Mining Exception-Handling Rules as Sequence Association Rules

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

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

Abstract

Programming languages such as Java and C++ provide exception-handling constructs to handle exception conditions. Applications are expected to handle these exception conditions and take necessary recovery actions such as releasing opened database connections. However, exception-handling rules that describe these necessary recovery actions are often not available in practice. To address this issue, we develop a novel approach that mines exception-handling rules as sequence association rules of the form "(FC_c^1...FC_c^n) $\wedge$ FC_a $\Rightarrow$ (FC_e^1...FC_e^m)". This rule describes that function call FC_a should be followed by a sequence of function calls (FC_e^1...FC_e^m) when FC_a is preceded by a sequence of function calls (FC_c^1...FC_c^n). Such form of rules is required to characterize common exception-handling rules. We show the usefulness of these mined rules by applying them on five real-world applications (including 285 KLOC) to detect violations in our evaluation. Our empirical results show that our approach mines 294 real exception-handling rules in these five applications and also detects 160 defects, where 87 defects are new defects that are not found by a previous related approach.

1 Introduction

Programming languages such as Java and C++ provide exception-handling constructs such as `try-catch` to handle exception conditions that arise during program execution. Under these exception conditions, programs follow paths different from normal execution paths; these additional paths are referred to as *exception* paths. Applications developed based on these programming languages are expected to handle these exception conditions and take necessary recovery actions. For example, when an application reuses resources such as files or database connections, the application should release the resources after the usage in all paths including *exception* paths. Failing to release the resources can not only cause performance degradation, but can also lead to critical issues. For example, if a database lock acquired by a process is not released, any other process trying to acquire the same lock hangs till the database releases the lock after timeout. A case study [21] conducted on a real application demonstrates the necessity of releasing resources in exception paths for improving reliability and performance. The case study found that there was a surprising improvement of 17% in performance of the application after correctly releasing resources in the presence of exceptions.

Software verification can be challenging for exception cases as verification techniques require specifications that describe expected behaviors when exceptions occur. These specifications are often not available in practice [10]. To address this issue, association rules of the form "$FC_a \Rightarrow FC_e$" are mined as specifications [22], where both FC_a and FC_e are function calls that share the same receiver object. These specifications are used to verify whether the function call FC_a is followed by the function call FC_e in all exception paths. However, simple association rules of this form are often not sufficient to characterize common exception-handling rules. The rationale is that there are various scenarios where FC_a is not necessarily followed by FC_e when exceptions are raised by FC_a, although both function calls share the same receiver object.

We next present an example using Scenarios 1 and 2 (extracted from real applications) shown in Figure 1. Scenario 1 attempts to modify contents of a database through the function call `Statement.executeUpdate` (Line 1.9), whereas Scenario 2 attempts to read contents of a database through the function call `Statement.executeQuery` (Line 2.8). Consider a simple specification in the form of an association rule "`Connection creation` $\Rightarrow$ `Connection rollback`". This rule describes that a `rollback` function call should appear in exception paths whenever an object of `Connection` is created. Although a `Connection` object is created in both scenarios, this rule applies only to Scenario 1 and does not apply to Scenario 2. The primary reason is that the `rollback` function call should be invoked *only* when there are any changes made to the database. This example shows that simple association rules of the form "$FC_a \Rightarrow FC_e$" are often insufficient to characterize exception-handling rules.

Scenario 1

```
1.1: ...
1.2: OracleDataSource ods = null; Session session = null;
     Connection conn = null; Statement statement = null;
1.3: logger.debug("Starting update");
1.4: try {
1.5:     ods = new OracleDataSource();
1.6:     ods.setURL("jdbc:oracle:thin:scott/tiger@192.168.1.2:1521:catfish");
1.7:     conn = ods.getConnection();
1.8:     statement = conn.createStatement();
1.9:     statement.executeUpdate("DELETE FROM table1");
1.10:    connection.commit(); }
1.11:    catch (SQLException se) {
1.12:              if (conn != null) { conn.rollback(); }
1.13:              logger.error("Exception occurred"); }
1.14: finally {
1.15:    if(statement != null) statement.close();
1.16:    if(conn != null) conn.close();
1.17:    if(ods != null) ods.close();
1.18: }
```

Scenario 2

```
2.1: Connection conn = null;
2.2: Statement stmt = null;
2.3: BufferedWriter bw = null; FileWriter fw = null;
2.3: try {
2.4:     fw = new FileWriter("output.txt");
2.5:     bw = BufferedWriter(fw);
2.6:     conn = DriverManager.getConnection("jdbc:pl:db", "ps", "ps");
2.7:     Statement stmt = conn.createStatement();
2.8:     ResultSet res = stmt.executeQuery("SELECT Path FROM Files");
2.9:     while (res.next()) {
2.10:              bw.write(res.getString(1));
2.11:    }
2.12:    res.close();
2.13: } catch(IOException ex) { logger.error("IOException occurred");
2.14: } finally {
2.15:    if(stmt != null) stmt.close();
2.16:    if(conn != null) conn.close();
2.17:    if (bw != null) bw.close();
2.18: }
```

Figure 1. Two example scenarios from real applications.

The insufficiency of simple association rules calls for more general association rules, hereby referred to as *sequence association rules*, of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$". This sequence association rule describes that function call FC_a should be followed by function-call sequence $FC_e^1...FC_e^m$ in exception paths only when preceded by function-call sequence $FC_c^1...FC_c^n$. Using this sequence association rule, the preceding example can be expressed as "$(FC_c^1 FC_c^2) \wedge FC_a \Rightarrow (FC_e^1)$", where

FC_c^1 : `OracleDataSource.getConnection`
FC_c^2 : `Connection.createStatement`
FC_a : `Statement.executeUpdate`
FC_e^1 : `Connection.rollback`

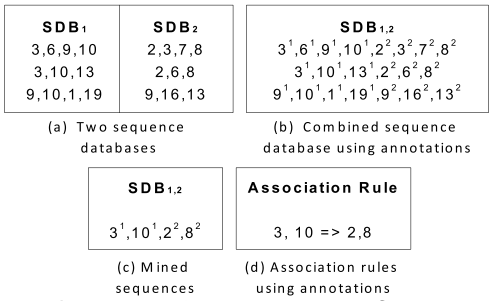
This sequence association rule applies to Scenario 1 and does not apply to Scenario 2 due to the presence of FC_a: `Statement.executeUpdate`. The key aspects to be noted in this rule are: (1) `Statement.executeUpdate` is the primary reason to have `Connection.rollback` in an exception path and (2) the receiver object of `Statement.executeUpdate` is dependent on the receiver object of `Connection.rollback` through the function-call sequence defined by $FC_c^1 FC_c^2$.

