A Type System for Regular Expressions

Eric Spishak

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

March 2012

Presentation of work given on ____________________

Thesis and presentation approved by ____________________

Date ____________________
Abstract

This paper describes practical experience building and using a type system for regular expressions. A regular expression is a pattern for matching certain strings of text. Regular expressions are often complex and different across programming languages. Therefore, it is easy for developers to make syntactic mistakes when writing regular expressions, resulting in runtime errors.

To detect errors in regular expression syntax, we have created a type system for regular expressions. This type system validates regular expression syntax and capturing group usage at compile time instead of runtime. This paper describes the type system, as well as its motivation and implementation. In an evaluation of five Java applications and nearly half a million lines of code the type system was easy to use and found 50 bugs.

1 Introduction

A regular expression is a pattern for matching certain strings of text. It is easy to make syntax mistakes in regular expressions because regular expressions are often complex. An example of a syntactically invalid regular expression is shown below. Note that this regular expression is syntactically invalid because of the unclosed parenthesis at the end of it.

```
(.* ([0-9]+)(
```

In Java, regular expressions are used as follows:

```
Pattern p = Pattern.compile("(.*) ([0-9]+)\);
Matcher m = p.match("number 447");
if (m.matches())) {
    System.out.println(m.group(2)); // output: 447
}
```

This code creates a regular expression with two capturing groups, each delimited by parentheses. Then, the code prints the text matched by the second capturing group.

This paper considers two types of errors associated with regular expression in Java. Errors in regular expression syntax are described in Section 1.1 and errors in usage of capturing group functionality are described in Section 1.2.

1.1 Regular Expression Syntax Errors

If a regular expression with invalid syntax is compiled, the `Pattern.compile` method will throw a `PatternSyntaxException` at runtime. An example of this is shown here:

```
Pattern p = Pattern.compile("(.*) ([0-9]+)\);
// invalid regular expression syntax,
// throws PatternSyntaxException at runtime
```

1.2 Capturing Group Errors

A capturing group is a part of a regular expression that is delimited by parentheses. The text that matches the part of a regular expression within a capturing group can be extracted.

If a capturing group number that does not exist in a regular expression is used then an `IndexOutOfBoundsException` is thrown at runtime. An example of this is shown here:

```
Pattern p = Pattern.compile("([0-9]+)\);
// compile a regular expression with 2 capturing groups
Matcher m = p.match("number 447");
if (m.matches())) {
    System.out.println(m.group(3)); // throws IndexOutOfBoundsException at runtime
}
```

Since group 3 does not exist in the regular expression, an `IndexOutOfBoundsException` is thrown.
1.3 Organization

This paper presents a type system that checks the validity of regular expressions and regular expression capturing group access at compile time rather than at run time. This approach helps developers find bugs in syntax and capturing group usage of regular expressions earlier and more effectively than the previous method of relying on run time errors.

This paper is organized as follows. Section 2 describes the type system for regular expressions. Section 3 describes implementation details of the type system. Section 4 presents results of using this type system on five Java applications. Section 5 discusses positives and negatives of the type system. Section 6 discusses related work. Section 7 concludes.

2 Type System

The regular expression type system distinguishes between Strings that are syntactically valid regular expressions and Strings that are not. This is so that the type system can verify that a syntactically valid regular expression is used where required. The type system also tracks the number of capturing groups in a regular expression. This is so that the type system can verify the correct parameter to a method call that takes a capturing group number.

The type system introduces a @Regex qualifier to qualify a type that is a syntactically valid regular expression (Section 2.1). The @Regex qualifier also takes an optional parameter qualifying how many capturing groups are in the given regular expression (Section 2.2).

2.1 @Regex for Valid Regular Expression Syntax

The first argument to String's replaceAll method must be a valid regular expression. To signify this, a @Regex qualifier is added to the type of the first argument as shown below.

```java
public String replaceAll(@Regex String regex, String replacement) {
    ...
}
```

Figure 1 shows the basic subtyping relationship of the type system's qualifiers. The @Regex qualifier is a subtype of the @Unqualified qualifier so that a type qualified with @Regex can be used where a type qualified with @Unqualified is required.

Following are some additional examples of the @Regex qualifier in use:

- @Regex String regex = "(reg)(ex)"; // regular expression literal
- @Regex String regex2 = regex + "(re)"; // concatenation of two regular expressions
- @Regex String nonRegex = "non ("; // compile-time error

The third line shows an example of a compile-time error. The String literal is not a valid regular expression, therefore it is an @Unqualified String and cannot be assigned to a @Regex String.

