Mining Exception-Handling Rules as Conditional Association Rules

[Qualifying Examination Report]
Suresh Thummalapenta
Department of Computer Science
North Carolina State University, USA
sthumma@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. Failing to take necessary recovery actions such as rolling back transactions can not only cause performance degradation, but also lead to critical issues. In this paper, we propose a novel approach that mines exception-handling rules, which describe expected behavior when exceptions occur. Existing mining approaches mine association rules of the form "$FC_a \Rightarrow FC_e$", which describes that the function call FC_a should be followed by the function call FC_e in all paths. In this paper, we develop the first novel mining algorithm to mine conditional association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^n)$", which describe that FC_a should be followed by a sequence of function calls $(FC_e^1...FC_e^n)$ only when FC_a is preceded by the sequence $(FC_c^1...FC_c^n)$. Such form of rules is required to characterize common exception-handling rules. We show the usefulness of these rules by applying these rules on five real-world applications to detect violations. In our evaluation, we show that our approach detects 294 real exception-handling rules in five benchmark applications including 285 kLOC and also finds 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. During these exception conditions, programs follow paths different from normal execution paths; these additional paths are referred 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 resource after the usage in all paths including *exception* paths. Failing to release the resource can not only cause performance degradation, but also lead to critical issues. For example, if a database lock taken by a transaction is not released, the process trying to acquire the same lock hangs till the database releases the lock after timeout. A case study [22] conducted on a real application demonstrates the necessity of releasing resources in exception paths for improving reliability and efficiency. The case study found that there is a surprising improvement of 17% in the performance of an 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 often do not exist in practice [10]. To address the preceding issue, association rules of the form "$FC_a \Rightarrow FC_e$" are used 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 the preceding form are often not sufficient to characterize common exception-handling rules. The rationale is that there are various scenarios where FC_a should not necessarily be followed by FC_e when exceptions are raised by FC_a, although both function calls share the same receiver object.

For example, consider Scenarios 1 and 2 in Figure 1 extracted from real applications. Scenario 1 attempts to modify the contents of the database through the function call `Statement.executeUpdate` (Line 1.9), whereas Scenario 2 attempts to read contents of the database through the function call `Statement.executionQuery` (Line 2.8). Consider a simple pattern "`OracleDataSource.getConnection` $\Rightarrow$ `Connection.rollback`", which suggests that a `rollback` should appear in exception paths whenever an object of `Connection` is created. Although the

`Connection` object is created in both the scenarios, this pattern applies only to Scenario 1 and does not apply to Scenario 2. The key aspect is that the `rollback` function should be invoked *only* when there are any changes made to the database. The preceding example shows that the simple association rules of the form "$FC_a \Rightarrow FC_e$" are not often sufficient to characterize exception-handling rules.

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

The preceding conditional 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 the 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 conditional association rules are a super set of simple association rules. For example, conditional 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 [11] available in the literature cannot be directly applied to mine such conditional association rules. Therefore, to bridge the gap, we developed a new mining algorithm by adapting closed frequent subsequence mining algorithm [20].

We further developed a novel approach, called CAR-Miner, that incorporates our new mining algorithm for the problem of detecting exception-handling rules in the form of conditional association rules by analyzing source code. Apart from mining conditional association rules, CAR-Miner addresses another challenge that is often faced by existing approaches [3, 11, 22], which gather specifications from a limited data scope, i.e., from only a few example applications. Therefore, these approaches may not be able to discover specifications that do not have enough supporting samples in those example applications and hence the related defects remain undetected by these approaches. To address the preceding challenge, CAR-Miner expands the data scope by leveraging a code search engine (CSE) for gathering related code samples from existing open source projects available on the web and reuses those code samples to mine exception-handling rules. We show the usefulness of mined exception-handling rules by applying these rules on several applications to detect violations. CAR-Miner tries to address the problems related to the quality of the code examples 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 conditional association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^n)$". 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.
- An approach for constructing precise EFG, which is an extended form of 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 the input application. These rules can help detect new defects in the input application.
- An Eclipse plugin that implements our approach and a set of evaluations to show the effectiveness of our approach. More specifically, CAR-Miner detects 294 real exception-handling rules in five different applications including 285 kLOC. 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 initial response from developers of HsqlDB [1] is encouraging. The developers responded on the first ten defects that we reported, where seven defects are *accepted* and only three defects are rejected.

