2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Search-Based Test Input Generation for String Data Types Using the Results of Web Queries

Phil McMinn, Muzammil Shahbaz and Mark Stevenson
University of Sheffield, Regent Court, 211 Portobello, Sheffield, UK, S1 4DP

***Abstract*—Generating realistic, branch-covering string inputs is a challenging problem, due to the diverse and complex types of real-world data that are naturally encodable as strings; for example resource locators, dates of different localised formats, international banking codes, and national identity numbers. This paper presents an approach in which examples of inputs are sought from the Internet by reformulating program identifiers into web queries. The resultant URLs are downloaded, split into tokens, and used to augment and seed a search-based test data generation technique. The use of the Internet as part of test input generation has two key advantages. Firstly, web pages are a rich source of valid inputs for various types of string data that may be used to improve test coverage. Secondly, the web pages tend to contain realistic, human-readable values, which are invaluable when test cases need manual confirmation due to the lack of an automated oracle. An empirical evaluation of the approach is presented, involving string input validation code from 10 open source projects. Well-formed, valid string inputs were retrieved from the web for 96% of the different string types analysed. Using the approach, coverage was improved for 75% of the Java classes studied by an average increase of 14%.**

***Keywords*-Automatic test data generation; search based testing; string inputs; web queries;**

I. Introduction

The automatic generation of structural software test data has received much attention in the literature of late. Different approaches have been proposed; including dynamic symbolic execution [1], [2] and search-based testing [3]. This particular paper concerns the search-based approach to test data generation. Search-based approaches employ meta heuristic search techniques, such as Genetic Algorithms, to optimise a fitness function describing the test goal, for example the coverage of a particular branch in a piece of software. Not only has the approach been shown to be an effective technique for generating software test data [4], but it is also extremely flexible, allowing different test objectives to be tackled by simply changing the fitness function. To date, search-based testing has been applied not just to structural testing [4], but also to functional testing [5], web testing [6], interaction testing [7] and stress testing [8], amongst others.

One important activity in structural test data generation involves deriving suitable values for string inputs. Strings present additional challenges above those of other basic data types, such as integers. In general, strings may be of variable length, contributing to enormous input domain sizes, and consequently very large search spaces. Many forms of real world data may be naturally represented as strings; for example resource locators, dates of different localised formats, international banking codes, and national identity numbers. In order to cover a particular program branch, a string may need to satisfy a series of complex conditions involving regular expressions, substring comparisons or checksum operations. An additional issue is the hitherto overlooked problem concerning the generation of realistic, comprehensible values that a human tester would be likely to generate and use to exercise a program. Automatic test generators take cues from program code only, leading to the likely generation of arbitrary values such as '`!&^@s.sd`' for an email address, rather than natural, instantly-readable strings such as '`bill@microsoft.com`'. This is an important consideration when test cases require manual evaluation by a human, due to the lack of an automated oracle (as is often the case in practice).

Internet web pages are a rich source of examples of string data that may be re-used as a natural source of inputs for a wide variety of programs, and one that has hitherto remained unexploited in structural test data generation. This paper presents a novel approach in which real examples of string inputs are sought from the Internet by performing web queries based on key identifiers appearing in the source code of the program under test. The resultant URLs are downloaded and tokenised, and the strings used to augment and seed a search-based test data generation technique. The paper presents an empirical study using 20 Java classes from 10 open source projects. The code studied centred on validation routines for a number of different data types based on strings. The web query approach was capable of retrieving examples of valid, well-formed string inputs for 96% of string types analysed. Furthermore, the string values obtained enabled the coverage of several 'hard-to-execute' branches. Branch coverage was improved for 75% of the Java classes studied by an average of 14%.

The contributions of this paper are therefore as follows:

1) An approach for generating test cases involving string inputs by formulating web queries and retrieving potential string values from the Internet
2) An empirical study demonstrating the effectiveness of the approach. The study shows that valid, well-formed string inputs can be found for the programs concerned by reformulating program identifiers into web queries, and that the strings found are capable of improving the test coverage of the source code of a program.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.26

```
1   class DDMMYYYYDate {
2     int d=0, m=0, y=0;
3     DDMMYYYYDate() {}
4     void parse(String s) {
5       int b1 = s.indexOf("/");
6       if (b1 != -1) {
7         int b2 = s.indexOf("/", b1+1);
8         if (b2 != -1) {
9           d = Integer.parseInt(s.substring(0, b1));
10          m = Integer.parseInt(s.substring(b1+1, b2))
11          y = Integer.parseInt(s.substring(b2+1));
12    } } }
13    boolean isValid() {
14      if (d > 0 && d <= 31)
15        if (m > 0 && m <=12)
16          return true;
17      return false;
18    }
    ...
```

Figure 1. Class intended to validate dates in the form 'DD/MM/YYYY'

The paper begins by reviewing important background (Section II). Section III presents our approach for using identifiers in web queries, and extracting example strings. Section IV describes how these strings are then incorporated into the test data generation process. Section V then evaluates the approach using Java code drawn from a number of open source projects. Section VI then presents related work, while Section VII closes with concluding remarks and avenues for future work.

II. Background

A. Search-Based Unit Testing of Object-Oriented Software

Search-based testing applies optimisation techniques to generate test cases for branch coverage. In the case of unit testing object-oriented software, the search must find a test case that instantiates the class under test with the correct constructor parameters, followed by a sequence of method calls to the object and accompanying input parameter values. For example, a test case causing '`isValid`' method of Figure 1 to return true would be:

```
DDMMYYYYDate d = new DDMMYYYYDate();
d.parse("12/10/2007");
d.isValid();
```

A popular choice of search technique for generating object-oriented unit tests are Evolutionary Algorithms [9], [10], [11]. Evolutionary Algorithms work to evolve test cases using a process inspired by Darwinian evolution and the principle of the survival of the fittest. The 'fitness' of a test case is computed by a fitness function that is to be minimised by the algorithm. Test cases deemed to be 'close' to executing the target structure are rewarded with lower values than those judged to be further away. A test case covering the target is awarded the optimal value of zero. The fitness computation involves the so-called *approach level* and the *branch distance*. The approach level measures how close the test case was to covering a target in terms of its execution path, based on the target's control dependencies. For a structured program, the approach level reflects how deeply the nesting structure surrounding the target is penetrated. For example, if the target is the '`return true`' statement (line 16) of Figure 1, the approach level is 1 or 0 if the execution path fails to take the true branches at lines 14 or 15 respectively. The approach level is 2 if the `isValid` method is not executed by the test case at all.

