Type Checking for Reliable APIs

Maria Kechagia and Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{mkechagia,dds}@aueb.gr

***Abstract*—In this paper, we propose to configure at compile time the checking associated with Application Programming Interfaces' methods that can receive possibly malformed values (e.g. erroneous user inputs and problematic retrieved records from databases) and thus cause application execution failures. To achieve this, we design a type system for implementing a pluggable checker on the Java's compiler and find at compile time insufficient checking bugs that can lead to application crashes due to malformed inputs. Our goal is to wrap methods when they receive external inputs so that the former generate checked instead of unchecked exceptions. We believe that our approach can improve Java developers' productivity, by using exception handling only when it is required, and ensure client applications' stability. We want to evaluate our checker by using it to verify the source code of Java projects from the Apache ecosystem. Also, we want to analyze stack traces to validate the identified failures by our checker.**

***Keywords*-application programming interfaces; exceptions; type systems;**

I. Introduction

Application programming interfaces (APIs) are bundles of interfaces, classes, methods, and fields that developers use to program the main functionalities of client systems and applications. Even though APIs are the builders of modern software, mostly the last five years there is growing research interest regarding APIs' usability [1]–[4] and evaluation [5]–[7]. Still, there is scant research concerning the automation of methods that can guarantee APIs' reliability. This possibly occurs because APIs run on diverse usage contexts and on many devices with different specifications, making APIs' debugging and testing challenging [8].

Currently, to ensure applications and systems' robustness, programming languages, such as Java, C++, C#, Objective-C and scripting languages, provide exception handling mechanisms on several flavors. Here, we take into account the controversial exception types of the Java programming language. We investigate how Java can be extended to improve the reliability of Java APIs and the productivity of client applications' developers.

In brief, Java has two types of exceptions: checked and unchecked.[1] If a client application can do something when an exceptional condition occurs, and this condition is unpredictable, then there should be used a **checked exception**. This guarantees that the client will handle the exceptional condition, preventing the application from execution failures. If the client application can predict and avoid an exceptional condition (e.g. by passing correct values), or if the client application cannot do anything to recover from that condition, then there should be used an **unchecked exception**. This does not force the client to write exception handling code that could be buggy [9].

In this work, we propose to configure at compile time the checking associated with API methods that can receive possibly malformed values and thus cause application execution failures (e.g. due to erroneous user inputs and problematic retrieved records from databases). To achieve this, we design a type system for checking methods that can possibly receive malformed values passed as external inputs.

We plan to implement a pluggable checker on the Java's compiler and find at compile time insufficient checking bugs related to invalid inputs. In particular, we wrap a method and throw a checked instead of an unchecked exception *only* when this method receives external inputs. This approach can improve: 1) the productivity of Java developers, by using checked exceptions when it is needed, and 2) the robustness of client applications. We expect that our technique will assist the building of modern mobile and web applications that their stability highly depends on received external inputs, such as: user inputs, data from network, and sensor inputs [10].

To evaluate our checker, we can use it to verify the source code of Apache Java projects. Also, we can analyze stack traces from these projects to cross-check the identified failures by our checker. Here, we list ten stack traces extracted from the Jira issue tracker that we manually analyzed to show execution failures that could have been avoided, by using our system.

The structure of this paper is as follows. Section II lists motivating examples of our proposal. Section III presents the design of the type system for the checker. Finally, Section IV presents related work and Section V outlines our conclusions and plans for future work.

II. Motivating Examples

An application execution failure (crash) can be related to user input. For instance, consider the case when the user passes an invalid URL to the arguments of a Java program. An error would occur and the application would crash. To avoid such errors, the *javadoc* of the URL class informs Java developers to *always* throw and handle a `MalformedURLException` (checked) exception when creating a new URL.[2]

[1] https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html.
All the referred URLs are archived in http://istlab.dmst.aueb.gr/∼mkehagia/checker.txt

[2] https://docs.oracle.com/javase/7/docs/api/java/net/URL.html#URL(java.lang.String)

Listing 1. Malformed URL

```
import java.net.MalformedURLException;
import java.net.URL;

public class URLReader {
   public static void main(String[] args) {
      try {
         /* URL signature:
          * URL(String spec) throws MalformedURLException
          */

         // Case 1: user input
         URL url1 = new URL(args[0]);

         // ...
      } catch(MalformedURLException e) {
         System.err.println("Invalid URL");
         // Give some new URL or use default URL ...
      }

      // Case 2: constant url
      URL url2 = new URL("http://www.example.com");

      // ...
   }
}
```

Java's compile-time checks that ensure whether developers handle specific types of exceptions (checked) have been controversial within the software engineering community [8], [11]. In the following, we explain when there is a need for exception static checking and when this is unnecessary.