The rest of the paper is organized as follows. Section 2 presents a formal definition of conditional association rules and describes our new mining algorithm. Section 3 describes key aspects of the CAR-Miner approach. Section 4 discusses 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 later describe conditional association rules

[1]`http://hsqldb.sourceforge.net/`

Scenario 1

```
...
OracleDataSource ods = null; Session session = null;
    Connection conn = null; Statement statement = null;
logger.debug("Starting update");
try {
    ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:scott/tiger@192.168.1.2:1521:catfish");
    conn = ods.getConnection();
    statement = conn.createStatement();
    statement.executeUpdate("DELETE FROM table1");
    connection.commit(); }
    catch (SQLException se) {
            if (conn != null) { conn.rollback(); }
            logger.error("Exception occurred"); }
finally {
    if(statement != null) { statement.close(); }
    if(conn != null) { conn.close(); }
    if(ods != null) { ods.close(); } }
}
```

Scenario 2

```
Connection conn = null;
Statement stmt = null;
BufferedWriter bw = null; FileWriter fw = null;
try {
    fw = new FileWriter("output.txt");
    bw = BufferedWriter(fw);
    conn = DriverManager.getConnection("jdbc:pl:db", "ps", "ps");
    Statement stmt = conn.createStatement();
    ResultSet res = stmt.executeQuery("SELECT Path FROM Files");
    while (res.next()) {
            bw.write(res.getString(1));
    }
    res.close();
} catch(IOException ex) { logger.error("IOException occurred");
} finally {
    if(stmt != null) stmt.close();
    if(conn != null) conn.close();
    if (bw != null) bw.close();
}
```

Figure 1. Two examples scenarios from real applications.

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 aspects 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 = $\{FC_1, FC_2, ..., FC_m\}$ and J = $\{FC_1, FC_2, ..., FC_n\}$ be two sets of items, where $I \subset F$ and $J \subset F$. Consider a sequence database as a set of tuples (*sid*, S_1, S_2), where *sid* is a sequence id, S_1 is a sequence of items belonging to *I*, and S_2 is a sequence of items belonging to *J*. For example, S_1 shall be of the form $FC_6FC_8FC_9$, where $\{FC_6, FC_8, FC_9\} \in I$. In essence, S_1 and S_2 belong to two sets of sequence databases, say SDB_1 and SDB_2, and there is a one-to-one mapping between both sequence databases. We define an association rule between sets of sequences as $X \Rightarrow Y$, where both X and Y are subsequences belonging to SDB_1 and SDB_2, respectively.

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 $X \in SDB_1$ and $Y \in SDB_2$. Therefore, 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 how we combine both databases into a single sequence database using annotations. We next mine frequent subsequences from the combined database, say $SDB_{1,2}$, using closed frequent subsequence mining [20].

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 ap-

SDB₁	**SDB₂**
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

SDB₁,₂
$3^1,6^1,9^1,10^1,2^2,3^2,7^2,8^2$
$3^1,10^1,13^1,2^2,6^2,8^2$
$9^1,10^1,1^1,19^1,9^2,16^2,13^2$

(b) Combined sequence database using annotations

SDB₁,₂
$3^1,10^1,2^2,8^2$

(c) Mined sequences

Association Rule
3, 10 => 2,8

(d) Association rules using annotations

Figure 2.

pear at least *min_sup* times in the sequence database. A sequence α = $\langle a_1a_2...a_n\rangle$ is defined as a subsequence of another sequence β = $\langle b_1b_2...b_n\rangle$, denoted as $\alpha \subseteq \beta$, if there exist integers 1 $\leq j_1 < j_2 < ... < j_m \leq$ n such that $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$,..., $a_m \subseteq b_{jm}$. Given a sequence *s*, it is referred as frequent if the support *sup(s)* $\geq$ *min_sup*. In our context, we are interested in frequent closed subsequences. A sequence *s* is referred as a frequent closed sequence, if *s* is frequent and no proper super-sequence of *s* is frequent. Figure 2c shows an example closed frequent subsequences from the combined sequence database. As sequence mining preserves the temporal order among elements, 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 the 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. The primary reason is that these multiple sequence databases can also be combined into a single sequence database using the similar mechanism described in Figure 2.