Our sequence association rules are a super set of simple association rules. For example, sequence association rules are the same as simple association rules when the sequence $FC_c^1...FC_c^n$ is empty. To the best of our knowledge, existing association rule mining techniques [2] cannot be directly applied to mine these sequence association rules. Therefore, to bridge the gap, we develop a new mining algorithm by adapting the frequent closed subsequence mining technique [19].

We further develop a novel approach, called CAR-Miner, that incorporates our new mining algorithm for the problem of detecting exception-handling rules in the form of sequence association rules by analyzing source code. Apart from mining sequence association rules, CAR-Miner addresses another challenge that is often faced by existing approaches [3, 11, 22], which mine rules from a limited data scope, i.e., from only a few example applications. Therefore, these approaches may not be able to mine rules that do not have enough supporting samples in those example applications, and hence the related defects remain undetected by these approaches. To address this challenge, CAR-Miner expands the data scope by leveraging a code search engine (CSE) for gathering relevant code samples from existing open source projects available on the web. From these relevant code samples, CAR-Miner mines exception-handling rules. We show the usefulness of mined exception-handling rules by applying these rules on five applications to detect violations. CAR-Miner tries to address problems related to the quality of code samples gathered from a CSE by capturing the most frequent patterns through mining.

This paper makes the following main contributions:

- A general mining algorithm to mine sequence association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$". Our new mining algorithm takes a step forward in the direction of developing new mining algorithms to address unique requirements in mining software engineering data, beyond being limited by existing off-the-shelf mining algorithms.
- An approach that incorporates the general mining algorithm to mine exception-handling rules that describe expected behavior when exceptions occur during program execution.
- A technique for constructing a precise Exception-Flow Graph (EFG), which is an extended form of a Control-Flow Graph (CFG), that includes only those exception paths that can potentially occur during program execution.
- An implementation for expanding the data scope to open source projects that help detect new related exception-handling rules that do not have enough supporting samples in an application under analysis. These rules can help detect new defects in the application under analysis.
- Two evaluations to show the effectiveness of our approach. (1) CAR-Miner detects 294 real exception-



SDB_1	SDB_2
3,6,9,10	2,3,7,8
3,10,13	2,6,8
9,10,1,19	9,16,13

(a) Two sequence databases

$SDB_{1,2}$
$3^1,6^1,9^1,10^1,2^2,3^2,7^2,8^2$
$3^1,10^1,13^1,2^2,6^2,8^2$
$9^1,10^1,1^1,19^1,9^2,16^2,13^2$

(b) Combined sequence database using annotations

$SDB_{1,2}$
$3^1,10^1,2^2,8^2$

(c) Mined sequences

Association Rule
3, 10 => 2,8

(d) Association rules using annotations

Figure 2. Illustrative examples of general algorithm.

handling rules in five different applications including 285 KLOC. (2) The top 50 exception-handling rules (top 10 real rules of each application) are used to detect a total of 160 real defects in these five applications, where 87 defects are new, not being detected by a previous related approach [22].

The rest of the paper is organized as follows. Section 2 presents a formal definition of sequence association rules and describes our new mining algorithm. Section 3 describes key aspects of the CAR-Miner approach. Section 4 presents evaluation results. Section 5 discusses threats to validity. Section 6 presents related work. Finally, Section 7 concludes.

2 Problem Definition

We next present a formal definition of general association rules and then describe sequence association rules required for characterizing exception-handling rules. Although we present our algorithm from the point-of-view of mining exception-handling rules, the algorithm is general and can be applied to other practical problems that fall into our problem domain.

Problem Domain

Let $F = \{FC_1, FC_2, ..., FC_k\}$ be the set of all possible distinct items. Let $I = \{FC_{i1}, FC_{i2}, ..., FC_{im}\}$ and $J = \{FC_{j1}, FC_{j2}, ..., FC_{jn}\}$ be two sets of items, where $I \subseteq F$ and $J \subseteq F$. Consider a sequence database as a set of tuples (*sid*, S_i, S_j), where *sid* is a sequence id, S_i is a sequence of items belonging to *I*, and S_j is a sequence of items belonging to *J*. In essence, S_i and S_j belong to two sequence databases, say SDB_1 and SDB_2, denoted as $S_i \in SDB_1$ and $S_j \in SDB_2$, respectively, and there is a **one-to-one** mapping between the two sequence databases. We define an association rule between sets of sequences as $X \Rightarrow Y$, where both X and Y are subsequences of $S_i \in SDB_1$ and $S_j \in SDB_2$, respectively. A sequence $\alpha = \langle a_1 a_2 ... a_p \rangle$ (where each a_s is an item) is defined as a subsequence of another sequence $\beta = \langle b_1 b_2 ... b_q \rangle$, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 < j_2 < ... < j_p \leq q$ such that $a_1 = b_{j1}$, $a_2 = b_{j2}$,..., $a_p = b_{jq}$.

