Reggae: Automated Test Generation for Programs using Complex Regular Expressions

Nuo Li Tao Xie
North Carolina State University
Raleigh, USA
Email:{nli3, txie}@ncsu.edu

Nikolai Tillmann Jonathan de Halleux Wolfram Schulte
Microsoft Research
Seattle, USA
Email:{nikolait, jhalleux, schulte}@microsoft.com

***Abstract*—Test coverage such as branch coverage is commonly measured to assess the sufficiency of test inputs. To reduce tedious manual efforts in generating high-covering test inputs, various automated techniques have been proposed. Some recent effective techniques include Dynamic Symbolic Execution (DSE) based on path exploration. However, these existing DSE techniques cannot generate high-covering test inputs for programs using complex regular expressions due to large exploration space; these complex regular expressions are commonly used for input validation and information extraction. To address this issue, we propose an approach, named Reggae, to reduce the exploration space of DSE in test generation. In our evaluation, we apply Reggae on various input-validation programs that use complex regular expressions. Empirical results show that Reggae helps a test-generation tool generate test inputs to achieve 79% branch coverage of validators, improved from 29% achieved without the help of Reggae.**

***Keywords*-test generation; dynamic symbolic execution; string generation;**

I. Introduction

Test coverage such as branch coverage is commonly measured to assess the sufficiency of test inputs. To achieve high test coverage, testers need high-covering test inputs, but it is tedious to generate high-covering test inputs manually. To reduce tedious manual efforts in generating high-covering test inputs, various automatic test-generation techniques have been proposed. Effective techniques such as Dynamic Symbolic Execution (DSE) [1], [2], [3], [4] have been recently proposed.

DSE first executes the program under test with arbitrary inputs, and collects symbolic constraints on inputs from the conditionals in branching statements being executed. DSE then negates a clause in the collected constraints and solves the resulting constraints to generate an input to cover a new path. Existing DSE techniques can effectively generate high-covering test inputs for various programs. However, these DSE techniques cannot deal with programs that use complex regular-expression (RegEx) operations such as a matching operation using a complex RegEx. Complex RegExs are commonly used in various programs, such as input validators of web applications and information-extraction applications. The main reason that existing DSE techniques cannot deal with such programs is that DSE's exploration space is often very large and the collected symbolic constraints may be too complex to be solved when a program under test includes complex RegEx operations.

To address the preceding challenges of generating high-covering test inputs for programs using complex RegExs, we present an approach, named Reggae, for improving test generation based on a notion of specialization. The program under test may invoke some generic functions[1], such as a RegEx-matching operation. In the program under test, the logic in such generic functions often need not be tested, as it is not the main testing focus when testing the program under test. However, when we use a DSE engine to generate test inputs for the program under test, the DSE engine explores the invoked generic functions, whose logic may be complex. The complex logic makes it difficult for the DSE engine to generate test inputs to achieve high branch coverage of the program under test within limited time and resources. To prevent the DSE engine from exploring large space in such generic functions, we transform the generic functions to specialized functions without changing the logic of the program under test but providing the DSE engine with much smaller exploration space.

We implement Reggae based on a DSE engine for testing .NET programs, including C# programs. Since RegEx-matching operations are relatively more complex than other string operations and have been commonly used in various programs, this paper focuses on how to improve a DSE engine to generate high-covering test inputs for a program using RegEx-matching operations (i.e., `IsMatch(s1,regex1)`, which determines whether the input string `s1` matches the RegEx specified in the pattern parameter `regex1`). However, the idea of Reggae is also

[1]In this paper, a generic function refers to a special type of function, whose return value depends on at least two parameters. One parameter defines a template. Given the template, the generic function calculates its return value based on the other parameter. Take `IsMatch(s1,regex1)` for instance, where `IsMatch` is a generic function and `regex1` defines a template. If an input string `s1` matches `regex1`, `IsMatch` returns `true`. Otherwise, `IsMatch` returns `false`.

```
01:public class MyAutomaton{
02: public enum States{STATE0, STATE1, STATE2, ...}
03: States _state = States.State0;
04: public bool AcceptChar(char c){
05:  switch (_state){
06:     case States.STATE0:
07:        if (-1 != (".").IndexOf(c)){
08:            _state = States.STATE3;
09:            return true; }
11:        if (-1 != ("abc").IndexOf(c)){
12:            _state = States.STATE0;
13:            return true; }
14:        ......
15:        return false;
16:     case States.STATE1:
17:         ...... } } }
```

Figure 1. Checking code of an automaton

applicable for other string operations.