The second component of the fitness function is the *branch distance*. Where the path diverges from the target, the branch distance assesses how close the test case was to executing the alternative outcome at the decision statement. Different branch distance formulas are applied depending on the type of predicate appearing in the branching statement. For predicates of the form '`a == b`' the branch distance is $|a - b| + K$, where K is a positive constant value ($K = 1$ in this paper). The closer `a` and `b` are to being equal, the lower the branch distance. (For a full list of predicates and associated branch distance formulas, see Tracey *et al.* [12].) The approach level and branch distance are combined into one value by normalising the branch distance and adding it to the approach level. In this paper, the normalisation function proposed by Arcuri [13] is used, $\frac{d}{d+1}$, where d is the branch distance.

An Evolutionary Algorithm consists of a number of key steps, maintaining a set of candidate solutions (*i.e.* test cases) to the problem at hand, called the 'population', with the aim of evolving fitter candidate solutions. Each candidate solution — referred to as an 'individual' — must be represented in a particular format for later manipulation, referred to as 'chromosomes'. The first generation of the population is randomly generated. The loop of the algorithm then begins, first assessing each individual for fitness, and then invoking a *selection* process, biased towards the fittest individuals, in which chromosomes are put forward for *crossover*. In crossover, two 'parent' individuals are spliced together to form two 'offspring' candidate solutions. In terms of test cases, this involves recombining the constructor and method call sequence. For example, the test cases

```
(p1) d = new DDMMYYYYDate();   (p2) d = new DDMMYYYYDate();
     d.parse("12/10/2007");         d.isValid();
     d.parse("7/4/2010");           d.isValid();
```

may be recombined after the second statement (with the crossover point denoted by the dotted line) to form two offspring test cases:

```
(o1) d = new DDMMYYYYDate();   (o2) d = new DDMMYYYYDate();
     d.parse("12/10/2007");         d.isValid();
     d.isValid();                   d.parse("7/4/2010");
```

The offspring are then *mutated* at random. In terms of a test case, this may involve inserting a method call or constructor, removing it entirely, or changing a parameter value. For both crossover and mutation, a stage of 'repair' may be required to remove references that are left unused as a result or either operator, or new objects being created using randomly-selected constructors if the original constructor

was removed or is not present in the offspring of a new test case. The final stage of the loop involves *reinsertion* of the newly generated individuals into the population to form a new *generation* of candidate solutions, in which some or all of the original population are replaced with the newly-generated offspring. The algorithm continues to iterate with each successive generation until an appropriate test case is found, or some stopping criterion is fulfilled (*e.g.*, a limit on the number of generations or fitness function evaluations).

B. Search-Based Generation of String Values

String inputs and parameter values present problems for automatic test generators, for two reasons:

1) Complexity. Conditions involving strings tend to be complex and hard to solve; potentially involving regular expressions, substring comparisons and checksum operations. A further confounding factor is that strings may be of a variable length, resulting in potentially infinite search spaces. Appropriate string values for a branch may therefore be very hard to find, forming pathological 'needle in a haystack' problems.

2) Realistic test data. Automatic test data generators tend to be based on information derived almost exclusively from the program itself, resulting in the production of arbitrary string values that can cover branches, but are not necessarily realistic or easily comprehensible from a human point of view. Such string values are a poor match with that which would be produced by a human tester. This results in test cases that are harder to manually check [14] where an automated oracle does not exist (as is often the case in practice).

Despite over 500 papers being written on Search-Based Testing [15], there have been very few papers addressing the problem of generating string values for test cases. Previous work in testing procedural C programs treated strings as fixed-length arrays of characters [16]. The *eToc* tool of Tonella [9] allows for variable length strings to be generated. Optional 'generators' may be supplied by a tester in order to generate specific types of required string. Ideally, however, string generation should be fully automatic.

Alshraideh and Bottaci [17] consider search-based string generation for programs written in the JScript language. A large improvement in coverage was found when string literals found in the program code were used to seed the first generation of the evolutionary search. String literals can act as full or partial examples of strings that are expected as inputs to a program; however, there is no guarantee that useful string literals will always be present in the code being tested.

This paper proposes a novel approach that seeks realistic examples of string inputs through targeted web queries, by using identifiers appearing in the program under test.

III. Using Web Queries To Find String Test Inputs on the Internet

This section describes our strategy for finding string inputs on the Internet, implemented in a tool called *Delver*. Web queries are generated from key identifiers found in the source code of the Java class under test. The web query strings are then inputted into a search engine. The search engine results are then harvested, with potential string inputs extracted for later use in test case generation.

A. Generating Web Queries

Program identifiers are intended to describe different types of data in order to aid human understanding of a program, and as such are likely to form useful web queries in finding information about the individual words from which they are formed. Where an identifier name corresponds to some common concept, the contents of the URLs returned in the search engine's results are likely to include examples of that concept. For example, a web search using the query *email address* is likely to return web pages that contain examples of email addresses.

