Extending Static Code Analysis with Application-Specific Rules by Analyzing Runtime Execution Traces

Ersin Ersoy[1] and Hasan Sözer[2(✉)]

[1] Turkcell Technology, İstanbul, Turkey
ersin.ersoy@turkcell.com.tr
[2] Ozyegin University, İstanbul, Turkey
hasan.sozer@ozyegin.edu.tr

Abstract. Static analysis tools cannot detect violations of application-specific rules. They can be extended with specialized checkers that implement the verification of these rules. However, such rules are usually not documented explicitly. Moreover, the implementation of specialized checkers is a manual process that requires expertise. In this work, application-specific programming rules are automatically extracted from execution traces collected at runtime. These traces are analyzed offline to identify programming rules. Then, specialized checkers for these rules are introduced as extensions to a static analysis tool so that their violations can be checked throughout the source code. We implemented our approach for Java programs, considering 3 types of faults. We performed an evaluation with an industrial case study from the telecommunications domain. We were able to detect real faults with checkers that were generated based on the analysis of execution logs.

1 Introduction

Static code analysis tools (SCAT) can detect the violation of programming rules by checking (violation of) patterns throughout the source code [1]. The detected violations are reported in the form of a list of alerts. Although SCAT have been successfully utilized in the industry [7,8,15], they have limitations as well. It is very hard or undecidable to show whether an execution path is feasible or infeasible without the runtime context information [11]. As a result, some faults might be missed. SCAT also fall short to detect the violation of application-specific rules [3]. For example, it might be necessary to check some of the arguments and/or return values before/after certain method calls. SCAT do not consider such application-specific rules by default.

One can extend SCAT with specialized checkers to detect the violation of application-specific rules [3]. However, the implementation of specialized checkers is a manual process that requires expertise. In fact, state-of-the-art SCAT provide special extension mechanisms for defining new rules, which can be then checked by these tools. Yet, such rules have to be defined manually and they are usually not documented explicitly or formally.

T. Czachórski et al. (Eds.): ISCIS 2016, CCIS 659, pp. 30–38, 2016.
DOI: 10.1007/978-3-319-47217-1_4

In this paper, we introduce an approach for extending SCAT, in which specialized checkers are generated automatically. Our approach employs offline analysis of execution traces collected at runtime. These traces comprise a set of encountered errors. The source code is analyzed to identify faults that are the root causes of these errors. One could consider just to fix these faults without systematically and formally documenting them. However, instances of the same fault can exist at other places in the source code. It might also be possible that the same fault is introduced again later on. Therefore, it is important to capture this information and systematically check for the identified faults in the overall source code regularly. In our approach, programming rules are inferred to prevent these pitfalls. Specialized checkers are automatically generated for these rules and they are introduced as extensions to SCAT. The extended SCAT can detect the violation of the inferred rules throughout the source code.

We performed an evaluation with an industrial case study from the telecommunications domain. We captured the execution logs of a previous version of a large scale system implemented in Java. A number of recorded errors are analyzed for 3 types of errors and the corresponding faults are identified. We generated rules and specialized checkers for these faults, which were already fixed. The SCAT that is employed in the company is extended with these checkers. Then, we were able to detect several new instances of the identified faults that had to be fixed.

The remainder of this paper is organized as follows. The following section summarizes the related studies. We present the overall approach in Sect. 3. The approach is illustrated in Sect. 4, in the context of the industrial case study. Finally, in Sect. 5, we conclude the paper.

2 Related Work

There have been studies for automatically deriving programming rules based on frequently used code patterns [4,5]. Hereby, pattern recognition, data mining and heuristic algorithms are used for analyzing the program source code and detecting potential rules. Then, the source code is analyzed again to detect inconsistencies with respect to these rules. These studies utilize only (models of) the source code to infer programming rules. They do not make use of runtime execution traces.

There are studies [2,14] that make use of the analysis of previously fixed bugs to derive application-specific programming rules. However, programmers have to define the rules applied to fix these bugs. Hence, they rely on manual analysis. In addition, they do not exploit any information collected during runtime execution.

There exist a few approaches [9,10,13] that exploit dynamic analysis and runtime execution traces. DynaMine [9] uses dynamic analysis for validating programming rules that are actually derived by mining the revision history. Another approach [13] relies on the analysis of console logs to detect anomalies [13]; however, deriving rules for preventing these anomalies was out of the scope of the study. Daikon [10] derives likely invariants of a program by means of

dynamic analysis. However, Daikon focuses on numerical properties of variables as system constraints rather than bug patterns that can represent a wider range of bug types.

