2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost

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

***Abstract*—The frequent non-availability of an automated oracle means that, in practice, checking software behaviour is frequently a painstakingly manual task. Despite the high cost of human oracle involvement, there has been little research investigating how to make the role easier and less time-consuming. One source of human oracle cost is the inherent unreadability of machine-generated test inputs. In particular, automatically generated string inputs tend to be arbitrary sequences of characters that are awkward to read. This makes test cases hard to comprehend and time-consuming to check. In this paper we present an approach in which a natural language model is incorporated into a search-based input data generation process with the aim of improving the human readability of generated strings. We further present a human study of test inputs generated using the technique on 17 open source Java case studies. For 10 of the case studies, the participants recorded significantly faster times when evaluating inputs produced using the language model, with medium to large effect sizes 60% of the time. In addition, the study found that accuracy of test input evaluation was also significantly improved for 3 of the case studies.**

***Keywords*-Search-Based Testing, Strings, Language Model, Crowd-Sourced Human Study**

I. Introduction

In many real cases, software behaviour must be checked by a human. While recent work on test data generation allows for the possibility of automatically generating inputs to cover all the branches of a program, the outputs of a software system must still be checked against those inputs in order to assess that the system is functioning as intended.

However, conventional approaches to automatic test data generation typically ignore the overheads of human effort involved in manually checking the inputs produced. This effort adds a cost to the testing process, referred to as the *human oracle cost* [1], [2]. The traditional goal of an automatic test input generator is the achievement of structural code coverage only [3], [4], [5], [6], and not also the simultaneous reduction of human oracle cost.

One source of human oracle cost is the difficulty of reading machine-generated inputs. In particular, string values generated by automatic test input generators often look like arbitrary sequences of characters. This results in test scenarios that are hard to interpret and test cases that are time-consuming to manually evaluate.

In this paper, we apply a natural language model to the automatic generation of string inputs, with the aim of generating readable test values that are easy for humans to comprehend. Language models have been used in a variety of areas including natural language processing [7], where one of their applications is to assist with automatic translation [8], and in speech processing [9], where they are used to choose between the possible outputs from a speech recognizer. In software engineering, they have been used to improve code completion in the Eclipse IDE [10].

A language model assigns a score to a string reflecting the "likeness" of that string to those occurring in a natural language. We show how this score can be used to form an additional component of the fitness function used in search-based structural test input generation. Once an input has been found to cover a branch, the language model component of the fitness function can be harnessed to improve string inputs from the perspective of human readability, evolving them from seemingly random combinations of characters to strings that involve common letter sequences and characteristics of comprehensible text.

We present an empirical study in which we evaluated the capabilities of the language model test input generation approach with human judgements. Programmers were invited to evaluate tests cases for a series of 17 Java case studies from open source projects. The study found that test inputs generated by the language model approach took significantly less time to evaluate for 10 case studies, with medium to large effect sizes recorded in 6 cases. For 3 case studies, the accuracy of test input evaluation was also significantly improved.

The contributions of this paper, therefore, are as follows:

1) A technique for incorporating a language model into the automatic test input generation process for branch-covering string inputs (Section III).
2) The results of a human study in which the language model technique is compared with a conventional, non-informed approach to generating branch-covering test suites, revealing cases where human participants were both faster and more accurate in making oracle judgements (Sections IV and V).

We begin by describing the functioning of the language model used in this paper, and how it is incorporated into the search-based input generation process.

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.11

II. Language Models

A statistical language model assigns a probability score to a string that estimates the likelihood of that string occurring in the language it models. A good language model for English, for example, assigns higher probability scores to strings that resemble well-formed words, such as "*testing*", and lower scores to strings that do not, e.g. "*Qu5$-ua*".

Language models are widely used in natural language and speech processing for a wide range of tasks, including machine translation [8] and automatic speech recognition [9]. The majority of applications use word-based language models, which model the language as sequences of words. In this paper, a character-based language model is used, where the language is represented as a sequence of characters. The same basic approach is used by both word and character-based models. The explanation provided here, however, is focused around the character-based approach later incorporated into test input generation in Section III.

Let c_1^n be a sequence of n characters (c_1, c_2, ... c_n). A language model aims to assign a value to the probability $P(c_1^n)$. This can be decomposed using the chain rule of probability, allowing the probability of each character c_i to be estimated based on the characters that preceded it in c_1^n:

$$\begin{aligned} P(c_1^n) &= P(c_1)P(c_2|c_1)P(c_3|c_1^2)...P(c_n|c_1^{n-1}) \\ &= \prod_{i=1}^{n} P(c_i|c_1^{i-1}) \end{aligned} \quad (1)$$

where $P(c_i|c_1^{i-1})$ is the probability of character c_i following the sequence c_1^{i-1}.

A language model estimates these probabilities by analyzing a corpus. The probability $P(c_i|c_1^{i-1})$ can be estimated by identifying all occurrences of the string c_1^{i-1} to find the proportion that are followed by c_i. However, the number of possible strings means that many of these sequences will not be found, even in an extremely large corpus, making these probabilities impossible to estimate directly. A language with c possible characters has c^n possible sequences of n characters. For example, if we assume that there are 26 characters in English (i.e. ignoring case, punctuation and whitespace) the number of possible 5 character sequences is over 11 million.

Consequently language models approximate the probability of strings by combining the probabilities of shorter sequences, for which more reliable probabilities can be inferred from the corpus. One approach is to estimate the probability of each character based only on the character that immediately precedes it:

$$P(c_1^n) \approx \prod_{i=1}^{n} P(c_i|c_{i-1}) \quad (2)$$

This type of language model is known as a bigram model. However, even when using a bigram model some pairs of characters will not be seen in large corpora, and in these cases the probabilities are estimated by combining the probabilities of individual characters, i.e. $P(c_i)$, computed using smoothing and back-off techniques (for more details, the reader is referred to the references [9], [11]).