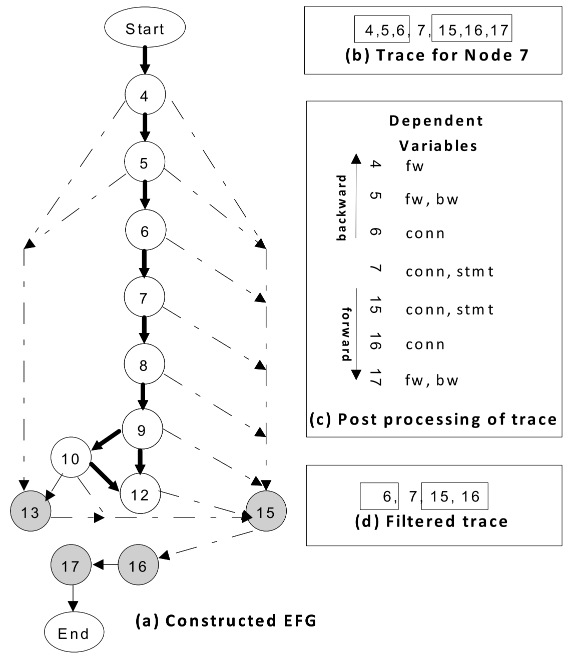


Figure 3. Illustrative examples of CAR-Miner approach.

General Algorithm

To the best of our knowledge, there are no existing mining techniques that can mine from sets of sequences such as SDB_1 and SDB_2 with resulting association rules as $X \Rightarrow Y$, where $X \sqsubseteq S_i \in SDB_1$ and $Y \sqsubseteq S_j \in SDB_2$. We combine both sequence databases in a novel way using annotations to build a single sequence database. These annotations help in deriving association rules in later stages. For example, consider two sequence databases shown in Figure 2a. Figure 2b shows a single sequence database using annotations combined from the two sequence databases. We next mine frequent subsequences from the combined database, denoted as $SDB_{1,2}$, using the frequent closed subsequence mining technique [19].

The frequent subsequence mining technique accepts a database of sequences such as $SDB_{1,2}$ and a minimum support threshold *min_sup*, and returns subsequences that appear at least *min_sup* times in the sequence database. Given a sequence *s*, it is considered as frequent if its support *sup(s)* $\geq$ *min_sup*. In our context, we are interested in frequent closed subsequences. A sequence *s* is a frequent closed sequence, if *s* is frequent and no proper super sequence of *s* is frequent. Figure 2c shows an example closed frequent subsequence from the combined sequence database. As sequence mining preserves temporal order among items, we scan each closed frequent subsequence and transform the subsequence into an association rule of the form "$X \Rightarrow Y$" based on annotations (as shown in Figure 2d). We compute confidence values for each association rule using the formula as shown below:

Confidence ($X \Rightarrow Y$) = Support (X Y) / Support (X)

Although we explain our algorithm using two sequence databases SDB_1 and SDB_2, our algorithm can be applied to multiple sequence databases as well. These multiple sequence databases can also be combined into a single sequence database using the similar mechanism illustrated in Figure 2.

Sequence Association Rules

In our current approach, our target is to mine exception-handling rules in the form of association rules. Therefore, we collect two sequence databases for each function call FC_a: a normal function-call-sequence (NFCS) database and an exception function-call-sequence (EFCS) database. We apply our mining algorithm to generate sequence association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^m$, where $FC_c^1...FC_c^n \sqsubseteq S_i \in$ NFCS and $FC_e^1...FC_e^m \sqsubseteq S_j \in$ EFCS. Such an association rule describes that FC_a should be followed by the function-call-sequence $FC_e^1...FC_e^m$ in exception paths, when preceded by the function-call-sequence $FC_c^1...FC_c^n$. As this association rule is specific to the function call FC_a, we append FC_a to the rule as $(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$.

3 Approach

Our CAR-Miner approach accepts an application under analysis and mines exception-handling rules for all function calls in the application. CAR-Miner detects violations of the mined exception-handling rules. We next present the details of each phase in our approach.

3.1 Input Application Analysis

CAR-Miner accepts an application under analysis and parses the application to collect each function call, say FC_a, in the application from the call sites in the application. For example, CAR-Miner collects the function call `Statement.executeUpdate` as an FC_a from Line 1.9 in Scenario 1. We denote the set of all function calls as FCS. CAR-Miner mines exception-handling rules for all these function calls.

3.2 Code-Sample Collection

To mine exception-handling rules for the function call FC_a, we need code samples that already reuse the function. To collect such relevant code samples, we interact with a code search engine (CSE) such as Google code search [9] and download code samples returned by the CSE. For example, we construct the query "`lang:java java.sql.Statement executeUpdate`" to collect code samples of the FC_a `Statement.executeUpdate`. Often code samples gathered from a CSE are partial as the CSE returns individual source files instead of complete projects. We use partial-program analysis developed in our previous approach [17] to resolve object types such as receiver or argument types of function calls in code samples. More details of our partial-program analysis are available in our previous paper [17]. As we collect relevant code samples from other open source projects that already reuse a function, our approach has an advantage of being able to detect additional rules that do not have enough supporting samples in the application under analysis.

3.3 Exception-Flow-Graph Construction

We next analyze the collected code samples and the application to generate traces in the form of sequence of function calls. Initially, we construct Exception-Flow Graphs (EFG), which are an extended form of Control-Flow Graphs (CFG). An EFG provides a graphical representation of all paths that might be traversed during the execution of a program, including exception paths. Construction of an EFG is non-trivial due to the existence of additional paths that transfer control to exception-handling blocks defined in the form of `catch` or `finally` in Java. We develop an algorithm inspired by Sinha and Harrold [16] for constructing EFGs with additional paths that describe exception conditions. Figure 3a shows the constructed EFG for Scenario 2, where each node is denoted with the corresponding line number of Scenario 2 in Figure 1.

Initially, we build a CFG that represents flow of control during normal execution and augment the constructed CFG with additional edges that represent flow of control after exceptions occur. We refer to these additional edges as *exception* edges and all other edges as *normal* edges. In the figure, *normal* and *exception* edges are shown in solid and dotted lines, respectively. For example, an exception edge is added from Node 5 to Node 13 as the program can follow this path when `IOException` occurs while creating a `BufferedWriter` object. As code inside a `catch` or a `finally` block gets executed after exceptions occur, we consider edges between the statements within `catch` and `finally` blocks also as exception edges. We show nodes related to function calls in normal paths such as those in a `try` block in white and function calls in exception paths such as those in a `catch` block in grey. Although function calls in a `finally` block belong to both normal and exception paths, we consider these paths as exception paths and show the associated nodes in grey. For simplicity, we ignore the control flow inside exception blocks.