Delver performs web queries using three program identifiers related to the string method parameter for which input values are sought:

1) *The identifier of the string method parameter*. The name of the method parameter for which values are sought is an obvious source of potential information about the types of values that it is supposed to be used for. This is the case for the string method parameter `emailAddress` in Figure 2a.
2) *The method identifier*. Sometimes programmers use non-informative generic method parameter names, such as *text*, *str*, or *value*. This may be because the concept is already embodied in the method name involving the string parameter, as seen in Figure 2b.
3) *The class identifier*. The method name may also have a non-informative generic name if the class has few responsibilities other than to represent a data type itself and the operations that may be performed on it. In this case, the name of the parameter's class may be a valuable source of keywords, as in Figure 2c.

Delver generates a set of queries for each identifier by applying a range of processing steps (summarised in the pull-out of Figure 4).

Generation of the base query. The first step involves creation of the *base query*. Program identifiers are often formed from concatenations of terms which need to be separated back into individual words to form useful web queries. The base query is therefore formed by splitting identifiers into words according to the underscoring or camel casing style used. For example '*anEmailAddress*' and '*an_email_address*' both become '*an email address*'. In addition, identifiers often contain stop words, very common words which are not useful

```
class Mailer {
  boolean isValid(String emailAddress) {
  ...
```

(a) Parameter name

```
class Util {
  boolean isEmailAddress(String str) {
  ...
```

(b) Method name

```
class EmailAddress {
  boolean isValid(String str) {
  ...
```

(c) Class name

Figure 2. How parameter, method and class identifiers may hold keywords indicating the type of information to be held in a string parameter

	Pluralised	Prefixed	Quoting style
email address (base query)	✗	✗	None
email addresses	✓	✗	None
"email address"	✗	✗	Full
"email addresses"	✓	✗	Full
list of email address	✗	✓	None
list of email addresses	✓	✓	None
"list of email address"	✗	✓	Full
"list of email addresses"	✓	✓	Full
"list of" "email address"	✗	✓	Separated
"list of" "email addresses"	✓	✓	Separated

Figure 3. Queries generated for the identifier '`emailAddress`'

for web queries, including 'the', 'and' and 'a'. Following standard practice in Information Retrieval [19] these are removed. Thus, '*an email address*' simply becomes '*email address*'.

Additional queries are generated from the base query by applying combinations of further operations, intended to increase the number of sources for examples of relevant string inputs. These are as follows:

Pluralisation. Delver generates pluralised versions of the base query by pluralising the last word using the ModeShape library [20]. For example, '*email address*' becomes '*email addresses*'.

Prefixing. In order to direct the web search towards pages containing lists of examples for the identifier, versions of the base query are formed by prefixing the query with *'list of'*; for example *'list of email addresses'*.

Quoting. Quoting words in a web query signals to the search engine that those terms must be found as a complete phrase in the web pages to be retrieved. This is useful to ensure that identifier words are kept together in the web pages returned in the search engine's results. *Delver* generates additional queries by adding quotes around the entire query, and in the presence of a 'list of' prefix, by adding quotes around the prefix and remainder of the query separately.

In total, 10 queries are generated for each identifier, as seen from the matrix in Figure 3, which shows the queries generated from the identifier '`emailAddress`'.

B. Performing Web Queries and Processing Resultant Pages

Queries are performed using Microsoft's Bing [18] – the only major Internet search engine providing free API access at the time this research was conducted. Delver uses version 2.0 of Bing's API [21] to retrieve search engine results. The localisation was 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 query results are processed by first downloading the contents of each URL and stripping out HTML tags. The remaining text is then tokenised according to whitespace, and placed into a list of unique tokens for use as potential string values in the test case generation process, as described in the next section.

IV. Test Case Generation using Web Query Results

For OO unit test case generation, finding inputs is only one part of the problem, since method call sequences are required to construct relevant objects and bring those objects into the correct state for a branch to be covered. Another issue is how the string values found from the web should be incorporated into the test generation process, since the web queries are likely to result in a large number of values.

Delver employs the following strategy, involving three distinct phases.

1) Run short evolutionary searches to generate initial method call sequences.
2) Inject tokens found by the web queries, one by one, into those method call sequences in place of existing values for string method parameters. Store any new test cases covering any new branches unexecuted in the previous phase.
3) Perform further evolutionary searches to cover any remaining uncovered branches, using the test cases stored as a result of the last step as seeded individuals in the first generation of the new search.

Phase 1. Initial Evolutionary Searches

Delver uses an improved version of the *eToc* tool [9] (hereon referred to as *eToc*$^+$) to search for branch-covering test cases, the general scheme for which was described Section II-A. Space does not allow for full details regarding *eToc* to be reproduced here (the interested reader is referred to [9]). The following paragraphs therefore serve to describe in detail the differences between *eToc*$^+$ and *eToc* only, in order for the experiments carried out in this paper to be replicated. *eToc*$^+$ makes use of the full fitness function combining approach level and branch distance as described in Section II-A (*eToc* uses the approach level