Bigram	Probability	Source
te	0.08548796	Direct
es	0.10209079	Direct
st	0.17185259	Direct
ti	0.07612985	Direct
in	0.30709963	Direct
ng	0.15313497	Direct

***score("testing")* = 0.17665935**

Bigram	Probability	Source
Qu	0.95987654	Direct
u5	0.00000009	Inferred
5$	0.00000005	Inferred
$-	0.00074450	Inferred
-u	0.00247280	Direct
ua	0.02245566	Direct

***score("Qu5$-ua")* = 0.00079785**

Figure 1. Computing language model scores for two strings. The word "*testing*" receives a higher score than the string "*Qu5$-ua*". For "*testing*", all bigram probabilities can be found directly in the corpus, whereas some bigrams for "*Qu5$-ua*" are not present and are inferred from probabilities computed for each individual characters of the bigram separately.

In general longer strings are less likely to occur than shorter ones and language models assign them lower probabilities. To avoid bias in favour of shorter strings the probability generated by the language model is normalized by taking the geometric mean, i.e. the score assigned to a string, $score(c_1^n)$, is computed as

$$score(c_1^n) = P(c_1^n)^{\frac{1}{n}} \quad (3)$$

Figure 1 shows the scores assigned by our bigram language model to the strings "*testing*" and "*Qu5$-ua*". The SRILM toolkit [12] was used to learn the model and the text used to train it was an electronic version of the classic novel *Moby Dick* [13] freely available at Project Gutenberg (*http://www.gutenberg.org/ebooks/2701*). This text contains 215,133 words and 1,235,150 characters, which is more than adequate to train the language model.

III. Incorporating a Language Model Into Search-Based Test Input Generation

Search-based test input generation has been applied extensively to the generation of structural test data [3]. The main feature of a search-based approach is the formulation of a *fitness function*. The fitness function underpins the test goal, rewarding inputs close to fulfilling the goal with good fitness values, while punishing inputs that are far away with weak fitness values. A metaheuristic search technique, such as an evolutionary or local search algorithm [4], is used to optimize the fitness function. The search favours exploration around input values with the best fitness values – on the assumption that they lie in the vicinity of inputs with even better fitness – with the aim of finding inputs that lead to the satisfaction of the current test goal of interest.

The conventional fitness function for generating inputs to cover individual branches is concerned with the control structure of the program and the values of variables at decision points only. The fitness function is to be minimized, with a zero fitness value representing the global optimum.

```
1   String toCamel(String str) {
2     StringBuffer sb = new StringBuffer();
3     boolean wasUnderline = false;
4     for (int i = 0; i < str.length(); i++) {
5       char c = str.charAt(i);
6       if (c == '_') {
7         wasUnderline = true;
8         continue;
9       }
10      if (wasUnderline) {
11        sb.append(Character.toUpperCase(c));
12        wasUnderline = false;
13        continue;
14      }
15      sb.append(Character.toLowerCase(c));
16    }
17    return sb.toString();
18  }
```

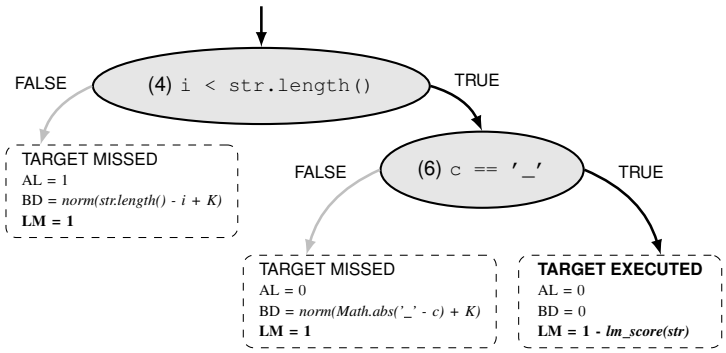

Figure 2. Fitness computation for the *toCamel* method and generation of a readable string that covers the true branch from node 6. The fitness function contains the usual approach level (AL) and branch distance (BD) components (*K* is a positive constant set to 1 in this paper), along with an extra metric from the language model (LM). LM is 1 until the branch is covered, at which point it assigns scores to the string `str` using *lm_score*, the probability score between 0 and 1 generated for `str` by the language model. The higher the value of *lm_score*, the lower the value of LM, and the higher the similarity `str` has with strings in the language.

The *approach level* (AL) scores how far down the control dependency graph the input penetrated with respect to a target branch. For example, with Figure 2 and execution of the true branch at line 6, inputs that do not enter the loop score 1, while inputs that reach line 6 score 0. Added to the approach level is the normalized *branch distance* (BD) metric, which scores how close the input was to taking the alternate desired branch. Example calculations are shown in the figure, and more details can be found in [3], [4], [14].

With this fitness function, the goal of the search is to cover a particular branch with any input that can be found. However, the inputs found tend to look random from a human perspective, and the information encoded in string inputs requires work to decipher due to arbitrary character sequences. For example, "*#qp}ˆbkJ';_ir9*" was generated for the *toCamel* method of Figure 2 during experiments reported later in Section V. Due to the frequent lack of an automated oracle, a human must often check that the outputs of a piece of software match those expected for some generated input. However, the harder the inputs are to comprehend, the more time-consuming and error-prone the task is likely to be. The *toCamel* method converts a program identifier string using the under_scoring style of joining words to the camelCase format, where the first letter of each word (bar the first) is a capital letter. The conversion process involves finding each underscore in a string, removing it, and capitalizing the character immediately after it. Thus the correct output for "*#qp}ˆbkJ';_ir9*" is "*#qp}ˆbkJ';Ir9*". From a human perspective, readable strings such as "*my_string*" would be preferred and should be automatically generated.