We have previously introduced an approach to generate runtime monitors based on SCAT alerts [12] These monitors identify alerts, which do not actually cause any failures at runtime. Then, filters are automatically generated for SCAT to supress these alerts. Hence, the goal is to reduce false positives and increase precision. In this work, we aim at reducing false negatives by detecting more faults as a result of checking application-specific rules. As such, the goal of the approach proposed in this paper is to increase recall instead.

3 Generating Rules from Execution Traces

Our approach takes runtime execution traces of a system as input. These traces should comprise the set of errors encountered and the set of software modules involved. The output is a set of checkers that are provided as extensions to SCAT. These checkers detect instances of faults that are the root causes of the logged errors. To be able to identify these faults and to generate the corresponding checkers, a library of analysis procedures and a library of checker templates are utilized, respectively. The scope of these libraries define the set of error and fault types that can be considered by the approach.

The overall process is depicted in Fig. 1, which involves 4 steps. First, *Log Parser* takes runtime logs as input, parses these logs, and generates the list of errors recorded together with the related modules and events (1). Then, this list is provided to *Root Cause Analyzer*, which analyzes the source code to identify the cause of the error by utilizing a set of predefined analysis procedures (2). For instance, if a null pointer reference error is detected at runtime, the corresponding analysis procedure locates the corresponding object and its last definition before the error. Let's assume that such an object was defined as the return value of a method call. Then, a rule is inferred, imposing that the return value of that particular method must be checked before use. The list of such rules are provided to *Checker Generator*, which uses a library of predefined templates to generate

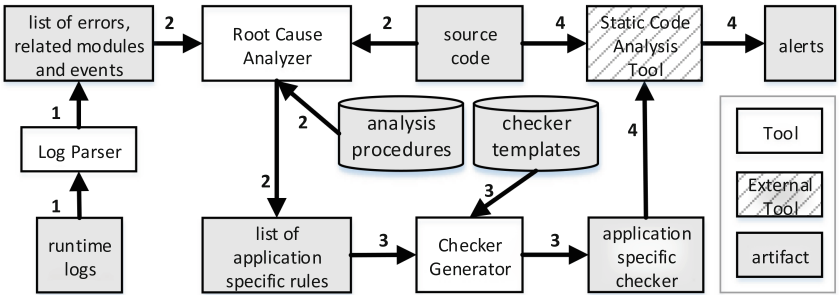

Fig. 1. The overall process.

a specialized checker for each rule (3). The generated checkers are included as extensions to SCAT, which applies them to the source code and reports alerts in case violations are flagged (4).

The overall process is automated; however, it relies on a set of predefined analysis procedures and checker templates. One analysis procedure should be defined for each error type and one checker template should be defined for each rule type. The set of rules and error types are open-ended in principle and they can be extended when needed. Currently, we consider the following types of errors and programming rules that are parametrized with respect to the involved method and argument names.

- *java.lang.IndexOutOfBoundsException*: The arguments of a method must be checked for boundary values before the method call, e.g., *if(x < MAX) m(x);*
- *java.lang.NullPointerException*: The return value of a method must be checked for null reference, e.g., *r = m(x); if(r != null)* {...} or *if(r == null)* {...}
- *org.hibernate.LazyInitializationException*: The JPA Entity[1] should be initialized at a transactional level (when persistence context is alive) before being used at a non-transactional level, e.g., object *a* is a JPA Entity with *LAZY* fetch type and it is an aggregate within object *b*. Then, *a* must be fecthed from the database when *b* is being initialized, for a possible access after the persistant context is lost.

In the following, we explain the steps of the approach in more detail with a running example. Then, in Sect. 4, we illustrate the application of the approach in the context of an industrial case study[2].

Analysis of Execution Logs: The first step of our approach involves the analysis of execution logs. In our case study, we had to utilize existing log files of a legacy system. Therefore, *Log Parser* is implemented as a dedicated parser for these files. However, it can be replaced with any parser to be able to process log files in other formats as well. Our approach is agnostic to the log file structure as long as the following information can be derived: *(i)* Sequence of events and in particular, encountered errors; *(ii)* The types of encountered errors; *(iii)* The location of the encountered errors in the source code, i.e., package, class, method name, line number. Even standard Java exception reports include such information together with a detailed stack trace. Hence, existing instrumentation and logging tools can be employed to obtain the necessary information. *Log Parser* is parametric with respect to the focused error types and modules of the system. We can filter out some error types or modules that are deemed irrelevant or uncritical.

[1] A JPA (Java Persistence API) entity is a POJO (Plain Old Java Object) class, which has the ability to represent objects in a database. They can be reached within a persistent context.

[2] Currently our toolset works on software systems written in Java. In principle, the approach can be instantiated for different programming languages/environments. Our design and implementation choices were driven by the needs and the context of the industrial case.