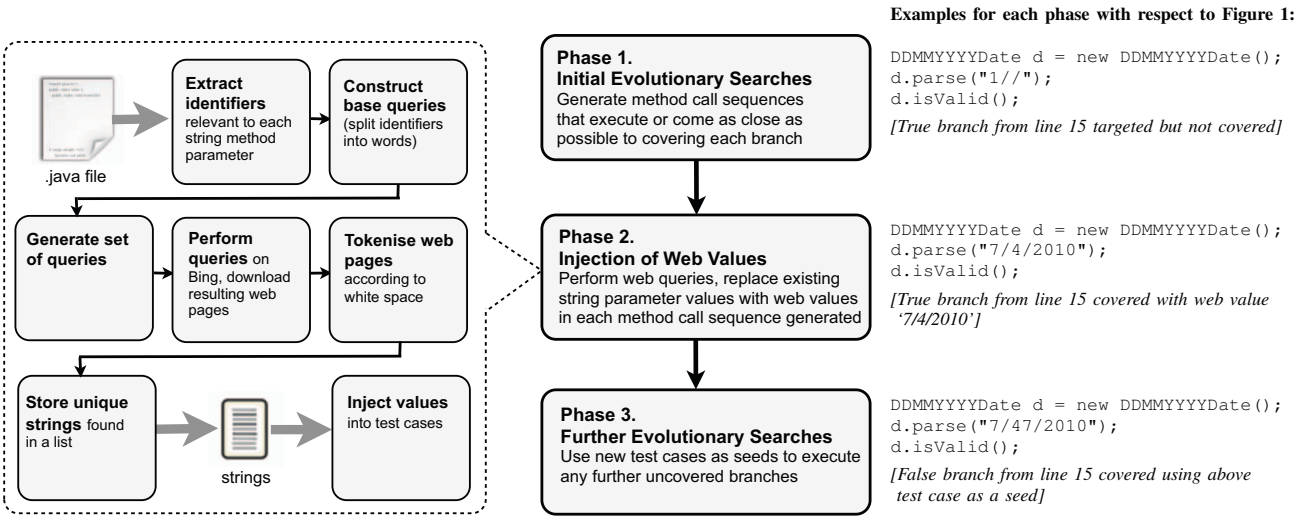


Figure 4. Overview of the steps involved in the test generation process behind *Delver*, utilising the results of web queries

only, resulting in a more coarse-grained fitness function). *eToc*$^+$ uses the chromosome encoding of test cases and the same crossover mechanism as *eToc*. Likewise, mutation of the constructor and method call sequence remains as for *eToc*, however mutation of parameter values is improved for *eToc*$^+$. Whereas *eToc* completely replaces values with new randomly-generated ones, *eToc*$^+$ applies a more incremental approach based on existing values. This allows the search to use mutation to improve parameter values with respect to fitness, rather than having to start over again following a value being overwritten with a random value. Gaussian mutation is used for numeric values. For string values, characters are inserted, removed or replaced with an equal probability at a randomly-chosen location. Characters are chosen randomly from the set of 95 printable ASCII characters (0x20–0x7E).

eToc$^+$ uses stochastic universal sampling [22], together with linear ranking [23] of individuals, which prevents super-fit individuals dominating the selection process. *eToc*$^+$ keeps hold of the very best individuals using elitist reinsertion. The top 10% of individuals in the population are automatically allowed to survive into the next generation, with the remaining 90% of individuals replaced with offspring.

eToc$^+$ attempts each branch one at a time, starting with the most deeply nested branches first. If a different branch is covered in the process of the search for a target branch, that test case is stored, meaning that a dedicated search for that branch is not required.

The initial evolutionary searches are run for 100 generations with a population size of 100 for each uncovered branch.

Phase 2. Injection of Web Values

For each branch uncovered in Phase 1, the method call sequence with the best fitness found is taken and used as a starting point for Phase 2. The starting test case is used as a skeleton into which string values found from the web queries are inserted. Web queries are performed as described in Section III, and the unique tokens found are stored in a list for each string parameter. The starting test case is taken, and the string values in it are replaced using the next token on the list for each respective parameter. The new test case is then executed to see if the branch is covered. If the branch was not covered, this step is repeated, until either the branch is covered, or web values on each parameter list are exhausted.

Phase 3. Further Evolutionary Searches Using Seeds

Since Phase 2 does not involve any evolutionary search, it does not allow for further branches to be covered that may be executed by simple crossover or mutation of test cases that rely on a string value found by a web query. For example, a string value corresponding to a web token may only need to be altered slightly through mutation to cover some hitherto uncovered branch. Or, some deeply-nested branch may be dependent on a web string value, but require some other aspect of the test case to be altered – *e.g.* through the insertion of an additional method call into the test case, or the alteration of a method parameter of a non-string type. Phase 3 allows for additional 'corner-case' branches of this type to be covered, with a further Evolutionary Search in which the first generation is seeded with test cases found to executed new branches from Phase 2. The generation size for the search consists of 100 individuals as for Phase 1. If there are more than 100 potential seeds, 100 seeds are chosen at random from the pool for the first generation. If there are fewer than 100 seeds, the population is filled up with randomly-generated test cases. The search proceeds for 100 generations for each uncovered branch.

A schematic of the overall process can be seen in Figure 4, with examples of test cases produced at each stage for the program of Figure 1. Phase 1 fails to generate a test case

Table I
CLASSES TESTED USING THE APPROACH

Project *(Source code URL)*	**Class**	**LOC**	**Branches**	**String data types validated**
Chemeval *(chemeval.sf.net)*	org.openscience.cdk.index.CASNumber	102	7	CAS registry numbers
Conzilla *(www.conzilla.org)*	se.kth.cid.identity.MIMEType	107	15	MIME types
	se.kth.cid.identity.PathURN	60	12	Path URNs
	se.kth.cid.identity.ResourceURL	72	15	Resource URLs
	se.kth.cid.identity.URI	228	41	URIs
	se.kth.cid.identity.URN	60	9	URNs
Efisto *(efisto.sf.net)*	com.efisto.util.Util	244	30	Dates in the java.text.SimpleDateFormat format 'dd.MM.yyyy' and 'EEE, dd MMM yyyy HH:mm:ss zzz'
GSV05 *(gsv05.sf.net)*	stempeluhr.validation.TimeChecker	82	8	24 hour times
JXPFW *(jxpfw.sf.net)*	org.jxpfw.util.CLocale	81	10	POSIX locale identifiers
	org.jxpfw.util.InternationalBankAccountNumber	481	52	Bank identifier codes (BICs) International bank account numbers (IBANs), IBAN country codes
LGOL *(lgol.sf.net)*	uk.gov.tameside.apps.validation.DateFormatValidator	105	9	Dates in the format 'dd/mm/yyyy'
	uk.gov.tameside.apps.validation.NumericValidator	86	9	Strings that represent integers
	uk.gov.tameside.apps.validation.PostCodeValidator	162	16	UK postcodes
OpenSymphony *(www.opensymphony.com)*	webwork.examples.userreg.Validator	150	24	Email addresses and US social security numbers (SSNs)
PuzzleBazar *(code.google.com/p/puzzlebazar)*	com.puzzlebazar.client.util.Validation	80	24	Email addresses
TMG *(tmgerman.sf.net)*	net.sf.dblp2db.dblpstat.db.fields.Isbn	238	36	International standard book numbers (ISBNs)
	net.sf.dblp2db.dblpstat.db.fields.Month	164	18	Month names ('January', 'February', etc.)
	net.sf.dblp2db.dblpstat.db.fields.Year	75	9	Four digit years
WIFE *(wife.sf.net)*	com.prowidesoftware.swift.model.BIC	84	12	Bank identifier codes (BICs)
	com.prowidesoftware.swift.model.IBAN	172	26	International bank account numbers (IBANs)
Total		2,833	382	