We argue that *only* when a method (or a constructor) receives external input, this method should throw a checked exception. For Case 1, in Listing 1, the programmer **should** handle a checked exception, because it is *unpredictable* if the user input will be well-formed.

On the contrary, for a constant value (see Case 2 in Listing 1) exception handling is **needless**, since the value of the URL is *known* at compile-time and its validity can be tested before releasing the software. Failures in such cases can be crashes due to a misconfiguration. These should be handled by specified IT personnel.

Listing 2. Case for unchecked exception

```
/*
 * URL(ThrowingUncheckedException dummy,
 * @WellformedURL String spec)
 * throws InvalidUrlUncheckedException
 */

// Case 3: Constant value
String u = "http://www.example.com/";
URL url3 = new URL(ThrowingUncheckedException.instance,
     @WellformedURL u);
```

In this context, we propose that our system will modify the program, **during compilation**, to use a different version of the URL constructor (see the block comment in Listing 2). The new constructor will throw an unchecked exception instead of a checked one. Then, Case 2 in Listing 1 should get modified (while compiling) into Case 3 in Listing 2, whereas Case 1 in Listing 1 should remain as it is. Consequently, programmers would not need to use exception handling for Case 2. This makes their source code cleaner, more maintainable, and, at the same time, more reliable.

Furthermore, methods that can currently raise unchecked exceptions can be similarly converted to also raise checked ones when needed. For instance, take into account Listing 3. According to the *javadoc* of the `compile` method, this method can throw an unchecked exception.[3] We agree that this is reasonable for Case 5, where the pattern is a constant value. However, when the pattern comes as a user input, we envisage that the client should wrap the `compile` method with a checked exception. Thus, Case 4 in Listing 3 should get modified (while compiling) into Case 6 in Listing 4, whereas Case 5 should remain as it is. Consequently, the programmer should write a `try–catch` block to handle the checked exception, `InvalidPatternCheckedException`, that the `compile` method would throw in Case 4.

Listing 3. Malformed pattern

```
import java.util.regex.InvalidPatternCheckedException;
import java.util.regex.Pattern;

public class Parser {

   public static void main(String[] args) {
      try {
         // Case 4: User input
         Pattern pattern1 = Pattern.compile(args[0]);

         // ...
      } catch(InvalidPatternCheckedException e) {
         System.err.println("Invalid pattern");
         // Give a new correct pattern ...
      }

     /* Pattern compile(String regex)
      *  throws PatternSyntaxException
      */

      // Case 5: Constant value
      Pattern pattern2 = Pattern.compile("^xy");

      // ...
   }
}
```

Listing 4. Case for checked exception

```
/*
 * Pattern compile(ThrowingCheckedException dummy,
 * String regex) throws InvalidPatternCheckedException
 */

// Case 6: User input
Pattern pattern = Pattern.compile(
   ThrowingCheckedException.instance, args[0]);
```

We want to build on top of the Java type system and add specific checks, so that the compiler can prove that an error due to a malformed external input will not manifest at runtime. To do this, we use the Checker Framework.[4] Given that this framework can strengthen the Java compiler, but it does **not** permit illegal Java programs, we have first to modify particular Java libraries. Notice that in Case 3 and Case 6 we use new overloaded methods. As a future work, we have also to consider the functional features of Java 8, because they can make method overloading unclear. This mainly occurs when in functions only method names are referred without formal or actual parameters.