In the constructed EFG, there is an exception edge from Node 5 to Node 13, but there is no exception edge from Node 6 to Node 13. The reason is that Node 13 handles a checked exception `IOException`, which is never raised by function call `DriverManager.getConnection` of Node 6. Therefore, we prevent such infeasible control flow through a sound static analysis tool, called Jex [14]. Jex analyzes source code statically and provides possible exceptions raised by each function call. For example, Jex provides that `IOException` can be raised by `BufferedWriter.Constructor` but not `DriverManager.getConnection`. While adding *exception* edges, we add only those edges from a function call to a `catch` block where the exception handled by the `catch` block belongs to the set of possible exceptions thrown by the function call. This additional check helps reduce potential false positives by preventing infeasible exception paths. If the `catch` block handles `Exception` (the super class of all exception types), we add exception edges from each function call to the `catch` block. We consider a `finally` block as similar to a `catch` block that handles `Exception`, and add exception edges from each function call to the `finally` block.

As gathered code samples are partial, we use intra-procedural analysis for constructing EFGs. Furthermore, before constructing an EFG for a code sample, we also check whether the code sample includes any $FC_a \in FCS$. If the code sample does not include any FC_a, we skip the EFG construction for that code sample.

3.4 Static Trace Generation

We next capture static traces that include actions that should be taken when exceptions occur while executing function calls such as $FC_a \in FCS$. For example, consider the FC_a "`Connection.createStatement`" and its corresponding Node 7 in the EFG. A trace generated for this node is shown in Figure 3b. The trace includes three sections: *normal function-call sequence* ($FC_c^1 ... FC_c^n$), FC_a, *exception function-call sequence* ($FC_e^1 ... FC_e^m$).

The $FC_c^1 ... FC_c^n$ sequence starts from the beginning of the body of the enclosing function (i.e., caller) of the FC_a function call to the call site of FC_a. The $FC_e^1 ... FC_e^m$ sequence includes the longest exception path that starts from the call site of FC_a and terminates either at the end of the enclosing function body or at a node in EFG whose outgoing edges are all normal edges. We generate such traces from code samples and input application for each $FC_a \in FCS$.

3.5 Trace Post-Processing

We next identify function calls in $FC_c^1 ... FC_c^n$ or $FC_e^1 ... FC_e^m$ that are not related to FC_a through data-dependency, and remove such function calls from each trace. Failing to remove such unrelated function calls can result in many false positives due to frequent occurrences of unrelated function calls as shown in the evaluation of PR-Miner [11]. For example, in the trace shown in Figure 3b, function calls in the normal function-call sequence related to Nodes 4 and 5 are unrelated to the FC_a of Node 7. Similarly, Node 17 in the exception function-call sequence is also unrelated to FC_a.

Figure 3c shows an example of our data-dependency analysis. Initially, we generate two kinds of relationships: var dependency of a variable and function association of a function call. The var dependency of a variable represents the set of variables on which a given variable is dependent upon. Similarly, a function association of a function call represents the set of variables on which a function call is associated with.

First, we compute the var-dependency relationship information from assignment statements. For example, in Scenario 2, we identify that the variable `res` is dependent on the variable `stmt` from Line 2.8 and is transitively dependent on `conn` as `stmt` is dependent on `conn` from Line 2.7. We compute the function-association relationship based on the var-dependency relationship. In particular, we identify that a function call is associated with all its variables including the receiver, arguments, and the return variable, and their transitively dependent variables. For example, applying the preceding analysis to the function call of Node 7, we identify that the associated variables are `conn` and `stmt`.

We use variables associated with each function call to identify function calls in the normal function-call sequence $FC_c^1 ... FC_c^n$ or the exception function-call sequence $FC_e^1 ... FC_e^m$ that are not related to FC_a. Starting from FC_a, we perform a backward traversal of the trace to filter out function calls in $FC_c^1 ... FC_c^n$ and a forward traversal to filter out function calls in $FC_e^1 ... FC_e^m$. Assume that variables associated with FC_a are $\{V_a^1, V_a^2, \ldots, V_a^s\}$. Assume that variables associated with a function call, say FC_{ce}^k, in the normal or exception function-call sequence are $\{V_{ce}^1, V_{ce}^2, \ldots, V_{ce}^t\}$.

In each traversal, we compute an intersection of associated variable sets of FC_a and FC_{ce}^k. If the intersection $\{V_a^1, V_a^2, \ldots, V_a^s\} \cap \{V_{ce}^1, V_{ce}^2, \ldots, V_{ce}^t\} \neq \phi$, we keep the FC_{ce}^k function call (either in the normal or exception function-call sequence) in the trace; otherwise, we filter out the FC_{ce}^k function call from the trace. The rationale behind our analysis is that if the intersection is a non-empty set, it indicates that the FC_a is directly or indirectly related to the FC_{ce}^k function call. For example, the intersection of associated variables for Nodes 6 and 7 is non-empty. In contrast, the intersection of associated variables for Nodes 5 and 7 is empty. Therefore, we keep Node 6 in the trace and filter out Node 5 during backward traversal. Similarly, during forward traversal, we ignore Node 17 since the intersection is an empty set. The resulting trace of "4,5,6,7,15,16,17" is "6,7,15,16", where

```
6 : DriverManager.getConnection
7 : Connection.createStatement
15 : Statement.close
16 : Connection.close
```

3.6 Static Trace Mining

We apply our new mining algorithm described in Section 2 on the set of static traces collected for each FC_a.

We apply mining on the traces of each FC_a individually. The reason is that if we apply mining on all traces together, rules related to a FC_a with only a small number of traces can be missed due to rules related to other FC_a with a large number of traces.

In the phase of static trace mining, we first transform traces suitable for our mining algorithm. More specifically, as each trace includes a normal function-call sequence and an exception function-call sequence, we build two sequence databases with normal and exception function-call sequences, respectively, from all the traces of a FC_a function call.