This paper makes the following major contributions:

- **Specialization technique for test generation**. We propose to transform generic functions in the program under test to specialized functions that incur much smaller exploration space for a DSE engine. In this paper, we focus on the specialization of RegEx-matching operations.
- **Implementation**. We implement Reggae for testing .NET programs based on a DSE engine called Pex [4]. Pex has been previously used internally at Microsoft to test core components of the .NET architecture and has found serious fault. Reggae improves Pex's capability in dealing with programs using complex RegEx-matching operations.
- **Evaluation**. We apply Reggae on various input-validation programs that use complex regular expressions. Empirical results show that Reggae helps a test-generation tool generate test inputs to achieve 79% branch coverage of validators, improved from 29% achieved without the help of Reggae.

II. Approach

To reduce the exploration space of a DSE engine, when a RegEx-matching operation appears in a program under test, we first transform the RegEx in the RegEx-matching operation into an automaton, which checks whether a string matches the RegEx. Then, we use the DSE engine to explore feasible paths in the program under test and the checking code of the automaton (instead of the standard `Regex.IsMatch` method).

The RegEx-matching operation, `IsMatch`, has two parameters: a RegEx (`regex`) that defines string formats, and a string (`msg`) that is to be validated with `regex`. Each RegEx is equivalent to a finite-state automaton. We first parse `regex` and construct a finite-state automaton for `regex`. We next automatically synthesize checking code for the automaton. Each state in an automaton is mapped to a `case` statement in the synthesized checking code of the automaton, and each transition in an automaton is mapped to an `if` statement in the synthesized checking code for the automaton. Figure 1 shows an example of our synthesized

```
01: public class Regex {
02:   public static bool IsMatch(string msg, string regex){
03:        bool flag = false;
04:        MyAutomaton myAuto = new MyAutomaton();
05:        for (int i = 0; i < msg.Length; ++i){
06:         char c = msg[i];
07:         flag = myAuto.AcceptChar(c);
08:         if (!flag)
09:           return false;
10:         else
11:           continue; }
12:        if (myAuto.State.ToString().Equals(FINAL_STATE))
13:          return true;
14:        else
15:          return false; } }
```

Figure 2. Synthesized `IsMatch` with single automaton

```
01:   public static bool IsMatch(string msg, string regex){
02:        ......
03:     switch (regex){
04:       case RegEx[0] :{
05:        RegEx0_Automaton auto0 = new RegEx0_Automaton();
06:           ...... }
07:       case RegEx[1] :{
08:        RegEx1_Automaton auto1 = new RegEx0_Automaton();
09:           ......} }
```

Figure 3. Synthesized `IsMatch` with multiple automata

automaton. Given a character `c`, the `AcceptChar` method in Figure 1 checks whether the value of `c` is acceptable in the current state, and updates the current state if so.

Next, we split `msg`, the second parameter of `IsMatch`, into a character array and use the automaton to check whether all the characters in the array can be accepted by the automaton (i.e., the last character can reach an acceptable state called a final state). We synthesize a method that implements the preceding operation with the same name of the original RegEx-matching method (i.e., `IsMatch`). In addition, we implement the method in a class (denoted as α) with the same name of the original RegEx class. To avoid compilation errors, α is in the same `namespace` of the program under test (which should be different from the `namespace` of the original RegEx class). Figure 2 shows an example of our synthesized `IsMatch` method. In Line 4 of Figure 2, we create an object (`myAuto`) of the automaton, whose sample code is shown in Figure 1. In Lines 5-11 of Figure 2, if any character in `msg` makes `myAuto.AcceptChar` return `false`, our `Regex.IsMatch` returns `false`. Line 12 of Figure 2 checks whether the last character in `msg` makes our automaton reach a final state.