that executes the true branch from line 15 of the class – failing to find an appropriate string in the 'DD/MM/YYYY' format. The method call sequence found with the best fitness for this branch is used in Phase 2 for the injection of string values found from the web queries for the parameter. The web value '7/4/2010' is injected as a string value, and the target is covered. This test case is then used to seed the discovery of test cases covering further unexecuted branches in Phase 3. The test case is mutated through the insertion of an additional digit to the month, resulting in an invalid date that covers the previously uncovered false branch from line 15.

V. EMPIRICAL EVALUATION

An empirical study was performed with two objectives; the first to evaluate the effectiveness of the web queries in finding appropriate string values, and second, to evaluate the effect of the approach on branch coverage. The research questions addressed by the study were therefore as follows:

RQ1. Do web searches using identifiers result in pages containing appropriate string values? Which types of web search query are the most effective in finding appropriate strings?

The first research question concerns the foundation of the approach. Can the use of web queries formed from program identifiers result in URLs containing examples for the string parameters concerned? For example, given an parameter representing an email address, do any of the tokens found using the generated web queries correspond to valid email addresses, and if so how many? Is there a particular type of query formulation (*i.e.* using prefixes, pluralisation or quotes) that is more effective in finding valid examples for an input parameter of a string type?

RQ2. What is the effect on coverage when using string values found using web queries?

Does the incorporation of web values improve the coverage of a program, when compared to standard search-based generation of test cases for object-oriented classes, and if so when and by how much?

The research questions were addressed using classes drawn from open source code, as described in the next section.

A. Case Studies

The case studies investigated in the empirical study concern input validators, program routines that are widely found in web and GUI applications to check inputs entered by an end user. Such case studies are ideal for use in evaluating *Delver*, since they tend to perform relatively complex operations and checks on different types of strings, for which generating valid inputs by standard search-based techniques is challenging.

Twenty Java classes were studied from 10 open source projects, which contained 24 different input validation routines for various types of string, along with further code involving string parameters, which was also tested. Table I provides more in-depth detail. 'Chemeval' is a chemical evaluation framework that examines molecular structure to

assist in hazard assessment. One class was tested, which is responsible for representing so-called 'CAS numbers', unique identifiers assigned by the Chemical Abstracts Service to chemical substances. A CAS number is a string consisting of up to 10 digits, separated by hyphens, the last digit serving as a check digit. CAS numbers begin at '50-0-0', the number for formaldehyde, while water is '7732-18-5'. 'Conzilla' is a knowledge management tool. Six classes were tested, including one representing valid MIME types, whilst the other five are responsible for validating and manipulating different types of URI. 'Efisto' is a tool for sending files via a web location. One class was tested, involving the manipulation and validation of dates supplied as strings in two different formats. 'GSV05' is a mobile attendance recorder. One class was tested, for validating 24 hour times supplied as strings. 'JXPFW' stand for 'Java eXPerience FrameWork', a utility library. Two classes were tested, which validate and manipulate of POSIX locale identifiers, Bank Identifier Codes (BICs) and International Banking Account Numbers (IBANs). 'LGOL' is a framework designed to assist Java application development for local governments in the UK. Three classes were tested, involving string inputs representing dates in the format 'dd/mm/yyyy', integer numbers and UK postcodes. 'OpenSymphony' is a web development framework. A utility class was tested for validating email addresses and US social security numbers (SSNs). 'PuzzleBazar' is a web-based platform for uploading and playing puzzles. One class was tested, which validates email addresses. 'TMG' stands for 'Text Mining for German documents' and contains classes for interfacing with the DBLP research publication database. Four classes were tested, involving the validation of International Standard Book Numbers (ISBNs), month names and year numbers. Finally, 'WIFE' is a framework for managing SWIFT messages between international banks. Two classes were tested, involving the validation of BICs and IBANs.

B. Answers to Research Questions

RQ1. Do web searches using identifiers result in pages containing appropriate string values? Which types of web search query are the most effective in finding appropriate strings?

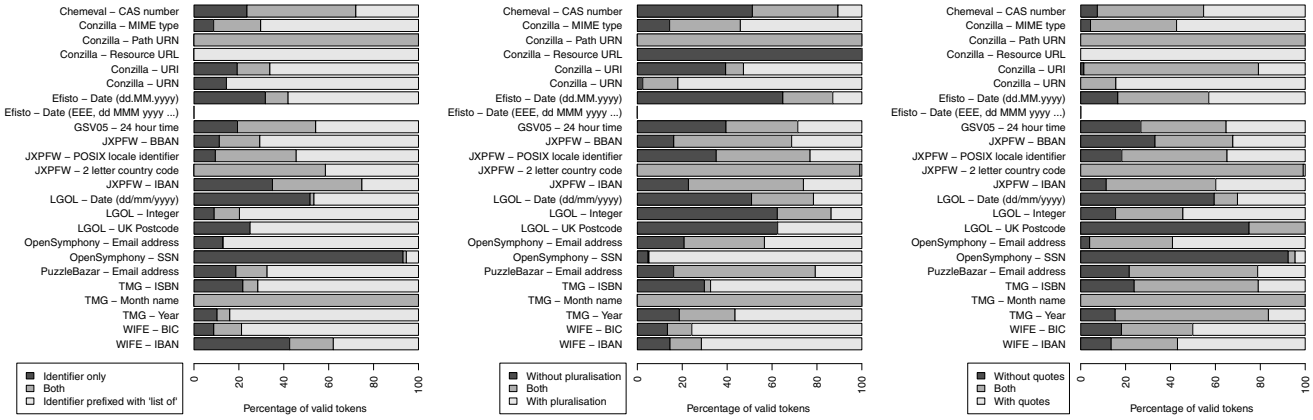
In order to answer this question, web values were run through the validation routines in each project, and the number of valid values found recorded. The web values assessed were those found as a result of web queries generated from the identifiers related to the string parameter to each validation routine in question.

