Automated Discovery of Valid Test Strings from the Web using Dynamic Regular Expressions Collation and Natural Language Processing

Muzammil Shahbaz, Phil McMinn, Mark Stevenson
University of Sheffield, UK
{m.shahbaz, p.mcminn, m.stevenson}@sheffield.ac.uk

***Abstract*—Classic approaches to test input generation – such as dynamic symbolic execution and search-based testing – are commonly driven by a test adequacy criterion such as branch coverage. However, there is no guarantee that these techniques will generate meaningful and realistic inputs, particularly in the case of string test data. Also, these techniques have trouble handling path conditions involving string operations that are inherently complex in nature.**

This paper presents a novel approach of finding valid values by collating suitable regular expressions dynamically that validate the format of the string values, such as an email address. The regular expressions are found using web searches that are driven by the identifiers appearing in the program, for example a string parameter called `emailAddress`. The identifier names are processed through natural language processing techniques to tailor the web queries. Once a regular expression has been found, a secondary web search is performed for strings matching the regular expression.

An empirical study is performed on case studies involving String input validation code from 10 open source projects. Compared to other approaches, the precision of generating valid strings is significantly improved by employing regular expressions and natural language processing techniques.

***Keywords*-test data generation; string inputs; valid inputs; web queries; regular expressions; natural language processing**

I. Introduction

There has been much work in the literature of late devoted to automated test input generation, for example dynamic symbolic execution (DSE) [16], [22] and search-based testing (SBT) [17]. These approaches are driven by a test adequacy criterion such as structural coverage. However, by concentrating only on structural information, they ignore the problem of generating meaningful and realistic inputs. Of particular concern are string values. The problems are two-fold:

1) While it is important to test programs with invalid inputs, it is also important to test them with *valid* values. The input generation in DSE and SBT is mostly governed by some branch or path coverage criteria, but they tend not to generate valid values. As an example, the Java method in Figure 1, from the open source project *TMG*[1], validates a given input as a month name: `January` — `December`. However, full coverage of this method does not imply the generation of a valid input (i.e., 12 month names). This is confirmed by *EclEmma* tool V2.1.1 [30], which reported 100% coverage of this method on an empty string input. This results in a program being tested in largely unrealistic scenarios for the application concerned.

```
boolean isMonth(String month) {
    // months is a set containing month names
    return months.contains(month);
}
```

Figure 1. Example of a method whose full coverage can be achieved by an invalid String value.

2) Automatically generated test inputs are hard to read and understand by human testers. Since a formal specification is frequently unavailable, a tester often assumes the role of an oracle and thus serves as a *human oracle* [23], determining whether the right outputs were produced for the generated inputs. This task is made harder when test inputs are not meaningful or realistic. For instance, it is hard to distinguish between arbitrary email addresses: `"b\2@3#t"@s3t` *(valid)* and `"b\2@3#"t@s3t` *(invalid[2])*, but instantly-readable values: `bob@mail.com` *(valid)* and `bob@mailcom.` *(invalid[3])* are easily distinguishable.

One promising approach to obtain test values is to perform Internet searches. In our previous work [24], the idea was presented with the objective to enhance structural coverage in SBT. In that approach, identifiers in the program under test were split into constituent words which are then used to construct web queries. The queries were performed using a search engine and the web pages returned in the search results were tokenized. All unique string tokens were then used as test values for the program. While the approach was found to increase coverage in a number of cases, the *precision* of the approach – i.e., the proportion of strings that were valid for the program under test (such as a month name or a well-formed email address) – was very low.

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

[2]quotes must be dot separated, or the outer characters of the local-part.
[3]dot must not be last in the domain-part.

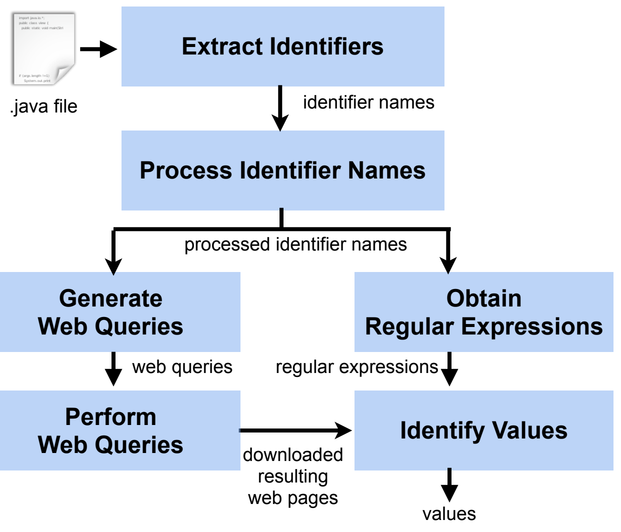

Figure 2. Overview of the Approach

This paper presents a novel approach that combines *Natural Language Processing (NLP)* techniques and dynamic *regular expressions* collation for finding valid String values on the Internet. The NLP techniques help tailoring the web queries that are made up of identifier names to collate appropriate regular expressions from the web. Then, a secondary web search is performed for strings matching the collated regular expressions. The intuition behind the approach is that regular expressions can be used to help target only valid values on the web, and NLP techniques can harness the web queries to search for such regular expressions. The contributions of this paper are therefore as follows:

