# Making Exceptions on Exception Handling

Tao Xie
*North Carolina State University*
*Raleigh, NC, USA*
*xie@csc.ncsu.edu*

Suresh Thummalapenta
*IBM Research - India*
*Bangalore, India*
*surthumm@in.ibm.com*

***Abstract*—The exception-handling mechanism has been widely adopted to deal with exception conditions that may arise during program executions. To produce high-quality programs, developers are expected to handle these exception conditions and take necessary recovery or resource-releasing actions. Failing to handle these exception conditions can lead to not only performance degradation, but also critical issues. Developers can write formal specifications to capture expected exception-handling behavior, and then apply tools to automatically analyze program code for detecting specification violations. However, in practice, developers rarely write formal specifications. To address this issue, mining techniques have been used to mine common exception-handling behavior out of program code. In this paper, we discuss challenges and achievements in precisely specifying and mining formal exception-handling specifications, as tackled by our previous work. Our key insight is that expected exception-handling behavior may be "conditional" or may need to accommodate "exceptional" cases.**

## I. Introduction

Modern programming languages such as Java, C#, and C++ provide a mechanism, called exception handling, to deal with exception conditions that may arise during program executions. To produce high-quality programs, developers are expected to handle these exception conditions and take necessary recovery or resource-releasing actions [9]. Failing to handle these exception conditions can lead to not only performance degradation, but also critical issues [3], [7].

Although the exception-handling mechanism has existed from many years, developers often neglect writing exception-handling code for various reasons, such as those discovered in a study conducted by Shah et al. [2]. Among them, one main reason for neglecting exception handling is that it requires much additional code, and especially that the code is executed only in rare scenarios. Therefore, developers often do not see strong need for writing exception-handling code since it may not be worth spending time on writing exception-handling code. The study shows that developers wait until the actual errors occur and then add necessary exception-handling code on demand.

Another important reason is that, although languages enforce developers to write exception-handling code, via compilation errors such as not handling a certain exception, it is easy to defeat the language enforcement. For example, consider the code example shown in Figure 1. The code example is well written from the perspective of exception handling. The example handles `SQLException` and also includes necessary code for taking a recovery action. In addition, the example includes resource-releasing actions to make sure that all resources are released after exception conditions are satisfied (e.g., using the `finally` construct in Figure 1). Figure 2 shows a functionally-equivalent code example handling exceptions improperly. As is shown in the figure, the developers can defeat the language enforcement by handling `SQLException` via simply creating an empty `catch` block along with using `SQLException`'s super class `Exception`.

```
01:Connection conn = null; Statement stmt = null;
02:BufferedWriter bw = null; FileWriter fw = null;
03:try {
04:  fw = new FileWriter("temp.txt");
05:  bw = new BufferedWriter(fw);
06:  conn = DriverManager.getConnection("jdbc", "ps", "ps");
07:  stmt = conn.createStatement();
08:  stmt.executeUpdate("delete from table1");
09:  bw.write("...");
10:  conn.commit();
11:} catch (SQLException se) {
12:  if(conn != null) conn.rollback();
13:} catch (IOException ie) {
14:  if(bw != null) bw.flush();
15:} finally {
16:  if(stmt != null) stmt.close();
17:  if(conn != null) conn.close();
18:  if(bw != null) bw.close();
19:}
```

Figure 1. A well-written code example handling exceptions properly.

```
01:Connection conn = null; Statement stmt = null;
02:BufferedWriter bw = null; FileWriter fw = null;
03:try {
04:  fw = new FileWriter("temp.txt");
05:  bw = new BufferedWriter(fw);
06:  conn = DriverManager.getConnection("jdbc", "ps", "ps");
07:  stmt = conn.createStatement();
08:  stmt.executeUpdate("delete from table1");
09:  bw.write("...");
10:  conn.commit();
11:  stmt.close();
12:  conn.close();
13:  bw.close();
14:} catch (Exception ex) {
15:}
```

Figure 2. A code example handling exceptions improperly.

In practice, developers may tend to write code similar to the code example in Figure 2, since it is much simple and easy to understand, and more importantly behaves exactly

```
01:Connection conn = null; Statement stmt = null;
02:ResultSet res = null;
03:try {
04:  conn = DriverManager.getConnection("jdbc", "ps", "ps");
05:  stmt = conn.createStatement();
06:  res = stmt.executeQuery("select col from table1");
07:  while(res.next())
08:    System.out.println(res.getString(1));
09:} finally {
10:  if(res != null) res.close();
11:  if(stmt != null) stmt.close();
12:  if(conn != null) conn.close();
13:}
```

Figure 3. A code example that retrieves data from a database.

the same way as the code example in Figure 1 under normal circumstances. Furthermore, the exception conditions that were handled in Figure 1 may not be satisfied during normal testing.

Even when developers write code similar to the code example in Figure 1, the developers may often fail to write complete exception-handling code (e.g., missing a recovery action or a resource-releasing action).

To address these problems in practice, developers can write formal specifications to capture expected exception-handling behavior, and then apply tools to automatically analyze program code for detecting specification violations. However, in practice, developers rarely write formal specifications [1]. To address this issue, mining techniques [8], [5] have been used to mine common exception-handling behavior out of program code [3], [4]. Such common exception-handling behavior could likely be expected behavior to be included in formal exception-handling specifications. In this paper, we discuss challenges and achievements in precisely specifying and mining formal exception-handling specifications, as tackled by our previous work [3], [4].

Our key insight is that expected exception-handling behavior may be "conditional" or may need to accommodate "exceptional" cases. The "conditional" nature [3] denotes that a behavioral rule needs to be followed only under some situations. The "exceptional" nature [4] denotes that a behavioral rule needs to be followed under some majority situations whereas a different behavioral rule needs to be followed under some minority (i.e., "exceptional") situations.