2.2 @Regex for Valid Capturing Group Usage

The @Regex qualifier can also take a parameter of at least the number of capturing groups in the qualified regular expression.

The full subtyping relationship of the type system's qualifiers is shown in Figure 2.

Here is an example of the @Regex qualifier used with the capturing groups parameter:

- @Regex(2) String regex = "(reg)(ex)"; // regular expression literal with 2 capturing groups
- @Regex(3) String regex2 = regex + "(re)"; // concatenation of @Regex(2) and @Regex(1) Strings
- @Regex(3) String invalidGroupCount = "(regex)"; // compile-time error
Figure 2: The full subtyping relationship of the type system’s qualifiers. Because the parameter to a @Regex qualifier is at least the number of capturing groups in a regular expression, @Regex qualifiers with more capturing groups are subtypes of @Regex annotations with fewer capturing groups.

The @Regex qualifier with capturing groups can also be used on Pattern and Matcher types to qualify the number of capturing groups in the type.

2.2.1 @Regex (groups) Qualifier

Moving down the qualifier hierarchy are @Regex annotations with increasing group counts. Any regular expression that has at least 3 groups also has at least 1 group, so the regular expression has both the types @Regex(3) and @Regex(1). Further, the @Regex(0) qualifier in Figure 2 is shorthand for the @Regex qualifier in Figure 1 because all syntactically valid regular expression have at least zero capturing groups.

Here is an example use of the @Regex qualifier with groups that demonstrates why this organization of the type hierarchy is beneficial (from Lucene/solr/src/java/org/apache/solr/core/Config.java, a program we used in the case studies described in Section 4):

```java
parsedMatchVersion = parsedMatchVersion.replaceFirst("(\d)\.(\d)\$, "LUCENE_$1$2");
```

String’s replaceFirst method requires a @Regex String as the first parameter. In the example above, the regular expression passed to replaceFirst has two capturing groups so it is a @Regex(2). Even though the types do not match here, this is still valid because @Regex(2) is a subtype of @Regex.

2.2.2 @Unqualified Qualifier

The @Unqualified qualifier encompasses all types, including types that are syntactically valid regular expressions and types that are not. Because @Unqualified is at the top of the type hierarchy, a type qualified with any of this type system’s qualifiers can be used where an @Unqualified type is required.

2.2.3 @RegexBottom Qualifier

The only use of the @RegexBottom qualifier is for the null literal. Since @RegexBottom is a subtype of all other qualifiers in the type system, this allows the null literal to be assignable to any type qualified with any other qualifier from this type system. The null literal is considered a @Regex for the convenience of initializing types qualified with
This is acceptable because this type system prevents usage of invalid regular expression syntax, not null pointer exceptions.

3 Regex Checker

The type system described in Section 2 is implemented as a pluggable type system called the Regex Checker in the Checker Framework [3]. This section describes important details of how the type system is implemented and used.

3.1 Defaulting

As a convenience to developers the Regex Checker adds default qualifiers to types, preventing the need to explicitly qualify each type.

3.1.1 String Literals

It would be irritating to always have to add a @Regex annotation to any String literal is a syntactically valid regular expression. Because of this, the Regex Checker implicitly adds a @Regex annotation with the correct capturing group count to any String literal that is a syntactically valid regular expression. In addition, the @Regex annotation on a String is propagated from the String to the Pattern produced by compiling the String and to the Matcher produced by matching the Pattern. An example of this is shown here:

```java
String regex = "(\.* \[0-9\]+)"; // regular expression literal, @Regex(2) added to type
Pattern p = Pattern.compile(regex); // @Regex(2) propagated from String to Pattern
Matcher m = p.match("number 447"); // @Regex(2) propagated from Pattern to Matcher
if (m.matches()) {
    System.out.println(m.group(2)); // output: 447
    System.out.println(m.group(3)); // compile-time error, m only has 2 groups
}
```

Because the @Regex annotation is propagated to the Matcher, the Regex Checker can verify that calls to the group method are passed valid capturing group numbers.