1) An approach for finding valid string inputs on the web, that involves a) natural language processing of identifier names, and b) searching for appropriate regular expressions to identify valid values.
2) An empirical study on 24 open source case studies and comparison with the previous approach [24].
3) An analysis of the proposed approach against two contemporary test data generation tools in DSE and SBT techniques.

The rest of this paper is organized as follows. Section II presents the approach. Section III evaluates the approach. Section IV discusses the potential threats to validity. Section V presents the related work. Section VI concludes the paper.

II. Approach

Figure 2 presents the overview of the approach that takes a (Java) program source as input. First the information about the program identifiers is extracted from the source code. This information is refined by processing the identifier names in order to infer the format of the targeted String type. The processed identifier names are used to 1) obtain regular expressions dynamically, and 2) generate web queries. The web queries are then sent to a search engine. Finally, the textual contents of the URLs produced in the search results are downloaded, from which the values are identified using regular expressions obtained previously.

The rest of the section provides details for each part of the approach.

A. Extracting Identifiers

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

The approach considers three types of identifiers and aims to identify input values for them: 1) The method parameter identifier, 2) The method identifier, and 3) The class identifier. The names of these identifiers are an obvious source of information about the types of values they expect.

The implementation uses Java 6.0 Compiler API to extract the information about the identifiers in the Java source code.

B. Processing Identifier Names

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

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 in web queries. Conventions for concatenating strings are to separate tokens using camel casing and underscores [12]. Identifiers are split into tokens by replacing underscores with whitespace and adding a whitespace before each sequence of upper case letters. For example, "`an_Email_Address_Str`" becomes "`an email address str`" and "`parseEmailAddressStr`" becomes "`parse email address str`". Finally, all characters are converted to lowercase.

2) PoS Tagging: Identifier names often contain words such as articles ("a", "and", "the") and prepositions ("to", "at" etc.) that 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, "`parseEmailAddressStr`" is supposed to parse an email address. The key information for the input value is contained in the noun "`email address`", rather than the verb "`parse`". The part-of-speech category in the identifier names can be identified using a NLP tool called Part-of-Speech (PoS) tagger [18], and thereby removing any non-noun tokens. Thus, "`an email address str`" and "`parse email address str`" both become "`email address str`".

The implementation uses Stanford Log-linear Part-Of-Speech Tagger Version 3.0.4 [29] to perform PoS tagging. The default options are used including the pre-trained bidirectional model [26] for English language.

3) Removal of Non-Words: Identifier names may include non-words which can reduce the quality of search results. Therefore, names are filtered so that the web query should entirely consist of meaningful words. This is done 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.

The implementation uses an edited version of SCOWL word lists [8] that consists of 573,120 English language words and common abbreviations. A modified version of the Jazzy tool [3] is used to carry out the word lookup.

C. Obtaining Regular Expressions

The regular expressions for the identifiers are obtained dynamically from two ways: 1) RegExLib Search, and 2) Web Search. The methods are explained in the following.

1) RegExLib: RegExLib [6] is an online regular expression library that is currently indexing around 3300 expressions for different types (e.g., email, URL, postcode) and scientific notations. It provides an interface to search for a regular expression using keywords. The search can also be filtered by ratings on the expressions that have been given by the online users of the library according to their quality.

The approach accesses the search interface by generating HTTP requests on-the-fly for a processed identifier name. The regular expressions are then collected from the search results (in the HTTP response). Only the expressions having highest ratings in the results are selected.

2) Web Search: When RegExLib is unable to produce regular expressions, the approach performs a simple web search. The search query is formulated by prefixing the processed identifier names with the string "`regular expression`". For example, the regular expressions for "`email address`" are searched by applying the query "`"regular expression" email address`".

The first 50 URLs returned in the search results are then downloaded and parsed using the JSoup parser [4]. The regular expressions are collected by identifying any string that starts with ^ and ends with `$` symbols. These regular expressions are further filtered with the *Pattern.compile()* method from Java 6 that helps in discarding any malformed expressions in the search results.