We next apply our mining algorithm that initially annotates corresponding normal and exception function-call sequences and combines the annotated sequences into a single call sequence. The mining algorithm produces sequence association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^m$. As this sequence association rule is specific to FC_a, we add FC_a to the rule as $(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$. The preceding sequence association rule describes that the function call FC_a should be followed by $FC_e^1...FC_e^m$ in exception paths only when preceded by $FC_c^1...FC_c^n$ in normal paths. In our approach, we use the frequent closed subsequence mining tool, called BIDE, developed by Wang and Han [19]. We used the *min_sup* value as 0.4, which is set based on our initial empirical experience. We repeat the preceding process for each FC_a and rank all final sequence association rules based on their support values assigned by the frequent subsequence miner.

3.7 Anomaly Detection

To show the usefulness of our mined exception-handling rules, we apply these rules on the application under analysis to detect violations. Initially, from each call site of FC_a in the application, we extract the normal function-call sequence, say $C_c^1 C_c^2 \ldots C_c^a$, from the beginning of the body of enclosing function of FC_a to the call site of FC_a. If $FC_c^1 \ldots FC_c^n \sqsubseteq C_c^1 C_c^2 \ldots C_c^a$, then we extract the exception function-call sequence, say $C_e^1 C_e^2 \ldots C_e^b$, from the call site of FC_a to the end of the enclosing function body or to a node (in the EFG) whose outgoing edges are all normal edges. We do not report a violation if $FC_e^1 \ldots FC_e^m \sqsubseteq C_e^1 C_e^2 \ldots C_e^b$; otherwise, we report a violation in the application under analysis. We rank all detected violations based on a similar criterion used for ranking exception-handling rules.

4 Evaluations

We next describe the evaluation results of CAR-Miner with five real-world open source applications as subjects. We use the same subjects (and same versions) used for evaluating a related approach called WN-miner [22] for the ease of comparison with the data provided by the WN-miner developer. We used five out of eight subjects used in WN-miner since related versions of the remaining three subjects are not currently available. In our evaluations, we try to address the following questions. (1) Do the exception-handling rules mined by CAR-Miner represent real rules? (2) Do the detected rule violations represent real defects in subject applications? (3) Does CAR-Miner perform better than the existing related WN-miner tool in terms of mining real rules and detecting real defects in an application under analysis? (4) Do the sequence association rules help detect any new defects that cannot be detected with simple association rules of the form "$FC_a \Rightarrow FC_e$"? The detailed results of our evaluation are available at `http://ase.csc.ncsu.edu/projects/carminer/`.

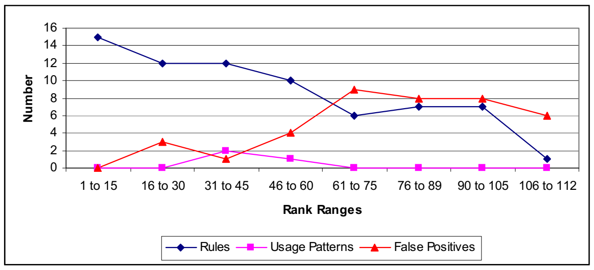


Figure 4. Distribution of classification categories with ranks for the Axion application.

4.1 Subjects

Table 1 shows subjects and their versions used in our evaluations. Column "Internal Info" shows the number of declared classes and functions of each application. Column "External Info" shows the number of external classes and their functions invoked by the application. Column "Code Examples" shows the number of code examples gathered by CAR-Miner to mine exception-handling rules. For example, CAR-Miner gathered 47783 code examples ($\approx$ 7 million LOC) from a code search engine for mining exception-handling rules of the Axion application. Column "Time" shows the amount of time taken by CAR-Miner in seconds for each application. The shown time includes the analysis time of the application and gathered code examples, and the time taken for detecting violations. The amount of processing time depends on the number of samples gathered for an application. All experiments were conducted on a machine with 3.0GHz Xeon processor and 4GB RAM.

4.2 Mined Exception-Handling Rules

We next address the first question on whether the mined exception-handling rules represent real rules that can help detect defects in an application under analysis. Table 2 shows the classification of exception-handling rules mined

Table 1. Characteristics of subjects used in evaluating CAR-Miner.

Subject	Lines of code	Internal Info #Classes	Internal Info #Functions	External Info #Classes	External Info #Functions	# Code Examples	Time (in sec.)
Axion 1.0M2	24k	219	2405	58	217	47783 (7M)	1381
HsqlDB 1.7.1	30k	98	1179	80	264	78826 (26M)	2547
Hibernate 2.0 b4	39k	452	4321	174	883	88153 (27M)	1125
SableCC 2.18.2	22k	183	1551	21	76	47594 (15M)	1220
Ptolemy 3.0.2	170k	1505	9617	477	2595	70977 (21M)	1126

Table 2. Classification of exception-handling rules.

Subject	#Total	Real Rules		Usage Patterns		False Positives	
		#	%	#	%	#	%
Axion	112	70	62.5	3	2.68	39	34.82
HsqlDB	127	89	70.08	3	2.36	35	27.56
Hibernate	121	86	71.07	1	0.82	34	28.09
SableCC	40	12	30	2	5	26	65
Ptolemy	94	37	39.36	5	5.32	52	55.32
AVERAGE			54.6		3.24		42.16

by CAR-Miner. Column "Total" shows the total number of rules in each application. We classify these rules into three categories: real rules, usage patterns, and false positives. Real rules describe the behavior that must be satisfied while using function calls such as FC_a, whereas usage patterns suggest common ways of using FC_a. The violations of real rules and usage patterns can be defects and hints, respectively. A hint, which was originally proposed by Wasylkowski et al. [20], helps increase readability and maintainability of source code of an application. We used the available on-line documentations, JML specifications[1], or the source code of the application for classifying mined exception-handling rules into these three categories. Our results show that real rules are 54.61% and false positives are 42.16%, averagely.