Our technique incorporates the language model described in the last section into the fitness function for covering individual branches. Language model probability scores can be viewed as a measure of "likeness" or similarity of a string to natural words. As such, the probability score forms an ideal output of a fitness function, since it can be used to guide the search towards more natural and inherently readable strings. Language model scores begin to have an impact on fitness once a branch is covered, as shown in Figure 2. Before an input has been found to cover a branch, the language model fitness component (LM) is always 1, and is added to AL and BD. Once the target branch is covered – the point at which the search would normally terminate – the search instead continues to optimize the input for language model score. With the AL and BD scores both 0, the LM component returns $1 - lm(str)$, where str is the string input and $lm(str)$ is the language model function that returns a probability score for the string str. In other words, lower values of LM reflect "better" strings. If the search makes a "move" to improve the readability of the string, but the new string now fails to cover the branch, the fitness function punishes it by scoring for the approach level and branch distance elements again, while the LM score reverts back to 1.

Using LM in the fitness computation means that it is almost impossible for the search to reach the global optimum, and so the search must always be stopped at some suitable fitness evaluations limit. (In this paper, 100,000 fitness evaluations is used as the termination criterion.)

Our work seeks to compare the *conventional approach*, which does not include the LM component (i.e. LM is essentially always 0 for the fitness function described in Figure 2), against the *language model approach*, which includes the LM metric. In contrast to "*#qp}ˆbkJ';_ir9*", generated by the conventional approach for the true branch from line 6, the language model approach generated strings such as "*inererof_yo*" in our experiments. The hypothesis that strings such as the latter are easier to read and quicker for humans to check is tested in an empirical study, detailed in the next section.

IV. Experimental Study Methodology

The primary aim of our empirical study was to discover whether automatically generated branch-covering string inputs were evaluated more quickly and accurately by humans when those strings had been generated with the assistance of a language model. We begin by detailing the case studies used as a basis for generating string inputs to be assessed in the human evaluation.

A. Case Studies

The case studies used in the empirical evaluation were Java methods with string arguments taken from 17 open source projects. A summary can be found in Table I. Since each case study would be the subject of human evaluation, it was required that the operation of each Java method be amenable to being understood from no more than a paragraph of text. The primary reason for avoiding complicated methods was to avoid so-called *fatigue effects* and having participants tire quickly during progression of the study, potentially influencing results or increasing the number of unusable responses. On the one hand, therefore, the methods tend to be relatively simple in nature with few branches. On the other hand, this is reflective of good Java programming style, where short methods are best practice [15].

The projects and methods outlined in Table I are now described in more detail.

Bots'n'Scouts is a multiplayer game, from which one method was used – *lesseqString*. The method checks whether the first string argument `a` is lexicographically less than or equal to another string argument `b`, returning true if so, otherwise false.

CodeHaggis is an Eclipse plugin with code generation capabilities. One method was used in the study, *toCamel*, introduced in the last section as the subject of Figure 2.

Daikon is an invariant generator, well-known to the software engineering research community. Two methods were used. *getClassName* extracts a Java class name from a string, based on the location of the last dot character. *protectQuotations* takes a string and places a backslash in front of each quotation mark.

Germoglio is a compilation of solutions to various programming problem contests. One method was used, *translate*, which translates a word string to "Pig Latin". If the first letter of the string is not a vowel, it is appended to the end of the string. Then – regardless of the first character – the string is appended with "ay". Thus "input" becomes "inputay" and "string" becomes "tringsay".

The project *Jake* manages information about online academic resources. One method was used, *isValidUsername*, which checks whether its string argument is at least three characters long, and consists of alphanumeric characters only.

JavaMail is the email and messaging API for use with Java. The *isSimpleAddress* method checks whether the string argument is a valid URL address, i.e. is free of certain forbidden characters including brackets, square brackets, semi-colons etc. The method *isGroup* checks whether a string is a group address, according to RFC822, i.e. it contains a colon and ends with a semi-colon.

JOX manages data transfer between XML documents and Java beans. The *stripName* method was used, which returns the lowercase version of its argument, with dash, underscore, dot, and colon characters removed.

Muffin is a project for filtering Internet pages. The *containsChar* method used in the study tests if a character argument is present in another supplied string argument.

OpenJDK is the well-known open-source implementation of the Java platform. The method *isHostNameLabel* takes a string as an argument, and returns true if the string is a valid host name – a string of which the first and the last characters are alphanumeric. The remaining characters may be alphanumeric or hyphens. *composeName* takes two strings `name` and `prefix` as arguments. If one of `name` or `prefix` are null or empty, the method returns the non-null/non-empty argument, else the method returns `prefix` appended by a forward slash followed by `name`.

PuzzleBazar is a web-based platform for uploading and playing puzzles. One method was used in the study, which validates email address strings.

Rife is a Java web development framework. The *capitalize* method takes a string and converts the first character to upper case if it is a lower case letter. *encodeClassname* takes a string and converts it to a valid Java class name, by replacing any characters that are not letters, digits or underscores with underscores. *needsUrlEncoding* inspects whether or not a string argument requires encoding by checking whether certain characters are present.

Finally, *Subsonic* is a media streaming application. The *containsIgnoreCase* method checks whether a substring is present in another string, ignoring casing differences.

For the purposes of increasing the number of branch targets for input generation, the return statements of the *isSimpleAddress*, *isGroup*, *containsIgnoreCase* methods, involving the evaluation of boolean expressions, were modified (without changing the method's functionality) to "if" statements containing the expression with corresponding "return true" and "return false" parts.