D. Generating Web Queries

Once the regular expressions are obtained, valid values can be generated automatically, e.g., using automaton. However, the objective here is to generate not only valid values but also realistic and meaningful values. Therefore, a secondary web search is performed to identify values on the Internet matching the regular expressions.

This section explains the generation of web queries for the secondary search to identify valid values. The web queries include different versions of pluralised and quoting styles explained in the following.

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 ModeShape library [5] is used in the implementation to achieve grammar based pluralisation.

2) Quoting: The approach generates queries with or without quotes. The former style enforces the search engine to target web pages that contain all words in the query as a complete phrase. The latter style is a general search to target web pages that contain the words in the query.

In total, 4 queries are generated for each identifier name that represent all combinations of pluralisation and quoting styles. For a processed identifier name "`email address`", the generated web queries are: `email address`, `email addresses`, `"email address"`, `"email addresses"`.

E. Performing Web Queries

The web queries are performed using Microsoft's Bing [1] – the only major Internet search engine providing free API access at the time this research was conducted. The implementation uses version 2.0 of Bing's API [2] to retrieve search engine results. The localisation is set to "`en-GB`", with URL results of a non-HTML content type (e.g., PDFs, Word document files) to be ignored. The API limits the results to the first 50 web pages for each query. The textual contents of the web pages are then downloaded.

F. Identifying Values

Finally, the collated regular expressions and the downloaded web pages are used to identify valid values. 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.

III. Empirical Evaluation

The evaluation was 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 result in producing valid string values? If yes, what is the precision?*

This is a basic question about the validity of the approach. It seems intuitively plausible that regular expressions can be used to help identify valid values. However, it is important to analyse the practicality of the approach when the regular

Table I
DETAILS OF THE CASE STUDIES WITH CLASS, METHOD AND PARAMETER NAMES

Project	**Class**	**Method**	**Parameter**	**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)

expressions and the web queries are generated dynamically using the information from the program identifiers. More specifically, the question is about measuring the strength of the approach by computing the average precision of valid values from the generated values.

RQ2. *Which web search strategies are significant in finding valid string values?*

This question is related to the different processing methods for the identifier names. In other words, which NLP techniques (tokenisation, PoS-tagging and Non-Word Removal) are significant in finding valid values. For regular expressions, how effective is the approach when using only RegExLib versus using both RegExLib and the web search?

RQ3. *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?

A. Case Studies

The research questions have been addressed on the case studies drawn from 10 Java open source projects. They mainly include input validation routines that are integrated in interactive applications to check inputs entered by an end user. These routines perform relatively complex operations on strings, for which generating valid values is a challenging task, and thus are ideal for evaluation.

There were 20 Java classes selected which contained 24 different input validation routines for various types of string, comprised of 2833 lines of code. Many of these routines were non-monolithic programs, i.e., there existed several calls to sub-routines. Table I provides details of the case studies including the names of class, method and String parameters validated. The details for each case study is 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 designed to allow users to peruse related concepts in a browser interface. Six classes were selected, including one validates MIME types, whilst the other five are responsible for validating different types of URI.

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 mobile attendance recorder two-tier J2ME application. One class was selected that validates 24 hour time format supplied as strings.

Table II
STRATEGIES FOR IDENTIFYING VALID VALUES

Strategy	Regular Expression Method		NLP Method	
	RegExLib	Web Search	POS-Tagging	Remove Non-Words
S0	✗	✗	✗	✗
S1	✓	✗	✗	✗
S2	✓	✗	✓	✗
S3	✓	✗	✓	✓
S4	✓	✓	✗	✗
S5	✓	✓	✓	✗
S6	✓	✓	✓	✓

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 GWT based framework for the development of web-based puzzles. One class was selected to validate email addresses.

TMG ***(tmgerman.sf.net)*** stands for 'Text Mining for German documents' aiming for scientific/engineering text processing tasks. Four classes were selected, involving the validation of International Standard Book Numbers (ISBNs), month names and a year number format.

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

B. Answers to Research Questions

In order to answer the research questions, different strategies were explored for processing identifier names and obtaining regular expressions dynamically. In total, 7 strategies were experimented which are described in the following.

The first strategy S0 is the simplest in which no regular expressions were sought and the identifier names were processed only through tokenisation. The web queries were generated from the processed identifier names and performed using the search engine. The search results were processed by first downloading the contents of each URL and stripping out HTML tags. The remaining text was then tokenised according to whitespace, and placed into a list of unique tokens as potential String values. This strategy was proposed in our previous work [24] aiming to enhance branch coverage in the SBT technique. In this paper, this strategy is used to compare with the current approach to analyse the generation of valid values using NLP techniques and regular expressions.

