Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions

Muzammil Shahbaz*, Phil McMinn, Mark Stevenson

University of Sheffield, UK

Abstract

Classic approaches to automatic input data generation are usually driven by the goal of obtaining program coverage and the need to solve or find solutions to path constraints to achieve this. As inputs are generated with respect to the structure of the code, they can be ineffective, difficult for humans to read, and unsuitable for testing missing implementation. Furthermore, these approaches have known limitations when handling constraints that involve operations with string data types.

This paper presents a novel approach for generating string test data for string validation routines, by harnessing the Internet. The technique uses program identifiers to construct web search queries for regular expressions that validate the format of a string type (such as an email address). It then performs further web searches for strings that match the regular expressions, producing examples of test cases that are both valid and realistic. Following this, our technique mutates the regular expressions to drive the search for invalid strings, and the production of test inputs that should be rejected by the validation routine.

The paper presents the results of an empirical study evaluating our approach. The study was conducted on 24 string input validation routines collected from 10 open source projects. While dynamic symbolic execution and search-based testing approaches were only able to generate a very low number of values successfully, our approach generated values with an accuracy of 34% on average for the case of valid strings, and 99% on average for the case of invalid strings. Furthermore, whereas dynamic symbolic execution and search-based testing approaches were only capable of detecting faults in 8 routines, our approach detected faults in 17 out of the 19 validation routines known to contain implementation errors.

Keywords: Test data generation, web searches, regular expressions

*Corresponding author

Email addresses: muzammil.shahbaz@gmail.com (Muzammil Shahbaz), p.mcminn@sheffield.ac.uk (Phil McMinn), m.stevenson@dcs.shef.ac.uk (Mark Stevenson)

1. Introduction

There has been much work in the literature of late devoted to automated test input generation [1], however handling string input types remains a challenging task [2, 3]. This is due to the inherent complexity of real-world data that is naturally encoded as strings—e.g., dates of different formats, banking codes, registration numbers, etc.—which have very large input domains, and consequently, involve a huge search space for test data generation.

To date, a number of approaches have been investigated, including symbolic execution [4] and search-based testing [5]. However, since they are driven by the need to obtain high levels of structural coverage—e.g., branch coverage—the test suited produced have the following deficiencies:

Low test effectiveness: Test suites that achieve high coverage are not necessarily effective, particularly where string data types are concerned, since it is possible to cover program structure without generating any inputs similar to those actually supplied in practice when the software is deployed. For example, the Java method below—`isMonth` (from the open source project *TMG*[1])—validates whether a given string input is a month name, i.e. '`January`' to '`December`'. However, the method can be fully "covered" without an actual month name being supplied, through execution of the method with an arbitrary (possibly empty) string:

```
// declaration of a set
Set months = new HashSet();
// initialisation
months.add("January");
...
months.add("December");
```

```
// method body
boolean isMonth(String month) {
    return months.contains(month);
}
```

Difficult-to-read test inputs: Automatically generated test inputs tend to be hard for human testers to read and understand. Since a formal specification is frequently unavailable, a tester often assumes the role of a *human oracle* [6]—that is, manually determining whether the right outputs were produced for the generated inputs. This task is made harder when test inputs are not easy to read [7]. For instance, it is harder for a human to distinguish between arbitrary email addresses such as '`"b\2@3#t"@s3t`' (valid) and '`"b\2@3#"t@s3t`' (invalid[2]), than '`bob@mail.com`' (valid) and '`bob@mailcom.`' (invalid[3]).

Inability to test missing implementation: Roughly 35% of program implementation errors result from missing functionality [8]. One way to detect such errors is to test programs with invalid values. However, automated techniques guided by program structures cannot produce such values due to missing logic

[1] `http://tmgerman.sf.net`

[2] quotes must be separated by '.', or they must be the outer characters of the local-part.

[3] '.' must not be the last character in the domain-part.

paths, or so-called "sins of omissions". For example, an email validation program in the open source project *LGOL*[4], misses a check for rejecting values containing more than one '@' symbol. Hence, the address '`i.am@invalid@for.sure.com`' passes the validation test.

This paper builds upon our previous work [3] that proposed an approach for generating valid values using tailored web searches and regular expressions (which were also sought dynamically from web sources). The web searches are conducted through web queries that are generated using information extracted from program identifiers following the application of natural language processing techniques.

In this paper, we extend the approach for generating invalid values using regular expression mutation, and further define a testing procedure using the generated valid and invalid values to find potential program errors—in particular missing logic paths. The paper furnishes an empirical study conducted on 24 string input validation routines collected from 10 open source projects. The results of the study show that the approach was capable of finding a number of valid and invalid values for different string types, with an average accuracy of approximately 34% for valid values and 99% for invalid values. The approach also detected that 17 out of the 19 routines contained implementation errors when using the values generated. The approach has been analysed against two contemporary test data generation tools implementing dynamic symbolic execution [9] and search-based testing [2, 10] techniques. These tools were only able to generate a very low number of values, and detected errors in only 8 routines.

The rest of the paper is organized as follows. Section 2 provides an overview of the proposed approach. Sections 3–7 explain different steps of our approach in detail. Section 8 then reports the empirical and comparative study of the approach, while Section 9 discusses potential inherent threats to validity in our evaluation. Section 10 details related work, and finally Section 11 concludes the paper with directions for future work.

2. Overview of the Approach

Before describing the approach, we define the following terms used throughout the paper:

- **Validation routine:** a program function that takes a value as an input and returns true if it is accepted, or false if it is rejected.
- **Valid/Invalid value:** a generated value that is assumed to be accepted/rejected by a validation routine.

[4] `http://lgol.sf.net/`

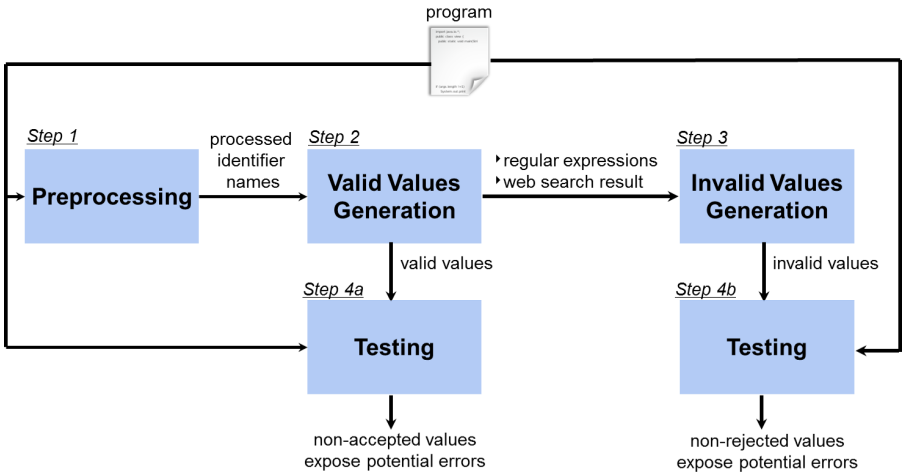

Figure 1: Overview of our approach

Figure 1 presents an overview of our approach, which consists of four steps. Step 1 performs preprocessing on the identifier names found in the source code of a Java class, in order to gain information on the types of string that are involved. Step 1 is explained in the next section, Section 3.

Step 2 takes the processed identifier names as an input from Step 1 and searches for suitable regular expressions using web searches. It then generates web queries and performs them using a search engine, collecting the resulting web pages. Finally, it extracts valid values from those pages collected using regular expressions matches. Step 2 is explained in Section 4.

Step 3 takes regular expressions and web pages as inputs from Step 2 and generates invalid values. It first mutates the regular expressions, and then uses them to extract string matches from the web pages. Step 3 is explained in Section 6.

Finally, Step 4 performs testing using the valid values (in Step 4a) and the invalid values (in Step 4b) generated from Steps 1–3. The valid values that are not accepted, and the invalid values that are not rejected by the program are reported as exposing potential errors. Step 4 is explained in Section 7.

3. Preprocessing

Step 1 in Figure 1, the preprocessing step, takes a Java class source file and extracts all the identifiers contained in it. Each identifier name is then refined

using natural language processing techniques into a form that may help infer the format of the string data corresponding to the parameters of the routines in the original source.

Figure 2 shows a snippet of a Java class that will be used as an example of the identifier names in the subsequent sections. The goal in this example is to generate values for the string parameter named "`an_Email_Address_Str`". The rest of this section provides details on each part of this step.

```
// class body
class Validator {
  // method body
  boolean parseEmail(String an_Email_Address_Str) {
    ...
  }
}
```

Figure 2: Snippet of a Java class used as an example for identifier refinement

3.1. Extracting Identifier Names

The key idea behind our approach is to extract important information from program identifiers, and then use them to construct web query search queries that are likely to return search results (web pages) containing examples of valid values for those identifiers. For example, an identifier name including the word '`email`' is a strong indicator that its value is expected to be an email address. A web query string containing '`email`' can be used to retrieve example email addresses from the Internet.

For a given Java method, our approach extracts 1) the identifier of the class containing the method, 2) the identifier of the method itself, and 3) the string type parameter identifier. These identifiers are of interest because they may give clues about the types of string values that the method expects. In the example of Figure 2, the following identifier names are extracted:

Identifier Type	Class	Method	String Parameter
Extracted Name	`Validator`	`parseEmail`	`an_Email_Address_Str`

We use the Java 6.0 Compiler API in order to extract the information about the identifiers in the source code under analysis.

3.2. Processing Identifier Names

Once identifiers are extracted, their names are processed using the following techniques.

3.2.1. Tokenisation

Identifier names are often formed from concatenations of words and need to be split into separate words (or tokens) before they can be used as part of natural language web queries. Conventions for concatenating token to form identifiers are to use camel casing and/or underscores [11]. Identifiers are split into tokens by replacing underscores with whitespace and/or adding a whitespace before each sequence of upper case letters. Finally, all characters are converted to lowercase. For example, "`parseEmail`" becomes "`parse email`" and "`an_Email_Address_Str`" becomes "`an email address str`".