Conditional Association Rules

In the current approach, our target is to mine exception-handling rules in the form of association rules. Therefore,

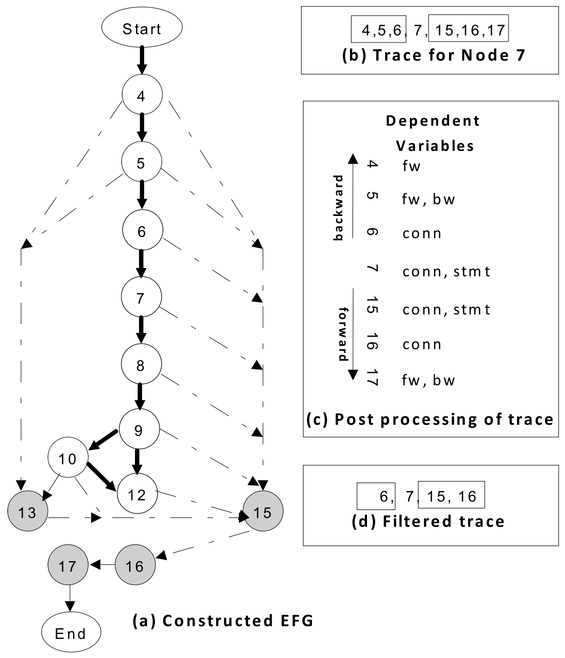

Figure 3. Illustrative examples of CAR-Miner approach.

we collect two sequence databases for each FC_a: normal function-call-sequence (NFCS) database and exception function-call-sequence (EFCS) database. We apply the preceding mining algorithm to generate association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^n$, where $FC_c^1...FC_c^n$ $\subseteq$ NFCS and $FC_e^1...FC_e^n \subseteq$ EFCS. Such association rules describe that FC_a should be followed by the function-call-sequence $FC_e^1...FC_e^n$ 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 transform the rule into a conditional association rule as $FC_c^1...FC_c^n \wedge FC_a \Rightarrow FC_e^1...FC_e^n$.

3 Approach

Our CAR-Miner approach accepts an input application and mines exception-handling rules for the function calls in the input application. CAR-Miner applies mined exception-handling rules to detect violations. We next present the details of each phase in our approach.

3.1 Input Application Analysis

CAR-Miner accepts an input application and parses the application to collect the function calls, say FC_a, 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 refer the set of all function calls as FCS.

3.2 Code-Sample Collection

To identify exception-handling rules that characterize the function call FC_a, we need samples that are already reusing the function call. To collect such related samples, we interact with a code search engine such as Google code search [9] and download related code samples returned by the code search engine. For example, we construct the query “`lang:java java.sql.Statement executeUpdate`” to collect the code samples of the FC_a `Statement.executeUpdate`. Often the code samples from a code search engine are partial. We reused the type heuristics developed in our previous approach called PARSEWeb [18] to resolve object types in those code samples. As we collect related code samples from other open source projects that are already reusing the function calls, our approach has an advantage of being able to detect additional patterns that do not have enough supporting samples in the given input application.

3.3 Exception-Flow-Graph Construction

We next analyze the code samples and input 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 a control-flow graphs (CFG). 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 several additional paths that transfer control to exception-handling blocks defined in the form of `catch` or `finally` in Java. We developed an algorithm inspired by Sinha and Harrold [17] for constructing EFGs with additional paths that describe exception conditions. Figure 3a shows the constructed EFG for Scenario 2.

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 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 the `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 between Nodes 5 and 13, but there is no exception edge between Nodes 6 and 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 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 each function call to `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 the potential false positives by preventing infeasible exception paths. Before constructing EFG, 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

The objective of our trace generation is to capture static traces that include the 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* ($FC_e^1 ... FC_e^n$).