Each Java method can be placed into one of two categories – *string validation* and *string conversion* routines. Validation routines take a string and return either true or false, and include *containsChar*, *containsIgnoreCase*, *isGroup*, *isHostNameLabel*, *isSimpleAddress*, *isValidUsername*, *lesseqString*, *needsUrlEncoding*, and *validateEmail*. Conversion routines take one or more string inputs and return some string output. These include *capitalize*, *composeName*, *encodeClassName*, *getClassName*, *protectQuotations*, *stripName*, *toCamel* and *translate*.

Table I
CASE STUDIES

Project	Class	Methods used (No. of branches)
Bots'n'Scouts (http://botsnscouts.sourceforge.net)	de.botsnscouts.util.H	lesseqString (2)
CodeHaggis (http://sourceforge.net/projects/codehaggis)	net.sf.haggis.actions.stringConverter.StringConverter	toCamel (6)
Daikon (http://pag.csail.mit.edu/daikon)	daikon.split.SplitterJavaSource	getClassName (2), protectQuotations (4)
Germoglio (http://code.google.com/p/germoglio-programming)	P492	translate (2)
Jake (http://sourceforge.net/projects/jake)	org.jakedb.UserManager	isValidUsername (6)
JavaMail (http://www.oracle.com/technetwork/java/javamail)	javax.mail.internet.InternetAddress	isSimpleAddress (2), isGroup (4)
JOX (http://sourceforge.net/projects/jox)	com.wutka.jox.JOXBeanOutput	stripName (6)
Muffin (http://muffin.doit.org)	sdsu.util.SimpleTokenizer	containsChar (4)
OpenJDK (http://openjdk.java.net)	com.sun.jndi.dns.DnsName com.sun.jndi.toolkit.url.GenericURLContext	isHostNameLabel (4) composeName (4)
PuzzleBazar (http://code.google.com/p/puzzlebazar)	com.puzzlebazar.client.util.Validation	validateEmail (24)
Rife (http://rifers.org)	com.uwyn.rife.tools.StringUtils	capitalize (4), encodeClassname (2), needsUrlEncoding (6)
Subsonic (http://www.subsonic.org)	net.sourceforge.subsonic.service.SearchService	containsIgnoreCase (4)

B. Generation of Inputs

The first phase of the experiments involved generating test inputs for each of the methods, using the conventional approach and the language model informed approach, as described in Section III. Full branch coverage of each method was attempted using a simple (1+1) Evolutionary Algorithm (EA). A (1+1) EA maintains one test input and modifies it through random changes called mutations. If the modified input scores a better fitness than the original, the original is replaced with the modified input. Inputs were generated using the IGUANA toolset [16]. IGUANA is a tool for C programs, but was easily adapted to Java in this instance, since all the methods were static; and if not static, stateless. That is, there was no requirement to generate a method call sequence other than the simple invocation of a constructor. The representation of strings for the search was an array of characters, s, of length 30, followed by an additional integer l, for controlling string length. For example, if $l = 5$, the first five characters of s were used to form a string of length 5. The upper bound of 30 was chosen in order to guarantee feasibility of as many branches as possible over all the case studies employed in the evaluation. In practice, this value could be adjusted if necessary by the tester. Each character of s was in the ASCII printable range of 32–126. Mutations were made at a probability of $\frac{1}{l+1}$ using uniform mutation.

The search for test data involving each branch of each method was given up to 100,000 fitness evaluations. The conventional approach may terminate before this limit if test data is found to cover the branch, whereas the language model approach continues to optimize the branch-covering string input to achieve the best language model score. Due to the stochastic nature of the search algorithm, the search with each approach and Java method branch was repeated 30 times using an identical set of random seeds. This resulted in both approaches covering exactly the same branches, since the search using the language model is identical to the conventional approach up to the coverage of the target branch. During test data generation, 100% branch coverage was achieved for all methods, except for one infeasible branch in *validateEmail*.

C. Human Test Input Evaluation Study

The human study required participants to have expertise in the Java language and complete an online questionnaire involving one of the case study methods. The participant was presented with the method's signature and a paragraph of text describing the operation of the method. (Supplementary information was additionally presented for the *lesseqString* method, in the form of ASCII codes.)

The participant was then required to answer 8 questions based on a selection of string inputs generated for that case study, drawn at random from the repetitions of searches performed as described in the last section. That is, each set of 8 questions featured a mixture of string inputs generated for any of branches of the case study using either the two approaches – conventional or language model.

In order to answer a question, the participant was simply required to type the expected output for the Java method. For a method in the *string validation* category, this would be a boolean value (i.e., the user would be expected to

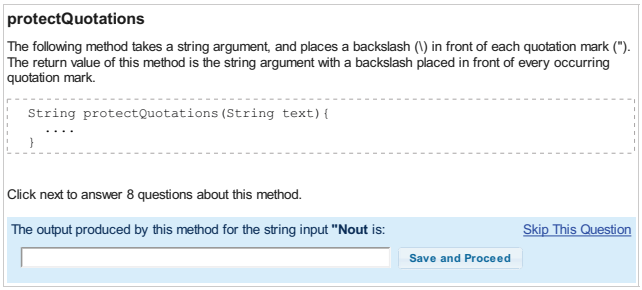

Figure 3. Evaluating string inputs in the online questionnaire system

enter "true" or "false"). For a *string conversion* method, the string return value was required. The time taken from the presentation of the question to the user clicking "next" and having entered their response was recorded, with the response logged internally as "correct" if it matched the actual return value of the method. In order to familiarize the participant with the case study, the first two questions were designated as practice questions, the answers to which were not used in the analysis presented in the next section. Figure 3 shows a screenshot of the questionnaire with an example question. Once a participant had completed a questionnaire for a Java method, they were not allowed to go back and change any answers or re-take with different questions.

