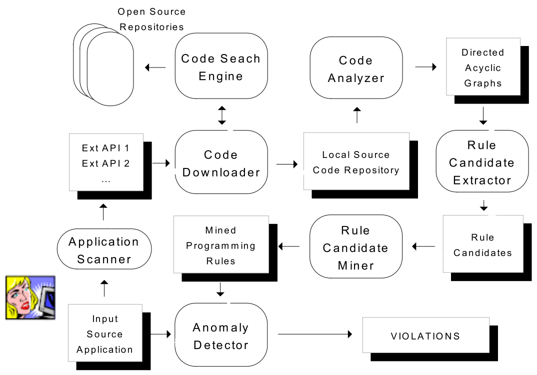

Figure 2: Overview of NEGWeb framework

4.2 Code Downloader

The code downloader accepts the set of external classes and methods, and interacts with a CSE for searching and gathering related code examples. A code example gathered from CSE for the query “`lang:java Verifier`” related to the `Verifier` class of the BCEL library is shown in Figure 1. The code samples returned by a CSE are often partial and not compilable, as CSE retrieves only individual source files instead of entire projects. The code downloader stores gathered code examples in local file system and is referred as a local source code repository.

Our framework uses Google code search (GCSE) [8] for collecting related code examples 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 code examples.

4.3 Code Analyzer

The research question addressed by the code analyzer is how to analyze the partial code examples stored in the local source code repository and build a control flow graph that can be exploited for extracting implicit programming rules around individual API calls. To analyze these partial code examples, the code analyzer uses several type heuristics that help identify object types in these code examples. Our heuristics are based on rules used by a type checker of a Java compiler. For example, a type checker verifies whether left and right hand sides of an assignment statement are compatible to each other. Our heuristics use an opposite form of type checking, where we infer the type of an unknown expression from expressions with known object types. For example, consider the method invocation in Statement 6 of the code sample shown in Figure 1. As we do not have access to the method signature of the method `doPass1` of the `Verifier` class, we cannot directly look up the return type of this method invocation. However, we can infer the return type from the left hand side of the assignment statement as `VerificationResult` or its sub class.

The code analyzer analyzes code examples statically through abstract syntax trees and transforms into a Directed Acyclic Graph (DAG). 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 represent other 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 (suffix indicates the statement id), the code analyzer also extracts all variables, say $\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node. The code analyzer identifies the type of condition check that is used to determine the template type. The extracted information is associated with the control node in the constructed DAG. Therefore, each control node CT_i includes a set of pairs $\{(V_1, T_{type1})$,(V_2, T_{type2}),..,$(V_n, T_{typeN})\}$. For example, the control node CT_5 includes the {(`verf`, `direct-null-check`)} pair.

While constructing the DAG, the code analyzer identifies nodes in the graph that include external methods and marks those nodes as API Nodes, 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 rule-candidate extractor that extracts rule candidates around each external method invocation. In the example code sample, the code analyzer identifies nodes related to Statements 6, 9, 13, and 14 as API Nodes.

4.4 Rule-Candidate Extractor

The research question addressed by the rule-candidate extractor (RCExtractor) is how to extract rule candidates around individual API calls by exploiting program dependencies among rule elements. Failure to consider these program dependencies may result in rules that are not semantically related as shown in the limitations of the PR-Miner [11] and DynaMine [12] approaches. To exploit program dependencies, RCExtractor uses the concept of *dominance* with a blend of *control-flow* and *data-flow* dependencies. RCExtractor performs an intra-procedural analysis through the constructed CFG and uses different algorithms for capturing preceding and succeeding rule candidates. If the same rule candidate appears more than once among gathered code examples, RCExtractor stores the number of times the rule candidate is detected. We next describe how RCExtractor identifies these preceding and succeeding rule candidates.

4.4.1 Preceding Rule Candidates

RCExtractor extracts preceding rule candidates by using the concept of dominance. The definition of dominance [3] is given below:

Definition 3: *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, RCExtractor identifies the dominant CT_i nodes for each AN_k node. For example, CT_5 dominates AP_6. RCExtractor computes the 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 {`receiver`, `argument1`, ..., `argumentN`}. If the intersection $\{V_1, V_2, ..., V_n\}$ ∩ {`receiver`, `argument1`, ..., `argumentN`} $\neq \emptyset$, RCExtractor 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 redefined in the path between CT_i and AN_k nodes. If the AN_k node is data dependent on the CT_i node, RCExtractor instantiates a rule template to create a rule candidate. For example, the associated rule candidate for nodes CT_5 and AN_6 in the example code sample is "`direct-null-check on receiver before doPass1`", which indicates that a `null` check must be done on the the receiver variable of the method `doPass1` before the call site of `doPass1`.