3.2.2. Part-of-Speech (PoS) Tagging

Identifier names often contain words that are articles ("a", "and", "the") and prepositions ("to", "at" etc.), which are not useful when included in web queries. In addition, method names often contain verbs as a prefix to describe the action they are intended to perform. For example, "`parseEmail`" parses an email address. The key information for the input value is contained in the noun "`email`", rather than the verb "`parse`". The part-of-speech category in the identifier names can be identified by applying a Part-of-Speech (PoS) tagger [12], and thereby removing any non-noun tokens. Thus, "`parse email`" becomes "`email`" and "`an email address str`" becomes "`email address str`".

The implementation uses Stanford Log-linear PoS Tagger Version 3.0.4 [13]. The default options are used, including the pre-trained bidirectional model [14] for the English language.

3.2.3. Removal of Non-Words

Identifier names may include non-words which can reduce the quality of search results. Therefore, names are filtered with the intention of producing web queries that consist solely of meaningful words. This is performed by removing any word in the processed identifier name that is not a dictionary word. For example, "`email address str`" becomes "`email address`", since "`str`" is not a dictionary word.

In order to implement this sub-step, we use an edited version of the SCOWL word lists [15] that consist of 573,120 English language words and common abbreviations. A modified version of the Jazzy tool [16] is used to carry out the word lookup.

The above three techniques are applied in the order presented. In the example of Figure 2, the extracted identifier names are processed as follows:

Extracted Identifier Name	`Validator`	`parseEmail`	`an_Email_Address_Str`
Tokenisation	`validator`	`parse email`	`an email address str`
	↓	↓	↓
PoS Tagging	`validator`	`email`	`email address str`
	↓	↓	↓
Removal of Non-Words	`<empty string>`	`email`	`email address`
Final Processed Identifier Name	`validator`	`email`	`email address`

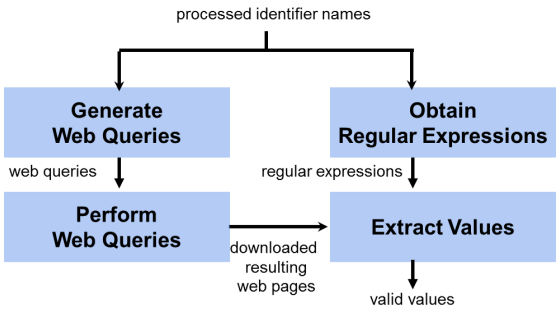

Figure 3: Valid Values Generation

It is important to note that for an identifier name that is processed as an empty string, the application of the last one or two processing techniques is reversed. For example, the identifier name "`Validator`" becomes empty string after removal of non-words, since it is not a dictionary word. In this case, the last processing step will be reversed to avoid generating an empty string, as shown in "Final Processed Identifier Name" above for the identifier name "`Validator`".

4. Valid Values Generation

Figure 3 presents a schematic view of the process of generating valid values (Step 2 in Figure 1). It takes the processed identifier names of class, method and string parameters as inputs from the previous step. Then, each identifier name is used to 1) generate web queries, and 2) obtain regular expressions dynamically. The web queries are performed using a search engine. Finally, the textual contents of the URLs produced in the search results are downloaded, from which the valid values are extracted using the regular expressions matches. The rest of this section provides details on each part of this step.

4.1. Generating Web Queries

This section explains the generation of web query strings, which include different combinations of pluralization and quotation styles. These combinations of queries increase the number of sources to acquire target string types, and are explained in the following.

4.1.1. Pluralization

The approach generates pluralised versions of the processed identifier names by pluralising the last word according to the grammar rules. For example, '`email address`' becomes '`email addresses`'.

The implementation uses the *Inflector* utility of the ModeShape library [17] for grammar-based pluralisation.

4.1.2. Quoting

If an identifier name consists of multiple words, the approach generates query strings with and without quotes. The latter style is a general search to target web pages that contain the words in the query string, whereas the former style forces the search engine to target web pages that contain all words in the query string appearing together as a phrase.

Using different combinations of pluralization and quotation style, a maximum of four query strings are generated for each identifier name. In the example of Figure 2, the following eight query strings are generated from the processed identifier names of class, method and method parameter identifiers.

Processed identifier name	`validator`	`email`	`email address`
Query strings	`validator`, `validators`	`email`, `emails`	`email address`, `email addresses`, `"email address"`, `"email addresses"`

4.2. Performing Web Queries

The query strings generated from class, method and string parameter names are used to perform web queries to generate values for the target string parameter. In the example of Figure 2, all eight query strings generated (as shown in Section 4.1) are applied one by one to a search engine to generate values for the string parameter with the identifier '`an_Email_Address_Str`'.

Web queries were performed using Microsoft's Bing [18]—the only major Internet search engine providing free API access at the time this research was conducted. The implementation used version 2.0 of Bing's API [19] to retrieve search results in the form of a list of URLs. The localisation is set to '`en-GB`', source type to '`web`' to obtain only textual contents (excluding images, videos etc.) and non-HTML content types (e.g., PDF, MS Word files) are ignored. The API limits the results to the first 50 URLs for each query.

The web pages given by the URLs are then downloaded using JSoup [20]—an open source parser for extracting and manipulating data from HTML pages.

4.3. Obtaining Regular Expressions

Regular expressions are obtained dynamically in two ways:
1) RegExLib Search, and 2) Web Search. These methods are explained in the following.

4.3.1. RegExLib

RegExLib [21] is an online regular expression library that currently indexes around 4000 expressions for different types (e.g., email, URL, postcode) and scientific notations. It provides an interface to search for regular expressions using keywords. The search can also be filtered using ratings on the expressions that have been given by the library users according to their quality. The ratings are given on the scale of 1 (poor) to 5 (the best).

The approach accesses the search interface by generating HTTP requests for a processed identifier name on-the-fly. The regular expressions are then collected from the search results (in the HTTP response). Only the best-rated expressions are selected in the results (i.e., those rated 5).

4.3.2. Web Search

When RegExLib is unable to produce regular expressions, the approach performs a simple web search using Bing's API [19]. The search query string is formulated by prefixing the processed identifier names with the string '`regular expression`. For example, the regular expressions for '`email address`' are searched for by applying the query string '`"regular expression" email address`'.

The web pages linked from the first 50 URLs returned in the search results are then downloaded and parsed using JSoup [20]. The regular expressions are extracted from the web pages by identifying any string that starts with `^` and ends with `$` symbols. These regular expressions are further filtered with the *Pattern.compile()* method from the Java Class Library, which helps in discarding any malformed expressions in the search results.

4.4. Extracting Data

Finally, the collated regular expressions are used to extract data from the downloaded web pages. For each web page, the HTML tags are stripped out and the regular expressions are matched on the remaining text one by one. All unique matches are then identified as potential valid string values.

4.5. Complexity Analysis

The complexity of generating valid values is bounded by the following parameters. Let b be the search time for one web query and x be the search time to obtain regular expressions for an identifier. Then, the maximum web search time for an identifier using q queries is $q(b + x)$.

Furthermore, let r be the number of regular expressions, d be the download time for a web page, e be the regular expression matching time to extract values from a web page. Then, the maximum time to extract values from u URLs using q queries is $qu(d + re)$.

From the above two equations, the worst case complexity to find valid values for one identifier using q queries is

$$q \times (b + x + u(d + re)) \tag{1}$$

There are no absolute limits on b and x and it is assumed that the search engine always responds in a finite time. There is also no limit on the number of regular expressions. The limits on d, e and u can be set at runtime but the default time for d per web page is set to 3 seconds by JSoup [20] and the default limit on u per query is set to 50 by the Bing API V2 [19].

5. Regular Expression Mutation

The mutation of regular expressions is performed as a part of the process of generating invalid values (Step 3 in Figure 1). There exist a number of operators (e.g., character substitution, BitFlip [22]) for string mutation but they cannot be applied directly to regular expressions because of their special characteristics. For example, a string can be mutated by replacing characters randomly or changing the length by adding or removing characters arbitrarily. However, a regular expression has a well-defined format that can also be represented as a deterministic finite automaton. Therefore, adding or removing characters randomly can distort a regular expression or make it invalid. Moreover, regular expressions have a special character set and operators that have semantic meanings, requiring a careful treatment before they can be altered.

This section presents an algorithm for regular expressions mutation that has been designed to produce invalid string values in our approach.

5.1. Preliminaries

A regular expression s is a finite string of elements defined over an alphabet Σ, i.e., $s \in \Sigma^+$. Let C be a set of alphabet classes such that for each class $c \in C$, $c \subseteq \Sigma$ and for two classes $c, c' \in C$, where $c \neq c'$, $c \cap c' = \emptyset$, i.e., all classes are disjoint. Table 1 shows the list of alphabet classes developed for the algorithm. The list is not exhaustive[5] but includes important characters for an effective mutation.

Let the length of a string or a set s be denoted by $|s|$. Also, each element in the string s is uniquely positioned in the order of its occurrence. For example, if $s = e_1\ e_2\ \cdots\ e_k$ of length $k = |s|$, where each $e_i \in \Sigma$, $1 \leq i \leq k$, then e_1 is indexed at the 1st, e_2 is indexed at the 2nd, ..., and e_k is indexed at the kth positions. Also assume that all positions have equal probability of being selected randomly.

Let $s_\downarrow$ be a projection on s that is obtained by erasing all elements in $\Sigma \setminus \bigcup_{c \in C} c$. That is, $s_\downarrow$ contains elements only belonging to the alphabet classes. For example, if $s = (a)*$, then $s_\downarrow = a*$, as per the alphabet classes in Table 1.

A string s is said to be *mutable* iff $|s_\downarrow| > 0$, i.e, if its projection is a non-empty string. Also, δ denotes a set consisting of range elements: **,** and **-**, i.e, each element in δ specifies a range in a regular expression.

