Test Image Generation using Segmental Symbolic Evaluation for Unit Testing

Tahir Jameel, Mengxiang Lin
State key Lab of Software Development Environment
Beihang University
Beijing, China
{tahir, mxlin}@nlsde.buaa.edu.cn

He Li, Xiaomei Hou
State key Lab of Software Development Environment
Beihang University
Beijing, China
{lihe, xmhou}@ nlsde.buaa.edu.cn

***Abstract*— This paper presents a novel technique to generate test images using segmental symbolic evaluation for testing of image processing applications. Images are multidimensional and diverse in nature, which leads to different challenges for the testing process. A technique is required to generate test images capable of finding program paths derived by image pixels. The proposed technique is based on symbolic execution which is extensively used for test data generation in recent years. In image processing applications, pixel operations such as averaging, convolution etc. are applied on a segment of input image pixels called window for a single iteration and repeated for the entire image. Our key idea is to imitate operations on pixel window using symbolic values rather than concrete ones to generate path constraints in the program under test. The path constraints generated for different paths are solved for concrete values using our simple SAT solver and the solutions are capable to guide program execution to the specific paths. The solutions of path constraints are used to generate synthetic test images for each identified path and the paths constraints which are not solvable for concrete pixel values are reported as infeasible paths. We have developed a tool IMSUIT that takes an image processing function as input and executes the program symbolically for the given pixels window to generate test images. Effectiveness of IMSUIT is tested on different modules of an optical character recognition system and the result shows that it can successfully create test images for each path of the program under test and capable of identifying infeasible paths.**

Keywords—symbolic execution, testing image processing, unit testing

I. INTRODUCTION

Recently, the role of image processing applications has been increased remarkably such as medical imaging, documents digitization, bioinformatics, remote sensing and a number of other applications. The significance of decision based image systems raises need of a well-tested reliable system. Software testing is a vital approach to identify bugs, which is accomplished by software analysis and generation of bug causing inputs. The aim of software testing is to show the absence of bugs but practically it only shows the bugs that manifest during the testing process. Furthermore, it is the most costly part of software development life cycle and may exceed 50% of overall cost [1]. Despite of its limitation and cost, software testing helps to improve software quality and reduces manual efforts to test the program.

Our goal in this research is to automate the testing process of image processing applications. In particular, we seek to automatically generate test images that can achieve a code coverage metric such as path coverage. In last decade, the significance of symbolic execution [4, 6] for automatic test input generation has been widely proved for sequential as well as concurrent programs written in different languages [5, 10]. However, testing of image processing applications pose different challenges that must be addressed.

Firstly, to test an image processing software, we deal with semantically meaningful images. In real systems, these images can be either specific or general e.g. face recognition system require faces and non-faces for testing, classification of cancerous and non-cancerous tissues requires MRI images and sets of similar images for content based image retrieval system. Generally, these systems are tested using the available class of images which they process and their output is analyzed. For example if we test a face recognition system we give different test images to the system and check whether it is capable to match them with the right face in the image database. However, by doing so we are analyzing the algorithm correctness rather than software. Our first observation is that, meaningful images are necessary for algorithm testing not for its software implementation. We can use synthetic images for testing different paths of a software and its output is analyzed using a test oracle. Secondly, an image is a multidimensional input data composed of different brightness levels and a 2-D grid of pixel positions; which makes testing process and test data generation more difficult. Our second observation is that, we can make use of a subset of image pixels for which operations are applied in a single iteration and repeated for the entire image. Thirdly, selection of a test oracle is based on semantics of image processing application. For example if a program classifies image pixels to different bands of gray levels then the test oracle should be based on the pixel ranges.

Manual testing is the most ready technique without using extra resources. For this purpose the code is analyzed and test cases are generated on the basis of the control flow or data flow of the program under test. However, the problem with this approach is its applicability. It is a tedious job to test the system manually especially, when the programs under test have multidimensional data. Exhaustive testing is a choice in safety critical systems [2] but in case of image processing applications,

978-1-4799-5604-3/14/$31.00 copyright 2014 IEEE

it is not feasible to test the program for all possible input images in a limited time. Random testing [3, 8, 9] is a simple approach but it generates untargeted tests. Time required to process images is high and untargeted testing lacks diversity. The requirement is to generate test images that satisfy certain code coverage such as path coverage.