Participants were continuously recruited via the CrowdFlower (*http://crowdflower.com*) crowdsourcing website until 250 answers (excluding those to practice questions) had been obtained to questions involving strings generated by each of the two input generation approaches for each Java method. CrowdFlower is one of a number of websites in which tasks or jobs can be uploaded for completion by workers for a fee. Crowdsourcing websites have been used in a number of other software engineering empirical studies, including the investigation of code smells [17], fault localization accuracy [18] and patch maintainability [19]. It has also been used to evaluate human linguistic annotations [20] and Wikipedia article quality [21].

The use of crowdsourcing provides access to a far wider set of participants than would be possible if participants had been identified through personal contacts (such as students or industrial practitioners) and allows judgements to be obtained from them far more quickly than would otherwise be the case. However, control over the selection of study participants is limited and, while we did require participants to have expertise in Java, expertise levels were indicated by the participants themselves. An additional problem is the possibility of participants trying to "game the system" for money by entering rushed and thoughtless responses.

However, these issues can be overcome by analyzing a subset of participants based on an independent metric, a method found to work well in other crowd-sourced human studies [18], [19], [20], [21]. Answers from participants who did not answer 50% or more of the test input questions were automatically discarded and did not count towards the target 250 answers being collated for each Java method and test input generation approach. This removed frivolously-entered responses from the analysis and the need to trust participants' assessments of their own level of ability. (The test input evaluation questions were not intended to challenge the participant's level of ability, and a programmer of even a basic level of competence should have been able to answer the majority of the questions accurately.) As each questionnaire involved a random selection of string inputs generated by both types of approach, this process does not bias our results towards one particular approach or the other.

D. Research Questions

The research questions to be answered by the human empirical study are as follows:

RQ 1. String input generation using the language model. The use of the language model as part of the fitness function is expected to improve the language model scores of the strings generated. Does it, and by how much?

RQ 2. Accuracy of judgements. Is there an improvement in the accuracy of evaluation of strings generated using the language model? That is, do the participants enter the correct expected outputs for a string input produced using the language model more frequently than for strings generated without the use of the language model?

RQ 3. Time to make judgements. Is there a decrease in time for evaluating strings produced using the language model? That is, do human participants enter the correct expected outputs for strings generated using the language model more quickly than those generated without the use of the language model?

V. Experimental Results

RQ 1. String input generation using the language model.

Figure 4 shows the impact of the language model on average probability scores of strings generated using the language model approach and the non-informed conventional approach. The chart shows that the scores for "unimproved" strings generated using the conventional approach are low – as would be expected – and that the evolutionary algorithm can indeed leverage the language model to improve strings to higher language model scores. Probability scores were at the very least doubled, and in the best cases improved by several orders of magnitude.

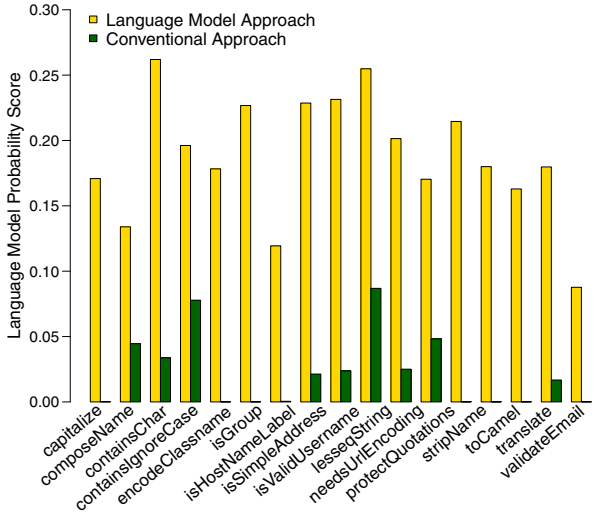

Figure 4. Average language model scores for strings generated using the conventional search-based approach to test input generation compared to those generated with the aid of a language model.

Table II
AVERAGE AND MAXIMUM NUMBER OF FITNESS EVALUATIONS FOR COVERING THE BRANCHES OF EACH JAVA METHOD

Case Study	Av. Evals	Max. Evals
capitalize	5,101.9	5,323
composeName	9,722.8	10,653
containsChar	78.8	86
containsIgnoreCase	6,564.9	7,456
encodeClassname	3,300.3	4,906
getClassName	279	279
isGroup	7,941.9	10,955
isHostNameLabel	243.3	254
isSimpleAddress	153.1	292
isValidUserName	934.1	1,167
lesseqString	196.8	303
needsUrlEncoding	7,670.8	12,179
protectQuotations	554.4	846
stripName	3,952.8	5,186
toCamel	395.8	618
translate	956.7	1,061
validateEmail	50,213.6	71,285

Java methods experiencing high language model scores for their string inputs, such as *containsChar* and *lesseqString*, were generally those involving few constraints regarding the presence of special types of character that do not form elements of natural words in English. Strings for *validateEmail* received the lowest language model score on average. These results may also be explained in terms of the "budget" of fitness evaluations available for improving string readability once a branch had been covered. The search for string inputs for each branch was terminated after 100,000 evaluations, but if a large number of these evaluations were spent covering the branch, few further evaluations would be spent post-branch coverage improving readability using the language model. Table II shows the average and maximum number of fitness evaluations required for covering branches for each case study. The average number of evaluations for the majority case studies was under 10,000 – leaving a budget of at least 90,000 further evaluations for improving string readability. The branches of *containsChar* and *lesseqString* were covered in fewer than 200 evaluations on average – leaving most of the 100,000 fitness evaluations budget for improving readability. However, *validateEmail* consumed just over 50,000 on average, leaving a smaller (but still considerable) number of around 50,000 evaluations for improving readability.