The remaining strategies S1-S6 can be divided into two groups. From S1 to S3, only RegExLib was sought for obtaining regular expressions. From S4 to S6, first RegExLib was sought, then web search method was applied if no regular expressions could be found by RegExLib. Each group of strategies used different NLP methods for processing identifier names. Tokenisation is a basic technique for determining the words that make up an identifier. Therefore, all strategies performed tokensiation as a first step. Then, a combination of PoS tagging and Non-Word Removing techniques were applied. Table II provides the summary of the strategies S0-S6.

The remaining of the section details the empirical evaluation and addresses the research questions.

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

To answer this question, the values generated by each strategy were run through the validation routines in each project, and the percentage of valid values for each type was computed. Table III records this percentage for each strategy for the 24 String types and their average.

The strategy S0 produced valid values in all cases, except in Resource URL (Conzilla) and one of the Date (Efisto) types. The strategies using regular expressions and NLP techniques, i.e., S1-S6, produced higher number of valid values compared to S0 on average. However, there are three cases where no values could be produced by strategies S1-S6. The analysis of each one of those is given as follows.

The first case was Path URN (Conzilla). There were no regular expressions generated by RegExLib for this type, so S1-S3 could not proceed any further. Strategies S4-S6 applied web search to further explore for regular expressions. However, Bing did not return any search results. Consequently none of the strategies were able to produce valid values 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 from the web search method which consequently produced several values. These values were mainly related to the Internet resource references (such as URLs). However, no value

Table III
% Comparison of Valid Values in different strategies with Average percentage and Rank-Sum scores

Project	**String Type**	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**
Chemeval	CAS number	18.48	48.77	95.5	95.51	46.95	95.5	95.5
Conzilla	MIME type	7.22	0.62	0	0	0.78	68.29	100
	Path URN	< 0.01	0	0	0	0	0	0
	Resource URL	0	0	0	0	0	0	0
	URI	5.94	1.86	2.25	2.33	2.05	2.17	1.99
	URN	0.05	0.1	0.12	0.13	0.08	0.13	0.11
Efisto	Date (dd.MM.yyyy)	1.79	0	0	0.13	0	0	0.17
	Date (EEE, dd MMM yyyy HH:mm:ss zzz)	0	0	0	0	0	0	0
GSV05	24 hour time	1.19	6.88	6.03	5.01	6	5.16	5.51
JXPFW	BBAN	8.48	52.63	95.11	94.92	66.67	94.87	94.87
	POSIX locale identifier	1.76	0	0	0	5.93	3.2	2.21
	2 letter country code	0.30	0.06	0.14	0.11	0.05	0.14	0.09
	IBAN	0.21	15.52	3.9	3.86	4.66	3.75	3.75
LGOL	Date (dd/mm/yyyy)	0.15	0.02	0.02	7.12	0.07	0.12	7.69
	Integer	1.76	3.79	3.45	0	2.76	4.34	0
	UK Postcode	< 0.01	0.01	0.01	100	0.07	0.37	100
OpenSymphony	Email address	6.35	70.9	56.71	97.62	22.5	58.7	99.16
	SSN	17.96	4.9	4.63	100	5.56	5.59	100
PuzzleBazar	Email address	0.37	0.32	0.11	0.11	0.39	0.1	0.12
TMG	ISBN	6.37	75.39	81.29	81.29	62.33	81.29	81.29
	Month	0.04	0.57	0.63	0.68	0.65	0.68	0.68
	Year	8.29	93.35	92.16	93.01	91.97	92.53	91.88
WIFE	BIC	7.74	100	100	100	100	100	100
	IBAN	0.07	80.43	83.58	83.58	20.44	83.58	83.58
Average		**3.94**	**23.17**	**26.07**	**36.06**	**18.33**	**29.19**	**40.36**
Rank-Sum scores		**76**	**85.5**	**92**	**112**	**86.5**	**108.5**	**111.5**

corresponded to the required type of the URL that is prefixed with "`res://`" – a rare representation. Therefore, the count for valid values in all cases remained zero.

The last case was Date (EEE, dd MMM yyyy HH:mm:ss zzz) (Efisto). In this case, RegExLib generated a number of regular expressions, however, none corresponded to this precise format. Hence, no valid values were produced for this case in any strategy.

All strategies produced a high number of valid values on average when compared to S0. In some cases 100% of the generated values were found to be valid, including MIME-Type (Conzilla), UK Postcode (LGOL) and BIC (WIFE). There are few cases where the precision fell below 1%. There were various causes for such performance that are discussed in the following.