Root Cause Analysis: Once *Log Parser* retrieves the relevant error records together with their context information, it provides them to *Root Cause Analyzer*. This tool performs two main tasks: *(i)* finding the root cause of the error, *(ii)* determining whether this root cause is application-specific or not. We are not interested in generic errors. Hence, it is important to be sure that the root cause of the error is application-specific. For instance, consider the code snippet in Listing 1.1. When executed, it causes a *java.lang.NullPointerException*; however, *Root Cause Analyzer* ignores this error because, the cause of the error is an object that is simply left unitialized. This is a generic error.

Listing 1.1. An sample code snippet for a generic error that is ignored by *Root Cause Analyzer*.

```
1 static Report aReport;
2 public static void print() { System.out.println(aReport); }
```

If the null value is obtained from a specific method in the application, then such an error is deemed relevant (See Listing 1.2). That means, the return value of the corresponding method (e.g., *getServiceReport*) must be always checked before use. This is a type of rule that is determined by *Root Cause Analyzer*.

Listing 1.2. A possible application-specific error that is considered by *Root Cause Analyzer*.

```
1 static Report aReport = getServiceReport();
2 public static void print() { System.out.println(aReport); }
```

Root Cause Analyzer employs a set of predefined analysis procedures that are coupled with error types. For example, the analysis procedure applied for null pointer exceptions is listed in Algorithm 1. Hereby, the use of the object that caused a null pointer exception is located as the first step. Second, the reaching definition is found for this use of the object. If this definition is performed with a method call, the procedure checks where the method is defined. If the method is defined within the application, then a rule is reported for checking the return value of this method.

Root Cause Analyzer provides the type of rule to be applied and the parameters of the rule (e.g., name of the method, of which return value must be checked) to *Checker Generator* so that a specialized checker can be created.

Algorithm 1. Root cause analysis procedure applied for null pointer exceptions.

```
1: u ← use of object that causes the exception
2: d ← reaching definition for u
3: if ∃ method m as part of d then
4:    p ← package of m
5:    if p ∈ application packages then
6:       reportRule(RETURNVALCHECK, m)
7:    end if
8: end if
```

Generation of Specialized Checkers: Most SCAT are extensible; they provide application programming interfaces (API) for implementing custom checkers. *Checker Generator* generates specialized checkers by utilizing PMD[3] as SCAT. PMD uses JavaCC[4] to parse the source code and generate its abstract syntax tree (AST). This AST can be traversed with its Java API to define specialized checkers for custom rules. These checkers should conform to the Visitor design pattern [6]. Each checker is basically defined as an extension of an abstract class, namely, *AbstractJavaRule*. The *visit* method that is inherited from this class must be overwritten to implement the custom check. This method takes two arguments: *(i) node* of type *ASTMethodDeclaration* and *(ii) data* of type *Object*. The return value is of type *Object*. This visitor method is called by PMD for each AST node (e.g., method).

Checker generation is performed based on parametrized templates. We defined a template for each rule type. Each template extends the *AbstractJavaRule* class and overwrites the necessary visitor methods. A checker is generated by instantiating the corresponding template by assigning concrete values to its parameters. For instance, consider a specialized checker that enforces the handling of possible null references returned from a method in the application. The corresponding pseudo code that is implemented with PMD is listed in Algorithm 2. Hereby, all variable declarations are obtained as a set (V at Line 1). For each of these declarations (v), the node ID (vid) is obtained (Line 3). The name of the method call (m) is also obtained, assuming that the declaration involves a method call (Line 4). If there indeed exists such a method call and if the name of the method matches the expected name (i.e., $METHOD$), then an additional check is performed (*isNullCheckPerformed* at Line 6). This check traverses the AST starting from the node with id *vid* and searches for control statements that compare the corresponding variable (v) with respect to null (i.e., *if(v != null)* {...} or *if(v == null)* {...}). If there is no such a control statement before the use of the variable, then a violation of the rule is registered (Line 8).

Checker Generator generates specialized checkers by instantiating the corresponding template with the parameters (e.g., $METHOD$) provided by *Root Cause Analyzer*. Hence, multiple checkers can be generated based on the same rule type.

Extension of Static Code Analysis Tool: PMD is extended with the custom checkers generated by *Checker Generator* and it is executed by Sonar[5] version 4.0. The extension is performed in two steps: *(i)* adding a jar file that includes the custom checker, and *(ii)* extending the XML configuration file for rule definition. The jar file basically contains an instantiation of a checker code template. The rule regarding the introduced checker is defined in the XML configuration file by a new entry pointing at this jar file. It also specifies the *name*, *message* and *description* of the rule, which are displayed to the user as part of the listed alerts, when violations are detected.

[3] http://pmd.sourceforge.net/.
[4] https://javacc.java.net/.
[5] http://www.sonarqube.org/.