[3] https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#-compile(java.lang.String)

[4] https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html

TABLE I
EXAMPLES OF API FAILURES IN APACHE PROJECTS

#	Project	Source	Sink Method	Root Unchecked Exception	Crash Cause
1	Hadoop	URI	URI.getHost	NullPointerException	Invalid host name
2	Lucene	file index	Long.parseLong	NumberFormatException	Invalid file name
3	Fop	factor	InputHandler.transformTo	IllegalArgumentException	Illegal symbol
4	Pivot	path	FileBrowserSheet.setRootDirectory	IllegalArgumentException	Invalid directory
5	Cassandra	node	Integer.parseInt	NumberFormatException	Malformed string
6	Spark	data file	Double.parseDouble	NumberFormatException	Wrong field separators
7	Tuscany	property	Integer.parseInt	NumberFormatException	Impossible data conversion
8	Mahout	CSV file	KMeansDriver.buildClusters	IllegalStateException	Invalid arguments
9	Olio	argument	Integer.parseInt	NumberFormatException	Invalid argument
10	Tapestry	URL	URLEncoderImpl.decode	IllegalArgumentException	Incorrect URL

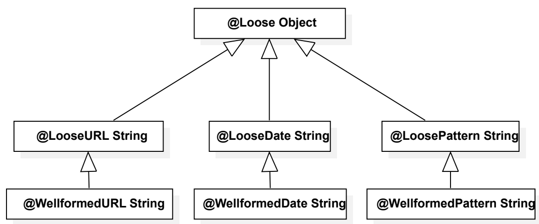

Fig. 1. Type hierarchy of the Well-formedness Type Checker

Table I shows API failures that could have been avoided if developers were forced by our system to catch the appropriate exceptions. We have manually extracted and analyzed these ten crash reports from the Jira issue tracker.[5]

III. TYPE CHECKER

We plan to implement a pluggable checker on the Checker Framework [12] and find at compile time bugs that can lead to execution failures due to invalid inputs. In this section, we describe the type system and the related type inference rules that we have designed. According to Papi et al. [12] and the framework's manual, we present the prerequisites for the new pluggable checker of the Checker Framework.

A. Type Qualifiers and Hierarchy

According to Weitz et al., a type qualifier (annotation) is attached to every occurrence of a type in the language [13]. Here, we define the annotations for the type system and the sub-typing relationships among qualified types. The well-formedness checker will implement a qualified type system where for every Java type, `@Wellformed` WFT is a subtype of `@Loose` LT.

In addition, we suggest the use of annotations for each type of well-formedness for the validation of the values passed in the arguments of a method. Specifically, for the URL type system, a reference of type `@LooseURL String` can have a malformed `String` value. By contrast, a reference of type `@WellformedURL String` always refers to a well-formed `String` value (e.g. a well-formed URL). Thus, an expression of type `@WellformedURL String` can never cause a crash related to a malformed URL. From an API designer's perspective, we can consider the signature of the URL constructor in the block comment of Listing 2 where we have annotated the `String spec` argument that should be always well-formed (`@WellformedURL`).

[5]https://issues.apache.org/jira/browse/ (login required)

For each type of well-formedess (e.g. URL, date, pattern) there will be used different type qualifiers. See the hierarchy in Figure 1. This prevents from passing a well-formed Date for instance into a context that expects a well-formed URL.

B. Type Inference Rules

In the following, we present the inference rules of our type system of well-formedness. We generalize our inference rules by using our generic annotations, namely, LT represents a `@Loose` LT type and WFT represents a `@Wellformed` WFT type. The $funct(t1, t2)$ refers to any operation between $t1$ and $t2$, such as $t1 + t2$. The first general rule (see equation 1) defines the operation (where T is a general type). The remaining rules say that if there is a loose type in an operation, the output type will be loose too (see equations 3 and 4); Otherwise, if all types are well-formed, the output type will be well-formed (see equation 2). We have based our rules on Pierce's semantics [14].

$$\Gamma \vdash funct : (T, T) \rightarrow T \tag{1}$$

$$\frac{\Gamma \vdash t1 : WFT \quad \Gamma \vdash t2 : WFT}{\Gamma \vdash funct(t1, t2) : WFT} \tag{2}$$

$$\frac{\Gamma \vdash t1 : WFT \quad \Gamma \vdash t2 : LT}{\Gamma \vdash funct(t1, t2) : LT} \tag{3}$$

$$\frac{\Gamma \vdash t1 : LT \quad \Gamma \vdash t2 : LT}{\Gamma \vdash funct(t1, t2) : LT} \tag{4}$$

C. Type Introduction Rules

Even though constants are theoretically always well-formed, they can also be validated (e.g. by the Checker Framework) at compile time [15]. Equation 5 refers that a constant type K should be anyway well-formed, WFT.

$$\Gamma \vdash K \quad : \quad WFT \tag{5}$$

We define as a **default** type for all values the loose one (`@Loose`), because not all values should be strictly well-formed, such as in the case of a URL, a date, or an SQL statement. This convention reduces the programmer's annotation burden, because they have to add **only** the `@Wellformed` annotation where a particular variable in a program should not be malformed.