After the code synthesis, we replace the original RegEx operation in the program under test with our synthesized RegEx operation, i.e., the `IsMatch` in the class α. In this way, when a DSE engine explores the program under test, the DSE engine explores our synthesized RegEx operation instead of the original RegEx operation, and the exploration space of our synthesized RegEx operation, which deals with a specific RegEx, is smaller than the exploration space of the original RegEx operation, which deals with any RegEx.

If a Regex is dynamically generated by a program, we can specialize `IsMatch` in this way: first, we run the DSE engine on the program under test and dynamically detect

the generated distinct RegExs as `RegEx[i]` (i=1,2,3 ...) at runtime; second, we construct a finite-state automaton for each `RegEx[i]` and synthesize checking code for the automaton; third, we synthesize our own `IsMatch` method as shown in Figure 3. The `IsMatch` method compares the value of its parameter `regex` with `RegEx[i]`. If the value of `regex` is equal to a `RegEx[i]`, `IsMatch` initializes the automaton of the `RegEx[i]` and invokes checking code of the automaton to check the input string `msg`.

III. Evaluation

As our DSE engine is designed for .NET programs, our current Reggae supports testing C# programs. However, Reggae is independent of any language and can be implemented for other programming languages by using a test generator that can generate test inputs for those languages. To evaluate Reggae, we need subject programs that use complex RegExs for matching input strings. One common type of such programs is input validators. In our evaluation, we use (1) the input-validation code of an open source web application and (2) a set of input-validation programs synthesized for using complex RegExs from a popular RegEx library [7].

A. Empirical Study of Open Source Code

The open source web application used in our empirical study is an Email-extraction component in a job board and recruiting system named FlashRecruit [8], which is implemented in 915 files with 62,007 LOC (excluding comments and blank lines). The files include Java, XML, Javascript, JSP, CSS, DTD, and HTML files. In the Java files, which include 42,114 LOC (excluding comments and blank lines), FlashRecruit uses an Email RegEx to find Email addresses in text, implemented in the `findEmailInText` method (as shown in Figure 4).

As our DSE engine is designed for .NET programs, we first use Microsoft Visual Studio to translate the program under test from Java to C#. Then, we use Pex to generate test inputs for `findEmailInText` with a time limit of 14 minutes. Pex generates 324 test inputs and achieves 80% branch coverage of the `findEmailInText` method. The 324 test inputs cannot cover Statements 11 or 18, as Pex cannot genetate a `Word` object with a `value` that can make the RegEx-matching operation return `true`.

We next use Reggae to replace the execution of `Regex.IsMatch` in Statements 10 and 17 with the execution of our own synthesized `IsMatch` code, and then use Pex to generate test inputs for the `findEmailInText` method with a time limit of 14 minutes again. Pex generates 320 test inputs and achieve 90% branch coverage of `findEmailInText`. The 320 test inputs include three `WordList` objects containing a `Word` object with a `value` that can make the RegEx-matching operation return `true`. These `WordList` objects are:

```
01: public static final String REG_EXP =
"^((?:(?:(?:[a-zA-Z0-9][\\.\\-\\+_]?)*)[a-zA-Z0-9])+)
\\@((?:(?:(?:[a-zA-Z0-9][\\.\\-_]?){0,62})
[a-zA-Z0-9])+)\\.([a-zA-Z0-9]{2,6})$";
02: public Set findEmailInText(WordList wordList) {
03:  ArrayList emails = new ArrayList();
04:  Word pw1, pw2, pw3 = null;
05:  IEnumerator iter = wordList.GetEnumerator();
06:  while (iter.MoveNext()) {
07:    Word word = (Word)iter.Current;
08:    if (word.Type.Equals("word")) {
09:      String value = word.Value;
10:      if ((value.Length>3)&&(value.IndexOf("@")>0)&&
(value.IndexOf("@")<20)&&(Regex.IsMatch(value, REG_EXP)))
11:            emails.Add(value); }
12:     pw3 = pw2;
13:     pw2 = pw1;
14:     pw1 = word;
15:     if ((pw3 != null)&&(pw2 != null)&&(pw1 != null)) {
16:       String value = pw3.Value + pw2.Value + pw1.Value;
17:       if ((value.Length>3)&&(value.IndexOf("@")>0)&&
(value.IndexOf("@")<20)&&(Regex.IsMatch(value, REG_EXP)))
18:             emails.Add(value); } }
19:  return(emails); }
```