RQ 2. Accuracy of judgements. The human study requires participants to enter the expected output for each input they are presented with. The input string could have been generated using the language model approach or the conventional, non-informed approach. In total, 250 answers were collated involving strings generated by each of the two test input generation approaches for each Java method (i.e., 500 for each Java method in total). Table III shows the percentage of inputs for each approach for which the participant correctly entered the output for the case study method in question. The

Table III
PERCENTAGE OF CORRECT JUDGEMENTS

The percentage of correct judgements for each Java method. A correct judgement is where the participant correctly entered the output for the case study given a string input produced by either the language model approach (**Lang.**) or the conventional, uninformed, approach (**Conv.**). A *p*-value in bold face indicates significance at a confidence level of 95% using Fisher's exact test.

Case Study	Lang. (%)	Conv. (%)	*p*-value
capitalize	74.4	79.6	0.635
composeName	80.4	82.0	0.894
containsChar	94.0	90.0	0.747
containsIgnoreCase	85.6	84.8	0.948
encodeClassname	**97.2**	**74.8**	**0.048**
getClassName	83.6	80.4	0.790
isGroup	95.6	96.8	0.949
isHostNameLabel	87.6	86.4	0.948
isSimpleAddress	94.4	90.4	0.747
isValidUsername	87.2	94.0	0.604
lesseqString	78.4	78.0	1.000
needsUrlEncoding	95.6	96.8	0.949
protectQuotations	88.0	84.8	0.793
stripName	**90.0**	**59.2**	**0.003**
toCamel	**90.4**	**59.2**	**0.003**
translate	88.4	84.8	0.793
validateEmail	72.0	89.6	0.108

table shows no significant difference between the language model approach and conventional approach for the majority of case studies, using Fisher's exact test on the numbers of questions correctly answered at a confidence level of 95%. No significant difference was found for any of the string validation routines. However, three of the string conversion methods did reveal a significant improvement when using the language model – *encodeClassName*, *stripName* and *toCamel*.

In answer to this research question, therefore, the evidence suggests that the language model *can* improve the accuracy of test input evaluation for certain types of case study. There was no evidence to suggest that inputs generated with the language model approach hindered accuracy in any of the cases, i.e. it did not significantly reduce accuracy for any of the methods under consideration.

RQ 3. Time to make judgements. The time for participants to enter the outputs for each input was logged. Table IV shows mean times for all judgements made by participants as well as mean times for correct judgements only (i.e. where the participant entered the correct output for the input). Statistical significance was tested for using the Wilcoxon rank-sum test at a confidence level of 95%. For both views of the data (mean times for all answers and mean times for correct answers only), significantly lower times were recorded for 10 of the 17 Java methods for inputs generated using the language model approach.

The computation of effect sizes, using Vargha and Delaney's $\hat{A}_{12}$ statistic [22], is recorded in Table IV. The guidelines presented in Vargha and Delany's paper class an effect size as *large*, *medium* or *small* as values that are less

Table IV
TIME TO MAKE JUDGEMENTS

Mean times taken for participants to make judgements on inputs for each of the Java methods using the language model (**Lang.**) and the uninformed conventional approach (**Conv.**). Part (a) shows mean times in seconds for judgements on all test cases, whereas part (b) is an average for correct judgements only (i.e. where the participant correctly entered the output for the case study in question). A *p*-value in bold face indicates significance at a confidence level of 95% using the Wilcoxon rank-sum test with the two sets of times obtained with the language model approach and the conventional approach respectively. For the $\hat{A}_{12}$ statistic, * indicates a small effect size , ** a medium effect size and *** a large effect size, according to the guidelines of Vargha and Delaney [22].

(a) All judgements

Case Study	**Lang.** (s)	**Conv.** (s)	***p*-value**	$\hat{A}_{12}$
capitalize	14.1	16.6	0.820	0.506
composeName	20.6	21.7	0.596	0.514
containsChar	**9.1**	**14.3**	**< 0.001**	*** **0.279**
containsIgnoreCase	10.4	10.2	0.877	0.496
encodeClassname	**15.1**	**40.3**	**< 0.001**	*** **0.100**
getClassName	**10.5**	**19.5**	**< 0.001**	** **0.328**
isGroup	**6.1**	**6.8**	**0.006**	* **0.429**
isHostNameLabel	**6.9**	**8.9**	**< 0.001**	* **0.414**
isSimpleAddress	**7.9**	**13.9**	**< 0.001**	** **0.350**
isValidUsername	6.2	5.5	0.946	0.502
lesseqString	**14.0**	**24.0**	**< 0.001**	* **0.407**
needsUrlEncoding	8.6	8.8	0.114	0.541
protectQuotations	**13.1**	**15.8**	**< 0.001**	* **0.380**
stripName	**20.4**	**42.7**	**< 0.001**	*** **0.194**
toCamel	**16.4**	**33.6**	**< 0.001**	*** **0.193**
translate	15.5	25.6	0.089	0.456
validateEmail	9.8	7.5	0.136	0.539