Inappropriate regular expressions

In some cases, the regular expressions obtained were not suitable for the required values. This was mainly due to the RegExLib search mechanism that generates expressions not particularly related to the search query. For example, the search query "`URN`" generates expressions related to date formats, XML tags, postal address etc. Thus, a large set of values generated by these expressions could not be validated as a proper URN.

Low informativeness in identifier names

There are cases where a peculiar format is required for the values to be valid but the identifier names could not guide 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) have produced a large set of values but few were regarded as valid.

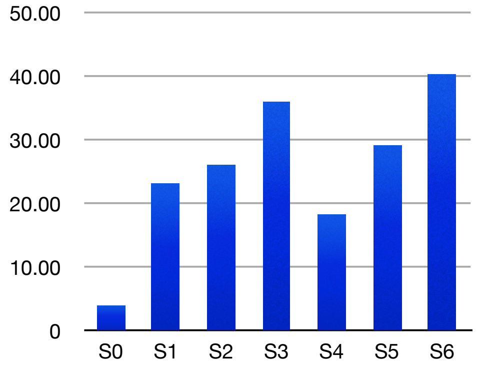

Figure 3. Comparison of Average Percentage of Valid Values in different strategies (summary of Table III).

Misguided search due to a general context

There are cases where the identifier names represent a general context, thereby causing too many values to be generated of which only few were valid. A good example is the comparison between Email Address in OpenSymphony and PuzzleBazar projects, where the former had produced ≈99% of valid values, but the latter had a marginal success. The main reason is the identifier names ("`Validation`", "`text`") in the PuzzleBazar project that represent a general context but not particularly related to an email address.

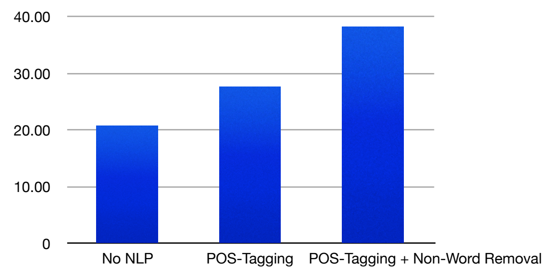

Figure 4. Aggregated mean comparison for different NLP techniques

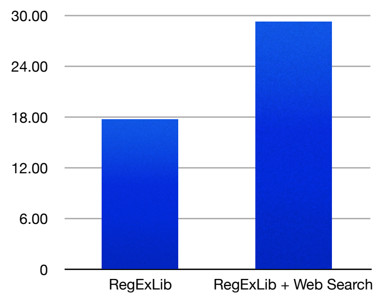

Figure 5. Aggregated mean comparison for different methods of obtaining regular expressions

Variations in the validation routines
It is observed that a large set of generated values were indeed valid in many cases but their validity check failed. The reason is that these validation routines vary in small details about which values are considered to be valid. For example, IBAN (JXPFW) requires the account numbers as a sequence of characters without space, whereas IBAN (WIFE) accepts spaces. The majority of IBANs generated by the approach contained spaces which failed to be regarded as valid in the former case, and thus produced low percentage compared to the latter case where a high percentage (≈83%) was obtained.

Figure 3 shows the comparison of strategies by their average performance. A general conclusion can be drawn from the chart that the use of regular expressions and web queries can find valid values based on the information extracted from the program identifier names. The quantitative result is partly due to finding the correct regular expressions and partly due to the information contained in the identifier name in order to form comprehensive web queries. This issue is further discussed in Section IV.

RQ2. *Which web search strategies are significant in finding valid string values?*

The comparison of average percentage of valid values shown in Figure 3 shows that the strategies using NLP techniques and regular expressions, i.e., S1-S6, outperformed the strategy S0 where a simplest approach was applied. The strategies were also analyzed according to the Rank-Sum test [9] in which each strategy was ranked in ascending order of their percentage computations for each case, and then adding all of the ranks together. For example, S0 was ranked 1 (lowest) and S6 was ranked 7 (highest) in Email address (OpenSymphony) case. Where tied ranks occurred, i.e., the same ranks occurred between two or more strategies, then the average rank was considered from the sum of the ranks concerned and divided by their number. For example, all strategies in the Resource URL (Conzilla) case are tied, thus given an equal rank of 4 = (1+2+3+4+5+6+7)/7. The rank-sum score was then calculated for each strategy by simply adding together their respective ranks for all cases. For simplicity, only the rank-sum scores are given in Table III, which shows that strategies S1-S6 are ranked higher than the strategy S0, and strategies S3 and S6 are ranked the highest by overall performance.