Figure 4. Email Extractor of FlashRecruit

```
wordList1("dH_X_Z@0-20.0k", "word"),
wordList2("L8-0-0@0B.080P", "word"),
wordList3("BS_9_0@9-c.a8", "word"),
```

which can cover Statement 11. However, the branch coverage of the `findEmailInText` method is still not 100% because the generated test inputs cannot cover Statement 18. Although Statement 18 is the same as Statement 11, the `value` in Statement 18 is a concatenation of the `values` in `pw3`, `pw2`, and `pw1`, which are three consecutive elements in `wordList`. In other words, the path constraints leading to Statement 18 are more complex than the path constraints leading to Statement 11. As a result, Pex can generate test inputs to cover Statement 11 but not covering Statement 18.

In summary, although Reggae does not achieve 100% branch coverage of the open source input-validator code, Reggae does help Pex to generate test inputs with higher branch coverage than the original test inputs.

B. Empirical Study of Synthesized Validators

This section presents our evaluation results used to answer the following research questions:

- RQ1: Does Reggae help Pex to generate test inputs that effectively improve the branch coverage of programs under test?
- RQ2: Does Reggae help Pex to generate test inputs that effectively improve capability of detecting faults in RegExs?

1) Subjects: We collect RegExs from *RegExLib* [7] to synthesize subject programs. Among the collected RegExs, we select 14 RegExs with complex structures based on three main criteria: RegExs represent different data types such as email and phone number, RegExs include selectable characters in a repeatable unit such as (\w|-)+, or RegExs include repeatable characters within a repeatable unit such as (@\w+)*. Each of the selected RegExs satisfies at least one of these criteria. Furthermore, as our implementation is based on the Brics Automaton [9], which cannot deal with

Table I
EMPIRICAL RESULTS OF APPLYING PEX WITHOUT AND WITH REGGAE ON 14 VALIDATORS WITH COMPLEX REGEXS

RegExs	Pex	Regae
\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*([,;]\s*\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)*		Y
\$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.\d{1,2}?)	Y	Y
([A-Z]{2}|[a-z]{2} \d{2} [A-Z]{1,2}|[a-z]{1,2} \d{1,4})?([A-Z]{3}|[a-z]{3} \d{1,4})?	Y	Y
[A-Za-z0-9]((_\.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)(([\.\-]?[a-zA-Z0-9]+)*)\. ([A-Za-z][A-Za-z]+)	Y	Y
(\w|-)+@((\w|-)+\.)+(\w|-)+		Y
[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?	Y	Y
((\w|\d|\-|\.)+)@{1}(((\w|\d|\-){1,67})|((\w|\d|\-)+\.(\w|\d|\-){1,67}))\.((([a-z]|[A-Z]|\d){2,4})(\.([a-z]|[A-Z]|\d){2})?)		
(([A-Za-z0-9]+_+)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|([A-Za-z0-9]+\++))*[A-Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}		
(http|https|ftp)\://([a-zA-Z0-9\.-]+(\:[a-zA-Z0-9\.%\$\-]+)*@)?((25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9])\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[1-9]|0)\.(25[0-5]|2[0-4][0-9]|[0-1]{1}[0-9]{2}|[1-9]{1}[0-9]{1}|[0-9])|([a-zA-Z0-9\-]+\.)*[a-zA-Z0-9\-]+\.[a-zA-Z]2,4)(\:[0-9]+)?(/[/]?[a-zA-Z0-9\.\,?'\/+%$#~ _-@]*)*		Y
(([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5}){1,25})+([;.](([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5}){1,25})+)*		Y
((http\://|https\://|ftp\://)|(www\.))+(([a-zA-Z0-9\.-]+\.[a-zA-Z]{2,4})|([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}))(/[a-zA-Z0-9%\:/-_\?\.']*)?		Y
((CN=((')|\w|\d|\s|\-|\.)+(/)*(,)*)+,\s*)*(OU=((\')|\w|\d|\s|\-|\.)+(/)*(,)*)+,\s*)*(DC=(\')|\w|\d|\s|\-)+[,]*\s*){1,}(DC=(\')|\w|\d|\s|\-)+\s*){1})		Y
((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+		Y
[a-zA-Z]\:\\((\w|[\u0621-\u064A]|\s)+\\)+(\w|[\u0621-\u064A]|\s)+(\.jpg|\.JPG|\.gif| \.GIF|\.BNG|\.bng)?		