Let r be the *mutation rate*, in the range $[0.0, 1.0]$. Then, the number of possible mutations in s is defined by $r \times |s_\downarrow|$. For example, if $r = 0.5$ and $|s_\downarrow| = 2$, then the number of possible mutations in s is $0.5 \times 2 = 1$. Finally, let n be a positive integer denoting the maximum *number of mutants* (or mutated s) to be generated.

[5] A complete reference for regular expressions and related information can be found at `http://www.regular-expressions.info/`

Table 1: Character classes defined for the regular expression mutation algorithm

Class	Elements	Description
c_1	`{\w, \W, \d, \D, \s, \S, \ }`	special characters for alphanumeric characters & spaces
c_2	`{a, ... , z, A, ... , Z}`	alphabetic characters
c_3	`{0, ... , 9}`	numeric characters
c_4	`{+, *, ?}`	quantification characters
c_5	`{?=, ?<=, ?!, ?<!}`	look-around characters

5.2. Mutation Algorithm

Given a regular expression string s, mutation rate r and the maximum number of mutants n, the regular expression mutation algorithm generates a set of at most n mutants. The algorithm is given in Figure 4 and explained in the following.

Line 1 initializes an empty set called *mutants*, which will hold the mutants generated during the run of the algorithm. Line 2 performs a check whether s is mutable. If s is not mutable, the algorithm returns an empty set.

From lines 3 to 11, the algorithm iterates to generate n mutants. Line 4 initializes an empty set called *positions*, which will hold the positions in s where mutations are performed during the generation of one mutant.

From lines 5 to 9, the algorithm iterates to generate one mutant. A mutant is generated when the number of mutations at different random positions reaches to $r \times |s_{\downarrow}|$. A mutation in the mutant is performed from lines 6 to 9 as follows.

Line 6 selects an element h in s at a random position p with two conditions: 1) h is also in the projection of s, and 2) p has not been selected previously. The first condition ensures that the element is selected from the defined classes. The second condition ensures that each time a different position is selected for mutation for the same mutant. Note that the position can be selected again but for another mutant, as the set *positions* is reinitialized at line 4.

Line 7 selects a replacement of h through the function $replacement(s, h)$. This function returns a random element h' ensuring that h and h' are different but belonging to the same class, and the selection of h' is under a valid range if h is a start/end of the range.

Once h' is selected, h is replaced with h' in s at the position p at line 8. Line 9 adds the position p to the set *positions* to ensure that p will not be selected again for the same mutant.

When the loop at line 5 terminates, i.e., a mutant is generated, line 10 adds the mutant into the set, and line 11 decrements the required number of mutants.

When the loop at line 3 runs for n times, line 12 returns the set of mutants.

5.3. Characteristics

Given a syntactically valid regular expression s, mutation rate r (in the range $[0.0, 1.0]$) and the maximum number of mutants $n > 0$, the algorithm generates at most n mutants, and for each mutant x, $x \neq s, |x| = |s|$, i.e., x and s are syntactically different but having the same length. All elements at distinct positions given by $r \times |s_{\downarrow}|$ are mutated in x. Furthermore, x is a syntactically valid regular expression. This is due to the correctness of the

Input: String s (where s is a syntactically valid regular expression)
Input: Mutation rate r (in the range [0.0, 1.0])
Input: Number of mutants n (where $n > 0$)
Output: Set of at most n mutants

1 $mutants = \emptyset$; `// set of mutants initially empty`
`// check if` s `is mutable`
2 **if** $|s_{\downarrow}| > 0$ **then**
  `/* generation of` n `mutants */`
3 **while** $n > 0$ **do**
    `// set of positions in` s `where elements will be mutated`
4    $positions = \emptyset$;
    `/* generation of one mutant */`
    `// repeat to obtain the required number of mutations`
5   **while** $|positions| \leq (r \times |s_{\downarrow}|)$ **do**
6   select an element $h \in s$ at position p randomly such that
        1. $h \in s_{\downarrow}$
        2. $p \notin positions$
7      $h' = replacement(s, p, h)$; `// selects a replacement of` h
8   replace h with h' in s at p ;
9   add p to $positions$; `// records` p
10   add s to $mutated$; `//` s `has now been mutated`
11    $n = n - 1$; `// decrement the number of mutants required`
12 **return** $mutated$

`/* Get replacement of element` h `at position` p `in string` s `*/`
`function replacement(s, p, h)`
select h' randomly from a class $c \in C$ using the four rules:

R1: $h \neq h'$; `//` h `and` h' `are different`

R2: $h, h' \in c, c \in C$; `//` h `and` h' `belong to the same class`

R3: If h is followed by a range element from δ, which is followed by an element $\rho \in c$, i.e., h is the start of a range that ends at ρ and

- if $h, \rho \in c_2$ then `'a'`$\leq h' \leq \rho$
- if $h, \rho \in c_3$ then `'0'`$\leq h' \leq \rho$

R4: If h is preceded by a range element from δ, which is preceded by an element $\rho \in c$, i.e., h is the end of a range that starts from ρ and

- if $h, \rho \in c_2$ then $\rho \leq h' \leq$`'z'`
- if $h, \rho \in c_3$ then $\rho \leq h' \leq$`'9'`

return h'

Figure 4: The regular expression mutation algorithm

replacement(s, p, h) function that replaces the element h at position p in s with a different element such that s remains syntactically valid. This is ensured by the application of four rules. Rules $R1$ and $R2$ ensure that h and h' are different but belong to the same class. Since each class has more than one character, it is always possible to select different elements from the same class. Moreover, $R2$ ensures that replacing h by h' does not affect the syntax since they belong to the same class. For instance, replacing quantification characters, e.g. '`*`' by '`+`', does not invalidate the syntax. The only case is the range where appropriate selection of h' is required to ensure syntax validity. This is ensured by rules $R3$ and $R4$. $R3$ ensures that if h is the start of a range that ends at an element ρ, then h' must be between 'a' and ρ, if h is an element representing alphabetic characters (class c_2), or between '0' and ρ, if h is an element representing numeric characters (class c_3). $R4$ ensures that if h is the end of a range that starts at an element ρ, then h' must be between ρ and 'z', if h is an element representing alphabetic characters (class c_2); or between ρ and '9', if h is an element representing numeric characters (class c_3).

5.4. Example

Consider a regular expression s, given as '`(a\d|b[c-f])+`', which accepts the occurrence of one or more strings consisting of 'a' followed by any element from '0' to '9', or 'b' followed by any element from 'c' to 'f'.

The mutation algorithm is demonstrated on this example using the list of alphabet classes given in Table 1. Suppose the mutation rate $r = 0.5$ and the number of mutants $n = 1$.

According to Table 1, the projection of s is obtained as $s_{\downarrow}$ = '`a\dbcf+`', which consists of 6 elements. The regular expression is mutable since the length of the projected string is greater than zero. Hence, the condition at line 2 will be evaluated to true. Since $n = 1$, the outer loop at line 3 will be executed only once. The inner loop at line 5 will run for $0.5 \times 6 = 3$ iterations and one mutated s will be generated. Each of the three iterations is illustrated in the following.

Iteration 1: In this iteration, *positions* $= \emptyset$, and each element in s is positioned in the following way.

$$s = \overbrace{(}^{1}\ \overbrace{a}^{2}\ \overbrace{\backslash d}^{3}\ \overbrace{|}^{4}\ \overbrace{b}^{5}\ \overbrace{[}^{6}\ \overbrace{c}^{7}\ \overbrace{-}^{8}\ \overbrace{f}^{9}\ \overbrace{]}^{10}\ \overbrace{)}^{11}\ \overbrace{+}^{12}$$

Then, at line 6, let a random element $h = b$ be selected at position $p = 5$. At line 7, a random replacement is selected as $h' = a$. At line 8, b is replaced with a at the position 5. At line 9, 5 is added to *positions*.

Iteration 2: In this iteration, *positions* $= \{5\}$, and each element in the string is positioned as follows.

$$s = \overbrace{(}^{1}\ \overbrace{a}^{2}\ \overbrace{\backslash d}^{3}\ \overbrace{|}^{4}\ \overbrace{a}^{5}\ \overbrace{[}^{6}\ \overbrace{c}^{7}\ \overbrace{-}^{8}\ \overbrace{f}^{9}\ \overbrace{]}^{10}\ \overbrace{)}^{11}\ \overbrace{+}^{12}$$

Then at line 6, let a random element $h = f$ be selected at position $p = 9$. At line 7, a random replacement is selected as $h' = d$. Note that h is preceded by a range element, which is preceded by c, therefore h' is selected randomly from the range $[c, z]$. At line 8, f is replaced with d at the position 9. At line 9, 9 is added to *positions*.

Iteration 3: In this iteration, $positions = \{5, 9\}$, and each element in the string is positioned as follows.

$$s = \overbrace{(}^{1}\ \overbrace{a}^{2}\ \overbrace{\backslash d}^{3}\ \overbrace{|}^{4}\ \overbrace{a}^{5}\ \overbrace{[}^{6}\ \overbrace{c}^{7}\ \overbrace{-}^{8}\ \overbrace{d}^{9}\ \overbrace{]}^{10}\ \overbrace{)}^{11}\ \overbrace{+}^{12}$$

At line 6, let a random element $h = \backslash d$ be selected at position $p = 3$. At line 7, a random replacement is selected as $h' = \backslash D$. At line 8, $\backslash d$ is replaced with $\backslash D$ at the position 3. At line 9, 3 is added to *positions*.

After the third iteration, $positions = \{5, 9, 3\}$, and hence the size is equal to the number of mutations required, i.e., 3. The loop at line 5 terminates. At line 10, the following mutated string is added to the set *mutated*: $(a\backslash D|a[c-d])+$.

At line 11, the number of mutants required is decremented, i.e., $n = 0$. Hence, the loop at line 3 terminates. At line 12, *mutants* is returned.