The $FC_c^1 ... FC_c^n$ sequence starts from the beginning of the method to the FC_a and the $FC_e^1 ... FC_e^n$ sequence includes the longest exception path that starts from the FC_a node and terminate either at the end of the enclosing method declaration or at a node that has normal outgoing edges only. We generate such traces from code samples and input application for each $FC_a \in FCS$.

3.5 Trace Post-Processing

The objective of our trace post-processing is to identify unrelated function calls of FC_a in $FC_c^1 ... FC_c^n$ or $FC_e^1 ... FC_e^n$ 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 the frequent occurrence of unrelated function calls as shown in the evaluation of PR-Miner [11]. For example, in the current 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 and method dependency. The var-dependency relationship represents the set of variables on which a given variable is dependent upon. Similarly, a method-dependency relationship represents the set of variables on which a function call is dependent upon.

First, we compute the var-dependency relationship information from the 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 method-dependency relationship based on var-dependency relationship. In particular, we identify that a function call is associated with all its elements including the receiver, arguments, and the return variable, and the dependent variables of these elements. For example, applying the preceding analysis to the function call of Node 7, we identify that the associated variables are `conn` and `stmt`.

Consider that the preceding analysis captures associated variables as follows:

$FC_a \Rightarrow \{FCV_a^1, FCV_a^2, \ldots, FCV_a^k\}$

$FC_j \in \{FC_c^1, \ldots, FC_c^n, FC_e^1, \ldots, FC_e^n\} \Rightarrow \{V_j^1, \ldots, V_j^p\}$

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 $FC_e^1 ... FC_e^n$. In each traversal, if $\{FCV_a^1, FCV_a^2, \ldots, FCV_a^k\} \cap \{V_j^1, \ldots, V_j^p\} \neq \phi$, we keep the function call; otherwise, we ignore the function call. 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 dependent on the current node. 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 ignore Node 5 during backward traversal. Similarly, during forward traversal, we ignore Node 17 as the intersection results in 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 group 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, the patterns related to a FC_a with a few number of traces can be missed due to the patterns related to other FC_a with a large number of traces. 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 traces.

We next apply our mining algorithm that initially annotates sequences of normal and exception function-call sequences and combines annotated sequences into a single call sequence. The mining algorithm produces association rules of the form $FC_c^1...FC_c^n \Rightarrow FC_e^1...FC_e^n$. As this association rule is specific to FC_a, we transform the association rule into a conditional association rule of the form $FC_c^1...FC_c^n \wedge FC_a \Rightarrow FC_e^1...FC_e^n$. The preceding conditional association rule describes that the function call FC_a should be followed by $FC_e^1...FC_e^n$ in exception paths only when preceded by $FC_c^1...FC_c^n$ in the normal path. In our approach we use the closed frequent subsequence mining tool, called BIDE, developed by Wang and Han [20]. We used the *min_sup* value as 0.4, which is based on our initial empirical experience. We repeat the preceding process for each FC_a and rank all final conditional association rules based on the 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 input application to detect violations. Initially, from each call site of FC_a in the input application, we extract a function call sequence, say $CC_1CC_2...CC_n$, from the beginning of the method to FC_a. If $\{FC_c^1,...,FC_c^n\} \subseteq \{CC_1, CC_2, \ldots, CC_n\}$, then we check whether the call site in the application has $\{FC_e^1,...,FC_e^n\}$ in possible exception paths. Any missing function calls of $\{FC_e^1,...,FC_e^n\}$ in the exception paths of input application are reported as violations. 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 as 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 violations represent real defects in subject applications? (3) Does CAR-Miner perform better than the existing related tool WN-miner in terms of mining real rules and detecting real defects in a given input application? (4) Do the conditional association rules help detect any new defects? The detailed results of our evaluation are available at `http://ase.csc.ncsu.edu/projects/carminer/`

Table 2. Classification of exception-handling rules.