RegExs that include special characters such as < and >, we select the RegExs that can be processed by the Brics Automaton.

Column 1 of Table I lists the selected RegExs, which include different types, such as email, URL, price, and scientific notation. For each RegEx, we synthesize a C# validator method that uses the RegEx to validate input strings. Each validator takes a string as input and uses a RegEx listed in Table I to validate the format of the input string. If the input string passes the checking of the RegEx, the validator returns `true`; otherwise, the validator returns `false`. Due to space limit, we do not list here the details of the validators, which however can be found at our project website [10]. As our validators take the result of a RegEx-matching operation as the condition for each `if` statement, we use branch coverage to evaluate whether Reggae helps generate high-covering test inputs.

2) Evaluation Setup: We perform the empirical study on a PC with 3G memory and 2GHz CPU with the following steps. (1) Pex uses about 26.5 minutes to generate 8912 test inputs for the validators. (2) Reggae transforms each RegEx-matching operation into a class that implements the corresponding automaton-checking code. (3) After including the automaton-checking code, Pex uses about 27 minutes to generate 2774 test inputs for the validators. (4) We measure the branch coverage of the validators based on the test suites generated in Steps 1 and 3.

3) Branch Coverage of the Validators under Test: We next answer the research question RQ1. To achieve high branch coverage of the validators under test, Pex need generate strings that match and strings that do not match the RegEx used in each validator under test. It is not challenging to generate a string that does not match a RegEx, but it is challenging to generate a string that matches a complex RegEx. For the RegExs used in our validators under test, Columns 2 and 3 of Table I list whether Pex generates strings to match the RegExs with and without using Reggae. ("Y" denotes Yes; blank entries denote No). Among the 14 RegExs, Pex generates matching strings for only 4 RegExs; Pex with Reggae generates matching strings for 11 RegExs.

As we synthesize one conditional for each RegEx, we have two branches for each synthesized validator. Then as Pex alone generates matching strings for only 4 RegExs, we know that the Pex-generated test inputs cover 29% (8/28) branches of our validators. On the other hand, as Pex with Reggae generates matching strings for 11 RegExs, we know that, with the help of Reggae, the Pex-generated test inputs cover 79% (22/28) branches of our validators. In summary, Reggae helps Pex to achieve 79% branch coverage of validators, improved from 29% achieved without the help of Reggae.

4) Fault-Detection Capability Improvement: We next answer the research question RQ2. The earlier presented empirical results show that Reggae improves the branch coverage of the subjects. In other words, Reggae improves the fault-detection capability, since a necessary precondition of detecting a fault is to generate test inputs that can cover faulty statements. In addition, we manually mutated all RegExs in our subjects and checked whether a mutant can be killed by the test inputs generated by Pex with and without Reggae. Here, killing a mutant indicates that Pex generates at least one test input that matches the mutated RegEx but does not match the original RegEx, or matches the original RegEx but does not match the mutated RegEx.

We applied four types of mutation operators on the RegExs in our subjects: inserting extra characters, removing mandatory characters, disordering character sequence, and modifying repeat times for characters. In total, we constructed 56 mutated RegExs for all RegExs. The test inputs generated by Pex with Reggae (in short as Reggae test inputs) kill 61% mutants among all the mutants, whereas the test inputs generated by Pex alone (in short as Pex test inputs) kill only 9% mutants among all the mutants. Note that the remaining 39% mutants not killed by the Reggae test inputs are not killed by the Pex test inputs either, but the 9% mutants killed by the Pex test inputs are also killed by the Reggae test inputs. In summary, Reggae helps Pex to generate test inputs to kill 61% mutants, improved from 9% mutants killed without the help of Reggae.