6. Invalid Values Generation

Figure 5 presents a schematic view of the process of generating invalid values (i.e., Step 3 in Figure 1). It takes regular expressions and web search results as inputs from the valid values generation step. Then, the regular expressions are mutated, and invalid values are produced either by extracting matches from the pages retrieved by the web search results from Step 2, or by generating them randomly using the mutated regular expressions, if no matches are found. The rest of the section provides details on each part of this step.

6.1. Mutating Regular Expressions

The regular expressions are mutated one by one using the mutation algorithm explained in Section 5. A unique set of mutated regular expressions is obtained such that the string representations of any two mutated regular expressions in the set must be different—i.e., the two mutated regular expressions are syntactically different.

Depending upon the length of the regular expression and the rate of mutation in the algorithm, it is possible that a generated mutated regular expression still accepts valid values. Therefore, all mutated regular expressions are passed through a sanity check such that no mutated regular expression accepts a valid value in the set of valid values generated in the previous step (i.e. Step 2 in Figure 1). Otherwise, such a regular expression is discarded.

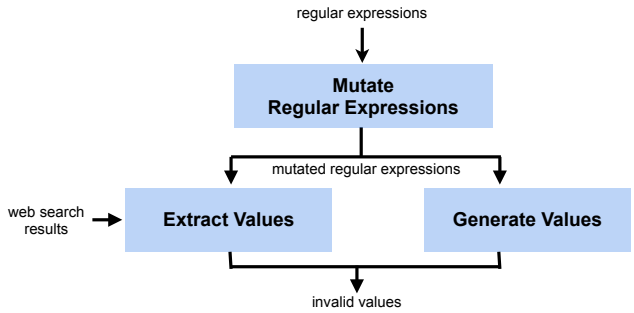

Figure 5: Invalid Values Generation

6.2. Extracting and Generating Values

All mutated regular expressions are applied to the textual contents of the web pages one by one. All unique matches are then identified as potential invalid string values.

If there are no matches found, the invalid values are generated automatically from the mutated regular expressions. Only one string is generated per regular expression. The implementation uses the Xeger library [23] that generates a random string from a given regular expression using an automaton approach.

6.3. Complexity Analysis

The complexity of generating invalid values is bounded by the following parameters. Let m be the mutation time for a regular expression, then the maximum time to mutate r regular expressions is rm.

Furthermore, let e be the regular expression matching time to extract values from a web page, u be the number of web pages, and g be the value generation time from a regular expression, then the maximum time to extract/generate values from r' mutated regular expressions is $r'(ue + g)$.

From the above two equations, the worst case complexity to find invalid values for one identifier is

$$rm + r'(ue + g) \tag{2}$$

The limit on m and r' depends upon the mutation rate and the maximum number of mutants respectively, which are inputs to the mutation algorithm (cf. Section 5.2). Similarly, the maximum time for e and g can be set at runtime.

7. Testing Procedure

The last step (i.e., Step 4 in Figure 1) tests the validation routine on the valid and invalid values produced in the previous steps (as explained in Sections 4 and 6 respectively). This is done in two parts (Step 4a and Step 4b).

Step 4a tests the routine on the valid values. Any *valid* value that is *not accepted* by the routine is reported as a value exposing a potential error.

Conversely, Step 4b tests the routine on the invalid values. Any *invalid* value that is *not rejected* by the routine is reported as a value exposing a potential error.

8. Empirical Evaluation

8.1. Research Questions

The evaluation study has been performed in the light of the following research questions.

RQ1. *Does the use of regular expressions and web queries formulated by the knowledge extracted from the program identifiers generate valid values? If yes, what is the precision?*

It seems intuitively plausible that regular expressions can be used to help identify valid values. However, this question analyses the feasibility of the approach in practice, measuring the average precision of valid values found for the case studies.

RQ2. *Does the use of mutated regular expressions result in producing invalid values? If yes, what is the precision?*

Similar to *RQ1*, this question concerns the use of regular expression mutation to obtain invalid values. Does mutating the regular expressions that were used to extract valid values result in producing invalid values?

RQ3. *Does the use of valid and invalid values generated by the approach reveal program errors?*

This question aims to evaluate the fault-finding capabilities of our techiques.

RQ4. *Is it feasible to run web searches and mutation in terms of computation time?*

The question concerns the practical limits of the approach. Is it computationally expensive to find valid and invalid values using web searches and regular expression mutation?

RQ5. *How effective is the approach compared to the other test data generation techniques for strings?*

It is important to study the approach in view of the other test data generation techniques for string types. Does the approach outperform the other techniques on average? If yes, by how much?

8.2. Case Studies

The research questions have been addressed on the case studies drawn from 10 open source Java projects. They include validation routines that are integrated in interactive applications to check inputs entered by end users. These

Table 2: Details of the case studies with class, method and parameter names

Project	Class Name	Method Name	Parameter Name	String data type validated
Chemeval	CASNumber	isValid	casNumber	CAS registry numbers
Conzilla	MIMEType	MIMEType	ntype	MIME types
	PathURN	PathURN	nuri	Path URNs
	ResourceURL	ResourceURL	nuri	Resource URLs
	URI	URI	nuri	URIs
	URN	URN	nurn	URNs
Efisto	Util	parse_ddmmyyyy_Date	date_string	Dates in format 'dd.MM.yyyy'
		parseDate	date_string	Dates in format 'EEE, dd MMM yyyy HH:mm:ss zzz'
GSV05	TimeChecker	TimeChecker	time	24 hour format
JXPFW	CLocale	toLocale	locale	POSIX locale identifiers
	InternationalBank AccountNumber	checkBasicBankAccount Number	bban	Bank Identifier Codes (BICs)
		isValidIBAN	iban	International Bank Account Numbers (IBANs)
		checkCountry	country	ISO 3166 country codes
LGOL	DateFormatValidator	isValid	str	Dates in format 'dd/MM/yyyy'
	NumericValidator	isValid	str	Strings that represent Integers
	PostCodeValidator	isValid	str	UK postcodes
Open Symphony	Validator	checkEmail	email	Email Address
		checkSsn	ssn	US Social Security Number (SSNs)
PuzzleBazar	Validation	validateEmail	text	Email Address
TMG	Isbn	Isbn	isbn	International Standard Book Numbers (ISBNs)
	Month	isMonth	month	Month names
	Year	Year	year	Four digit year
WIFE	BIC	BIC	bic	Bank Identifier Codes (BICs)
	IBAN	IBAN	iban	International Bank Account Numbers (IBANs)

routines perform relatively complex operations on strings, for which generating valid and invalid values is a challenging task, and as such are ideal for the evaluation study.

There were 24 different string input validation routines selected from 20 Java classes, comprised of 2,833 lines of code. Many of these routines were non-monolithic programs, i.e., there existed several calls to sub-routines. Table 2 provides the list of case studies, along with the names of class, method and string parameters used in the experiments. The parameters are of the `java.lang.String` type. The specific details of each case study are provided in the following.

Chemeval ***(chemeval.sf.net)*** is a chemical evaluation framework to assist hazard assessment in a molecular structure. One class was selected that validates unique identifiers, called Chemical Abstracts Service (CAS) numbers, which are assigned to every chemical substance described in the open scientific literature. A CAS Number is separated by hyphens into three parts, the first consisting of up to 7 digits, the second consisting of 2 digits, and the third consisting of a single digit serving as a checksum. CAS numbers begin at "`50-0-0`", the number for *formaldehyde*, and end at "`1346599-09-4`", the number for *naphthalenol*.

Conzilla ***(www.conzilla.org)*** is a knowledge management tool that is de-

signed to allow users to peruse related concepts in a browser interface. Six classes were selected, including one that validates MIME types, while the other five are responsible for validating different types of URIs.

Efisto ***(efisto.sf.net)*** is a tool for web file sharing. One class was selected that validates two types of Date formats.

GSV05 ***(gsv05.sf.net)*** is a J2ME application for mobile attendance recording. One class was selected that validates 24 hour time format supplied as strings.

JXPFW ***(jxpw.sf.net)*** stands for 'Java eXPerience FrameWork', a utility library used in commercial applications. Two classes were selected, which validate POSIX locale identifiers, Bank Identifier Codes (BICs) and International Banking Account Numbers (IBANs).

LGOL ***(lgol.sf.net)*** is a framework for building Java applications for local governments in the UK. Three classes were selected, which validate a date format, integer numbers and UK postcodes.

OpenSymphony ***(www.opensymphony.com)*** is a web development framework. One class was selected for validating email addresses and US Social Security Numbers (SSNs).

PuzzleBazar ***(code.google.com/p/puzzlebazar)*** is a web-based system for developing puzzles. One class was selected to validate email addresses.

TMG ***(tmgerman.sf.net)*** stands for "Text Mining for German documents", and performs text processing tasks. Four classes were selected that validate International Standard Book Numbers (ISBNs), month names and four-digit year.

WIFE ***(wife.sf.net)*** is a framework for parsing, writing and processing messages between international banks. Two classes were selected, involving the validation of BICs and IBANs.

8.3. Experimental Settings

The following parameter settings were used for the experiments:

#	Parameter	Value
1	Mutation rate per regular expression (cf. Sec. 5.1)	1.0
2	Maximum number of mutants per regular expression (cf. Sec. 5.1)	50
3	Maximum regular expression matching time per web page (cf. Sec. 4.5, 6.3)	5 seconds
4	Maximum number of URLs per query (cf. Sec. 4.5)	50
5	Maximum download time per web page (cf. Sec. 4.5)	3 seconds

Parameter #1 was set to the maximum possible mutation rate in order to mutate all "mutable" characters in a regular expression. The rationale was to have maximum chance of obtaining invalid values from the collated regular expressions. Parameter #2 was set to generate a reasonable number of mutants,

such that it would not be so small as to fail to produce invalid values, and not too large that it would make the experiments time-infeasible. Parameter #3 was set to an arbitrary time limit for a regular expression to match values in a web page. Parameters #4 and #5 were set according to the default settings in the Bing API V2 [19] and the JSoup library [20] respectively.