This paper presents a first effort to automatically generate test images using symbolic execution for testing of image processing applications. The proposed technique addresses the three challenges of testing image processing applications in the following way. The first challenge is addressed by the creating synthetic images based on program semantics. The proposed technique aims to achieve path coverage and test images are generated for each identified path. The paths for which no input images can be generated are identified as infeasible paths. Testing each path of the program using test images enhances confidence on the quality of the system. The second challenge of handling large scale multidimensional data is addressed by choosing a window of pixels (4 neighbors, 8 neighbors etc.) as symbolic variables and test images are generated by manipulating these variables. The key idea is to exploit the fact that certain operations are performed on a window of selected pixels and these operations are repeated for the entire image to produce output. The third challenge is addressed by selecting the test oracle automatically using semantics of program under test. For a test path, the pixels of resulting image occur in a specific ranges of gray levels which are used as test oracle.

In this paper we extend the traditional symbolic execution to image processing applications by using a subset of pixels called window. We have developed a prototype tool IMSUIT in Matlab which can take an image processing function written in Matlab as input and generates test images automatically. To do so, we generate constraints on image pixels during program execution on symbolic values. In some cases like pattern matching these constraints can be large enough, we use constraint simplification to boost up the speed of constraint solving. Path constraint is an expression with some mathematical and logical operations over pixel variables. For 8-bit pixels the value of gray levels ranging from 0 to 255. The constraint solvers [7] are used to solve the constraints. We have used a simple solver based on random number generation as the range of pixel values is not high. To test the effectiveness of IMSUIT, we applied IMSUIT on an optical character recognition (OCR) system and generated test images for its modules. The result shows that for each path in the program under test, images are generated to achieve path coverage.

This paper has three main contribution:

1. A novel idea to generate test images for unit testing of image processing applications;
2. Development of a working prototype IMSUIT;
3. Experimental results showing the effectiveness on real image processing applications.

The paper is organized as follows. In section 2, a brief overview of the approach is presented using a simple example. In section 3, the details of the approach are discussed. In section 4, implementation of the approach and its evaluation is presented

binariz.m

```
1:  function  binaryImage = binariz (grayImage)
2:  threshold = 128;
3:  r = 100;
4:  c = 100;
5:  for i = 2:r − 1
6:     for j = 2:c − 1
7:        p1 = grayImage(i − 1, j − 1);
8:        p2 = grayImage(i − 1, j);
9:        p3 = grayImage(i − 1, j + 1);
10:       p4 = grayImage(i, j − 1);
11:       p5 = grayImage(i, j);
12:       p6 = grayImage(i, j + 1);
13:       p7 = grayImage(i + 1, j − 1);
14:       p8 = grayImage(i + 1, j);
15:       p9 = grayImage(i + 1, j + 1);
16:       avg = (p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9)/9;
17:       if (avg < threshold)
18:          binaryImage(i − 1, j − 1) = 0;
19:       else
20:          binaryImage(i − 1, j − 1) = 255;
21:       end
22:    end
23: end
```

Figure 1 Example Code

using a real image processing application. Finally, section 5 concludes the discussion.

II. OVERVIEW

We use a simple example to illustrate the working of IMSUIT. Consider a function *binariz* in Fig. 1, which takes an input gray scale image of 100 by 100 pixels and generates its binary image. The program selects a window of eight neighboring pixels in terms of *i & j* for a given pixel location to take average of the window for smoothing purpose. Then the program applies thresholding to covert the gray pixels to black and white pixels in the resulting image. The program has two paths, one is followed when the average of selected window is less than a threshold while other is followed when the average is greater than or equal to a threshold. To generate test images for such functions, IMSUIT needs two kind of inputs, one is the program under test and the other is name of variables which are to be executed symbolically. In the above example, the variables p_1 to p_9 store the values of eight neighboring pixels of a centered pixel from input image in a single iteration. The values of p_1 to p_9 are changed iteratively for the next window of neighboring pixels until the whole image is traversed. These variable are given as input to IMSUIT to articulate that these variables are to be treated as symbolical variables during execution.

IMSUIT generates constraints from the program under test using symbolic evaluation. The program under test is parsed line by line and the states of the program variables are stored in different structures designed for variables, images, loops, branches and a stack is designed to resolve the scope of branching statements. These structures imitate program memory stack during symbolic execution. Table 1 shows the states of program variables executed symbolically in each step. In line 2, there is a simple assignment of a concrete value to a variable. For each simple assignment, it is evaluated that whether the variable on left hand side is a new variable or already declared.