(b) Correct judgements only

Case Study	**Lang.** (s)	**Conv.** (s)	***p*-value**	$\hat{A}_{12}$
capitalize	11.6	12.1	0.333	0.529
composeName	19.2	19.9	0.579	0.516
containsChar	**9.2**	**14.6**	**< 0.001**	*** **0.279**
containsIgnoreCase	10.3	9.4	0.707	0.511
encodeClassname	**15.0**	**35.8**	**< 0.001**	*** **0.089**
getClassName	**10.8**	**16.8**	**< 0.001**	** **0.329**
isGroup	**6.1**	**6.6**	**0.014**	* **0.435**
isHostNameLabel	**7.0**	**9.0**	**0.005**	* **0.422**
isSimpleAddress	**7.4**	**13.6**	**< 0.001**	** **0.335**
isValidUsername	6.2	5.3	0.756	0.508
lesseqString	**13.7**	**26.8**	**< 0.001**	* **0.367**
needsUrlEncoding	8.8	8.9	0.124	0.541
protectQuotations	**13.0**	**15.7**	**< 0.001**	* **0.371**
stripName	**20.3**	**45.0**	**< 0.001**	*** **0.134**
toCamel	**16.4**	**32.9**	**< 0.001**	*** **0.178**
translate	15.8	25.3	0.552	0.483
validateEmail	8.4	7.5	0.284	0.531

than 0.29, 0.36 and 0.44 respectively. The case studies *containsChar*, *encodeClassName*, *stripName* and *toCamel* all involved large effect sizes – the latter three having recorded significantly improved results for accuracy in the answer to the previous research question. A further 2 case studies experienced medium effect sizes, *isSimpleAddress* and *getClassName*, while the effect size was small for *isGroup*, *isHostNameLabel*, *lesseqString* and *protectQuotations*.

The remaining 7 case studies experienced no significant difference. The methods *capitalize*, *composeName* and *translate* do not require their string inputs to be fully comprehended to make judgements about outputs, which may explain why no significant difference was found in terms of evaluation times for the two approaches. *capitalize* merely requires the tester to examine the first character of the string. *composeName* merely outputs one or other of its arguments or the concatenation of both, with again, little need for the two string arguments to actually be read; while *translate*, in a similar style to *capitalize*, just requires examination of the first character of the string.

An analysis of the test inputs generated for *containsIgnoreCase* revealed that both approaches generated an empty string for the second string argument the majority of the time. *containsIgnoreCase* tests for the presence of the second string argument in the first, and so an empty string for the second argument leads to the method trivially returning false – accounting for a lack of difference in times recorded for the language model and conventional approaches.

Finally, the test data generated for *isValidUsername*, *needsUrlEncoding* and *validateEmail* seemed to present more straightforward tasks for the human participants for which the usage of a language model was incapable of producing significant differences.

VI. THREATS TO VALIDITY

Although the results of our empirical evaluation show that the accuracy and time for humans to evaluate string inputs generated using the language model are improved in many cases, there are several threats to validity associated with our study, and these are detailed as follows.

The first threat concerns a potential bias in the selection of case studies, such that the patterns observed in our study may not generalize in practice. In order to mitigate this, code was taken from open source projects developed by real programmers. Secondly, a number of methods with string arguments were used, so as to sample different characteristics of functionality involving string inputs. Section IV-A discussed these issues in more detail. A further potential threat involves the text used to train the language model. While our text contained an adequate number of character combinations for training such a model, the use of different training texts may result in different (and possibly improved) results. Details regarding the training text were given in Section II.

One well-known threat to validity in human studies is the "learning" or "training" effect, where participants perform significantly worse at the beginning of the study due to unfamiliarity with the task, including, for our empirical evaluation, an unfamiliarity with the case study on which the proceeding questions were based. One step taken to mitigate this effect was to have two practice questions at the start of the questionnaire. Converse to the training effect is the

"fatigue effect", which refers to the human trait of tiring towards the end of a long study, influencing results collated from the end of the process. Steps taken to mitigate the fatigue effect included keeping the form of the questions simple and limiting the number of questions to 8. The data we collated indicated that on average, each participant spent just under 3 minutes on a questionnaire, indicating that our study was not particularly onerous and unlikely to be subject to fatigue effects. However, the main step to remove any bias from learning and fatigue effects was to randomize the questions. That is, the ordering of the questions was not fixed such that one of the techniques being studied was more exposed to any potential learning or fatigue effects than the other. The test inputs that formed the basis of each question were selected at random from a large pool of inputs, and these were selected at random from the overall set of inputs generated for each stochastic approach. These issues were discussed further in Section IV-C.

Another source of bias concerns the use of the CrowdFlower crowdsourcing website, and the selection of participants, which was not under our control. CrowdFlower, and other crowdsourcing platforms like it, are open to users mis-reporting expertise levels or performing tasks frivolously in order to earn money quickly. This issue was discussed extensively in Section IV-C, where we described the usage of a subset of only the participants who answered 50% or more of the test input questions correctly. The analysis of subsets of participants based on an independent metric has been shown to be a successful tactic in human studies of this nature by other authors [18], [19], [20], [21].

Finally, significance was tested for using Fisher's exact test and the Wilcoxon rank-sum test, while effect size was tested for using Vargha and Delaney's $\hat{A}_{12}$ statistic. These are all non-parametric tests, which do not require the need to make or test assumptions about the normality of the sample means, avoiding the introduction of further potential sources of error into the study.

VII. Related Work

There has been much attention devoted to the problem of automatic structural test data generation, including search-based testing [3], [4] and dynamic symbolic execution [5], [6] – an improvement to symbolic execution techniques originally proposed in the 1970s [23]. There has also been work in the search-based testing, dynamic symbolic execution and constraint-solving literature devoted to the problem of generating string inputs, e.g. [24], [25], [26], [27], [28]. However, these techniques do not take into account human readability of the strings generated.

McMinn et al. [29] and Shahbaz et al. [30] source suitable string inputs from Internet web pages for commonly-available string types such as email addresses. Wherever possible, this method should be used, since Internet web page content tends to be produced by humans, and the string values found tend to be realistic and readable. However, the technique is reliant on the programmer using identifiers that contain useful keywords for web searches so that the required values can be found. Furthermore, the string types themselves may encode low level information – as with many of the case studies used in this paper – that is not easily found on the web. In these circumstances, an alternative approach needs to be taken, and that is the purpose behind using a language model presented in Section II.