The Regex Checker also implicitly adds a @Regex annotations with the correct capturing group count to concatenation of syntactically valid regular expressions. When two @Regex Strings are concatenated, the result is a @Regex String with a group count that is the sum of the initial two group counts. An example is show here:

```java
String regex = "(\.* \[0-9\]+)"; // regular expression literal with two groups, @Regex(2) added to type
String regex2 = regex + "\.*"; // concatenation of @Regex(2) and @Regex(1), @Regex(3) added to type
Pattern p = Pattern.compile(regex2); // @Regex(3) propagated from String to Pattern
Matcher m = p.match("number 447a"); // @Regex(3) propagated from Pattern to Matcher
if (m.matches()) {
    System.out.println(m.group(3)); // output: a
    System.out.println(m.group(4)); // compile-time error, m only has 3 groups
}
```

The Regex Checker implicitly qualifies the type of any String literal that is not a syntactically valid regular expression with @Unqualified.

3.1.2 Libraries

When using libraries, where the source code is not available, the Regex Checker will implicitly qualify all String types exposed from the libraries with @Unqualified. This allows the use of both @Regex Strings and @Unqualified Strings when interfacing with these binary classes. The one exception to this is explained in Section 3.4.

3.2 Validating User Input

A good developer who reads a regular expression from an external source (such as a file or user input) must validate it before using it. The Regex Checker supports this use case, and in fact ensures that the developer does not forget it, preventing potential run time errors.
The Regex Checker supplies some helper routines to aid the developer in doing the necessary input checking. These routines are recognized by the Regex Checker and cause it to be more precise in its type analysis by avoiding false positives.

The helper routines are as follows:

**RegexUtil.isRegex(String)** returns `true` if its argument is a valid regular expression.

**RegexUtil.regexError(String)** returns a `String` error message if its argument is not a valid regular expression, or `null` if its argument is a valid regular expression.

**RegexUtil.regexException(String)** returns the `PatternSyntaxException` `Pattern.compile(String)` throws when compiling an invalid regular expression. It returns `null` if its argument is a valid regular expression.

These are demonstrated in the example below.

```java
String regex = getRegexFromUser();
if (!RegexUtil.isRegex(regex)) {
    throw new Error("error parsing " + regex +": " + RegexUtil.regexError(regex));
}
// regex is a syntactically valid regular expression so @Regex is implicitly added to regex
Pattern p = Pattern.compile(regex);
```

There are also `isRegex`, `regexError` and `regexException` methods that take an additional capturing group count parameter. The `isRegex` method verifies that the first parameter is a syntactically valid regular expression with at least the given number of groups. The `regexError` and `regexException` methods return a `String` error or `PatternSyntaxException`, respectively, detailing why the given `String` is not a syntactically valid regular expression with at least the given number of capturing groups. An example of these methods is shown here:

```java
String regex = getRegexFromUser();
if (!RegexUtil.isRegex(regex, 2)) {
    throw new Error("error parsing " + regex +": " + RegexUtil.regexError(regex, 2));
}
// regex is a syntactically valid regular expression with at least 2 groups
// so @Regex(2) is implicitly added to regex
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher("number 447");
if (m.matches()) {
    System.out.println(m.group(2));
}
```

### 3.3 Enhancements

This section describes enhancements made to the Regex Checker to decrease the number of errors reported on legal regular expression usage.

#### 3.3.1 Pattern.LITERAL Flag

So far, we have assumed that `Pattern.compile` requires a regular expression as its argument. But, when invoked with `Pattern.LITERAL` as its second argument, as in:

```java
Pattern.compile(anyString, Pattern.LITERAL);
```

the `Pattern.compile` method escapes the first argument before it is compiled so it does not have to be a syntactically valid regular expression.

To fix this, the Regex Checker has a special case for the `Pattern.compile` method when called with the `Pattern.LITERAL` flag allowing the `Pattern.compile` method to take any `String`.

#### 3.3.2 @Regex char

The Regex Checker also supports the use of `chars` as regular expressions. For example, the `char ‘c’` is a syntactically valid regular expression and will be implicitly annotated as such by the Regex Checker. Concatenation works the same way with `chars` as it does with `Strings`.
### 3.4 Library Annotations

We added 18 `@Regex` annotations to library methods that take or return regular expressions. These are methods that provide an interface to the regular expression system provided by Java. Some of the more interesting annotated library methods are shown below.

```java
class Pattern {
    public static Pattern compile(@Regex String regex);
    public static @Regex String quote(String s); // escapes and returns argument
}

class String {
    public boolean matches(@Regex String regex);
    public String replaceAll(@Regex String regex, String replacement);
    public String[] split(@Regex String regex);
}
```

By annotating these methods, the Regex Checker can verify correct usage of these methods.

