International Journal of Networked and Distributed Computing, Vol. 2, No. 3 (August 2014), 135-147

Test Image Generation using Segmental Symbolic Evaluation

Tahir Jameel*
Department of Computer Science and Engineering, Beihang University
Beijing 100000, China

Mengxiang Lin
Department of Computer Science and Engineering, Beihang University
Beijing 100000, China
E-mail: mxlin@nlsde.buaa.edu.cn
www. .ev.buaa.edu.cn

Abstract

Image processing applications have played a vital role in modern life and they are required to be well tested due to their significance and human dependence on them. Testing of image processing application is difficult due to complex nature of images in terms of their generation and evaluation. The presented technique is first of its type to generate test images based on symbolic evaluation of program under test. The idea is based on the fact that, neighboring image operations are applied by selecting a segment of image pixels called a window, and iterated by sliding window over entire image. We imitate neighboring operations using symbolic values for the pixels rather than concrete values. The path constraint is extracted for each path in the program under test and solved for concrete solutions. Test images are generated based on solution of path constraints for each identified path. We have tested the proposed scheme on different programs and the results show that test images are successfully generated for each path to ensure the path coverage of the program under test and identifying infeasible paths.

Keywords: symbolic execution, image generation, testing, input generation, unit testing, synthetic images

1. 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 limitations 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 [2]

*State Key Lab of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 10000, China
tahirjameel8@gmail.com

for automatic test input generation has been revived and widely proved for sequential as well as concurrent programs written in different languages. 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. For example, a face recognition system requires images of faces and non-faces for testing, similarly classification of cancerous and non-cancerous tissues requires MRI images, and sets of similar images are required for testing of content based image retrieval system. Generally, these systems are tested using the available class of images for which they are meant to process and the 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 readily available technique and do not require extra resources. Image processing applications are tested manually using handcrafted or standard images. The outcome of program under test is analyzed manually to decide whether the program passes or fails. 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 multi-dimensional input data. Exhaustive testing is a choice in safety critical systems [3], but in case of image processing applications, it is not feasible to test the program for all possible input images in a limited time. Random testing [4, 5], is a simple approach but it generates untargeted tests. Generally, the time required to process an input image is high, furthermore, untargeted testing lacks diversity. The requirement is to generate test images automatically that satisfy certain code coverage criterion such as path coverage in a limited time.

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 the 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 multi-dimensional 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. 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. In symbolic execution, constraint solvers [6] 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 have applied IMSUIT on distance transform, robust filter and different modules of optical character recognition (OCR) system. The result shows that for each path in the implementation 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 background of symbolic execution and neighboring operations is presented. In section 3, an overview of the approach is presented using a simple example. In section 4, the details of the approach are discussed. In section 5, implementation and evaluation of the approach is presented using image processing applications. In section 6, a brief discussion on synthetic images and their usefulness is presented. Finally, section 7 concludes the discussion.

2. Background

Validity of testing is limited to dataset used in the testing process whereas program proving is highly dependent on specifications and steps of proof used in the formal method. Symbolic execution [2, 7] presented four decades ago as a practical approach between these two approaches. It is an alternative way to execute and analyze a program than its normal execution for the purpose of input generation and program analysis. The generated input can drive an execution to a specific program path and for each path there can be several or no inputs generated by symbolic execution. The advantage of executing program symbolically is to achieve abstract interpretation for each of program execution path. During the execution, the associated classes of symbolic inputs are pruned by the constraints to get a specific subclass of inputs that is specifically required to execute that path. Depending upon path constraint, the subclasses may have a single concrete input, a number of concrete inputs, or even no concrete input.