Subject	#Total	#Real Rules	#Usage Patterns	#False Positives
Axion	112	70	3	39
HsqlDB	127	89	3	35
Hibernate	121	86	1	34
SableCC	40	12	2	26
Ptolemy	94	37	5	52

4.1 Subjects

Table 1 shows subjects and their versions used in our evaluations. Column "Internal Info" shows the number of declared classes and methods of the input application. Column "External Info" shows the number of external classes and their methods referred by the input 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 input application and gathered code examples, and the time taken for detecting defects. 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. Column "URL" shows the location of the input application.

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 the input application. Table 2 shows the classification of exception-handling rules mined by CAR-Miner. Column "Total" shows the total number of rules in each input application. Columns "Real Rules", "Usage Patterns", and "False Positives" show the classification of mined rules through inspection of documentation and source code of the applications. 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 is originally proposed by Wasylkowski et al. [21], helps increase readability and maintainability of source code of the application. Figure 4 shows the percentage of each classification

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

Subject	Lines of code	Internal Info #Classes	Internal Info #Methods	External Info #Classes	External Info #Methods	# 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

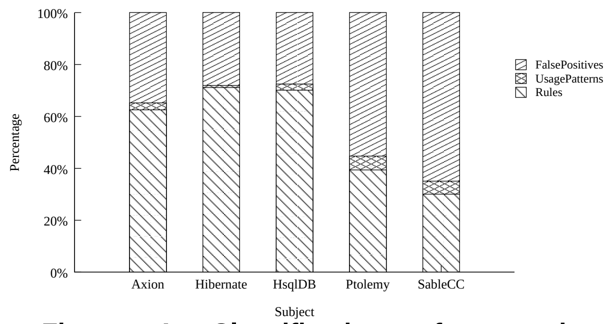

Figure 4. Classification of exception-handling rules.

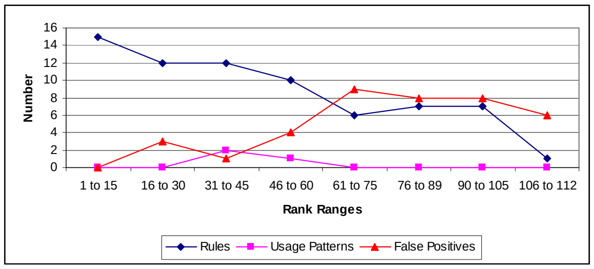

Figure 5. Distribution of classification categories with ranks for the Axion application.

category among the total number of mined rules for each application. Among all categories, real rules are around 54.61% and false positives are 42.16%, averagely.

Although false positives are 42.16% on average among the total number of rules, our mining heuristics for ranking detected exception-handling rules help give higher priority to real rules than false positives. We show a detailed distribution of all extracted rules for the Axion application in Figure 5. 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 efforts. 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.

4.3 Detected Violations

We next address the question on whether the detected violations represent real defects in the subject applications. 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 ten exception-handling rules and classified them into three categories: Defects, Hints, and False Positives.

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

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 buggy code such as method declarations with detected defects does not exist in the latest version. One reason could be due to the refactoring of such code, which can be considered as an indirect fix. We classified such defects as “Deleted” shown as a column in Table 4.

The results show that our CAR-Miner approach can detect real defects in the applications. The initial response from the developers of HsqlDB is quite encouraging. The developers responded on 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 these 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` $\Rightarrow$ `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 the HsqlDB application.

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 authors [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 the exception-handling rules and detected violations.

4.4.1 Comparison of exception-handling rules

We next present the comparison results of exception-handling rules mined by both approaches. Figure 6 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` $\Rightarrow$ `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: conditional 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 7 shows the percentage of conditional association rules, which cannot be detected by WN-Miner. The results show that conditional association rules are 20.37% of all real rules on average mined for all applications.

Figure 8 shows the percentage of real rules that cannot be mined by analyzing only the input application. For example, 44.28% of the real rules mined for the Axion application occur only from gathered code samples. As shown in the results, the increase in the data scope to open source repositories helps detect new exception-handling rules that do not have sufficient supporting samples in the input application. Furthermore, the increase in the data scope also helps to give higher priority to real rules than false positives.

4.4.2 Comparison of detected defects

We next present the number of real defects detected by CAR-Miner that were not found by WN-miner. To show that CAR-Miner can find new defects that are not detected by WN-miner, we identified the exception-handling rules that are detected only by CAR-Miner and not by WN-miner

Table 5. Defects detected by only CAR-Miner.