## II. Specifying and Mining Precise "Conditional" Specifications

Existing specification mining techniques [6] mine exception-handling specifications as rules of the simple form "$FC_a \Rightarrow FC_e$", where $FC_a$ and $FC_e$ are function calls. The preceding rule describes that the function call $FC_a$ should be always followed by $FC_e$ in all paths, including exception paths that are exercised when exception conditions are satisfied. However, such simple-form specifications may be imprecise or insufficient in characterizing expected exception-handling behavior.

To illustrate such issue, we use the code example shown in Figure 3. This code example is similar to the code example in Figure 1, where both code examples open database connections. However, the code example in Figure 1 modifies contents of the database, whereas the code example in Figure 3 retrieves some data from the database. Therefore, the code example in Figure 3 does not require the `rollback` operation, as it does not modify the database contents. Consider a simple-form specification "Connection creation $\Rightarrow$ Connection rollback". This rule describes that a `rollback` function call should appear in exception paths whenever a `Connection` is created. However, this rule should apply to only the code example in Figure 1 and should not apply to the code example in Figure 3; such rule in its simple form is in fact applicable to both code examples, causing a false warning of rule violation for the code example in Figure 3. The primary reason is that the rollback function call should be invoked only when there is any change made to the database.

To address this issue and to mine precise specifications, our previous work [3] proposed an approach, called CAR-Miner, that mines sequence association rules of the form: "$FC_c^1...FC_c^n\ FC_a \Rightarrow FC_e^1...FC_e^m$". 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 "$FC_c^1 FC_c^2\ FC_a \Rightarrow FC_e^1$", where

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

Then this sequence association rule applies to only the code example in Figure 1 and does not apply to the code example in Figure 3 due to the presence of $FC_a$: `Statement.executeUpdate` in the rule, causing no false warning of rule violation. In our previous work, we use these rules as inputs for a static verification tool and then apply the tool to detect violations of these rules.

## III. Specifying and Mining Precise "Exceptional" Specifications

Expected exception-handling rules may often need to capture alternative patterns, originally introduced in our previous Alattin approach [4]. In our Alattin approach, we introduced alternative patterns to capture necessary condition checks that should be performed before invoking an API function call or after invoking an API function call. The primary reason for the existence of alternative patterns is that developers write source code in different ways to achieve the same programming task. In addition, some of these ways

are more frequent compared to others, which may be just minority (i.e., "exceptional") ways.

Such alternative patterns also apply for exception-handling rules. For example, consider the code example shown in Figure 1. An exception-handling rule associated with handling file writing can be expressed as "$FC_c^1\ FC_a \Rightarrow FC_e^1$", where

$FC_c^1$ : `FileWriter.constructor`
$FC_a$ : `BufferedWriter.constructor`
$FC_e^1$ : `BufferedWriter.close`

However, an alternative way for closing the file handle is to invoke the `close` method on the `FileWriter` instance, expressed as "$FC_c^1\ FC_a \Rightarrow FC_e^2$", where

$FC_c^1$ : `FileWriter.constructor`
$FC_a$ : `BufferedWriter.constructor`
$FC_e^2$ : `FileWriter.close`

Therefore, the complete rule can be captured as an alternative rule as "$FC_c^1\ FC_a \Rightarrow FC_e^1 \vee FC_e^2$", where

$FC_c^1$ : `FileWriter.constructor`
$FC_a$ : `BufferedWriter.constructor`
$FC_e^1$ : `BufferedWriter.close`
$FC_e^2$ : `FileWriter.close`

## IV. Conclusion

To produce high-quality programs, developers are expected to handle exception conditions and take necessary recovery actions or resource-releasing actions. Failing to handle these exception conditions can lead to performance degradation or critical issues. After formal exception-handling specifications are written, program code can be automatically analyzed to detect specification violations. However, in practice, developers rarely write formal specifications. To address this issue, our previous work [3], [4] mines common exception-handling behavior out of program code as exception-handling specifications. Our key insight is that expected exception-handling behavior (1) may be "conditional", captured as sequence association rules [3], or (2) may need to accommodate "exceptional" cases, captured as alternative patterns [4]. We expect that sequence association rules along with alternative patterns could effectively help in precisely capturing expected exception-handling behavior. In future work, we plan to empirically study precise specifications for exception-handling behavior by conducting characteristic studies of exception-handling behavior documented in API documentation [9].

## Acknowledgments

This work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, and ARO grant W911NF-08-1-0443.

## References

[1] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, November 2003.

[2] H. Shah, C. Görg, and M. J. Harrold. Why do developers neglect exception handling? In *Proc. 4th International Workshop on Exception Handling (WEH 2008)*, pages 62–68, 2008.

[3] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proc. 31st International Conference on Software Engineering (ICSE 2009)*, pages 496–506, 2009.

[4] S. Thummalapenta and T. Xie. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering*, 18(3-4):293–323, 2011.

[5] S. Thummalapenta, T. Xie, and M. R. Marri. Mining API usage specifications via searching source code from the web. In D. Lo, S.-C. Khoo, J. Han, and C. Liu, editors, *Mining Software Specifications: Methodologies and Applications*. Taylor & Francis, 2011.

[6] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 461–476, 2005.

[7] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proc. 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 419–431, 2004.

[8] T. Xie, S. Thummalapenta, D. Lo, and C. Liu. Data mining for software engineering. *IEEE Computer*, 42(8):35–42, August 2009.

[9] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 307–318, 2009.