Symbolic execution uses symbolic inputs instead of concrete inputs and eventually the values of variables are represented as symbolic expressions. For a selected path in the program, symbolic evaluation generate a path constraint consisting of equalities and inequalities over symbolic inputs [2]. The path constraint is updated for a branching instruction to encode the constraints on inputs to reach that program point [8]. In contrast to concrete execution, both paths of branching instructions can be taken in a symbolic execution. However, the generated path constraint must be satisfied to traverse the path. Whenever symbolic execution along a specific path terminates (normally or erroneously), the program constraint is solved using constraint solvers [6, 9] and concrete inputs are generated by the solutions of the inequalities. When program is executed with the generated input, only a specific path is traversed for a deterministic code. If there exist no solution to a path constraint then the path is infeasible or unreachable. For recursive programs, the symbolic execution may produce infinite paths which is limited by restraining the execution to a certain depth. The efficacy of symbolic execution is twofold, one is automatic generation of test inputs and other is high coverage. Symbolic execution can be used for different purposes, such as bug detection, program verification, debugging, maintenance, and fault localization [10].

In contrast to classical symbolic execution, modern symbolic execution techniques try to decreases the complexity of path constraints [11 12 16], as constraint solving is computationally the most expensive part of symbolic execution. Recently, symbolic execution is used in combination with concrete execution [14 15 16] and uses program instrumentation avoiding a complete program interpreter required in classical symbolic execution. This hybrid program execution enhances coverage while avoiding the computational cost associated with full-blown symbolic execution which exercises all possible execution paths [15]. Techniques such as DART [16] instruments the program to calculate an input vector for the next execution during each execution. The input vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths. Similarly, CUTE [17] uses dynamic symbolic execution and represents all possible inputs using a logical input map. It use the symbolic execution to generate inputs that direct a

program to alternate paths, and to use the concrete execution to guide the symbolic execution along a concrete path. KLEE [11] performs a number of query optimizations to make them simpler so that reducing computational complexity of the constraint solver. Additionally, it represents program states compactly and uses search heuristics such as random path selection and coverage optimized search to get high code coverage. This makes it scalable to large programs.

Symbolic execution is used extensively in recent years for test input generation and program analysis. We have used the concept of symbolic evaluation to generate path constraints for image processing applications. These path constraints are solved and test images are generated on the basis of their solutions.

Typically, operations applied on images in spatial domain are point operations or neighboring pixels operations. Unlike point operations, the neighborhood operations perform modification of pixel value depending on the selected pixel and its neighboring pixels [18]. The neighborhood operators can be classified according to type of domain, type of neighborhood and their reclusiveness. The two types of domains consists of numeric or symbolic data. The numeric domain operators are arithmetic whereas symbolic domain operators are Boolean [19]. Usually, 4-connected or 8-connected neighborhood is used for neighboring operations. These neighborhood operations are useful in preprocessing algorithms, labelling and matching etc. We have studied applications using these neighborhood operations as test cases for generating test images with symbolic evaluation.

3. Motivation Example

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. The program applies thresholding to convert gray pixels to black and white pixels in the resulting image. It 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 given 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

binariz.m

```
1:  function   binaryImage=binarize (gray Image)
2:  threshold=128;
3:  [ r , c ] = size (grayImage);
4:  binaryImage = zeros (r-2,c-2);
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

which are to be executed symbolically. In the above example, the variables p_1 to p_9 contain the values of eight neighboring pixels of a centered pixel p_5 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.

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. 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. 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:

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

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 and stored in symbolic variable structure. 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:

$$avg = \sum_{x=1}^{9} P_x / 9 \tag{2}$$

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:

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

Whereas the alternate path condition is:

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

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 can range from 0 to 255 for an 8-bit image. A path condition is solved by generating arbitrary values from the range until the equation is satisfied. Path condition can have one specific solution or a number of different solutions. The above constraints have multiple solutions for each path ranging from 0~127and 128 to 255.

The solutions computed for the path condition are used to synthesize test images. 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 of path condition into a single test image. A test image ensures that the program under test will follow a specific path when executed. This creates ease of testing oracle and debugging process. In above example, two test

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						

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 a part of dead code.

4. Approach

IMSUIT takes an input program written as a set of statements, which consists of assignments, branching statements and loops. An assignment statement can be a simple or a mathematical equation. 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 branching statements have a Boolean expression followed by statements and then termination