4.4.2 Succeeding Rule Candidates

RCExtractor extracts the succeeding rule candidates by using the concept of post-dominance. Initially, RCExtractor identifies post-dominant CT_i nodes for each AN_k node and computes the intersection between the receiver and return object of AN_k node, say {`receiver`, `return`}, and the variable set associated with the CT_i node, say $\{V_1, V_2, ..., V_n\}$. If the intersection {`receiver`, `return`} ∩ $\{V_1, V_2, ..., V_n\}$ $\neq \emptyset$, RCExtractor checks whether the identified CT_i node is data-dependent on the AN_k node. A succeeding rule candidate between AN_6 and CT_7 nodes is "`direct-const-check` on `return after doPass1` with `VerificationResult.VR_OK`".

4.4.3 Absence of Rule Candidates

Sometimes the call sites of external method invocations do not have any preceding or succeeding condition checks; such call sites may include potential locations for neglected conditions. To store the number of such call sites that do not have any condition checks, the rule-candidate extractor associates an attribute called *No Rule Candidates* (`NRC`) with each such external method invocation. This attribute is used by the rule-candidate miner while mining for frequent rule candidates around an external method invocation.

4.5 Rule-Candidate Miner

The rule-candidate miner (RCMiner) mines frequent rule candidates, referred as mined programming rules, among all extracted rule candidates of an external method invocation. The primary objective of RCMiner is to reduce the number of false positives while computing the mined rules. We use the notation RC_i to refer to the i_{th} rule candidate of an external method invocation and RCF_i to refer to frequency of the rule candidate, i.e., the number of times the rule candidate is detected among gathered code examples. Initially, RCMiner computes support for each rule candidate.

Definition 4: *The support of a rule candidate, RCS_i, is defined as*

RCS_i = RCF_i / ($\sum_{j=1}^{N}$ RCF_j + NRC)

`where N is the number of` RC_i `for a method invocation.`

The rationale behind using NRC in computing RCS_i is

```
Input: $\{(RC_1,RCS_1), ..., (RC_n,RCS_n)\}$, $NRC$, $LT$, $UT$
Output: Set of mined programming rules
Initialize $MPRSet$;
//Step 1
if $(NRC \geq UT)$ or $(All\ RCS_i < LT)$ then
    return $null$;
end
//Step 2
if $Any\ RCS_i \geq UT$ then
    Append all $RC_i$ whose $RCS_i \geq UT$ to $MCPSet$;
    Set Confidence of those $RC_i$ to $HIGH$;
    return $MCPSet$;
end
Set $MAX\_SUP$ to $\mathbf{max}\{RCS_1, ..., RCS_n\}$;
//Step 3
if $All\ RCS_i$ are near to $MAX\_SUP$ then
    Append all $RC_i$ to $MCPSet$;
    Set Confidence of all $RC_i$ to $HIGH$;
    return $MCPSet$;
end
//Step 4
if $MAX\_SUP < NRC$ then
    Set $CONF\_LEVEL$ to $LOW$;
end
else
    Set $CONF\_LEVEL$ to $AVERAGE$;
end
for each $RC_i$ do
    if $RCS_i$ is near to $MAX\_SUP$ then
        Append $RC_i$ to $MCPSet$;
        Set Confidence of $CP_i$ to $CONF\_LEVEL$;
    end
end
return $MCPSet$;
```
Algorithm 1: Mining algorithm in NEGWeb.

to identify the actual support of a rule candidate among all call sites of the method invocation. The consideration of NRC can help reduce the number of false positives among the mined programming rules.

RCMiner uses the algorithm shown in Algorithm 1 for mining rule candidates. The algorithm is executed for each external method invocation to identify mined programming rules of that method invocation. RCMiner uses two threshold values Upper Threshold (UT) and Lower Threshold (LT) for computing mined programming rules and for classifying the mined rules into three confidence levels: `HIGH`, `AVERAGE`, and `LOW`. We describe the need for these confidence levels while explaining our algorithm. These confidence levels are later used for sorting the mined programming rules.

The mining algorithm accepts the set of extracted rule candidates (and their support values) of an external method invocation, along with the associated NRC value. We next describe the steps in our algorithm along with the rationale behind those steps.

Step 1. if the value of NRC is greater than UT or the support values of all rule candidates are less than LT, RCMiner ignores all the extracted rule candidates for the current external method invocation. The rationale behind this mechanism is that the external method invocation has many call sites with no rule candidates around. Therefore, all extracted programming rules of this external method invocation may not be significant and can be ignored.