There is also a general trend observed among the strategies S1-S6. This trend can be studied in two further questions:

RQ 2.1: *Which combination of NLP techniques is more significant for processing identifier names?*

To answer this question, the aggregated data from Table III for strategies S1, S4 (that use no NLP techniques), S2, S5 (that use PoS-Tagging) and S3, S6 (that use PoS-Tagging + Non-Word Removal) is compared in Figure 4. It is seen that using NLP techniques is useful for identifying more valid values, with an average difference between no NLP and PoS-Tagging of ≈7%, whereas between no NLP and PoS-Tagging + Non-Word Removal the difference is ≈17.5% in the study.

RQ 2.2: *Which method for obtaining regular expression is more significant in finding valid values?*

To answer this question, the aggregated data for strategies S1, S2, S3 (that use only RegExLib) and for strategies S4, S5, S6 (that use RegExLib + web search) is compared in Figure 5. It is seen that using both methods helps in finding more valid values which had an average increase of ≈12% in the evaluation study.

In order to test the statistical significant difference between the strategies on a general population, Wilcoxon Matched-Pairs Signed-Ranks test [9] was conducted at a confidence level of 0.95. The p-values were calculated using R tool [19] for different comparisons shown in Table IV. At this confidence level, where a p-value < 0.05 indicates significance, a significant difference was identified for strategies S3-S6 against strategy S0. To address, *RQ 2.1* and *RQ 2.2*, the test was conducted on different combination of strategies. The p-values given in Table V shows a significant difference between performance obtained using NLP techniques (PoS-Tagging + Non-Word Removal) and not using them (No

Table IV
p-VALUES FOR WILCOXON MATCHED-PAIRS SIGNED-RANKS TEST FOR ALL STRATEGIES AT CONFIDENCE LEVEL 95%. THE SIGNIFICANT DIFFERENCE IS GIVEN IN BOLD.

	S0	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**
S0	-	0.0605	0.0952	**0.0123**	**0.0312**	**0.0091**	**0.0022**
S1	0.0605	-	0.9794	0.0949	0.3860	0.1313	**0.0290**
S2	0.0952	0.9794	-	0.1820	0.3048	0.0736	0.3318
S3	**0.0123**	0.0949	0.1820	-	0.0500	0.2446	0.4897
S4	**0.0312**	0.3860	0.3048	0.0500	-	**0.0313**	0.0594
S5	**0.0091**	0.1313	0.0736	0.2446	**0.0313**	-	0.2585
S6	**0.0022**	**0.0290**	0.3318	0.4897	0.0594	0.2585	-

Table V
p-VALUES FOR WILCOXON MATCHED-PAIRS SIGNED-RANKS TEST FOR DIFFERENT COMBINATIONS AT CONFIDENCE LEVEL 95%. THE SIGNIFICANT DIFFERENCE IS GIVEN IN BOLD.

Comparison between		**p-value**
No NLP (S1,S4)	PoS-Tagging (S2,S5)	0.1313
No NLP (S1,S4)	PoS-Tagging + Non-Word Removal (S3,S6)	**0.0458**
PoS-Tagging (S2,S5)	PoS-Tagging + Non-Word Removal (S3,S6)	0.338
RegExLib (S1,S2,S3)	RegExLib + Web Search (S4,S5,S6)	0.6407

NLP). However, no other differences were found to be significant.

RQ3. *How effective is the approach compared to the other test data generation techniques for strings?*

There are no known tools available for generating valid String values in the same settings as provided in this paper. The closest techniques are test data generation techniques that usually aim for code coverage. Two main techniques: dynamic symbolic execution and search-based testing, 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. Then the test data was extracted from the test cases for the 24 String types to analyse the percentage of valid values through the validation routines.

For dynamic symbolic execution, *Symbolic PathFinder* (*SPF*) [25] was used that performs symbolic execution of Java bytecode with model checking and constraint solving. Default options were used for the decision procedure: *Choco* [27], and for the string solving approach: *automata*. *SPF* was run till termination, i.e., either normally (when tests were generated), or abnormally (due to the tool's internal error).

For search-based testing, an improved version of *eToc* [28], called *eToc*$^+$ [24] 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. This way, each uncovered branch got at least 100,000 fitness evaluations, possibly even more if progress was being made.

Table VI provides the average percentage of valid values collected from the test cases generated by both tools.

Table VI
AVERAGE PERCENTAGE OF VALID VALUES IN DIFFERENT TEST GENERATION TOOLS

	eToc$^+$	*SPF*
Average	10.28	4.17

Evidently, these tools have produced very low number of valid values on average in comparison with the proposed approach (cf. Table III). 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. 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

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

Then for each a, average effectiveness was taken for all s. Figure 6 presents the comparison of the average effectiveness for all approaches. It reflects the conclusion from *RQ 2* that strategies using regular expressions and NLP techniques are more effective, where S6 outperformed all approaches. *eToc*$^+$ is seen to be almost as effective as the strategies which employ no NLP techniques (i.e., S1 and S4). *SPF* is seen to be the least effective. The main reason is that String constraint solving in SPF is currently work-in-progress[4], and does not recognize complex object types (e.g., *java.text.SimpleDateFormat*, *java.lang.Integer*), and throws runtime exception during test generation.