The results for each of the 24 types can be found in Table II. The table shows that valid string values were found in every case apart from one of the date string types in the Efisto project. The web pages returned by the search engine *did* involve dates of this format, but due to whitespace tokenisation procedure in *Delver*, and the use of space characters in the date format, the relevant portion of text was always split into different tokens. This resulted in the string parameter being set to only a part of the overall valid string, during phase 2 of the test case generation process. Furthermore, the common representation for a UK postcode also involves a space, and as such the figure for the number of valid post codes is restricted to the smaller number of postcode examples found without spaces in results with LGOL.

Table II
AGGREGATED RESULTS OF WEB QUERIES FOR EACH STRING TYPE
'Valid' is the number of web values found for each string type that passed the corresponding validity check routine in the project from the total number of values found. The 24-hour validation routine of 'GSV05' was found to contain a bug, so a secondary figure is reported in brackets which refers to the number of valid values following correction of the bug

Project	String type	Valid	Total
Chemeval	CAS number	16,869	181,805
Conzilla	MIME type	7,065	124,924
	Path URN	1	216,403
	Resource URL	3	125,218
	URI	6,457	112,621
	URN	83	87,884
Efisto	Date (dd.MM.yyyy)	1,839	222,045
	Date (EEE, dd MMM yyyy HH:mm:ss zzz)	0	283,458
GSV05	24 hour time	985 (1,058)	112,239
JXPFW	BBAN	9,063	89,873
	POSIX locale identifier	2,108	151,310
	2 letter country code	234	138,266
	IBAN	123	101,896
LGOL	Date (dd/mm/yyyy)	116	148,594
	Integer	4,275	170,020
	UK Postcode	8	182,353
OpenSymphony	Email address	14,776	123,169
	SSN	22,293	178,059
PuzzleBazar	Email address	439	156,636
TMG	ISBN	10,256	139,099
	Month	12	253,308
	Year	22,436	126,102
WIFE	BIC	11,995	185,692
	IBAN	1,023	209,252

High numbers of valid inputs were found in all other cases, with a few exceptions. The representation for a Path URN used in Conzilla is extremely rare, and as such only one value was found – an example documented in the Path URN specification itself. The representation of a resource URL prefix of '`res://`' is similarly rare, and only 3 values were found. The number of unique values for months appears low relative to the other types, but the complete set of 12 month names, 'January' through to 'December', was successfully enumerated by the web searches. Furthermore, as discovered later when evaluating test cases generated as part of RQ2, the code for validating 24-hour time strings was found to contain a bug. Table II therefore records two values, the second value for the corrected version of the code, which successfully validated a further 73 strings.



(a) With and without the prefixing the query with 'list of' *(b) With and without pluralisation of identifiers* *(c) With and without the use of quotes*

Figure 5. The percentage of unique valid string values found using different query formulations (as described in Section III-A). In each case, the 'both' category refers to the percentage of valid values found independently by both types of query being compared

In order to determine which types of query were most effective, valid tokens were placed into different sets according to the strategy used and counted. Figure 5a shows the proportion of valid tokens found when using the 'list of' prefix in the query as opposed to not doing so. With the exception of SSN strings for the OpenSymphony project, the majority of valid strings were found when the 'list of' prefix was used. For path URNs (Conzilla) and month names (TMG), all valid values were returned by both types of query; *i.e.* with and without the prefix. Pluralization of identifiers has no such obvious trend, as seen in Figure 5b. All valid values were found without pluralisation in the case of resource URLs (Conzilla). Neither does a clear trend exist with the use of quotes, as seen in Figure 5c, where the results depend heavily on the identifiers string type concerned.

In conclusion to this research question, appropriate values can be found by performing web queries based on program identifiers related to string parameters. The number of appropriate values may vary in relation to a number of factors such as the formatting of the string (*i.e.*, the inclusion of spaces), and whether the string type relates to a common concept for which examples are plentiful on the web. However, no specific query formulation stands out as being a clearly better than any of the others; instead the inclusion of all web query variants helped increased the number of unique valid values found overall.

RQ2. What is the effect on coverage when using string values found using web queries?

The approach implemented in the Delver tool was compared against extended use of $eToc^{+}$. For each uncovered branch, $eToc^{+}$ continues searching until there has been no improvement in the best fitness value found in the last 1,000 generations, *i.e.* the search had stagnated. As such, each uncovered branch gets at least 100,000 fitness evaluations, with the possibility of more if progress is being made.