Algorithm 2. visit method of a specialized checker for a custom rule, i.e., *handle possible null pointer after calling the method.*

```
1: V = getChildrenOfType(ASTVariableDeclarator);
2: for all v ∈ V do
3:    vid = v.getID();
4:    m = v.getMethodCall();
5:    if m! = null & m.name == METHOD then
6:       isChecked = isNullCheckPerformed(vid)
7:       if !isChecked then
8:          addViolation(vid)
9:       end if
10:   end if
11: end for
```

4 Industrial Case Study

We performed a case study on a Sales Force Automation system maintained by Turkcell[6]. The system comprises more than 200 KLOC. It is operational since 2013, serving 2000 users. We downloaded all the log files regarding a previous version of this system. *Log Parser* identified an error in these files. The corresponding source code snippet is listed in Listing 1.3, where the object *opty* turns out to be null. Then, *Root Cause Analyzer* located the point in the source code, where this object was last defined (Line 1). The definition is coming from a method call, i.e., *templateDao.find(Opty.class, optyNo);*. This method creates and returns an object by utilizing information from a database; it returns null if the required information cannot be found.

Listing 1.3. The code snippet corresponding to the logged error.

```
1  Opty opty = templateDao.find(Opty.class, optyNo);
2  if (opty.getCoptycategory().equals(...)) { ... }
```

Then, an application-specific rule is inferred as: the return value of the method *find* must be checked for null references before use. A specialized checker is automatically generated based on this rule. It checks the whole code base and searches for initialized objects using the return value of the method *find* without a null reference check. As the last step, Sonar is extended with the specialized checker.

After the extension, 25 additional alerts were generated. All the alerts were true positives and the corresponding code locations really required to be fixed. In fact, we have seen that 3 of these locations caused errors afterwards and they were fixed in a later version of the source code. If our approach were applied and all the reported alerts were addressed, these errors would not occur at all. As a result, 25 real faults were detected with specialized checkers and 3 of them were

[6] http://www.turkcell.com.tr.

activated during operational time. This result shows the importance and high potential of information collected at runtime as a source for improving recall in static analysis.

5 Conclusion

In this work, we extracted application-specific programming rules by analyzing logged errors. We automatically generated specialized checkers for these rules as part of a static code analysis tool. Then, the tool can check for potential instances of the same type of error throughout the source code. We conducted an industrial case study from the telecommunications domain. We were able to detect real faults, which had to be fixed later on. In the future, we plan to extend our approach to cover more than 3 types of errors and rules. We also plan to conduct more case studies.

Acknowledgments. This work is supported by The Scientific and Research Council of Turkey (113E548).

References

1. Johnson, B., et al.: Why don't software developers use static analysis tools to find bugs?. In: Proceedings of the 35th International Conference on Software Engineering, pp. 672–681 (2013)
2. Sun, B., et al.: Automated support for propagating bug fixes. In: Proceedings of the 19th International Symposium on Software Reliability Engineering, pp. 187–196 (2008)
3. Sun, B., et al.: Extending static analysis by mining project-specific rules. In: Proceedings of the 34th International Conference on Software Engineering, pp. 1054–1063 (2012)
4. Chang, R., Podgurski, A.: Discovering programming rules and violations by mining interprocedural dependences. J. Softw. Mainten. Evol. Res. Pract. **24**, 51–66 (2011)
5. Chang, R., Podgurski, A., Yang, J.: Discovering neglected conditions in software by mining dependence graphs. IEEE Trans. Softw. Eng. **34**(5), 579–596 (2008)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston (1995)

7. Zheng, J., et al.: On the value of static analysis for fault detection in software. IEEE Trans. Softw. Eng. **32**(4), 240–253 (2006)
8. Krishnan, R., Nadworny, M., Bharill, N.: Static analysis tools for security checking in code at motorola. ACM SIG Ada Lett. **28**(1), 76–82 (2008)
9. Livshits, B., Zimmerman, T.: Dynamine: finding common error patterns by mining software revision histories. SIGSOFT Softw. Eng. Not. **30**, 296–305 (2005)
10. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
11. Ayewah, N., et al.: Using static analysis to find bugs. IEEE Softw. **25**(5), 22–29 (2008)
12. Sozer, H.: Integrated static code analysis and runtime verification. Softw. Pract. Exp. **45**(10), 1359–1373 (2015)
13. Xu, W., et al.: Detecting large-scale system problems by mining console logs. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, pp. 117–132 (2009)
14. Williams, C., Holingsworth, J.: Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Softw. Eng. **31**, 466–480 (2005)
15. Yuksel, U., Sozer, H.: Automated classification of static code analysis alerts: a case study. In: Proceedings of the 29th IEEE International Conference on Software Maintenance, pp. 532–535 (2013)