Table 1 Symbolic States

line	avg	thresh	r	c	i	j	pc
1	?	?	?	?	?	?	?
2	?	128	?	?	?	?	?
3	?	128	100	?	?	?	?
4	?	128	100	100	?	?	?
5	?	128	100	100	2	?	i≤r-1
6	?	128	100	100	2	2	i≤r-1&j≤c-1
7~15	?	128	100	100	2	2	i≤r-1&j≤c-1
16	$\frac{\sum_{n=1}^{9} p(n)}{9}$	128	100	100	2	2	i≤r-1&j≤c-1
17	$\frac{\sum_{n=1}^{9} p(n)}{9}$	128	100	100	2	2	i≤r-1&j≤c-1 & $\frac{\sum_{n=1}^{9} p(n)}{9}$<128
18	The pixel of output image is assigned 0						
19	$\frac{\sum_{n=1}^{9} p(n)}{9}$	128	100	100	2	2	i≤r-1&j≤c-1 & $\frac{\sum_{n=1}^{9} p(n)}{9}$ ≥128
20	The pixel of output image is assigned 255						

If a new variable is declared, its name and value are stored in the variable structure otherwise the value of already declared variable is overridden. Line 3 and 4 are also simple assignments to new variables. In the case of *for* loops, the loop condition must be satisfied to execute at least a single iteration. The loops in line 5 and 6 are nested, to execute the statements in nested loop the conditions of both loops must be satisfied. So the path condition becomes:

$$\text{PC:} \quad i \leq r-1 \;\&\; j \leq c-1 \tag{1}$$

The statements 7 to 15 initialize the symbolic variables with the pixel values of input image. The variables p_1 to p_9 are treated as symbolic variables, as given in the input. These variables are stored in symbolic variable structure and the variables whose values are based on these symbolic variables are also treated as symbolic variables. In line 16, symbolic variables are used to compute the average and the variable *avg* along with its symbolic value are stored in symbolic variable structure. Equation 2 shows the value assigned to variable *avg* in symbolic terms:

$$avg = [p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_9 + p_9]/9 \tag{2}$$

A branch splits the program execution into two paths and both paths are executed in symbolic execution. IMSUIT analyses branch conditions and generates path constraints for both true and false conditions. In the line 17, one path condition is as follows:

$$\text{PC:} \quad i \leq r-1 \;\&\; j \leq c-1 \;\&\; \frac{\sum_{n=1}^{9} p(n)}{9} < 128 \tag{3}$$

Whereas the alternate path condition is:

$$\neg\text{PC:} \quad i \leq r-1 \;\&\; j \leq c-1 \;\&\; \frac{\sum_{n=1}^{9} p(n)}{9} \geq 128 \tag{4}$$

Input: ***Test Program***
Symbolic Variables
Output: ***Test Images***

IMSUIT (***program, symb_varb***)

1: ***line = getLine*** ( );
2: ***While line ≠ Null***
3: ***SymbolicEvaluator*** (***line, symb_varb***)
4: ***writeLogFile*** ( )
5: ***const = symbolicExecuter*** ( )
6: ***line = getLine*** ( );
7: ***endWhile***
8: ***simp_constraint = simplifier*** (***const***)
9: ***solutions = constraintSolver*** (***sim_const***)
10: ***if solution ≠ NULL***
11: ***testImages = generateImages*** (***solutions***)
12: ***else***
13: ***report infeasible paths***
14: ***end***

Figure 2 Algorithm

The keyword *end* is associated to the correct scope of *for* loop or *if/ elseif* branch using a stack. The path conditions extracted from symbolic execution are solved for concrete values using a simple constraint solver based on random number generation. A pixel value ranges from 0 to 255 and to solve a path condition arbitrary values are generated from the range. If these values are able to satisfy the path condition then it is a possible solution, otherwise the process is repeated for another set of arbitrary numbers. A path condition can have one specific solution or a number of solutions. The above constraints have multiple solutions for each path ranging from 0~127and 128 to 255.

The solutions computed for the path conditions are used to synthesize test images and for each path in the program under test at least one test image is created. As in above case, a path has multiple solutions and to generate test images, IMSUIT randomizes different solutions. A test image ensures that the program under test will follow a specific path when executed by giving the test image as input. This creates ease of testing oracle and debugging process. In above example, two test images are created to test its each path. When a test image is given as input to a program, the assertion should not be violated. In Fig 1, line 18 and 20 define the pixel values of output image which are used as test oracle. For example for path 1, when the test image is executed, all the pixels of resultant image must be zero and for path 2 all the pixels of a resulting image must be 255. If there exists a path for which no test image can be created then the path is infeasible or can be a part of dead code.