Table III
BREAKDOWN OF PERFORMANCE BY BRANCH

'Sig.' denotes the number of branches for which the success rate of coverage was significantly improved using *Delver* compared to $eToc^{+}$ alone (to search stagnation). 'New' is the number of branches that were only ever covered using *Delver*, *i.e.* were never covered using $eToc^{+}$ alone. 'Total' refers to the total number of branches in each class

Project	Class	Sig.	New	Total
Chemeval	CASNumber	3	3	7
Conzilla	MIMEType	0	0	15
	PathURN	8	5	12
	ResourceURL	7	3	15
	URI	0	0	41
	URN	4	1	9
Efisto	Util	5	2	30
GSV05	TimeChecker	2	2	8
JXPFW	CLocale	1	0	10
	InternationalBankAccountNumber	7	2	52
LGOL	DateFormatValidator	1	1	9
	NumericValidator	0	0	9
	PostCodeValidator	0	0	16
OpenSymphony	Validator	0	0	24
PuzzleBazar	Validation	1	0	24
TMG	Isbn	10	10	36
	Month	2	1	18
	Year	0	0	9
WIFE	BIC	0	0	12
	IBAN	0	0	26
Total		51	30	382

Table III provides a breakdown of branches which experienced significantly improved coverage using the web value approach. The number of times a branch was successfully covered in each of the 50 runs for *Delver* and extended $eToc^{+}$ searches was compared using a two-sided Fisher's Exact Test at a confidence level of 0.999. The test indicated a significant difference in the two sets of 50 runs with each tool for 51 branches. In each significant case, *Delver* covered the branch in a higher number of runs than $eToc^{+}$. That is, 30 of these 51 branches were covered by *Delver* but never by $eToc^{+}$. $eToc^{+}$ run to stagnation does not cover any branches

Table IV
COVERAGE USING THE DIFFERENT ALGORITHMS

The performance of *Delver* compared to the use of *eToc*$^+$ alone – without the benefit of web searches, and run to stagnation (no improvement in fitness for the last 1,000 generations). Where not tied, the figure for the tool with the overall highest coverage is shown in bold

Project / Class	**Mean Coverage % (Standard Deviation)**			
	Delver			*eToc*$^+$
	Phase 1	Phase 2	Phase 3	
Chemeval				
CASNumber	57.1 (0.0)	100.0 (0.0)	**100.0** (0.0)	57.1 (0.0)
Conzilla				
MIMEType	86.7 (0.0)	86.7 (0.0)	86.7 (0.0)	86.7 (0.0)
PathURN	0.0 (0.0)	50.0 (2.9)	**60.2** (7.6)	0.5 (3.5)
ResourceURL	36.8 (5.6)	66.7 (3.7)	**82.1** (7.9)	41.3 (5.4)
URI	39.4 (4.2)	46.3 (0.3)	**46.3** (0.0)	46.2 (1.0)
URN	42.2 (7.9)	75.6 (4.4)	**88.4** (2.2)	54.4 (12.9)
Efisto				
Util	70.3 (1.0)	86.7 (0.0)	**86.7** (0.0)	72.3 (2.1)
GSV05				
TimeChecker	78.3 (5.4)	100.0 (0.0)	**100.0** (0.0)	75.0 (0.0)
JXPFW				
CLocale	89.4 (4.2)	100.0 (0.0)	**100.0** (0.0)	95.6 (4.9)
InternationalBankAccountNumber	77.6 (3.7)	97.6 (1.0)	**97.7** (0.7)	86.3 (2.4)
LGOL				
DateFormatValidator	88.9 (0.0)	100.0 (0.0)	**100.0** (0.0)	88.9 (0.0)
NumericValidator	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
PostCodeValidator	81.3 (0.0)	81.3 (0.0)	81.3 (0.0)	81.3 (0.0)
OpenSymphony				
Validator	51.7 (2.6)	54.2 (0.0)	54.2 (0.0)	54.2 (0.0)
PuzzleBazar				
Validation	83.6 (4.8)	99.5 (1.3)	**99.5** (1.3)	96.3 (2.5)
TMG				
Isbn	66.9 (3.0)	91.7 (0.0)	**97.0** (1.1)	69.4 (0.0)
Month	88.9 (0.0)	100.0 (0.0)	**100.0** (0.0)	89.0 (0.8)
Year	77.8 (0.0)	77.8 (0.0)	77.8 (0.0)	77.8 (0.0)
WIFE				
BIC	95.3 (4.1)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
IBAN	89.2 (1.5)	92.3 (0.0)	**92.3** (0.0)	92.0 (1.0)
Average			**87.5**	73.2

that *Delver* does not. It achieves a higher success rate than *Delver* for covering one particular branch (covering it in all 50 runs – *Delver* covers its only 41 times), but the difference was not significant at the 0.999 level.

Results comparing *Delver* with extended *eToc*$^+$ searches for branch coverage of each individual class are shown in Table IV. As the table shows, coverage was improved for 15 of the 20 classes, comparing the final coverage figure for *Delver* after phase 3 with that of *eToc*$^+$ run to search stagnation (far right column). Coverage levels were identical for the remaining six classes. The class experiencing the biggest improvement was the `PathURN` class of the Conzilla case study. With evolutionary search alone, it is difficult to obtain any coverage at all. The constructor of the class calls the `URN` superclass, and if the string is not a valid URN, an exception is thrown before any of the code in `PathURN` can actually be executed. Large improvements were also obtained for `CASNumber` (Chemeval), `ResourceURL` and `URN` (Conzilla), `Isbn` and `Months` (TMG). In each case, a string value found via a web search for the parameter led to the coverage of several additional nested branches. These values could not be found using evolutionary search alone.

The *Delver* approach was also responsible for finding a bug in GSV05. After tokenising the inputted string, the check for a 24-hour time includes the branching sub-condition '`minute > 0`', which should instead be correctly written '`minute >= 0`'. This bug was exposed by a test case from each of the 50 runs of Delver, which incorrectly deemed the web value '`8:00`' to be invalid. By contrast, *eToc*$^+$ run to search stagnation, never found a test case that could be used to expose this particular problem.

In conclusion for this research question, the use of web values has the effect of improving test coverage. Furthermore, a bug was exposed with *Delver* that could not be exposed using the standard evolutionary approach alone.