Step 2. RCMiner checks whether there are any rule candidates with support greater than UT and identifies those candidates as mined rules with confidence `HIGH`. The rationale behind this mechanism is that these rule candidates are of the highest support compared to other rule candidates of the current external method invocation.

Step 3. if no rule candidate has support greater than UT, instead, the support values of all rule candidates are near to the maximum support value, referred as `MAX_SUP`, RCMiner identifies all rule candidates as mined programming rules with confidence level `HIGH`. The rationale behind this mechanism is that a group of rule candidates can often appear together in gathered code examples. For example, consider that two rule candidates of a method invocation MI, say "`direct-const-check` on `return after MI` with `SUCCESS`" and "`direct-const-check` on `return after MI` with `FAILURE`" appeared together in 10 code examples. The computed support values for each rule candidate will be 0.5, resulting in a low confidence level. However, these rule candidates appeared in all code examples and should have more importance. We introduced confidence levels to handle these scenarios where some rule candidates, although with low support values, are classified as programming rules with `HIGH` confidence.

Step 4. we use the value of NRC to classify non-high rule candidates into other confidence levels `AVERAGE` and `LOW`. If the maximum support value is less than NRC, we set the confidence level as `LOW`. The rationale behind this mechanism is that the number of call sites with no rule candidates is greater than the number of call sites of the rule candidate with the maximum support. Therefore, these rule candidates might not be of more importance. In case the maximum support value is greater than NRC, we set the confidence level as `AVERAGE`.

RCMiner sorts all mined programming rules based on three attributes: confidence level, support (RCS_i), and frequency (RCF_i). We used values 0.75 for UT and 0.1 for LT. These values are based on our empirical investigation with different subjects.

4.6 Anomaly Detector

The anomaly detector accepts mined programming rules as input and detects violations of these programming rules in an input application. The anomaly detector extracts the rule candidates for each external method-invocation call site in the input source application, and checks whether the newly extracted rule candidates contain mined programming rules. Any missing mined rules are reported as violations. For each detected violation, the anomaly detector inherits attributes such as confidence level and support of the associated programming rule. The anomaly detector sorts all detected violations based on confidence level and support.

In addition, the anomaly detector uses two heuristics to reduce the number of false positives.

Anomaly Heuristic 1:*A violation detected for succeeding programming rules 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 condition check often appears at the call site of the enclosing method declaration.

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

The rationale behind this heuristic is that call sites with no condition checks can have a higher chance of being a defect than call sites with a few condition checks around.

5. EVALUATION

We conducted four different evaluations on NEGWeb to show that NEGWeb can effectively mine real rules from related code examples 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 programming rules and manually confirmed the extracted top 25 programming rules 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 programming rules in a novel way to detect violations in available open source applications. As NEGWeb mainly targets 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/`.

5.1 Open Source Applications

In this section, we describe programming rules mined by NEGWeb for five subject applications that vary in size and purpose. Table 2 shows the subject applications used in our evaluation and their characteristics. 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 multi-tier server-side Java applications. The common reason for selecting these applications is the presence of programming rules as described by their associated documentations that can help confirm the real rules.

NEGWeb usually accepts an input application and mines programming rules for external APIs used by the input source application. In this evaluation, as we plan to mine programming rules of methods provided by input application, we configured NEGWeb to accept a set of classes and methods directly, and mine programming rules of those classes and methods. We extracted classes and methods of these five subject applications and provided as input to NEGWeb. Column "Input Application" and its sub-columns "Classes" and "Methods" of Table 2 show the number of classes and methods in each application.

Sub-column "Samples" shows the number of code examples gathered from CSE for each application. For example, NEGWeb gathered and analyzed 49, 858 code examples for Java Util packages. The number of programming rules mined for each application is shown in Column "Prog. Rules".

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

Table 3: Mined programming rules of Java Util package

Object Type	Template Type	#Total	#Rule	#UP	#FP
`receiver`	all types in table 1	5	3	0	2
`argument`	all types in table 1	6	5	0	1
`return`	all types in table 1	19	12	7	0
`receiver`	`must-happen-before`	12	11	0	1
`receiver`	`must-happen-after`	22	5	10	7
SUM		**64**	**36**	**17**	**11**

UP: usage pattern, FP: false positive

5.1.1 Analysis of Programming Rules

We manually analyzed the first 25 rules of each application and classified them into three categories: real rules, usage patterns, and false positives. Real 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[2], or the source code of the application for classifying the mined programming rules into these three categories.