Subject	# Defects		
	# Total	#Common	# Only
Axion	13	0	13
HsqlDB	51	35	16
Hibernate	12	0	12
Sablecc	45	0	45
Ptolemy	39	38	1

[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.

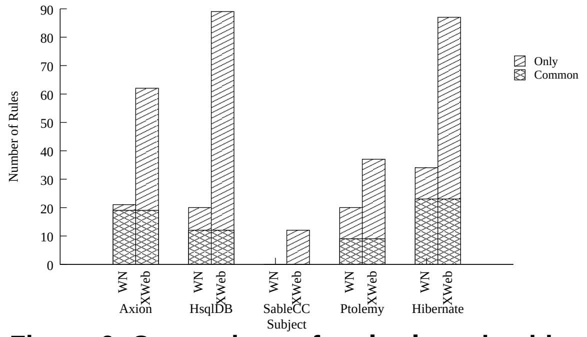

Figure 6. Comparison of real rules mined by CAR-Miner and WN-miner.

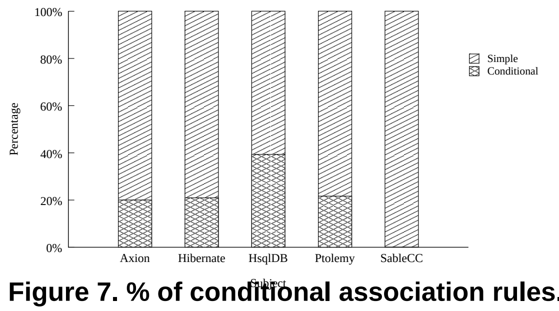

Figure 7. % of conditional association rules.

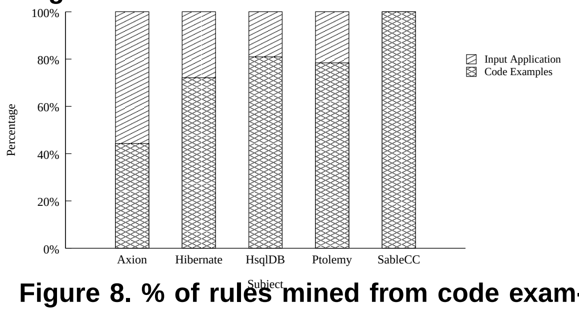

Figure 8. % of rules mined from code examples.

among top ten 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 ten exception-handling rules. Column "Common" and "Only" show the number of defects common to CAR-Miner and WN-miner, and defects that are detected by CAR-Miner only, respectively. 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 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 Conditional Association Rules

We next address the last research question on whether the conditional association rules mined by CAR-Miner are helpful in detecting new defects. Table 6 shows the number

Table 6. Defects detected by Conditional 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

of conditional association rules that are used to detect real defects in all applications. The results show that these rules mined 21 real defects among all applications.

We next describe a defect in the HsqlDB application to show the significance of conditional association rules, which cannot be mined by existing approaches such as WN-miner. The related code snippet from the `saveChanges` method 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` method, the `rollback` method should be invoked on the `Connection` object. Failing to invoke the `rollback` can make the database state inconsistent. This result shows that conditional 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 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 the available specifications and also 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. 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 detect rules that include a FC_e function call such as `Connection.rollback`, where FC_e can appear *only* in exception paths. Our approach significantly differs and improves upon their approach as we mine conditional association rules of the form "$(FC_c^1...FC_c^n) \wedge FC_a \Rightarrow (FC_e^1...FC_e^n)$" that can characterize more exception-handling rules. Our approach also addresses the problem of lacking enough supporting samples for these rules in the given application by expanding the data scope to open source repositories through a code search engine.

Acharya and Xie [1] proposed an approach for detecting API error-handling bugs in C code by approximating run-time API error behaviors through a push-down model checker. Their approach is also based on a similar technique used by Weimer and Necula for mining rules. Therefore, their approach also cannot detect the types of defects where FC_e appears only in exception paths. Furthermore, their approach is limited to certain types of defects, referred as critical defects, because their approach extracts rules from only those paths that exit the entire application. Our approach can detect more-general types of defects in exception paths that violate mined rules.