VI. RELATED WORK

Despite the large amount of work devoted to structural test case generation, there has been comparatively little work on generating string inputs. Previous work in search-based testing has largely ignored the problem, treating strings as fixed-length arrays of characters [16], or has required specialist 'generators' to be written [9]. Alshraideh and Bottaci [17] proposed new fitness functions for string generation, but found the improvements made were only moderately successful compared with the seeding of string literals found in the source code of the program into the first generation of the test data search. However, useful string literals may not always be available. In contrast, this paper proposes an approach where useful strings may be incorporated into the search-based test case generation process that are sourced from Internet web pages.

Dynamic Symbolic Execution (DSE) [1], [2] executes a program under test simultaneously on symbolic and concrete inputs in order to exercise branch constraints. Symbolic PathFinder [24] and Reggae [25] are two DSE tools that have handling for string constraints. There are further solvers available with the attempt for string constraint solving, such as HAMPI [26] and Kaluza [27] which currently lack integration with DSE tools. However, like search-based techniques using structural information only, DSE tends not to generate realistic strings.

Realistic test cases are important for human comprehension of test cases and lowering 'oracle costs' – manual evaluation whether a test case produces the correct result or not. McMinn *et al.* discuss realistic test input generation in relation to the oracle cost problem and propose using human-provided seeds for search-based test case generation [14]. Fraser and Zeller [28] propose the mining of software repositories to incorporate common usage patterns of APIs in object-oriented test case generation. Bozkurt and Harman [29] investigate realistic test data generation for service-oriented software. They propose using the outputs of other known and existing web services for re-use as inputs to the services under test (for example, an ISBN from a book web

service). However, none of these works use web queries and the extraction of tokens from web pages to improve structural test data generation involving string input, which is the contribution of this paper.

VII. Conclusions and Future Work

This paper has introduced an approach for generating test inputs for string types by performing web queries. The web values obtained are used in conjunction with evolutionary search. An empirical study presented in the paper showed that valid string values can be obtained using this method, and those values can be used to improve coverage of classes when compared to running evolutionary searches to stagnation without the benefit of web queries.

There are several avenues for future work, including the incorporation of the wealth of recent work in the area of program identifier analysis, which may help refine the web queries used and the corresponding results obtained. The approach to splitting identifiers into constituent words used in this paper is based on underscoring and camel casing. However more advanced algorithms can be employed, for example that of Madani *et al.* [30] which uses techniques from the field of speech recognition. The work of Lawrie *et al.* [31] may help in expanding programmer-shortened words and abbreviations back into their original form, for example the use of '`str`' instead of '`string`'. The extraction of domain information with respect to program identifiers may also prove useful in constructing web queries and removing identifier words that are not useful search terms. Abebe and Tonella [32] perform a deeper analysis of identifiers to extract more general concepts, while Binkley *et al.* [33] propose algorithms for improving identifier informativeness using part of speech information.

The use of web queries to find example strings may also aid the DSE approach to test case generation, and future work will also evaluate the method in the context of symbolic execution tools.

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

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *Programming language design and implementation (PLDI 2005)*, 2005, pp. 213–223.

[2] N. Tillmann and J. de Halleux, "Pex – white box test generation for .NET," in *Tests and Proofs (TAP 2008)*, 2008, p. 134–153.

[3] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[4] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[5] O. Buehler and J. Wegener, "Evolutionary functional testing," *Computers and Operations Research*, vol. 35, no. 10, pp. 3144–3160 2008.

[6] A. Marchetto and P. Tonella, "Using search-based algorithms for Ajax event sequence generation during testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.

[7] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *International Conference on Software Engineering (ICSE 2003)*, 2003, pp. 38–48.

[8] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Genetic and Evolutionary Computation Conference (GECCO 2005)*, 2005, pp. 1021–1028.

[9] P. Tonella, "Evolutionary testing of classes," in *International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[10] J. Ribeiro, M. Zenha-Rela, and F. de Vega, "Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software," *Information and Software Technology*, vol. 51, pp. 1534–1548, 2009.

[11] G. Fraser and A. Arcuri, "Whole test suite generation," in *International Conference on Software Quality (QSIC 2011)*, 2011.

[12] N. Tracey, J. Clark, and K. Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *International Workshop on Dependable Computing and Its Applications*, 1998, pp. 169–180.

[13] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *International Conference on Software Testing, Verification and Validation (ICST 2010)*, 2010, pp. 205–214.

[14] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *1st International Workshop on Software Test Output Validation (STOV 2010)*, 2010, pp. 1–4.

[15] *The SEBASE repository*, `http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/`

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

[17] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification and Reliability*, pp. 175–203, 2006.

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

[19] R. Baeza-Yates and B. Ribeiro-Neto, "Modern Information Retrieval", Addison Wesley, 1999.

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

[21] *Bing Developer Tools*, `http://www.bing.com/toolbox/bingdeveloper`

[22] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *International Conference on Genetic Algorithms and their Application (ICGA 1987)*, 1987, pp. 14–21.

[23] D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," in *International Conference on Genetic Algorithms (ICGA 1989)*, 1989, pp. 116–121.

[24] C. S. Păsăreanu, P. C. Mehlitz, D. H. 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 *International Symposium on Software Testing and Analysis (ISSTA 2008)*, 2008, pp. 15–26.

[25] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Reggae: Automated test generation for programs using complex regular expressions," in *24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009.

[26] V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A string solver for testing, analysis and vulnerability detection," in *CAV*, 2011, pp. 1–19.

[27] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[28] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *International Conference on Software Testing, Verification and Validation (ICST 2011)*, 2011, pp. 80–89.

[29] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," in *International Symposium on Service-Oriented System Engineering*, 2011.

[30] N. Madani, L. Guerrouj, M. Di Penta, Y. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *European Conference on Software Maintenance and Reengineering (CSMR 2010)*, 2010, pp. 68–77.

[31] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Working Conference on Reverse Engineering (WCRE 2010)*, 2010, pp. 3–12.

[32] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *International Conference on Program Comprehension (ICPC 2010)*, 2010, pp. 156–159.

[33] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Working Conference on Mining Software Repositories (MSR 2011)*, 2011, pp. 203–206.