As shown in Table 2, most of the programming rules 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 programming rule that appeared among these false positives. This common rule describes that an `instance-check` with the class `HttpServletRequest` must be performed on the `receiver` object before invoking the methods of the class `ServletRequest`. Although this pattern is a common usage, the available specification of Java Servlet does not confirm this extracted programming rule 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.

We manually classified all mined programming rules 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 rules. Table 3 shows classification categories of all mined rules. Among all mined patterns, the real rules constitute 56.25% (36/64), usage patterns constitute 26.56% (17/64), and false positives constitute 17.18% (11/64). 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 programming rules based on object types (O_{type}) of a rule template. NEGWeb is effective in extracting and mining real rules for object type `return` and template type `must-happen-before`, which are usually the main sources of neglected conditions. The template type `must-happen-after` has the largest number of false positives.

We next describe mined rules for the `Matcher` class of Java Util pacakges. The `Matcher` class is an engine that performs matching operations on a character sequence by interpreting a given regular expression. The rules detected by NEGWeb are as follows:

```
01: direct-boolean-check on return after find
02: direct-boolean-check on return of find
          must-happen-before start
03: direct-boolean-check on return of find
          must-happen-before group
04: direct-boolean-check on return of find
```

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

Table 2: Programming rules mined by NEGWeb

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

```
        must-happen-after start
05: direct-null-check on return of group
        must-happen-after find
06: direct-boolean-check on return of find
        must-happen-after group
```

Rule 1 describes that a `boolean` check must be performed on the return value of the `find` method. Rules 2,3,4, and 6 describe that the `find` method must precede and succeed methods `start` and `group`. These rules indicate that the methods `start` and `group` are often used inside a loop with a boolean check on the `find` method.

NEGWeb also mined undocumented programming rules in the subject applications. We confirmed these programming rules by inspecting the source code and comments in source files. For example, in the BCEL library, the `copy` method of the `Instruction` class cannot be used for its child class `Select`. Therefore, before using the `copy` method, a condition check on the receiver variable must be performed. NEGWeb detected the programming rule "`instance-check` on `receiver before copy` with `Select`", which describes that an instance check with the `Select` class must be done before invoking the `copy` method. Similarly, NEGWeb detected that while using the methods `doPass3a` and `doPass3b` of the `Verifier` class, 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.

5.1.2 Defects in Open Source World

As NEGWeb mines programming rules by gathering related code examples from available open source applications, we applied the mined rules in a novel way to detect violations in the gathered code examples themselves. This feature of NEGWeb could be useful while dealing with APIs such as security APIs to check whether there are any security holes in applications on the web. We used the same subject applications used in the preceding evaluation and applied mined programming rules on gathered code examples to detect violations.

Given the large number of detected violations, we manually analyzed violations of a mined rule (from the top 25 mined rules 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. [16]. 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 represent the scenario where mined rules 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.

The chosen programming rule from each application and the classification categories for the first 50 violations of the programming rule are shown in Table 4. Column "Support" gives the support value of the programming rule. Column "Open Source" gives the total number of open source projects that contain those detected violations. The total number of violations are shown in Column "Violations". 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 method used from this application is "`ServletConfig,getInitParameter(String)`" and the programming rule used is "`direct-null-check` on `return` of `getInitParameter`", which indicates that a `null` 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 input parameter does not exist. However, 31 open source projects violated this mined programming rule. Among the top 50 violations, 38 violations are classified as defects in our inspection. We found some 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 `null` 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 `null` check can cause `NullPointerException`. We also found that the same piece of code with 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, 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.

5.2 Real defects from the literature

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

Table 4: Analysis of violations detected in open source world

Application	Programming rule	Support	#Open Source	# Violations	Categories of first 50 violations				
					#Defect	#CS	#WP	#Hint	#FP
Java Util APIs	Matcher: `direct-boolean-check` on `return` of `find` `must-happen-before` `group`	0.810	7	21	0	12	2	3	4
BCEL	Type: `direct-boolean-check` on `return` of `equals`	0.967	2	7	0	0	0	1	6
Hibernate	Filter: `direct-null-check` on `receiver` of `setParameterList`	0.875	1	4	4	0	0	0	0
Java Servlet APIs	ServletConfig: `direct-null-check` on `return` of `getInitParameter`	0.577	31	50	38	2	0	8	2
Java Transaction APIs	TransactionManager: `direct-null-check` on `return` of `getTransaction`	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[3] [16] and two defects in Java SSE Library and Joeq detected by DIDUCE [9]. We next describe details of these defects and explain the evaluation results with NEGWeb.