IV. THREATS TO VALIDITY

One major hypothesis this approach is built on is that the identifier names include relevant domain knowledge. This

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

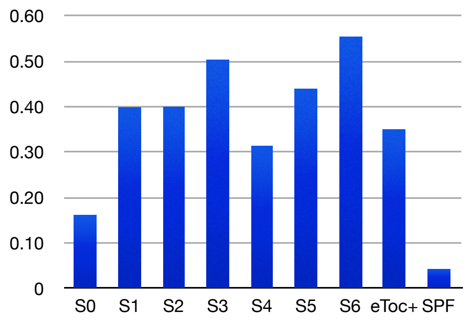

Figure 6. Comparison of Average Effectiveness of different approaches

is investigated by Butler et al. [12] in an empirical study of 28,000 identifier names drawn from 60 open source Java projects and concluded that programmers tend to enclose the domain knowledge in concise names using conventional styles of camel casing and underscores. In our experiments, 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. In general, many valid values were obtained by the approach using the identifier names in the empirical study. Arguably, the identifier names that are more expressive have been more useful in drawing valid values, but evidently, using NLP processing techniques improved the usefulness of the identifier names in the web queries.

Another major threat is about obtaining inappropriate regular expressions. This becomes more sensitive in using the web search method when the contents of a web page are presented in a highly unstructured way. The chances can be reduced by using dedicated libraries such as RegExLib that provide more formal structure for searching regular expressions. However, RegExLib is not very comprehensive at present and only provides a basic search engine of limited effectiveness. Moreover, the approach is currently using a very simple technique for collating regular expressions from the web, but more appropriate algorithms can be employed to check out their relevance with the target String type.

Another threat involves the potential bias in the comparison of different strategies. Simple mean calculation showed that the strategy S6 has outperformed the others in the evaluation study, however, more rigorous statistical analysis was required to check whether the results are not due to the selection of case studies. A common and reliable method is to perform Wilcoxon Matched-Pairs Signed-Ranks test [9] – a non-parametric statistical hypothesis test that does not require any assumptions about the shape of the distribution. The test was performed to show the significance of strategies over each other at the 95% confidence level. The test was repeated several times to increase the probability of the error.

Our previous work [24] showed that the use of valid values produced by the strategy S0 in search based testing increased the coverage by 14% on average on the same case studies. Since many more valid values have been produced by the current approach (S1-S6), it is likely that the approach would also be effective in enhancing the coverage.

V. Related Work

There has been significant work on test data generation, but little of it has addressed the problem of string input generation in the past. Many of the existing approaches tackle this problem by limiting the length of the string value. Even for fixed length, the input space is too large for the exhaustive testing of all inputs. Alternatively, random exploration of the input space could be feasible in practice, but the probability that it can exercise the deep branches is very low [22] [11]. Elbaum et al. [13] have proposed to collect input data from previous user sessions. This turns out to be an effective approach for generating test cases at a little expense. However, this work is mostly related to web applications in which users normally leave behind the footprints of their data. On the contrary, the use of our approach goes beyond the web applications; moreover, it searches for input data from scratch without assuming any historic usage of the application.

Dynamic Symbolic Execution (DSE) is an effective technique for test generation but most implementations get stuck at path conditions which include complex input structure (such as strings). Several approaches have been proposed to reduce the input space to generate syntactically valid inputs. Li et al. [21] have proposed Reggae tool based on a DSE engine for testing .NET programs that transforms the generic functions to specialized functions which provide the DSE engine with a smaller exploration space. Majumdar and Xu [22] have proposed the CESE tool that reduces the input size for DSE by pre-generating bounded-sized strings from the symbolic grammar that is abstracted from the concrete input syntax. A similar approach has been presented by Godefroid et al. [15] to leverage white-box fuzzing using a custom grammar based constraint solver. However, these approaches assume knowledge of the valid input type provided by some context-free grammar. Contrary to the approach in this paper, this knowledge is actually inferred from the program identifier names and formalised with the help of the relevant regular expressions that are also obtained dynamically.

Further string constraint solvers, e.g., HAMPI [14], are available but they currently lack support in existing DSE tools; hence it is not straightforward to compare them with the approach. Symbolic PathFinder [25] integrates external string constraint solvers, but it is still at a preliminary stage.