Although false positives are 42.16% on average among the total number of mined rules, our mining heuristics for ranking exception-handling rules help give higher priority to real rules than false positives. Figure 4 shows a detailed distribution of all extracted rules for the Axion application. In Figure 4, x-axis shows distribution of mined rules in different ranges (each range is of size 15) with respect to assigned ranks and y-axis shows the number of rules that are classified into the three categories for each range. The primary reason for selecting the Axion application is that the application is a medium-scale application that is amenable to a detailed analysis with reasonable effort. As shown in the figure, the number of false positives is quite low among the exception-handling rules ranked between 1 to 60. These results show the significance of our mining and ranking criteria. Our results in Table 2 also show that more exception-handling rules exist in applications such as Axion, HsqlDB, and Hibernate that deal with resources (such as databases or files) compared to other applications.

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

Table 3. Classification of detected violations.

Subject	#Total Violations	#Violations of first 10 rules	#Defects	#Hints	#FP
Axion 1.0M2	257	19	13	1	5
HsqlDB 1.7.1	394	62	51	0	10
Hibernate 2.0 b4	136	22	12	0	10
Sablecc 2.18.2	168	66	45	7	14
Ptolemy 3.0.2	665	95	39	1	55

Table 4. Status of detected defects in new versions of subject applications.

	# Defects	New Version	#Fixed	#Deleted	#Open
Axion 1.0M2	13	1.0M3	4	8	1
HsqlDB 1.7.1	51	1.8.0.9	2	9	40
Hibernate 2.0 b4	12	3.2.6	0	8	4
Sablecc 2.18.2	45	4-alpha.3	0	43	2
Ptolemy 3.0.2	39	3.0.2	0	0	39

4.3 Detected Violations

We next address the question on whether the detected violations represent real defects. Table 3 shows the violations detected in each application. Column "Total Violations" shows the total number of violations detected in each application. The HsqlDB and Hibernate applications include test code as part of their source code. As test code is often not written according to specifications, we excluded the violations detected in the test code of those applications from the results. Given a high number of violations in each application, we inspected the violations detected by the top 10 exception-handling rules and classified them into three categories: Defects, Hints, and False Positives.

Column "Violations of first 10 rules" shows the number of violations detected by the top ten exception-handling rules mined for each application. Column "Defects" shows the total number of violations that are identified as defects in each application. As we used the same versions (an earlier version than the latest version) used by the WN-miner approach for the ease of comparison, we verified whether the defects found by our approach are fixed, deleted, or still open in the latest version of each application. Column

"New Version" of Table 4 shows the latest version used for our verification. The defect's sub-categories "Fixed" and "Open" indicate that the defects found by our approach in the earlier version are fixed or still open in the new version, respectively. We reported those open defects to respective developers for their confirmation. Sometimes, we find that the defective code such as function body with detected defects does not exist in the latest version. One reason could be the refactoring of such code, which can be considered as an indirect fix. We classified such defects as "Deleted" (shown in Table 4).

The results show that our CAR-Miner approach can detect real defects in the applications. The number of defects shown in Columns "Fixed" and "Deleted" provide further evidence that these defects detected by CAR-Miner are real since these defects are fixed directly or indirectly in newer versions of the applications. The initial response from the developers of HsqlDB is quite encouraging. The developers responded on the first ten defects that we reported, where seven defects are *accepted* and only three defects are rejected. The bug reports for these ten defects are available in the HsqlDB Bug Tracker system[2] with IDs #1896449, #1896448, and #1896443[3]. Although the three rejected defects are violations of real rules, developers described that the violation-triggering conditions of these defects cannot be satisfied in the context of the HsqlDB application. For example, a rejected defect is a violation of real rule "`DatabaseMetaData.getPrimaryKeys` ⇒ `ResultSet.close`". The preceding rule describes that the `close` function call should be invoked on `ResultSet`, when `getPrimaryKeys` throws any exceptions. The response from the developers (Bug report ID: #1896448) for this defect is "*Although it can throw exceptions in general, it should not throw with HSQLDB. So it is fine.*", which describes that the violation-triggering condition cannot be satisfied in the context of HsqlDB.

4.4 Comparison with WN-miner

We next address the third question on whether our CAR-Miner approach performs better than the related WN-miner tool. As the WN-miner tool is not currently available, the WN-miner developer provided the mined specifications and static traces of their tool. We developed Perl scripts to detect violations of mined specifications in static traces as described by the WN-Miner developer [22]. We used the same criteria described in Sections 4.2 and 4.3 for classifying rules and violations detected by their approach, respectively. We compared both mined exception-handling rules and detected violations.

[2]`http://sourceforge.net/tracker/?group_id=23316&atid=378131`

[3]We reported multiple defects in the same source file as a single bug report.

Table 5. Defects detected or missed by CAR-Miner.

Subject	# Defects			
	# Total	#Common	# Only	# Missed
Axion	13	0	13	1
HsqlDB	51	35	16	13
Hibernate	12	0	12	7
Sablecc	45	0	45	0
Ptolemy	39	38	1	11
TOTAL	160	73	87	32

4.4.1 Comparison of exception-handling rules

We next present the comparison results of exception-handling rules mined by both approaches. Figure 5 shows the results for the classification category "real rules" between WN-miner and CAR-Miner. For each subject and approach, the figure shows the total number of rules mined by each approach along with the number of common rules between the two approaches. For example, CAR-Miner detected a total of 70 rules for the Axion application. Among these 70 rules, 43 rules are newly detected by CAR-Miner and 27 rules are common between CAR-Miner and WN-miner. CAR-Miner failed to detect 2 real rules that were detected by WN-miner.

The primary reason for these two real rules not detected by CAR-Miner and detected by WN-miner is due to the *ranking* criterion used by WN-miner. WN-miner extracts rules "$FC_a \Rightarrow FC_e$" when FC_e appears at least once in exception-handling blocks such as `catch` and ranks those rules with respect to the number of times FC_e appears after FC_a among normal paths. As shown in their results, such a criterion can result in a high number of false positives such as "`Trace.trace` ⇒ `Trace.printSystemOut`" in the HsqlDB application, where FC_e often appears after FC_a in normal paths and is used once in some `catch` block. CAR-Miner ignores such patterns due to their relatively low support among exception paths of FC_a.