All experiments were conducted on a dedicated machine running OS X 10.6.8, equipped with 2.6 GHz Intel Core 2 Duo and 4 GB RAM, on a shared Ethernet network having a theoretical speed of 100 Mbit/s. The remaining of the section details the empirical evaluation and addresses the research questions.

8.4. Evaluation Strategy

In order to verify the results of the experiments while answering the research questions, a set of oracles was implemented for each string type. These oracles were consulted to analyse the performance of the presented approach (and other approaches for comparison). Specifically, they were used to furnish the accuracy figures, e.g., the percentage of valid/invalid values generated by the approach, and the number of true/false positives received during the testing procedure.

The types of oracles for each string type are given in Table 3. Where possible, an official source was used. For example, the Chemical Abstracts Service [24] has defined a format for issuing CAS numbers and provided a regular expression for their validation. Thus, the oracle for CAS number (Chemeval) was implemented by using the regular expression. For a few string types, an exhaustive list of all possible values was used as the oracle. For example, the *ISO 3166-1 alpha-2* standard has defined a fixed list of all country codes. Thus, the oracle for 2 letter country code (JXPFW) was implemented by using the list. In most cases, the *Apache Commons Validator* API [25] was used as an oracle. For the rest, different programs acquired from 3rd-party sources were used as oracles. In this case, all results were verified manually.

8.5. Answers to Research Questions

RQ1. *Does the use of regular expressions and web queries formulated by the knowledge extracted from the program identifiers generate valid values? If yes, what is the precision?*

To answer this question, the identifier names in each string type were pre-processed to obtain regular expressions from the web. Then, web queries were performed to collect web pages from which valid values were extracted using the regular expressions.

The approach was also studied for comparison with the simple approach of randomly generating synthetic strings directly from the collated regular expressions. For this experiment, 50 random strings were generated from each regular expression using Xeger [23].

The values generated by both approaches were inputted to the oracles (Table 3), and the percentage of the accepted values was computed for each string type. Table 4 shows the number of regular expressions collated dynamically,

Table 3: Types of oracles used to analyse potential errors

The column 'Oracle Type' lists the types of oracles for each string type that has been implemented to evaluate the approach while answering the research questions. The *Apache Commons Validator* API [25] was used as an oracle in most cases. For others, 'Official' means that oracles have been derived from an official source, 'Exhaustive' means that oracles have enumerated all possible values, '3rd party' means that oracles have been acquired from an external source. For the '3rd party' type, the results have been verified manually.

Project	Class	Oracle Type
Chemeval	CAS number	Official [24]
Conzilla	MIME type	Exhaustive [26]
	Path URN	Official [27]
	Resource URL	3rd party
	URI	3rd party
	URN	3rd party
Efisto	Date (dd.MM.yyyy)	Apache Commons Validator [25]
	Date (EEE...)	Apache Commons Validator [25]
GSV05	24 hour time	Apache Commons Validator [25]
JXPFW	BBAN	Exhaustive (all country formats)
	POSIX locale identifier	3rd party
	2 letter country code	Exhaustive (all country formats)
	IBAN	Exhaustive (all country formats) + 3rd party checksum validator
LGOL	Date (dd/mm/yyyy)	Apache Commons Validator [25]
	Integer	Apache Commons Validator [25]
	UK Postcode	Official [28]
OpenSymphony	Email address	Apache Commons Validator [25]
	SSN	3rd party
PuzzleBazar	Email address	Apache Commons Validator [25]
TMG	ISBN	Apache Commons Validator [25]
	Month	Exhaustive (all month names)
	Year	Apache Commons Validator [25]
WIFE	BIC	3rd party
	IBAN	Exhaustive (all country formats) + 3rd party checksum validator

the number of total values generated and the percentage for the 24 string types in both approaches.

The values generated by the random generation approach included only five cases where the values were accepted by the oracles, with the average precision of 8.23%. Whereas, the values generated by the proposed approach were accepted by the oracles in all but three cases, with the average precision of 33.8%. These three cases are explained in the following.

The first was Path URN (Conzilla). No regular expressions were generated by RegExLib for this type. The web search produced five regular expressions were found but no data could be extracted from the search results. Consequently, no valid values could be produced for this case.

The second case was Resource URL (Conzilla). No regular expressions were generated by RegExLib for this type either. However, many expressions were collected using the web search method and these produced several values. These values were mainly related to the Internet resource references (such as URLs). However, none of the values generated corresponded to the required URL format, i.e. a string prefixed with "`res://`". Therefore, the count for valid values remained zero.

The last case was 'Date (EEE, dd MMM yyyy HH:mm:ss zzz)' (Efisto). In

Table 4: Analysis of valid values for each case study using the approach

'Total regex' denotes the number of regular expressions collated for each string type. 'Random Gen.' states the result of the experiments for the approach of generating synthetic strings directly from the regular expressions. 'Web Search' states the results of the proposed approach. 'Total Values' states the total number of generated values and 'Valid Values' is the percentage of valid values accepted by the respective oracles for each string type.

Project	**String Type**	**Total regex**	**Random Gen.**		**Web Search**	
			Total Values	**Valid Values**	**Total Values**	**Valid Values**
Chemeval	CASNumber	1	50	0.0%	12,753	95.5%
Conzilla	MIMEType	121	6,050	0.0%	708	0.1%
	PathURN	5	250	0.0%	0	n/a
	ResourceURL	21	1,050	0.0%	8,234	0.0%
	URI	20	910	50.0%	19,796	2.3%
	URN	15	687	0.0%	26,140	0.1%
Efisto	Date (dd.MM.yyyy)	12	600	0.0%	19,768	0.1%
	Date (EEE, $\cdots$)	1	48	0.0%	5,007	0.0%
GSV05	24 hour time	11	457	10.94%	3,365	7.0%
JXPFW	BBAN	1	50	0.0%	30	33.3%
	POSIX locale identifier	38	1,900	0.0%	13,559	1.7%
	2 letter country code	23	1,150	0.0%	18,972	0.1%
	IBAN	1	50	0.0%	58	82.8%
LGOL	Date (dd/mm/yyyy)	117	5,844	0.0%	650	7.7%
	Integer	23	1,150	3.13%	39,697	4.4%
	UK Postcode	22	1,100	0.0%	1,032	100.0%
Open-Symphony	Email Address	2	100	0.0%	5,237	57.7%
	SSN	3	150	0.0%	250	100.0%
PuzzleBazar	Email Address	49	2,444	0.0%	42,935	0.1%
TMG	Isbn	15	634	0.0%	1,117	81.2%
	Month	2	10	100.0%	1,760	0.7%
	Year	24	260	0.0%	1,339	51.7%
WIFE	BIC	24	144	33.33%	103	100.0%
	IBAN	9	337	0.0%	67	83.6%
	Average			**8.23%**		**33.8%**

this case, RegExLib generated a number of regular expressions, however, none corresponded to this precise date format and no valid values were produced.

The study of the experiments of the two approaches has shown that the approach of random generation was only successful in cases where a precise regular expression for the required string type was available. For instance, a regular expression for Month (TMG) had the names of all the months specified[6]. Hence, it was easy to produce a month name from a random walk. On the other hand, it was hard to produce a string that requires checksums or other logical encoding by a random walk on a regular expression. The proposed approach of extracting web values produced more valid strings on average in the experiments. In some cases, 100% of the generated values were found to be valid, for example MIMEType (Conzilla), UK Postcode (LGOL) and BIC (WIFE). There were a few cases where the precision fell below 1%. There were various reasons for such performance which are described below.

Inappropriate regular expressions

[6]Snippet of the regular expression: `(Jan(uary)?|Feb(ruary)?|...|Nov(ember)?|Dec(ember)?)`

In some cases, the collated regular expressions were not suitable for the required formats. This was mainly due to the RegExLib search method generating expressions that are not related to the search query. For example, the search query "`URN`" generates expressions related to date formats, XML tags and postal addresses. Thus, a large set of values generated by these expressions could not be validated as an URN.

Low informativeness in identifier names
There are cases where a particular format is required for the values to be valid but the identifier names did not provide enough information for the web queries to generate the required format. For example, the cases related to date and time formats (Efisto, GSV05, LGOL) and ISO country codes (JXPFW) produced a large set of values but few were regarded as valid due to the specific formats required.

Misguided search due to a general context
There are cases where the identifier names represent a general context, thereby causing large numbers of values to be generated. For example only 0.1% of the values for Email Address in the PuzzleBazar project are valid. The main reason is the identifier names used in this project ("`Validation`", "`text`") are not particularly related to email addresses. On the other hand the identifier names for Email Address in the OpenSymphony project are more suitable search terms, leading to 57.7% of values being valid.

RQ2. *Does the use of mutated regular expressions result in producing invalid values? If yes, what is the precision?*

To answer *RQ2*, the experiments for generating invalid values were conducted using the regular expressions and the web search results obtained in the previous experiments of generating valid values (i.e., in answering *RQ1*). Moreover, the mutation rate was set to 1.0 and the maximum number of mutants for each regular expression was 50 for the mutation algorithm.

The experiments were repeated 10 times for each string type in order to account for random variations in the mutation algorithm. In each of these 10 runs, the invalid values generated by the approach were inputted to the oracles (cf. Table 3), and the percentage of the rejected values was computed for each string type. Table 5 shows the number of mutated regular expressions, the number of total values generated from the mutated regular expressions, and records the percentage of invalid values for each string type averaged over 10 runs. As shown, a very high volume of invalid values was generated by the approach for all cases, averaging 98.8%.

RQ3. *Does the use of valid and invalid values generated by the approach reveal program errors?*

In the testing process, the generated valid and invalid values were tested on the validation routines to expose errors. The valid values which were not

Table 5: Analysis of invalid values for each case study using the approach.

The experiments were conducted using the mutation rate of 1.0 and the maximum number of mutants for each regular expression was 50. The experiments were repeated 10 times for each string type. The column 'Total mutants' denotes the number of mutated regular expressions generated for each string type in each run. 'Total Values (Av.)' states the number of invalid values extracted/generated using the mutated regular expressions, and 'Invalid Values (Av.)' is the percentage of invalid values from the total values given by the respective oracles, for each string type averaged over 10 runs.