III. APPROACH

IMSUIT takes a program as input written in the form of set of statements, which consists of assignments, branching statements and loops. An assignment statement can be a simple or a mathematical equation having an assignment operator. The assignment statement assigns the value evaluated from right hand side to a variable on left hand side. The assignment can be through a function call, a simple digit, variable, or equation. The

```
Input:   Program Line
         Symbolic variables
Output:  Update structures and stack

symbolicEvaluator(line, symb_vars)
    ignore spaces & comments
    Token = getToken(line)
    if Token ∈ keyword
        Parse line
        Update corresponding structure
        Update Stack
    else
        if Token ∈ reservedWords
            Parse line
        else
            check the assignment
            if assignment == fucntionCall
                Parse function call
                Update Corresponding Structure
            else
                if assignment == simple
                    if LHS ∈ variable structure
                        override variable value
                    else
                        new varriable initialized
                        write to structure variable
                    end
                else
                    assignment == equation
                    parse RHS of equation
                    write to LHS variable structure
                end
            end
        end
    end
```

Figure 3 Symbolic Evaluator

branching statements have a Boolean expression followed by statements and then termination its scope. A branch condition can be a simple consisting of single condition or complex consisting of multiple concatenated conditions. There can be statements for false condition of Boolean expression as well. A loop statement have a loop variable with a start value and a terminating value. Inside the loop, there are statements and then termination of its scope. IMSUIT aims to extract program paths according to above program structures and generates test images for each path.

To execute program symbolically we need to know about the program constructs and semantics. Fig.3 shows an overview of IMSUIT algorithm. A test program is taken as input with a list of variables required to execute symbolically. The program is read line by line and passed to a symbolic evaluator, which parses the statement and extracts the useful information necessary to execute the program symbolically. This information is used to update stack and structures that store the program states. After extracting the necessary information, the statement is executed symbolically and path constraints are updated. Once the program is processed, path constraints for different paths are extracted. These path constraints can be complex, which are simplified to expedite constraint solver as it

```
Input:   Constraint
Output:  Solution to Constraint

constraintSolver(constraint)
    get the symbolic variables
    while (¬satisfied)
        generate a set of random numbers
        set symbolic variables values
        solve the constraint
        if constraint is solved
            satisfied = 1
        end if
    end while
```

Figure 4 Constraint Solver

is computationally most expensive part. To find concrete solutions, the simplified path constraint are passed to a constraint solver. The concrete solutions are used to generate test images. Whereas, if the path constraint cannot be solved then the path is infeasible. The infeasible paths are reported to the user.

In symbolic evaluation, information in the program under test is extracted and analyzed without executing it. The aim of symbolic evaluation is to extract semantics of the program structure. Different structures are designed to store program states e.g. variables, symbolic variables, images, loops and branches. A stack is designed to resolve the scope of loops and branching statements. The symbolic evaluator takes a statement from the program under test and a list of symbolic variables. Fig. 3 shows how symbolic evaluator works. Firstly, the string token of the statement is examined for keywords specific to programming language e.g. *if, else, for* etc. If the token is a keyword then the line is parsed according to the expected values specific to that keyword. For example, if the keyword is *if* then symbolic evaluator expects a Boolean expression after the *if* keyword. Secondly, if the token is not a keyword then it is checked for a reserved word specific to the programming language e.g. in Matlab *clc, figure* etc. These reserved words do not contribute in symbolic execution but only used to display images, close files etc. Thirdly, if the token is not a reserved word, then the line is checked for an assignment statement. If the line has an assignment operator, then its right hand side is parsed. The right hand side of the equation can be a function call whose output is assigned to the variable on the left hand side. Otherwise, the statement can be either a simple assignment having a digit or variable on right hand side or an equation consisting of variables, digits and operators whose value is computed and assigned to the variable on left hand side. The variable states are updated in the variables structure accordingly. Branches are important in symbolic execution. When an *if* or *elseif* keyword is found its Boolean condition is parsed. A condition can be simple or combinations of several conditions. The multiple conditions are concatenated using *&* or | operators. Condition variables and logical operators are stored to a structure which is maintained for branch statements. As discussed earlier, symbolic execution follows both paths of a branch, for this purpose the false condition is also computed and stored to the conditions structure along with the true condition. In nested branches, the true condition of parent is concatenated