### 4 Case Studies

#### 4.1 Methodology

To evaluate the Regex Checker we wanted to find large, actively developed Java projects that used regular expressions. We chose Daikon [4], Plume-lib [6], Apache Lucene [2], Apache Chukwa [1] and YaCy [7].

Our methodology was to repeatedly run the Regex Checker, examine its warnings, and add `@Regex` annotations where necessary, until the project was completely annotated. Along the way, we sometimes had to correct errors that the Regex Checker had discovered, or suppress false positive warnings.

#### 4.2 Results

Figure 3 shows results for the five case studies we annotated. Bugs are described in detail in Section 4.3. False positives are described in detail in Section 4.4.

#### 4.3 Bugs

We found 50 bugs in our case study applications. The types of bugs can be split into two different groups: regular expression syntax bugs (Section 4.3.1) and capturing group bugs (Section 4.3.2).

##### 4.3.1 Syntax Bugs

We found 50 bugs where input was not validated or escaped before use. These could lead to run time errors from invalid regular expression syntax. Here is an example of one the bugs we saw from Plume-lib/java/src/plume/Lookup.java:
Figure 4: Summary of the calls to Matcher methods that take a group parameter.

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Calls</th>
<th>Total calls with int literal parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>172</td>
<td>171</td>
</tr>
<tr>
<td>start</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>end</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>TOTALS</td>
<td>183</td>
<td>178</td>
</tr>
</tbody>
</table>

For this example the Regex Checker reports an error when `keyword` is compiled since `keyword` is built from input and cannot be verified as a syntactically valid regular expression. By speaking with the developer, we found that the intention in this example is to match the literal argument. The solution to this error is to quote `keyword` before embedding it in a regular expression.

This is another example with a similar problem from Chukwa/trunk/src/main/java/org/apache/hadoop/chukwa/extraction/demux/processor/mapper/TsProcessor.java:

```java
datePattern = jobConf.get("TsProcessor.time.regex." + chunk.getDataType(), datePattern);
```

In this example, a regular expression is retrieved from a configuration file. Because this regular expression is input it needs to be verified to have valid syntax before use. Additionally, after this regular expression is compiled the first capturing group is extracted from text that matches this regular expression. That shows that this input does need to be a valid regular expression and should not be quoted. The solution to this bug is as follows:

```java
datePattern = jobConf.get("TsProcessor.time.regex." + chunk.getDataType(), datePattern);
if (!RegexUtil.isRegex(datePattern, 1)) { // verify datePattern is a legal regular expression with at least 1 group
    throw new Error("error parsing regex " + datePattern + ": " + RegexUtil.regexError(datePattern, 1));
}
```

In all 50 bugs we found, a `PatternSyntaxException` would be thrown with invalid regular expression input, printing out a stack trace. This is not very useful to a user of the program. Requiring the developer to validate input also requires the developer to produce a more useful response for invalid input, potentially highlighting a specific value in a config file that is an invalid regular expression. This makes it much easier for the user to fix the error in the regular expression.

### 4.3.2 Capturing Group Bugs

We did not find any bugs from the invalid use of capturing groups, but will still discuss the utility of this validation and potential reasons why we found no bugs.

For the Regex Checker to verify the correct parameter to a method that takes a capturing group number the parameter must be a compile-time constant.

Figure 4 shows how many calls to Matcher's methods are passed `int` literals. Of the 183 total calls, only 5 are not passed `int` literals. 3 of the 5 calls that were not passed `int` literals were in deprecated code, so we added a trivial warning suppression. This confirms that the Regex Checker will be able to verify correct capturing group usage for most calls.

One potential reason why no capturing group bugs were found is because these are simple bugs to find during development. It is likely that a developer will get a run time error from this kind of bug while testing so it is likely
4.4 False Positives

A false positive is when the Regex Checker incorrectly reports an error in regular expression syntax or capturing group usage. This is when the developer is able to confirm valid regular expression usage, but the Regex Checker is not.

Figure 5 shows a table of the occurrences of false positives in the case studies. These false positives can be split up into three groups: unavoidable (Section 4.4.1), fixable (Section 4.4.2) and unfixable (Section 4.4.3).