The results show that CAR-Miner is able to detect most of the rules mined by WN-miner and also many new rules that are not detected by WN-miner. CAR-Miner performed better than WN-miner due to two factors: sequence association rules and increase in the data scope. To further show the significance of these factors, we classified the real rules mined by CAR-Miner based on these two factors. Figure 6 shows the percentage of sequence association rules among all real rules. The results show that sequence association rules are 20.37% of all real rules on average mined for all applications.

Figure 7 shows the percentage of real rules that cannot be mined by analyzing only the application under analysis. For example, 44.28% of the real rules mined for the Axion application occur only from gathered code samples. Our

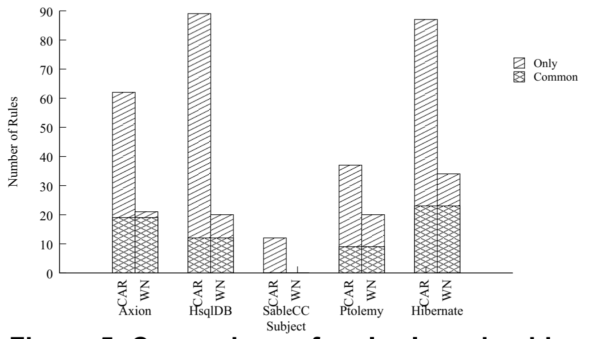


Figure 5. Comparison of real rules mined by CAR-Miner and WN-miner.

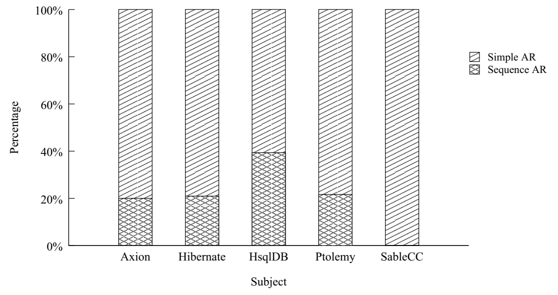


Figure 6. % sequence association rules.

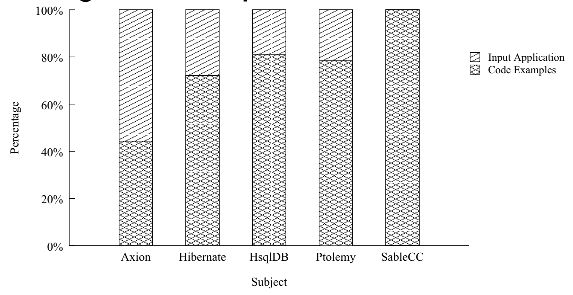


Figure 7. % rules mined only from code examples.

Table 6. Defects detected by Sequence Association Rules.

	# Rules	# Violations	# Defects	# Hints	# False Positives
Axion	3	6	4	0	2
HsqlDB	6	14	8	0	6
Hibernate	4	10	8	0	2
Sablecc	0	0	0	0	0
Ptolemy	1	1	1	0	0

results show that increase in the data scope to open source repositories helps detect new exception-handling rules that do not have sufficient supporting samples in the application. Furthermore, increase in the data scope also helps give higher priority to real rules than false positives.

4.4.2 Comparison of detected defects

We next present the number of real defects that were detected by CAR-Miner but not detected by WN-miner. To show that CAR-Miner can find new defects that were not detected by WN-miner, we identified the exception-handling rules that are mined only by CAR-Miner and not by WN-miner among top 10 shown in Table 3 and verified the defects detected by those rules. The results are shown in Table 5. Column "Total" shows the number of violations detected by the top 10 exception-handling rules. Column "Common" and "Only" show the number of defects commonly detected by CAR-Miner and WN-miner, and defects that are detected by CAR-Miner only, respectively. Column "Missed" shows the number of defects detected by WN-miner only. The results show that CAR-Miner detected 87 new defects (among all applications) that were not detected by WN-miner. When inspecting all violations detected by CAR-Miner, we expect that the preceding number of new defects detected by CAR-Miner can be much higher. CAR-Miner missed 32 defects that were detected by WN-miner. These missed defects are due to the missing patterns as described in Section 4.4.1.

4.5 Significance of Sequence Association Rules

We next address the last research question on whether sequence association rules mined by CAR-Miner are helpful in detecting new defects that cannot be detected by simple association rules. Table 6 shows the number of sequence association rules that are used to detect real defects in all applications. The results show that these rules help detect 21 real defects among all applications.

We next describe a defect in the HsqlDB application to show the significance of sequence association rules, which cannot be mined by existing approaches such as WN-miner. The related code snippet from the `saveChanges` function of `ZaurusTableForm.java` is shown as below:

```
public boolean saveChanges()
{ ...
  try {
    PreparedStatement ps =
         cConn.prepareStatement(str);
    ps.clearParameters(); ...
    for (int j=0; j<primaryKeys.length; j++){
      ps.setObject(i + j + 1,
         resultRowPKs[aktRowNr][j]); }
    ps.executeUpdate();
  } catch (SQLException e) { ...
      return false;
  } ...
}
```

CAR-Miner detected a defect in the preceding code example as the code example violated the exception-handling rule $FC_c^1 \wedge FC_a \Rightarrow FC_e^1$, where

FC_c^1:`Connection.prepareStatement`
FC_a :`PreparedStatement.clearParameters`
FC_e^1 :`Connection.rollback`

The preceding rule describes that when an exception occurs after executing the `clearParameters` function, the `rollback` function should be invoked on the `Connection` object. Failing to invoke `rollback` can make the database state inconsistent. This result shows that sequence association rules are helpful in detecting new defects.

5 Threats to Validity

The threats to external validity primarily include the degree to which the subject applications and CSE used are representative of true practice. The current subjects range from small-scale applications such as Axion to large-scale applications such as Ptolemy. 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 external validity also include the quality of code examples collected from a CSE. We tried to reduce this threat to some extent by capturing most frequent patterns among these code examples. The threats to internal validity are instrumentation effects that can bias our results. Faults in our CAR-Miner prototype might cause such effects. There can be errors in our inspection of source code for confirming defects. To reduce these threats, we inspected available related specifications and call sites in source code.