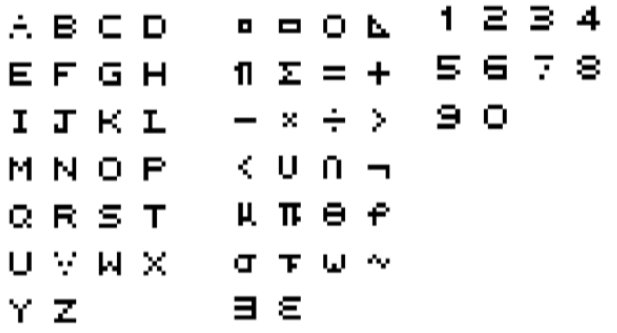

Figure 5 Test Image Generated for OCR

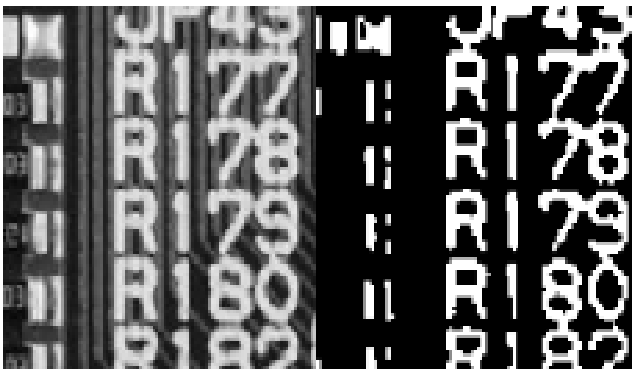

Figure 6 Binarized Image

with the child. Whereas in case of *elseif* the false condition of the parent *if* condition is concatenated with the child's Boolean condition using *&* operator. These branch conditions are used to update the path constraint in symbolic execution. The condition variables are mostly the symbolic variables or variables computed using symbolic variables because the conditions are usually applied to classify pixels or function of pixels. Loops in the program have a Boolean condition to execute the loop iterations and its termination. The loop condition is also concatenated with the path condition after extracting from the statement. Whenever a condition or a branch occurs, the stack is updated to resolve their scope.

For an assignment statement, if the type is a simple assignment then the variable at left hand side is either a new variable or already declared. Whenever a variable is parsed, it is searched in the existing structure of variables. The right hand side of equation is also parsed if it is a number then the value is written to the value field of the variable structure. But if the right hand side is a variable then the value of the variable is extracted from variable structure and copied to the variable on the left hand side in its value field in variables structure. If the equation is a mathematical or logical then the equation is parsed for its each operand and operator. The equation is stored in terms of variables and symbolic variables which are evaluated during symbolic execution. At the same time, the variables are checked if they are symbolic then the values are stored to symbolic variable structure. The symbolic variables are usually the pixels of images on which different operations are performed. They are initialized by the image pixels and then their symbolic values are changed during execution. The variables whose values are computed using symbolic variables are also treated as symbolic variables and their symbolic values are stored in symbolic variable structure.

After extracting the information by symbolic evaluation, the statement is executed symbolically. During symbolic evaluation, a message is also generated for symbolic execution containing the semantics information. For example, semantic information is generated for different statements such as branch, simple assignment, image read, symbolic variable override etc. and specific flags are set. The symbolic evaluator finds the flag and performs the corresponding task. Symbolic evaluator executes the program on symbolic value rather than concrete values. Whenever it finds a branch it follows both the paths. The path constraint is also updated for each branching statement execution. The condition of a *for* loop is similar to an *if* condition. Path conditions are generated for each path aggregating all the branch conditions over symbolic variables. The path condition can be a sequence of concatenated logical and mathematical statements. In some cases, the path condition can be simplified. Constraint solving is computationally the most expensive part of the system. By simplifying the path condition we can make it faster. We use a constraint simplifier for constraint simplification.

A constraint solver checks whether a path condition can be solved for concrete value. We have developed a constraint solver shown in Fig. 4, based on random number generation. Usually the symbolic variables are pixels whose values are stored in 8-bit variables ranging from 0 to 255. Range of random numbers is small and a random number generator based SAT solver can find the concrete solution for a path constraint quickly. Once the solution to the path condition is computed, it is used to generate test images. Multiple images can be generated if there exists multiple solutions to the path constraint. A single image containing different concrete solutions can also be generated. These test images are given as input to program under test and the output of the program is evaluated. For example, in global thresholding the pixel value of gray image is checked. If pixel is less than 128, then pixel of the resulting binarized image is assigned 0 otherwise 255. There exist multiple solutions to their path conditions. The multiple solutions are randomized in the test images. When the test image of path 1 is executed the resulting image is all black. Whereas when the test image for path 2 is executed the resulting image is all white.