In Search-Based-Testing (SBT), the problem has been largely ignored treating strings as fixed-length characters [17] or acquiring specialist generators [28]. Alshraideh and Bottaci [7] exploited the string literals found in the program under test and guided the search operators to focus the search in the region of such string literals. This resulted in an ameliorative branch coverage in the evolutionary testing empirically. Bozkurt and Harman [10] studied the valid input generation for service-oriented software. They proposed reusing the outputs of the other existing web services as inputs to the services under test. In contrast to these works, the proposed approach in this paper uses web queries to find potentially valid string values that might already exist on the Internet. Our recent work [24] has actually proposed the original idea, which is called the strategy S0 in this paper, to enhance branch coverage in SBT by seeding web values. However, achieving high coverage does not imply generating valid values (as suggested by the analysis of S0 in Section III-B). The current approach uses regular expressions and NLP techniques with web searches that have appeared to be more effective compared to the existing approaches.

VI. Conclusion and Future Work

This paper has presented an approach for generating values for String data types by using tailored web searches, dynamic regular expressions and NLP techniques. The empirical study showed that the valid values can be obtained using the approach. Another benefit is that the generated values are also realistic rather than arbitrary-looking – as often the case with the most automatic test data generation techniques. This is because the values are obtained from the Internet which is a rich source of human-recognizable data. When an automated oracle is non-existent, the test cases using such values help in reducing human-oracle cost [23] in terms of time and effort involved in interpreting results. More empirical evidence is required to hold the claim that is planned in the future work.

Future work also includes reducing the limitations of the approach by employing more sophisticated algorithms for processing identifier names, e.g., name expansion (i.e., converting "`str`" to "`string`") [20]. Moreover, advanced algorithms for obtaining and filtering regular expressions shall be investigated to improve the precision of valid values.

The use of valid values generated by this approach may also be useful for a DSE approach instead of using random concrete values. Also, fitness functions in SBT can benefit from these values to reduce the time in the evolutionary searches [24]. Further investigation is planned in these directions with more empirical studies.

Acknowledgment

This work was funded by the EPSRC project RE-COST (REducing the Cost of Oracles for Software Testing, grant no. EP/F065825), http://recost.group.shef.ac.uk.

References

[1] *Bing search engine*. http://www.bing.com.

[2] *Introducing Bing API Version 2.0*. http://www.bing.com/developers/s/APIBasics.html.

[3] *Jazzy*. http://sourceforge.net/projects/jazzy.

[4] *JSoup*. http://www.jsoup.org.

[5] *ModeShape library*. http://www.jboss.org/modeshape.

[6] *RegExLib: Regular Expression Library*. http://regexlib.com/.

[7] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators. *STVR*, 2006.

[8] K. Atkinson. *Spell Checking Oriented Word Lists (SCOWL)*. http://wordlist.sourceforge.net/.

[9] S. Boslaugh and P. A. Watters. *Statistics in a nutshell - a desktop quick reference*. O'Reilly, 2008.

[10] Mustafa Bozkurt and Mark Harman. Automatically generating realistic test input from web services. In *SOSE*, 2011.

[11] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*. IEEE Computer Society, 2008.

[12] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *ECOOP*, 2011.

[13] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *ICSE*, 2003.

[14] V. Ganesh, A. Kiezun, S. Artzi, P. Guo, P. Hooimeijer, and M. Ernst. Hampi: A string solver for testing, analysis and vulnerability detection. In *CAV*, pages 1–19, 2011.

[15] P. Godefroid, A. Kiezun, and M. Levin. Grammar-based whitebox fuzzing. In *PLDI*, pages 206–215. ACM, 2008.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

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

[18] D. Jurafsky and J.H. Martin. *Speech and Language Processing*. Prentice Hall, 2 edition, 2008.

[19] P. Kuhnert and W. N. Venables. *An Introduction to R: Software for Statistical Modelling & Computing*. CSIRO, Canberra, Australia, 2005.

[20] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *ICSM*, pages 113–122, 2011.

[21] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE*, 2009.

[22] R. Majumdar and R. Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[23] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[24] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *ICST*, pages 141–150, 2012.

[25] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, pages 15–26. ACM, 2008.

[26] L. Shen, G. Satta, and A. Joshi. Guided Learning for Bidirectional Sequence Classification. In *ACL*, 2007.

[27] CHOCO Team. choco: an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[28] P. Tonella. Evolutionary testing of classes. In *ISSTA*, 2004.

[29] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL '03*, pages 173–180, 2003.

[30] L. Williams, B. Smith, and S. Heckman. *Test Coverage with EclEmma*. North Carolina State University, March 2009.