6 Related Work

WN-miner by Weimer and Necula [22] extracts simple association rules of the form "$FC_a \Rightarrow FC_e$", when FC_e is found at least once in exception-handling blocks (i.e., `catch` or `finally` blocks). Their approach mines and ranks these rules based on the number of times FC_e appears after FC_a in normal paths. Due to their ranking criteria, their approach cannot mine rules that include a FC_e function call such as `Connection.rollback`, where FC_e can appear *only* in exception paths. Acharya and Xie [1] later proposed a similar approach for detecting API error-handling defects in C code. Our approach significantly differs and improves upon these previous approaches as we mine sequence association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^m)$" that can characterize more exception-handling rules. Our approach also addresses the problem of lacking enough supporting samples for these rules in the application under analysis by expanding the data scope to open source repositories through a code search engine.

CodeWeb [13] mines association rules from source code as framework reuse patterns. CodeWeb mines association rules such as application classes inheriting from a library class often create objects of another class. PR-Miner [11] uses frequent itemset mining to extract implicit programming rules in large C code bases and detects violations. DynaMine [12] uses association rule mining to extract simple rules from version histories for Java code and detects rule violations. Engler et al. [5] proposed a general approach for finding defects in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Wasylkowski et al. [20] mines rules that include pairs of API calls and detect violations. Perracotta [23] mines patterns such as $(ab)^*$ and includes techniques for handling imperfect traces. Schäfer et al. [15] mine association rules that describe usage changes in framework evolution. All these preceding approaches mine simple association rules that are often not sufficient to characterize complex real rules as shown in our approach. In contrast, our approach can mine more complex rules in the form of sequence association rules.

Our approach is also related to other approaches that analyze exception behavior of programs. Fu and Ryder [6] proposed an exception-flow analysis that computes chains of semantically related exception-flow links across procedures. Our approach uses intra-procedural analysis for constructing exception-flow graphs. The Jex [14] tool statically analyzes exception flow in Java code and provides a precise set of exceptions that can be raised by a function call. We use Jex in our approach to prevent infeasible exception edges in a constructed EFG. Fu et al. [7] present a *def-use*-based approach that helps gather error-recovery code-coverage information. Our approach is different from their approach as our approach detects defects that violate mined rules rather than focusing on coverage of exception-handling code.

Chang et al. [3] applies frequent subgraph mining on C code to mine implicit condition rules and detect neglected conditions. Their approach targets at different types of defects called neglected conditions. Moreover, their approach does not scale to large code bases as graph mining algorithms suffer from scalability issues. Finally, DeLine and Fähndrich [4] proposed an approach that allows programmers to manually specify resource management protocols that can be statically enforced by a compiler. However, their approach requires manual effort from programmers and also requires the knowledge of the *Vault* specification language to specify domain-specific protocols. In contrast, our approach does not require any manual effort or the knowledge of any specific specification languages.

Javert [8] uses a pattern-based specification miner to mine smaller patterns such as $(ab)^*$, called *micro patterns*, and then compose these patterns into larger specifications. Their approach does not require the user to provide any templates. Similar to their approach, our approach also does not require the user to provide any templates. However, their mined patterns cannot characterize exception-handling rules mined by our approach.

Our previous approaches PARSEWeb [17] and SpotWeb [18] also exploit code search engines for gathering related code samples. PARSEWeb accepts queries of the form "*Source* → *Destination*" and mines frequent function-call sequences that accept *Source* and produce *Destination*. SpotWeb accepts an input framework and detects hotspot classes and functions of the framework. Our new approach CAR-Miner significantly differs from these previous approaches. CAR-Miner constructs EFGs and includes new techniques for collecting and post-processing static traces related to exception handling. Furthermore, CAR-Miner incorporates our new mining algorithm for detecting exception-handling rules as sequence association rules.

7 Conclusion

We have developed an approach, called CAR-Miner, that mines exception-handling rules in the form of sequence association rules. Unlike simple association rules of the form "$FC_a \Rightarrow FC_e$", these sequence association rules of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 ... FC_e^m)$" can characterize more complex exception-handling rules. As existing mining algorithms cannot mine these sequence association rules, we proposed a novel mining algorithm based on frequent closed subsequence mining. CAR-Miner also tries to address the problems of limited data scopes faced by existing approaches by expanding the data scope to open source projects available on the web. We have evaluated our approach with five real-world open source applications and shown that CAR-Miner mined 294 real exception-handling rules. We have also shown that CAR-Miner finds 160 defects, where 87 are new defects, not being found by a previous related approach [22]. Our approach takes a step forward in the direction of developing new mining algorithms to address unique requirements in mining software engineering data, beyond being limited by existing off-the-shelf mining algorithms.

Acknowledgments

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

References

[1] M. Acharya and T. Xie. Mining API error-handling specifications from source code. In *Proc. FASE*, 2009.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

[3] 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.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. PLDI*, pages 59–69, 2001.

[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] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proc. ICSE*, pages 230–239, 2007.

[7] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *Proc. ISSTA*, pages 23–33, 2004.

[8] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. FSE*, pages 339–349, 2008.

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

[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. ESEC/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] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. ICSE*, pages 167–176, 2000.

[14] M. P. Robillard and G. C. Murphy. Analyzing exception flow in Java programs. In *Proc. ESEC/FSE*, pages 322–337, 1999.

[15] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proc. ICSE*, pages 471–480, 2008.

[16] S. Sinha and M. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, 2000.

[17] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. ASE*, pages 204–213, 2007.

[18] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. ASE*, pages 327–336, 2008.

[19] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. ICDE*, pages 79–88, 2004.

[20] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proc. ESEC/FSE*, pages 35–44, 2007.

[21] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proc. OOPSLA*, pages 419–431, 2004.

[22] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proc. TACAS*, pages 461–476, 2005.

[23] J. Yang and D. Evans. Perracotta: mining temporal API rules from imperfect traces. In *Proc. ICSE*, pages 282–291, 2006.