5) Summary: Overall the evaluation answers the research questions raised at the beginning of this section. Reggae does help Pex generate test inputs that effectively improve the branch coverage of programs under test. For the validators, Reggae helps Pex to achieve 79% branch coverage, improved from 29% without Reggae's help. Reggae does help Pex to generate test inputs that effectively improve capability of detecting faults in RegExs. Reggae helps improve the percentage of killed mutants among 56 mutated RegExs from 9% to 61%.

IV. Related work

Symbolic execution for test generation was first proposed in the 1970's [11]. In recent years, this field has been especially active as more powerful constraint solvers became available. However, most existing symbolic execution engines [1], [2], [12] cannot provide specific techniques for helping generate strings that match complex RegExs. There exist some approaches [13], [14] that focus on gathering symbolic constraints on string values based on symbolic execution and generate test inputs based on the gathered symbolic constraints. However, if there exist complex string operations in an explored path during symbolic execution, these approaches cannot solve the complex symbolic constraints, and therefore cannot generate test inputs to cover those related statements. White-box fuzz testing [16] uses a generational search algorithm using heuristics to find faults as fast as possible in an incomplete search. Different from white-box fuzz testing, Reggae reduces the exploration space for a symbolic execution engine based on specialization technique. Different from Hampi [15], a dedicated string-constraint solver, by tightly integrating into a DSE engine, Reggae does not require to limit the length of generated strings and can generate more than one matched string during test generation.

Besides symbolic execution, there exist other techniques that can determine the values of string expressions such as JSA [17]. JSA statically determines the values of string expressions in Java programs based on automaton analysis, and generates all possible values of string expressions at a certain execution point. Compared with JSA, Reggae combines the techniques of string analysis and DSE to generate test inputs for a program including string operations that can or cannot be expressed as automaton checking.

V. Conclusion

Programs that use complex string operations including RegEx operations pose challenges for existing test generation tools [12], [4], [2]. To address the challenges in generating string test inputs, we proposed a specialization-based approach, named Reggae. In particular, we transform generic functions invoked by a program under test to specialized functions that can help a test-generation tool to generate high-covering test inputs. We implemented Reggae for testing .NET programs, including C# programs. Although this paper focuses on how to improve a test-generation tool to generate test inputs for a program with RegEx-matching operations, Reggae is applicable for other string operations related to RegExs.

Acknowledgment

This work is supported in part by NSF grants CCF-0725190 and CCF-0845272, and ARO grants W911NF-08-1-0443 and W911NF-08-1-0105.

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. PLDI*, 2005, pp. 75–84.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proc. ESEC/FSE*, 2005, pp. 263–272.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *Proc. CCS*, 2006, pp. 322–335.

[4] N. Tillmann and J. de Halleux, "Pex - White box test generation for .NET," in *Proc. TAP*, 2008, pp. 134–153.

[5] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *Proc. TACAS*, 2009, pp. 307–321.

[6] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. DSN*, 2009.

[7] "Regexlib.com," 2008, http://regexlib.com/.

[8] "Flashrecruit - job board/applicant track," 2007, http://sourceforge.net/projects/flashrecruit/.

[9] "dk.brics.automaton," 2008, http://www.brics.dk/automaton/.

[10] "Reggae: Automated test generation for programs using complex regular expressions," 2009, https://sites.google.com/site/asergrp/projects/reggae.

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[12] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A symbolic execution extension to Java PathFinder," in *Proc. TACAS*, 2007, pp. 134–138.

[13] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proc. ISSTA*, 2007, pp. 151–162.

[14] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proc. ISSTA*, 2008, pp. 249–260.

[15] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for string constraints," in *Proc. ISSTA*, 2009, pp. 105–116.

[16] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. NDSS*, 2008, pp. 151–166.

[17] A. S. Christensen, A. Mller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. SAS*, 2003, pp. 1–18.