Project	String Type	Total mutants	Total Values (Av.)	Invalid Values (Av.)
Chemeval	CASNumber	299	519.1	99.5%
Conzilla	MIMEType	6,964	363.7	100.0%
	PathURN	401	39.3	100.0%
	ResourceURL	1351	130.1	100.0%
	URI	550	17392.0	100.0%
	URN	1,311	2363.0	99.5%
Efisto	Date (dd.MM.yyyy)	509	34047.4	100.0%
	Date (EEE, $\cdots$)	50	72023.3	100.0%
GSV05	24 hour time	1,009	1552.3	94.7%
JXPFW	BBAN	16	43224.9	100.0%
	POSIX locale identifier	2,558	1657.8	99.7%
	2 letter country code	2,157	61028.8	99.8%
	IBAN	16	34147.1	99.9%
LGOL	Date (dd/mm/yyyy)	8,235	122.4	97.3%
	Integer	1,186	90426.6	99.8%
	UK Postcode	1,396	144114.1	100.0%
OpenSymphony	Email Address	98	145.0	87.3%
	SSN	150	278.9	99.9%
PuzzleBazar	Email Address	3,742	133472.6	100.0%
TMG	Isbn	1,076	27012.1	99.4%
	Month	51	940.6	100.0%
	Year	5,255	47754.6	98.5%
WIFE	BIC	246	1098.9	99.1%
	IBAN	836	820.0	97.5%
	Average		29778.1	**98.8%**

accepted, and the invalid values which were not rejected by the routines were assumed to expose potential errors.

In order to check whether these values indicate real program errors, the non-accepted values (denoted by α) and non-rejected values (denoted by β) were inputted to the respective oracles (cf. Table 3) for each string type. If the oracle accepts a value in α, or rejects a value in β, the value exposes a real error and thus labeled as a true positive. Otherwise, the error is spurious and the value is labeled as a false positive.

Out of the 24 routines, five (underlined in Table 6) contained no known errors. The remaining 19 contained errors regarding either the rejection of well-formed strings, or the acceptance of malformed strings. Some of these errors were related to subtle violations in the input format, and could easily go undetected unless they were exposed by a particular input string. For example, one of the valid values generated by the approach for 24 hour time (GSV05) was "`8:00`", which had not been accepted by the validation routine. This is due to the branching sub-condition "`minute > 0`", which should have been correctly written as "`minute >= 0`".

In case of invalid values, an example value generated by a mutated regular

Table 6: Analysis of program errors exposed by generated values

The validation routines were tested on the valid/invalid values generated by the approach. The rejected valid values and the accepted invalid values indicate potential errors in the routines. The tables below provide an analysis for the valid and the invalid values respectively. The number of distinct values exposing potential errors is stated in '# Values Exposing Errors'. The true and false positives were computed using oracles, whose numbers are stated in 'True Positives' and 'False Positives' respectively. 'Precision' denotes the precision of finding values that expose errors as per Eq. 3. 'n/a' denotes the case when precision cannot be computed due to the absence of values exposing errors. All routines contain errors except the five that are underlined. The valid values have exposed errors in 2, whereas the invalid values have exposed errors in 17 out of 19 erroneous routines.

(a) Analysis of program errors exposed by valid values

Project	**String Type**	**# Values Exposing Errors**	**True Positives**	**False Positives**	**Precision**
Chemeval	CASNumber	0	n/a	n/a	n/a
Conzilla	MIMEType	13	0	13	0.0%
	PathURN	0	n/a	n/a	n/a
	ResourceURL	8234	0	8234	0.0%
	URI	19692	0	19692	0.0%
	URN	26107	0	26107	0.0%
Efisto	Date (dd.MM.yyyy)	19734	0	19734	0.0%
	Date (EEE ⋯)	5007	0	5007	0.0%
GSV05	24 hour time	3202	33	3169	1.0%
JXPFW	BBAN	10	0	10	0.0%
	POSIX locale identifier	13125	0	13125	0.0%
	2 letter country code	18954	0	18954	0.0%
	IBAN	224	54	170	24.1%
LGOL	Date (dd/mm/yyyy)	600	0	600	0.0%
	Integer	35615	0	35615	0.0%
	UK Postcode	0	n/a	n/a	n/a
Open	Email Address	2163	0	2163	0.0%
Symphony	SSN	0	n/a	n/a	n/a
PuzzleBazar	Email Address	42892	0	42892	0.0%
TMG	Isbn	209	0	209	0.0%
	Month	1748	0	1748	0.0%
	Year	113	0	113	0.0%
WIFE	BIC	0	n/a	n/a	n/a
	IBAN	11	0	11	0.0%
	Average	8357.7	5.7	10023.6	**1.3%**

(b) Analysis of program errors exposed by invalid values averaged over 10 runs. The precision is given with the standard deviation to analyse the variation in all runs.

Project	**String Type**	**# Values Exposing Errors**	**True Positives**	**False Positives**	**Precision (St. Dev.)**
Chemeval	CASNumber	0	n/a	n/a	n/a
Conzilla	MIMEType	72.0	72.0	0.0	100.0% (0.0)
	PathURN	0.0	n/a	n/a	n/a
	ResourceURL	0.0	n/a	n/a	n/a
	URI	578.0	404.0	174.0	69.9% (0.0)
	URN	28.6	4.7	23.9	16.3% (6.1)
Efisto	Date (dd.MM.yyyy)	13.6	12.7	0.9	93.4% (7.0)
	Date (EEE ⋯)	0.0	n/a	n/a	n/a
GSV05	24 hour time	17.0	5.0	12.0	29.4% (0.0)
JXPFW	BBAN	1659.4	1655.0	4.4	99.7% (0.1)
	POSIX locale identifier	33.8	32.1	1.7	95.0% (5.8)
	2 letter country code	157.5	0.0	157.5	0.0% (0.0)
	IBAN	9.7	1.4	8.3	14.4% (7.3)
LGOL	Date (dd/mm/yyyy)	5.4	1.0	4.4	18.5% (9.2)
	Integer	145.3	0.0	145.3	0.0% (0.0)
	UK Postcode	8.3	4.0	4.3	48.2% (10.0)
Open	Email Address	101.0	80.0	21.0	79.2% (0.0)
Symphony	SSN	75.1	75.1	0.0	100.0% (0.0)
PuzzleBazar	Email Address	18.0	11.8	6.2	65.6% (14.2)
TMG	Isbn	269.8	2.2	242.0	0.9% (1.1)
	Month	0.0	n/a	n/a	n/a
	Year	1842.8	920.8	661.0	58.2% (3.1)
WIFE	BIC	67.8	60.0	7.8	88.5% (9.8)
	IBAN	32.5	6.0	26.5	18.5% (4.8)
	Average	214.0	139.5	62.5	**69.0%**

expression for URI (Conzilla) was '`dttps://41.5vYMM: /:s`', which was not rejected by the validation routine. The routine performs various checks on the format of the given URI but did not include checking of whitespaces. Similarly, an invalid value extracted from a web page by a mutated regular expression for BIC (WIFE) was "`WildBoyz`", which was not rejected either. In this case, the routine performs checks about the length of the string that must be 8 or 11 characters long and its 5th and 6th characters must be an ISO country code. The specified value has 8 characters and the 5th and 6th characters ("`Bo`") make the country code for Bolivia.

Table 6 (a) and (b) show the results of testing valid and invalid values respectively for all string types. The number of distinct values that have exposed errors are stated in the column '# Values Exposing Errors'. Note that these errors are not necessarily unique—i.e, more than one distinct values may expose the same error in the program. All these values were verified by the respective oracles for each string type and true/false positives were computed as follows.

$$precision = \frac{True\ Positives}{True\ Positives\ +\ False\ Positives} \times 100 \tag{3}$$

Table 6 (a) shows that the valid values exposed errors in two routines, i.e, 24 hour time (GSV05) and IBAN (JXPFW). However, there were many false positives, and therefore, the overall average precision is low (1.3%). One of the reasons for low precision is the fact that these routines are obtained from older projects, which have already been tested on well-formed input strings. As a result, only a few routines contained errors related to non-acceptance of valid values, which were exposed by the approach.

Table 6 (b) provides the results for testing the invalid values averaged over 10 runs of the experiments. It shows that the invalid values exposed errors in 17 out of the 19 routines known to contain faults (in each of the 10 runs). Out of the five routines for which errors were found to be error-free, there were only two cases, i.e., '2 letter country code' (JXPFW) and Integer (LGOL), where the invalid values exposed potential errors but which were actually false positives. The precision in this case is given along with the standard deviation to analyse the variations in all runs. The few cases where the variation is slightly higher is due to the identifier names that represent a general context. The highest variation is 14.2 that was found in Email Address (PuzzleBazar) that encapsulates the target concept only in the method name, i.e., "`validateEmail`". The class and parameter names, "`Validation`" and "`text`" respectively, produced regular expressions that could not extract any values from the given web pages. Consequently, the random generator (Xeger [23]) was applied to generate values that tended to deviate from one run to the other. There is only one other case, i.e, UK Postcode (LGOL) where the deviation is given in double figures. However, the deviations are much smaller than the respective averages in all cases. The overall average precision of 69% was calculated for all string types.

The accuracy of these results is subjective to the quality of oracles used in the experiments. However, these oracles were acquired from the respective

authorities (i.e., official, exhaustive, Apache Commons) where possible. For other sources, the results have been manually verified. But overall results show that the approach is capable of finding errors using valid and invalid values that were extracted or generated by regular expressions and their mutations respectively. In case of testing valid values, the precision was low; however, the approach surpassed the other testing techniques in exposing erroneous routines (see *RQ5* for details). In case of testing invalid values, high precision was obtained.

RQ4. *Is it feasible to run web searches and mutation in terms of computation time?*

The time taken by the approach for generating valid and invalid values for all string types is given in Table 7 and measured in seconds.