5.2.1 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 (confidence level: `HIGH`) as the code sample violated the pattern "`direct-boolean-check` on `return` of `hasNext` `must-happen-after` `next`" of the class `Iterator`. NEGWeb confirmed the other defect reported by JADET that is also related to a similar scenario.

5.2.2 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 examples gathered from the code search engine, NEGWeb detected that only 5 code examples have the `null` check on their return object. 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 programming rules that can cause potential defects and to reduce the number of false positives among the detected violations.

5.3 Case Study: Columba

Columba 1.4[4] 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 that Columba reuses 541 external classes and 2225 external methods and mined programming rules for those external methods by interacting with a CSE. NEGWeb gathered and analyzed $309,757$ code examples ($\approx$ 50 million LOC) for mining these programming rules.

NEGWeb mined a total of 559 programming rules related to the external methods reused by Columba. Among these 559 programming rules, 189 rules did not detect any violations. The remaining 370 programming rules were used to detect 1647 violations. We manually analyzed violations of the first 25 programming rules that were used to detect 70 violations. We classified these 70 violations into different violation categories using the same classification criteria described in Section 5.1.2.

The results of our evaluation are shown in Table 5. Each row in the table represents a programming rule. Columns "Supp." and "RC" give the support and manually assigned category of each rule, respectively. Column "Total" gives the total number of violations detected by that rule. Among the first 25 mined rules, 19 programming rules 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 programming rules do not have any false positives. As each programming rule 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 rule 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 rules. 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

[3]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.

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

Table 5: Evaluation results of Columba case study.

SNo	Rank	Supp.	RC	#	Categories of violations				
				Total	#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			

expressions in assignment statements. We plan to address these limitations in near future work and these limitations can be addressed without any difficulty.

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*(<[ŝ<>]+>)s*`", no elements will be added to the `entries` list. The preceding code sample violated the mined rule "`direct-boolean-check` on `return` of `hasNext` `must-happen-before` `next`", 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.

The third defect is related to the JPIM[5] 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 "`direct-null-check` on `return after unmarshallContacts`", which indicates that a `null` check must be performed on the return value of the `unmarshallContacts` method. As this pattern is not available in the documentation, 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 `null` check on the return variable was done. The absence of the `null` check can cause a `NullPointerException`.

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

We next describe a code smell detected in the method `readInByteArray` of the class `StreamUtils`. We show the code example of the `readInByteArray` method as below:

```
...
byte[] result = new byte[in.available()];
in.read(result);
in.close();
return result;
```

The preceding code example violated the mined programming rule "`direct-const-check` on `return` of `read`", which describes that the return value of the `read` function must be verified. This return value gives the number of bytes that are actually read from the input stream. Although this violation can be a defect, we classified this violation as a code smell because this public method is not currently invoked in the application.

6. DISCUSSION AND FUTURE WORK

In our current implementation, we define a subset of possible template types that we found during our preliminary investigation with the BCEL library. Our main objective is to focus on the template types that are most common and could detect real defects in a given application while producing low false positives. We next describe an example template type that is not defined by our current implementation. We defined a template type called `must-happen-after` that is specific to the receiver object a method invocation. This template type captures the succeeding calls of the current method invocation on the same `receiver` object. A similar template type can be defined related to succeeding calls on the `return` object. In our future work, we plan to extend NEGWeb to handle several other template types that we found during our preliminary investigation and develop heuristics to reduce their induced false positives.

The current prototype does not handle a few Java constructs such as `switch` statements or conditional expressions in assignment statements. In future work, we plan to extend NEGWeb to handle those constructs. We also plan to extend NEGWeb framework to the C programming language.

7. 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 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.

8. RELATED WORK

The most related work to our NEGWeb approach is the approach developed by Chang et al. [5] 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 related code examples from open source code available on the web. 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 statistical approach to mine programming 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 [11] uses frequent itemset mining to extract implicit programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann [12] 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 [17] 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 [17], 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. [6] 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.

9. CONCLUSION

We developed a framework, called NEGWeb, that accepts a source application as input and detects neglected conditions around individual API calls in the application by mining programming rules from the open source code available on the web. NEGWeb attempts to address the issue of lacking relevant data points faced by the existing static defect finding approaches that mine programming rules from one or a few project code bases by leveraging a code search engine. NEGWeb defines rule templates for describing common neglected conditions and uses analysis heuristics that can help reduce false positives among detected violations. 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 intend to extend NEGWeb framework to the C programming language.

10. 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] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. In *IEEE Software*, pages 35–39, 2003.

[11] 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.

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

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

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

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

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

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

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

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