4.4.1 Unavoidable False Positives

The unavoidable false positives are “Tests whether s is a regex”. These are false positives that are intrinsic to the type system. These false positives represent declarations of methods like `isRegex` or `regexError` where it is necessary to compile an arbitrary `String` to determine if it is a valid regular expression or what the error is with the regular expression. To eliminate these false positives a warning suppression is added to the declarations of these methods.

4.4.2 Fixable False Positives

The fixable false positives are “Flow-sensitivity”, “Partial regex concatenation”, “StringBuilder/StringBuffer/char[]”, “line.separator property is a legal regex” and “Deprecated code”.

The “Flow-sensitivity” false positives are ones where we had to cast input to a `@Regex String` before using it. These casts are all immediately after verifying that a regular expression is valid with the `isRegex` method. A new Dataflow Framework is in development and will support this use case. All of these false positives can be removed once the new Dataflow Framework is complete.

The “line.separator property is a legal regex” false positive is for calls to `System.getProperty("line.separator")` to get the system path separator. System path separators are always valid regular expression so this call will always return a legal regular expression. For now, this warning has to be suppressed. A fix for this false positive would be to add a special case to the Regex Checker for the “line.separator” property, but this has not yet been implemented.

The “Deprecated code” false positive is because the Regex Checker reported errors in a section of deprecated code. Since this code was deprecated we suppressed the warning and moved on.

The “Partial regex concatenation” false positive is demonstrated by the following example, taken from Lucene/solr/src/java/org/apache/solr/spelling/SpellingQueryConverter.java.

```java
final static @Regex String NMTOKEN;

... // NMTOKEN assigned here

class final static @Regex String PATTERN = "(?:(?!\|) + NMTOKEN + ":|\d+))\]|\p{L}_\-\d+);";
```

Figure 5: False positive statistics. Unavoidable false positives are ones that are intrinsic to the type system. Fixable false positives are ones that we can fix by enhancing the Regex Checker but have not yet. Unfixable false positives are ones that require more information about the context of the code than the Regex Checker has to determine correctness.

that these bugs were discovered before a release. However, the Regex Checker would be useful during development in finding these bugs at compile time, instead of waiting for tests to find these bugs at run time.
The last line declares and initializes a @Regex String PATTERN that is the result of concatenating a String literal with a @Regex String and another String literal. However, these String literals are not valid regular expressions because the parentheses are not balanced. But, when the expression is look at as a whole, the three opening parentheses in the first String literal are closed by the three closing parentheses in the second String literal, so this actually does produce a valid regular expression. However, the Regex Checker is unable to determine that this String is a valid regular expression because it can only determine that concatenation of @Regex Strings produce a regular expression.

One potential solution for partial regex concatenation would be to add another qualifier, called @PartialRegex. The @PartialRegex qualifier would be added implicitly to a String literal that is not syntactically valid regular expression and carry along the partial regular expression as a parameter to the annotation. When the partial regular expression is concatenated with another partial regular expression, the Regex Checker could look at the parameter to each of the annotations, concatenate those two together and see if the result is a syntactically valid regular expression. If so the concatenation evaluates to a @Regex String, otherwise the concatenation evaluates to another @PartialRegex String with the concatenation of the two initial partial regular expressions as a parameter.

The “StringBuilder/StringBuffer/char[]” false positive is demonstrated by the following example from Daikon/asm/-Opendan.java.

```java
StringBuilder b = new StringBuilder();
b.append("eax");
b.append("|ebx");
b.append("|ecx");
b.append("|edx");
...
registers32BitRegExp = Pattern.compile(b.toString());
```

In this example the Regex Checker issues an error on the call to Pattern.compile because the Regex Checker is unable to determine that b.toString() returns a @Regex String. This is because the Regex Checker cannot track the state of the StringBuilder (this and the following also apply for a StringBuffer or char[]). To do this, the Regex Checker would need to have a typestate system to modify the type of the StringBuilder as it is mutated. This is not something that can be done easily with the current Checker Framework, but once the new Dataflow Framework is in place this should be much simpler. For now, solving this problem requires suppressing the warning.

### 4.4.3 Unfixable False Positives

The unfixable false positives are “Substring operation” and “Verified variable group count”. These are cases that require information that cannot be represented in the type system.