The time analysis for the valid values generation is given in the second, third and fourth columns according to Equation 1. The second column states the web search time and the regular expression search time for all queries. The third column states the download time and the value extraction time for all web pages, regular expressions and queries. The fourth column totals the previous two columns. The average time for generating the valid values is given as 323.08 seconds or about 5.4 minutes.

The time analysis for the invalid values generation is given in the fifth, sixth and seventh columns according to Equation 2, averaged over 10 runs of the experiments. The fifth column states the mutation time for all regular expressions. The sixth column states the value extraction time for all web pages and mutated regular expressions, plus the value generation time from the mutated regular expressions in case no values could be extracted. The seventh column totals the previous two columns. The average time for generating the invalid values is given as 694.26 seconds or about 23 minutes.

The web page downloading and value extraction processes are a little time consuming (cf. columns 3 and 6 of Table 7) but are not an overhead that is infeasible or particularly burdensome if the goal is to obtain valid and invalid values in high volumes.

RQ5. *How effective is the approach compared to the other test data generation techniques for strings?*

There are no known tools available for valid and invalid values generation in the same settings as provided in this paper. The closest are test data generation techniques that usually aim for code coverage. Two main techniques: dynamic symbolic execution [4] and search-based testing [5], were considered for comparison with the approach. For each technique, a relevant tool was obtained that could generate data for string types. Each class from the case studies was run on these tools that generated several branch-covering test cases. For each class, the tools ran for 10 times, where each run was bounded by the maximum time taken by the proposed approach for that class in any of the 10 runs (see Table 7). Following this, the test data was extracted from the test cases for all string types to analyse the percentage of valid values through the validation routines.

Table 7: Time Analysis for valid and invalid values Generation in seconds

The time analysis for valid and invalid values generation according to Equations 1 and 2 respectively. The second, third and fourth columns state the valid values generation time. The second column is the web search time and the regular expression search time for all queries, the third column is the download time and the value extraction time for all regular expressions, web pages and queries. The fourth column is the total of previous two columns. The fifth, sixth and seventh columns state the invalid values generation time. The fifth column is the mutation time for all regular expressions. The sixth column is the value extraction time for all web pages and the value generation time using all mutated regular expressions. The seventh column is the total of previous two columns.

Project / String Types	**Valid Values Generation Time (sec)**			**Invalid Values Generation Time (sec)**		
	$q(b+x)$	$qu(d+re)$	$q(b+x)+$ $qu(d+re))$	rm	$r'(ue+g)$	$rm+$ $r'(ue+g)$
Chemeval						
CASNumber	6.00	336.67	342.66	0.16	35.57	35.73
Conzilla						
MIMEType	5.71	1230.62	1236.33	11.39	787.04	798.43
PathURN	6.46	108.69	115.15	0.95	90.11	91.06
ResourceURL	6.29	236.48	242.76	2.98	127.91	130.89
URI	6.21	284.34	290.55	243.67	788.98	1032.65
URN	8.08	314.46	322.54	7.15	520.17	527.32
Efisto						
Date (dd.MM.yyyy)	9.36	30.49	39.84	124.28	2665.38	2789.66
Date (EEE $\cdots$)	10.24	29.35	0.66	3.28	3.94	
GSV05						
24 hour time	8.29	212.42	220.72	1.99	111.24	113.23
JXPFW						
BBAN	9.95	38.69	48.64	144.48	5752.11	5896.59
POSIX locale identifier	15.32	654.87	670.18	5.09	477.53	482.61
2 letter country code	10.19	60.66	70.85	32.67	250.23	282.90
IBAN	7.64	38.68	46.32	0.01	1.56	1.56
LGOL						
Date (dd/mm/yyyy)	12.25	558.83	571.08	1.53	46.50	48.03
Integer	7.42	523.69	531.11	17.30	348.68	365.98
UK Postcode	8.10	503.50	511.59	19.04	837.77	856.81
OpenSymphony						
Email Address	9.78	57.21	66.99	34.61	331.81	366.43
SSN	5.84	45.60	51.43	0.46	54.98	55.44
PuzzleBazar						
Email Address	10.92	520.94	531.86	195.87	1302.04	1497.91
TMG						
Isbn	11.05	570.65	581.70	0.97	311.80	312.78
Month	10.56	129.34	139.90	2.19	95.70	97.88
Year	8.09	531.51	539.59	53.21	385.00	438.22
WIFE						
BIC	8.65	207.64	216.29	12.98	23.82	36.80
IBAN	6.87	319.29	326.15	67.53	331.94	399.47
Average	**8.72**	**314.36**	**323.08**	**40.88**	**653.38**	**694.26**

For dynamic symbolic execution, *Symbolic PathFinder* (*SPF*) [9] was used, which performs symbolic execution of Java bytecode with model checking and constraint solving. The default options were used for the decision procedure: *Choco* [29], and for the string solving approach: *automata*. *SPF* was run until either the timeout occurred, or terminated due to the tool's internal error.

For search-based testing, an improved version of *eToc* [10], called *eToc*$^+$[2] was used. The tool performed evolutionary searches for 100 generations of randomly-generated values with a population size of 100 for each branch. For an uncovered branch, *eToc*$^+$ continued searching until there had been no improvement in the best fitness value found in the last 1000 generations, i.e., the search had stagnated. Thus, each uncovered branch received at least 100,000 fitness evaluations, possibly more, if progress did not stagnate.

The experiments with *eToc*$^+$ and *SPF* were repeated for 10 times to minimize the effect of randomness. Later, the values from the test cases generated by each tool for each class were collected for analysis. Table 8 provides the average percentage of valid values verified by using the oracles. Evidently, these tools produced a very low number of valid values on average in comparison with the proposed approach. Although these tools are primarily test case generation tools aiming to achieve a specific type of coverage, they were used as a baseline comparison to measure the effectiveness of the proposed approach for valid values generation in the following way.

Let a be an approach and s be a string type in the case studies, the effectiveness function eff of a for s is calculated as

$$eff(a, s) = \frac{\%\ of\ valid\ values\ for\ s\ given\ by\ a}{Maximum\ \%\ of\ valid\ values\ for\ s} \tag{4}$$

When the denominator is zero, i.e., no valid values were generated by any approach for a specific s, the effectiveness of all approaches is zero for that s. Table 8 shows the average effectiveness computed for all approaches and for all s. Clearly, the proposed approach has been the most effective in producing valid values on average compared to other approaches. *eToc*$^+$ generated a number of values but only 0.01% values were valid, with the average effectivness of 0.0035. *SPF* generated few values were valid, and therefore, the average effectiveness was 0. The main reason is that string constraint solving in SPF is currently work in progress[7], and does not recognize complex object types (e.g., *java.text.SimpleDateFormat*, *java.lang.Integer*), and throws exceptions during test generation.

Another comparison was performed regarding the error exposing capabilities of these approaches. The values generated by *eToc*$^+$ and *SPF* were inputted to oracles to check for disagreement with the respective validation routines. There were in total 19 routines that contained known errors. *eToc*$^+$ exposed errors in 8 routines, whereas *SPF* exposed errors only in 1 routine. In comparison, the

[7] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc`

Table 8: Comparison with other test data generation approaches

The proposed approach is compared with $eToc^+$ and *SPF* tools. The experiments with these tools were repeated for 10 times for each string type. The second column states the average percentage of valid values for each string type over 10 runs. The figure for the proposed approach has been repeated from Table 4 for this column (which was run only once). The third column states the effectiveness of the approaches calculated according to Eq. 4.

Approach	**% of Valid Values (Av.)**	**Effectiveness**
Dynamic Symbolic Execution (*SPF* [9])	0.00%	0.0000
Search-Based Testing ($eToc^+$ [10, 2])	0.01%	0.0035
Proposed Approach	33.8%	0.8750

approach presented exposed errors in 17 routines (see *RQ3*).

In conclusion, the proposed approach has been the most effective in i) generating valid values, and ii) exposing errors, in the empirical study compared to the selected tools in dynamic symbolic execution and search-based testing.

9. Threats to Validity

Threats to validity might come from how the empirical evaluation was carried out. One possible threat is the choice of case studies, which is common for any empirical analysis. Naturally, it is impossible to capture the full diversity of all possible programs. However, the case studies were obtained from different open source portals in a variety of domains that contain different levels of expressiveness. Also in some cases, more than one program in a similar domain but from different sources were considered.

A threat to validity that might come from the random mutation of regular expressions. In order to reduce the bias, 50 mutants were generated for each collated regular expression, and the procedure was repeated 10 times for each case study to generalise the results. Another threat is linked to the mutation algorithm that did not consider all possible characters in the regular expression grammar (cf. Table 1). However, the experimental results in Table 5 show that the algorithm has produced regular expressions that achieved ≈99% of invalid values and also exposed errors in the case studies. There were only three cases where a high number of false positives (> 100) was found. But this was mainly due to the generic contexts inferred from the identifier names.

Another possible threat to validity comes from the experimental setup that did not take into account all combinations of parameter settings. In this paper, default settings were considered where possible, e.g., 50 URLs for each web query, otherwise parameters were chosen intuitively, e.g., mutation rate was set to one, to mutate all "mutable" characters in a regular expression. In theory, there might exist parameter settings affecting the generalisation of the current results. To analyse this issue, more experiments are required with tunable settings to account for different combinations.

Finally, a threat to validity that might come from the time analysis (*RQ4*). Sections 4 and 6 provide the generalisation of the cost of the computational time

required by the approach for valid and invalid values generation. Moreover, the actual time taken by the approach in the evaluation study has also been reported in Table 7. In order to perform a realistic time analysis, the approach was implemented without using concurrency (e.g., multi-threading). This means that all operations: web searches, downloading, value extraction and mutation, were conducted in the sequential mode. However, memorisation was used to avoid re-performing the previously processed web queries, but this could not affect the extraction of values from the static web pages if those queries had been repeated.