IV. IMPLEMENTATION AND EVALUATION

We have developed a tool IMSUIT in Matlab to generate test images for functions written in Matlab. It consists of 2K lines of codes. IMSUIT is tested on a real optical character recognition system, which has five different modules. Table 1, shows the results of test images created and their paths. The first function is *global threshold*, which binarizes the given image using a threshold values. This function has two paths and to test we need at least one image for each path. There exist 128 different solutions to both path constraint. The first image created has

Table 2 Results of Modules of OCR

Sr.	Function	Lines of Code	No of Paths	No Test Images
1	Global Threshold	31	2	2
2	Smooth Threshold	64	4	4
3	OCR Alphabets	164	27	27
4	OCR Numbers	106	11	11
5	OCR Special Characters	106	27	27

random values of pixels less than or equal to 128 and the second has random values greater than 128. When the first test image is given as input to the function, its output is all black whereas the output of second image is all white. Violation of the test oracle points to a bug in the function. The second function *smooth threshold* binarizes the input image and also smoothes the output image to eliminate noise which causes problems in character recognition. This function uses three different bands of pixel values to binarize input image. This function uses 8 neighbors of a centered pixel and computes their average. The average is compared to the three different thresholds. These 9 pixels are considered symbolic variables in the execution. The function have four different paths and for each path a test image is created. The third function is *OCR Alphabets* which searches alphabet patterns in the given image. This function has 27 paths and 27 images are generated for each path. The fourth function is *OCR Numbers* which searches number patterns in the given test image. IMSUIT generates test image for numbers using the path constraints. It has 11 paths and 11 test images are generated to test this function. The fifth function is *OCR special characters* which searches special character patterns in the input image. There are 26 different paths for this function and 26 test images are generated to test them.

Fig. 5 shows the input images generated for alphabets, numbers and special characters. The images created for other functions are visually meaningless that's why they are not shown. Fig. 6 shows a gray scale image on the left side which is pre-processed using *smooth threshold* function. Smoothing effects are clearly visible in the binarized image at the top right of the figure. Alphabets and numbers are clearly visible in the image and this image is ready to be given as input to OCR alphabets, OCR numbers and OCR special characters. However, these functions can be tested using test images shown in Fig 5. If a set of test images are tested covering all paths of a function then the function can be used for real images with higher confidence.

V. CONCLUSION

Testing a program with the help of test data is a basic way to check the functionality of the program under test. In this paper, we have used segmental symbolic evaluation for the generation of test images for image processing applications and demonstrated its usefulness with the help of optical character recognition system. This is the first effort to generate test images automatically for unit testing of image processing modules. The tool IMSUIT, is a prototype tool developed for Matlab programs however, it can be implemented for other programming languages. IMSUIT is capable of taking input function and generating test images for all of its paths using symbolic evaluation and reporting the infeasible paths. We have developed a simple SAT solver based on random number generation. The path constraints of image processing systems comprise of pixels and values of pixels are 0 to 255 for 8-bit images. This fact makes possible the use of a simple SAT solver. The tool is tested on different modules of optical character recognition system. The result shows that it has successfully created test images for each program path.

ACKNOWLEDGMENT

I would like to thank Chinese scholarship council for providing a conducive research environment.

REFERENCES

[1] Beizer, B., 1990. Software Testing Techniques, second ed. International

[2] Knight, John C., Kevin G. Wika, and Shannon Wrege. "Exhaustive Testing As A Verification Technique." Submitted to the International Symposium on Software Testing and Analysis. 1996. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[3] Ciupa, Ilinca, et al. "Finding faults: Manual testing vs. random+ testing vs. user reports." Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on. IEEE, 2008..

[4] King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394..

[5] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008

[6] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.

[7] Armand, Michaël, et al. "A modular integration of SAT/SMT solvers to Coq through proof witnesses." Certified Programs and Proofs. Springer Berlin Heidelberg, 2011. 135-150.

[8] Korel, Bogdan. "Automated software test data generation." Software Engineering, IEEE Transactions on 16.8 (1990): 870-879.

[9] Thevenod-Fosse, Pascale, and Helene Waeselynck. "STATEMATE applied to statistical software testing." ACM SIGSOFT Software Engineering Notes 18.3 (1993): 99-109.

[10] Anand, Saswat, Corina S. Păsăreanu, and Willem Visser. "JPF–SE: A symbolic execution extension to java pathfinder." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2007. 134-138.