The following is an example of a “Substring operation” false positive from Chukwa/trunk/src/main/java/org/apache/hadoop/chukwa/database/Macro.java:

```java
public String computeMacro(@Regex String macro) {
    ...
    // first param must be @Regex String
    table = dbc.findTableName(  
        macro.substring(macro.indexOf("\") + 1,  
        macro.indexOf("\") + 1),
        start,
        end);

    In the above example, a @Regex String is taken as a parameter. This parameter is used in the first parameter to the dbc.findTableName method, which requires a @Regex String as its first parameter. The computeMacro method is always called with a String in the following format:
    
    "group_avg(user_util)"

    With input formatted like this, the substring call (which extracts the text in the parentheses of the input) will always be a regular expression. In this case, the developer is able to confirm that the first parameter to the dbc.findTableName call will be a valid regular expression, but the Regex Checker is unable to reach the same conclusion. The developer knows the format of the String, but this is information that cannot be communicated to the Regex Checker. The only way to fix this false positive is to suppress the warning.

    The “Verified variable group count” false positives are ones where both a regular expression and capturing group number were taken as input. The regular expression was verified to be syntactically valid and have at least the required
number of capturing groups. However, there is no way to communicate to the type system that a variable represents a valid capturing group number for a regular expression String. The solution to this problem is suppressing the warning.

5 Discussion

The following are some positives and negatives of the Regex Checker.

5.1 Ease of Use

I was not familiar with any of the five case study applications when I began working on the case studies. Yet, adding annotations to the code was simple. The Regex Checker highlighted all of the areas I had to get started annotating, then all I had to do was work backwards from those points. That meant that I had to understand very little of the code, only the part of that code that had to do with regular expressions that the Regex Checker could not already figure out on its own.

This is promising. If it was easy for me to annotate code with very little understanding of the applications, it would be easier for a developer working on the project who understands the code to add annotations.

Unfortunately, I was unable to get timing numbers on how long it took to annotated these case studies. My best guess is that the first case study I annotated took on the order of two to three hours. After this, the case studies began to take less and less time. The overhead to add annotations to a whole project is low. The overhead to add annotations to a project as it is being developed is even lower since the developer should have an idea of which Strings need to be valid regular expressions. All the developer would have to do is add annotations to these types.

5.2 Improved Documentation

The @Regex annotations improve documentation in code. Instead of hoping a developer comments that a parameter must be a valid regular expression with three capturing groups, the Regex Checker requires that this annotation be on the type.

5.3 Cleaner Code

Some of the more difficult sections of code to annotate were ugly sections of code, such as embedding regular expression Strings in other Strings and then extracting them out later, or storing regular expression Strings in the same data structures as other Strings. My hope is that if a developer was attempting to annotate this then the developer would realize that the code is difficult to annotate so it should be refactored and cleaned up.

Further, when validating input the Regex Checker encourages developers to use the isRegex method. While the same task could be accomplished by catching the PatternSyntaxException, this requires knowledge that Pattern.compile throws a PatternSyntaxException when passed a syntactically invalid regular expression. By explicitly checking if input is a valid regular expression the intent is more clear.

5.4 False Positives

The Regex Checker does report a number of false positives that require the developer to suppress warnings. This can be frustrating for developers but we plan to improve the Regex Checker and eliminate some false positives like flow-sensitivity, partial regex concatenation and StringBuilder/StringBuffer.

6 Related Work

FindBugs has three bug checks for regular expressions:

1. Invalid regular expression syntax. This reports statements that will throw PatternSyntaxExceptions when executed.
2. `File.separator` cannot be used as a regular expression. This reports times when `File.separator` is used as a regular expression. This is illegal because the Windows file separator is a backslash, which is not a valid regular expression.

3. `"."` used for regular expression. This reports times where a single period is a regular expression. It may have been intended to escape the period rather than have a regular expression that matches anything.

Item 1 (invalid regular expression syntax) in FindBugs is only able to determine invalid regular expression syntax in compile time constants. The Regex Checker extends this and is able to determine correct regular expression syntax with variables, parameters and input. Furthermore, every error we found was of this variety.

Item 2 (`File.separator` cannot be used as a regular expression) is implemented in the Regex Checker, but by default, not as a special case.

Item 3 (`"."` used for regular expression) is not implemented in the Regex Checker. The Regex Checker does not validate the use of the correct expression usage. Rather, it just determines the syntactic validity of regular expressions.

7 Conclusion

The Regular Expression Type System found 50 bugs in five Java applications. It is easy to use because it requires low overhead and little understanding of the code being annotated. The type system improves documentation and encourages developers to use cleaner code.

References