D. Produced Exceptions

During compilation, there will be applied the following conventions regarding the exceptions that methods that receive possibly malformed external inputs can throw. Equation 6 says that a method with well-formed arguments can throw an unchecked exception (Case 2 in Listing 1 and Case 5 in Listing 3). On the contrary, equation 7 says that a method that can receive malformed external input can throw a checked exception (Case 1 in Listing 1 and Case 4 in Listing 3).

$$f(WFT) \Rightarrow \quad throws \quad unchecked \quad exception \tag{6}$$

$$f(LT) \Rightarrow \quad throws \quad checked \quad exception \tag{7}$$

IV. Related Work

Several studies have been conducted concerning the reliability of modern APIs. Existing work mainly refers to security issues [5], [7] and changes in API source code that may introduce bugs [6], [16]. We investigate API design deficiencies, regarding exception handling, that hinder developers to productively write robust applications.

A significant body of research also focuses on the study of the exception handling mechanisms in Java. Empirical studies show that exception handling in Java programs makes programming difficult and buggy [8], [9], [17]. This has led researchers to develop static and dynamic analysis tools to predict and simplify the use of exceptions [8], [18], [19]. Contrary to previous approaches, we propose a type system for checking (while compiling) methods that can possibly accept malformed values passed as external inputs.

V. Conclusion

We discuss a well-known problem that several client applications' developers of Java APIs face regarding the handling of redundant checked exceptions. To alleviate this, we designed a type system for the implementation of a pluggable checker to find at compile time bugs that can lead to application crashes due to malformed inputs. We envisage to wrap methods and throw checked instead of unchecked exceptions *only* when these methods receive unpredictable external inputs.

As a future work, we want to run the type checker on Apache Java projects' source code to find methods that can potentially accept malformed external inputs and crash client applications. Also, we want to use stack traces from the Jira issue tracker to cross-check *which* methods actually cause crashes because of problematic external inputs. Finally, we would like to run a trial on Java professionals to evaluate the usefulness of our system.

Acknowledgment

We thank Leonidas Lampropoulos for help in verifying the type inference rules and Michael Ernst for his internal reviews and comments on the paper. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732223.

References

[1] M. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[2] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do API documentation and static typing affect API usability?" in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 632–642.

[3] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 511–522.

[4] B. A. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, pp. 62–69, May 2016.

[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[6] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 477–487.

[7] L. Li, T. F. D. A. Bissyande, Y. Le Traon, and J. Klein, "Accessing inaccessible Android APIs: An empirical study," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016, p. 12.

[8] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, Apr. 2003.

[9] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in Java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015.

[10] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 190–203.

[11] Y. Zhang, G. Salvaneschi, Q. Beightol, B. Liskov, and A. C. Myers, "Accepting blame for safe tunneled exceptions," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 281–295.

[12] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ACM, 2008, pp. 201–212.

[13] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst, "A type system for format strings," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 127–137.

[14] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002, see Chapter 8.

[15] V. Karakoidas, D. Mitropoulos, P. Louridas, and D. Spinellis, "A type-safe embedding of SQL into java using the extensible compiler framework J%," *Computer Languages, Systems & Structures*, vol. 41, pp. 1–20, 2015.

[16] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the Android ecosystem," in *2013 IEEE International Conference on Software Maintenance*, Sept 2013, pp. 70–79.

[17] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining programmer practices for locally handling exceptions," in *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 2016, pp. 484–487.

[18] W. Weimer and G. C. Necula, "Exceptional situations and program reliability," *ACM Transactions on Programming Language Systems*, vol. 30, no. 2, pp. 8:1–8:51, 2008.

[19] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity JVMs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 83–101.