10. Related Work

The input validation problem has been addressed in the classic software testing literature [30]. The earlier approaches assume the provision of grammars from which the inputs can be generated. Beizer [30] has presented approaches for syntax testing, also called grammar-based testing, where the syntax of the input is expressed in a formal specification, such as BNF (Backus-Naur Form), and its equivalent graph. Then the valid inputs are generated by "covering" the graph. Invalid values can also be generated by employing heuristics such as interchanging terminal and non-terminal symbols (in the BNF), replacing numeric with alphabetic values and generating extra delimiters in the valid values – the so-called "error-condition" rules [31].

There has been a significant progress in the input generation techniques in the later research; however, dealing with complex string formats has remained a challenge. Well-known approaches including symbolic execution [4] and search-based testing [5] have trouble executing path conditions containing string operations. Several improvements have been proposed by reducing the search exploration space [5][32], or by employing custom grammar based constraint solvers [33][34].

Contrary to the above techniques, our approach does not assume the provision of grammars but it searches for inputs on the Internet by inferring the input formats from the identifier names, which is then formalised with the help of relevant regular expressions that are also obtained dynamically. The regular expressions limit the input space for arbitrary strings, as well as produce relevant data for program testing.

Elbaum et al. [35] have proposed an approach for generating valid values based on collecting input data from previous user sessions. This technique was designed for web applications where users normally leave behind the footprints of their data. Similarly, an approach for using outputs of the existing web services as inputs to the services under test has been evaluated for service-oriented-architectures [36]. Contrary to these works, our approach searches input data from scratch without assuming existing usage data.

One key objective of the current approach is to generate invalid data. Constraint based approaches [37] can also be employed for this purpose. This is similar to generating valid values except that a constraint is inversed, e.g., returning a number less than a threshold while a greater number is expected.

This is normally practiced in symbolic execution to explore *if-else* paths of programs. Other string constraint solvers exist, e.g, HAMPI [38] that uses context-free grammar to solve strings whose lengths are predetermined. It performs a bitvector encoding of all positional shifts for each regular expression in a constraint in order to find an optimal solution. It was observed that much of that encoding work is unnecessary if the goal is to find a single string assignment as quickly as possible [39]. Recently, Java String Testing (JST) tool [40] has been proposed that enhanced the symbolic execution engine of the SPF platform [9] to tackle complex data structures. The tool implements a hybrid approach that searches solutions for string variables using both numeric and string solvers for fast convergence. Furthermore, special rules and heuristics are devised for different kinds of string constraints in Java language, which helped in achieving better coverage in an evaluation study. However, it is known that solving arbitrary constraints is an undecidable problem [41].

Fuzzing [34] has been popular in generating malformed inputs using randomized techniques. This technique is more popular in attacking system vulnerabilities such as crashes or failing assertions. An important difference with our approach is that fuzzing mutates the given valid values to generate invalid ones, whereas our approach mutates a regular expression that matches valid values to produce invalid ones. Also, fuzzing requires the provision of valid values but our approach searches for values on the web.

In search-based testing, an approach has been proposed to generate test data for the purpose of raising specified exceptions in a program [41]. This approach employs meta-heuristic techniques and uses directed search methods that aim to find optimal solutions for specific fitness functions. On the contrary, our approach does not use program structure to target exceptions, instead it uses only identifiers to extract information about input formatting.

The foundation of this approach has been laid by our previous work [2] that presented the original idea of harvesting the Internet for finding test data. There, the objective was to enhance branch coverage in search-based testing by seeding web values. But achieving high coverage does not imply generating valid and invalid data (cf. Section 1), which is aimed in the current approach. Furthermore, the current approach provides a testing framework to find out program errors which are difficult to detect unless specific data is used.

11. Conclusion

This paper has presented an approach for generating valid string values by using tailored web searches and regular expressions, which are also collated dynamically from different web sources. Furthermore, the approach generates invalid string values by using mutated regular expressions. Finally, testing is performed using the valid and invalid values to find potential program errors. The empirical study showed a number of errors were exposed in 17 out of 19 validation routines known to contain faults. These faults were related to violations in the required input format which can easily go undetected unless they are unveiled by specific test data.

One benefit of the approach is that the generated values are realistic, rather than the arbitrary-looking strings typically produced by automatic test data generation techniques. The strings generated are realistic because they are sourced from the Internet, a rich source of human-readable data. Furthermore, when there is no automated oracle, test cases utilizing such values help to reduce so-called human oracle cost [6]—that is, reducing the time and effort required of a human in interpreting the test inputs.

Acknowledgements

This work was funded by the EPSRC project RE-COST (REducing the Cost of Oracles for Software Testing, grant no. EP/I010386/1).

References

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, An orchestrated survey on automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[2] P. McMinn, M. Shahbaz, M. Stevenson, Search-based test input generation for string data types using the results of web queries, in: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, 2012, pp. 141–150.

[3] M. Shahbaz, P. McMinn, M. Stevenson, Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing, in: Proceedings of the 12th International Conference on Quality Software (QSIC), 2012, pp. 79–88.

[4] C. S. Păsăreanu, W. Visser, A survey of new trends in symbolic execution for software testing and analysis, International Journal on Software Tools for Technology Transfer 11 (4) (2009) 339–353.

[5] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: Local, global and hybrid search, IEEE Transactions on Software Engineering 36 (2010) 226–247.

[6] A. P. Mathur, Foundations of Software Testing, 1st Edition, Addison-Wesley Professional, 2008.

[7] S. Afshan, P. McMinn, M. Stevenson, Evolving readable string test inputs using a natural language model to reduce human oracle cost, in: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), 2013, pp. 352–361.

[8] R. L. Glass, Frequently forgotten fundamental facts about software engineering, IEEE Software 18 (3) (2001) 112–111.

[9] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing nasa software, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, 2008, pp. 15–26.

[10] P. Tonella, Evolutionary testing of classes, SIGSOFT Softw. Eng. Notes 29 (4) (2004) 119–128.

[11] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Improving the tokenisation of identifier names, in: ECOOP 2011–Object-Oriented Programming, Vol. 6813 of Lecture Notes in Computer Science, Springer, 2011, pp. 130–154.

[12] D. Jurafsky, J. Martin, Speech and Language Processing, 2nd Edition, Prentice Hall, 2008.

[13] K. Toutanova, D. Klein, C. D. Manning, Y. Singer, Feature-rich part-of-speech tagging with a cyclic dependency network, in: Proceedings of the North American Chapter of the Association for Computational Linguistics on Human Language Technology, Association for Computational Linguistics, 2003, pp. 173–180.

[14] L. Shen, G. Satta, A. Joshi, Guided learning for bidirectional sequence classification, in: Annual Meeting-Association for Computational Linguistics, Vol. 45, 2007, p. 760.

[15] K. Atkinson, Spell Checking Oriented Word Lists (SCOWL), `http://wordlist.sourceforge.net/`.

[16] Jazzy, `http://sourceforge.net/projects/jazzy`.

[17] ModeShape library, `http://www.jboss.org/modeshape`.

[18] Bing search engine, `http://www.bing.com`.

[19] Introducing Bing API Version 2.0, `http://www.bing.com/developers/s/APIBasics.html`.

[20] JSoup, `http://www.jsoup.org`.

[21] RegExLib: Regular Expression Library, `http://regexlib.com/`.

[22] D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Artificial Intelligence, Addison-Wesley, 1989.

[23] Xeger, A Java library for generating random text from regular expressions, `http://code.google.com/p/xeger/`.

[24] Chemical Abstracts Service, Check Digit Verification of CAS Registry Numbers, `http://www.cas.org/content/chemical-substances/checkdig`.

[25] Apache, Commons Validator, http://commons.apache.org/validator/.

[26] Internet Assigned Numbers Authority, MIME Media Types, http://www.iana.org/assignments/media-types/index.html.

[27] D. LaLiberte, M. Shapiro, The Path URN Specification, http://tools.ietf.org/html/draft-ietf-uri-urn-path-00 (1995).

[28] Cabinet Office's UK Data Standards Catalogue, Government Data Standards Catalogue (2001).

[29] N. Jussien, G. Rochart, X. Lorca, The choco constraint programming solver, Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP) (2008) 1–10.

[30] B. Beizer, Software Testing Techniques, International Thomson Computer Press, 1990.

[31] J. Hayes, J. Offutt, Input validation analysis and testing, Empirical Software Engineering 11 (4) (2006) 493–522.

[32] N. Li, T. Xie, N. Tillmann, J. de Halleux, W. Schulte, Reggae: Automated test generation for programs using complex regular expressions, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2009, pp. 515–519.

[33] R. Majumdar, R.-G. Xu, Directed test generation using symbolic grammars, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, 2007, pp. 134–143.

[34] P. Godefroid, M. Y. Levin, D. Molnar, Sage: Whitebox fuzzing for security testing, Queue 10 (1) (2012) 20:20–20:27.

[35] S. Elbaum, S. Karre, G. Rothermel, Improving web application testing with user session data, in: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, 2003, pp. 49–59.

[36] M. Bozkurt, M. Harman, Automatically generating realistic test input from web services, in: Proceedings of the 6th International Symposium on Service Oriented System Engineering (SOSE), IEEE, 2011, pp. 13–24.

[37] R. DeMilli, A. J. Offutt, Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.

[38] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, M. D. Ernst, Hampi: A solver for string constraints, in: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA), ACM, 2009, pp. 105–116.

[39] P. Hooimeijer, W. Weimer, Strsolve: solving string constraints lazily, Autom. Softw. Eng. 19 (4) (2012) 531–559.

[40] I. Ghosh, N. Shafiei, G. Li, W.-F. Chiang, Jst: An automatic test generation tool for industrial Java applications with strings, in: Proceedings of the 35th International Conference on Software Engineering (ICSE), IEEE Press, 2013, pp. 992–1001.

[41] N. Tracey, J. Clark, K. Mander, J. McDermid, Automated test data generation for exception conditions, Software - Practice and Experience 30 (1) (2000) 61–79.