Checking test inputs by hand is a laborious and time-consuming task, yet only a few works have considered how to generate realistic test cases in general that make the job for the human tester easier. McMinn et al. [2] proposed the seeding of human-supplied test cases into the first stage of a search-based approach, so that the search is injected with an element of "realism" originating from the real testers themselves. Other approaches to improve realism of machine-generated test inputs include that of Bozkurt and Harman [31], who proposed a testing approach for service-oriented software in which the real outputs of one service (e.g., real ISBNs for a publication) are used as inputs for another. Related to these works is that of Fraser and Zeller [32], who embodied elements of realism in test cases consisting of Java method call sequences by biasing a search-based approach to common usage patterns of classes found in open source repositories.

Aside from individual test inputs, a further means of reducing oracle cost is to reduce the number of test cases in a test suite. Traditionally this has been applied as a post-processing step to an existing test suite, e.g. [33]. However, there has been recent work in search-based testing that has sought to combine test input generation and test suite reduction into one phase to produce smaller test suites, for example the work of Harman et al. [1], Ferrer et al. [34] and Fraser and Arcuri [35] with the EvoSuite tool.

VIII. Conclusions, Discussion and Future Work

This paper has presented an approach to automatic string input generation where the strings are not only optimized for test coverage but also according to a language model, in order to improve human readability. The paper showed, through empirical experiments involving human participants, that the string inputs produced are quicker to evaluate in several cases, and in certain cases, the accuracy of test input evaluation with respect to outputs is also improved. Since automated oracles are frequently unavailable in software engineering practice, this work therefore has an important bearing on lowering the costs of human involvement in the testing process as a manual oracle.

Our technique improves readability of strings without weakening the test adequacy criterion. However, for non-stringent criteria like branch coverage (as used our study),

where several inputs may execute a branch, the use of less readable strings may be more effective at exposing corner cases and discovering flaws in an implementation. In this case, the use of a language model may represent a trade-off for a tester. Given a certain time budget in which to perform testing, readable inputs may be preferred in order to reduce oracle checking time and thus increase the number of test cases that may be considered. Given unlimited time, more faults may be found with strings produced without the aid of a language model.

Further work needs to establish whether readable inputs are less likely to reveal flaws and, if so, what trade-offs are involved in terms of time spent versus faults found. Further work is also required to study the types of texts used to train the models that result in improved readability of strings produced and their potential effects on fault-finding capability. For example, certain types of text may be more suitable for certain domains of programs. Finally, additional work is also required to empirically evaluate different search algorithms and fitness functions involving language models, so that even better probability scores can be obtained for string test data.

Acknowledgements

The authors would like to thank Chris Wright for useful discussions, and also the anonymous referees for several useful comments.

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

References

[1] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," *International Workshop on Search-Based Software Testing (SBST 2010)*, pp. 182–191.

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

[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] 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.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 213–223, June 2005.

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

[7] H. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press, 1999.

[8] A. Lopez, "Statistical machine translation," *ACM Computing Surveys*, vol. 40, no. 3, pp. 8:1–8:49, 2008.

[9] D. Jurafsky and J. Martin, *Speech and Language Processing*, 2nd ed. Pearson, 2009.

[10] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *International Conference on Software Engineering (ICSE 2012)*, pp. 837–847.

[11] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35, no. 5, pp. 400–401, 1987.

[12] A. Stolcke, "SRILM – an extensible language modeling toolkit," in *International Conference on Spoken Language Processing*, pp. 257–286, 2002.

[13] H. Melvile, *Moby Dick*. Harper and Brothers, 1851.

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

[15] "Android code style guidelines, http://source.android.com/source/code-style.html#write-short-methods."

[16] P. McMinn, "IGUANA: Input generation using automated novel algorithms. A plug and play research tool," Department of Computer Science, University of Sheffield, Tech. Rep. CS-07-14, 2007.

[17] K. T. Stolee and S. Elbaum, "Exploring the use of crowdsourcing to support empirical studies in software engineering," *International Symposium on Empirical Software Engineering and Measurement (ESEM 2010)*.

[18] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," *International Conference on Software Maintenance (ICSM 2010)*, pp. 1–10.

[19] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," *International Symposium on Software Testing and Analysis (ISSTA 2012)*, pp. 177–187.

[20] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng., "Cheap and fast – but is it good?: evaluating non-expert annotations for natural language tasks," *Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, pp. 254–263, 2008.

[21] A. Kittur, E. H. Chi, and B. Suh, "Crowdsourcing user studies with mechanical turk," *SIGCHI conference on Human Factors in Computing*, pp. 453–456, 2008.

[22] A. Vargha and H. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[23] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT – A formal system for testing and debugging programs by symbolic execution," *International Conference on Reliable Software*, pp. 234–244, 1975.

[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," *International Symposium on Software Testing and Analysis (ISSTA 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," *International Conference on Automated Software Engineering (ASE 2009)*, pp. 515–519.

[26] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.

[27] 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," *International Conference on Computer Aided Verification (CAV 2011)*, pp. 1–19.

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

[29] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," *International Conference on Software Testing, Verification and Validation (ICST 2012)*, pp. 141–150.

[30] M. Shahbaz, P. McMinn, and M. Stevenson, "Automated Discovery of Valid Test Strings from the Web using Dynamic Regular Expressions Collation and Natural Language Processing," *International Conference on Quality Software (QSIC 2012)*, pp. 79–88.

[31] M. Bozkurt and M. Harman, "Automatically generating realistic test input from web services," *International Symposium on Service-Oriented System Engineering (SOSE 2011)*, pp. 13–24.

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

[33] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[34] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software Practice and Experience*, vol. 42, pp. 1331–1362, 2012.

[35] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference on Quality Software (QSIC 2011)*, pp. 31–40.