CodeWeb [13] developed by Michail pioneered mining association rules [2] from source code. CodeWeb mines association rules such as application classes inheriting from a library class often create objects of another class. PR-Miner developed by Li and Zhou [11] uses frequent itemset mining to extract implicit programming rules in large C code bases and detects violations. DynaMine developed by Livshits and Zimmermann [12] uses association rule mining to extract simple rules from software revision histories for Java code and detects violations of these rules. Engler et al. [5] proposed a general approach for finding bugs in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Wasylkowski et al. [21] mines rules that include pairs of API calls and detect violations. These API pairs represent an implicit ordering among these API calls. Recently, Schäfer et al. [16] developed an approach to 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. Furthermore, other approaches [11, 13] mine rules from a single sequence database and cannot handle multiple sequence databases. In contrast, our approach can handle multiple sequence databases and can mine more complex rules in the form of conditional association rules. Furthermore, approaches such as PR-Miner or DynaMine may suffer from issues of high false positives as their rule elements are not necessarily associated with program dependencies, which are considered in our approach.

Our approach is also related to other approaches that analyze exception behavior of programs. Fu and Ryder [7] 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 developed by Robillard and Murphy statically analyzes exception flow in Java source 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 the constructed EFG. Fu et al. [8] presents a *def-use*-based approach that helps to gather error-recovery code-coverage information. They also developed a visualization tool, called ExTest [6], that displays their static analysis results [8]. Our approach is different from their approach as we try to detect defects that violate mined rules rather than focusing on coverage of exception-handling code. Robillard and Murphy [15] developed an approach to simplify the exception structure at the design level and thereby improve the robustness and changeability of the program. In contrast, our approach targets at detecting exception-handling defects at the implementation level.

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. 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* programming language to specify domain-specific protocols. In contrast, our approach does not require any manual effort or the knowledge of any specific programming languages.

Finally, our previous approaches PARSEWeb [18] and SpotWeb [19] also exploit code search engines for gathering related code samples. PARSEWeb accepts queries of the form "*Source* → *Destination*" and mines frequent method-invocation sequences that accept *Source* and produce *Destination*. PARSEWeb constructs CFG and extracts method-invocation sequences that serve as solutions for the preceding query. SpotWeb accepts an input framework and detects hotspot classes and methods of the framework. SpotWeb detects hotspots using metrics such as the number of times the framework classes and methods are used by gathered code samples. Our new approach CAR-Miner significantly differs from these previous approaches. CAR-Miner constructs EFG and includes new techniques for collecting and post-processing the static traces. Furthermore, CAR-Miner incorporates our new mining algorithm for detecting exception-handling rules in the form of conditional association rules.

7 Conclusion

We developed an approach, called CAR-Miner, that mines exception-handling rules in the form of conditional association rules. Unlike simple association rules of the form "$FC_a \Rightarrow FC_e$", these conditional association rules of the form "$(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 ... FC_e^n)$" can characterize more complex exception-handling rules. As existing mining algorithms cannot detect these conditional association rules, we proposed a novel mining algorithm based on closed frequent 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 evaluated our approach with five real-world open source applications and showed that CAR-Miner extracted 294 real exception-handling rules. We also showed that CAR-Miner finds 160 new defects, where 87 are new defects, not being found by a previous related approach [22]. We also showed that our new conditional association rules are helpful in detecting 21 new defects. 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. In future work, we plan to adapt our new mining algorithm to frequent itemset mining and evaluate the strengths and weaknesses of both algorithms.

References

[1] M. Acharya and T. Xie. Static detection of API error-handling bugs via mining source code. Technical Report TR-2007-35, North Carolina State University Department of Computer Science, Raleigh, NC, October 2007.

[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. Navigating error recovery code in Java applications. In *Proc. ETX*, pages 40–44, 2005.

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

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

[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. 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] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. In *Proc. FSE*, pages 2–10, 2000.

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

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

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

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

[20] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. ICDE*, page 79. IEEE Computer Society, 2004.

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

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