```
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

of 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 a false condition of a 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 to extract necessary information required to execute the program symbolically. Stacks and structures are used to store the program states are updated using the extracted information. After parsing and updating program state, the statement is executed symbolically and path constraint is 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 is computationally most expensive part of symbolic execution. To find concrete solutions, the simplified path constraint are passed to a constraint solver. The concrete

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

1:  symbolicEvaluator(line, symb_vars)
2:      ignore spaces & comments
3:      Token = getToken(line)
4:      if Token ∈ keyword
5:          Parse line
6:          Update corresponding structure
7:          Update Stack
8:      else
9:          if Token ∈ reservedWords
10:             Parse line
11:         else
12:             check the assignment
13:             if assignment == fucntionCall
14:                 Parse function call
15:                 Update Corresponding Structure
16:             else
17:                 if assignment == simple
18:                     if LHS ∈ variable structure
19:                         override variable value
20:                     else
21:                         new varriable initialized
22:                         write to structure variable
23:                     end
24:                 else
25:                     assignment == equation
26:                     parse RHS of equation
27:                     write to LHS variable structure
28:                 end
29:             end
29:         end
30:     end
```

Figure 3 Symbolic Evaluator

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

```
Input:    Constraint
Output:  Solution to Constraint

1:    constraintSolver(constraint)
2:       get the symbolic variables
3:       while (¬satisfied)
4:                generate a set of random numbers
5:                set symbolic variables values
6:                solve the constraint
7:                if constraint is solved
8:                         satisfied = 1
9:                end if
10:     end while
```

Figure 4 Constraint Solver

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 *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 statement is checked for an assignment statement by expecting an assignment operator. 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 which can be a simple or combinations of several conditions. The multiple conditions are concatenated using logical *AND* or *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 with the child. Whereas in case of *elseif* the false condition of the parent if condition is concatenated with the child's Boolean condition. 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 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

Table 2 Results of Modules of OCR

Sr.	Function	Lines of Code	No of Paths	No Test Images
1	Robust Filtering	56	3	3
2	Distance Transform	95	7	7
3	Global Threshold	31	2	2
4	Smooth Threshold	64	4	4
5	OCR Alphabets	164	27	27
6	OCR Numbers	106	11	11
7	OCR Special Characters	106	27	27

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 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.

5. Results

We have developed a tool IMSUIT in Matlab to generate test images for functions written in Matlab. It consists of 2000 lines of code. IMSUIT is tested on different functions and modules, Table 1 shows the results of test images created and their paths.

IMSUIT is tested on different modules and functions to show its applicability and effectiveness. The first function is *Robust filtering,* which removes salt and pepper noise from input image. The program has three different paths which classify input pixels on the basis of minimum and maximum values of 8-neighboring pixels. There can be different combinations of the neighboring pixels to fulfill the path constraints. For first path, the value of current pixel is greater than the maximum value of the 8-neighbors. Figure 5(A) shows the test image generated by IMSUIT and the plot shows the difference between the generated input test image and the resultant output image. The values in the plot are all positive which shows that the corresponding value of pixel is greater than the maximum value of the neighboring pixels. This path represents white noise removal in the image. For the second path, the current pixel value is replaced with the minimum value of neighboring window if the value is less than the minimum value. Figure 5(B) shows the test image created for this path and the plot shows the difference of generated input test image and the resultant output image. All the values are negative in the plot which shows that the value of center pixel in a 3x3 window of input image is less than the corresponding vale of output image. The plots in 5(A) and 5(B) shows the spread of different input combinations generated randomly for satisfying path constraint. The second path represents black noise removal in the image. For the third case, if the pixel is between minimum and maximum values of the neighboring pixels then the value of current image remains same. Figure 5(C) shows the test image generated for this path and plot shows that there is no difference between input and output image. In this way we can test the input program for different combinations of inputs and images are generated for each path of the program under test.

The second function *Distance Transform* finds distance of current white pixel with the nearest black pixel in a window of 25x25 pixels. Figure 6 (C) shows a

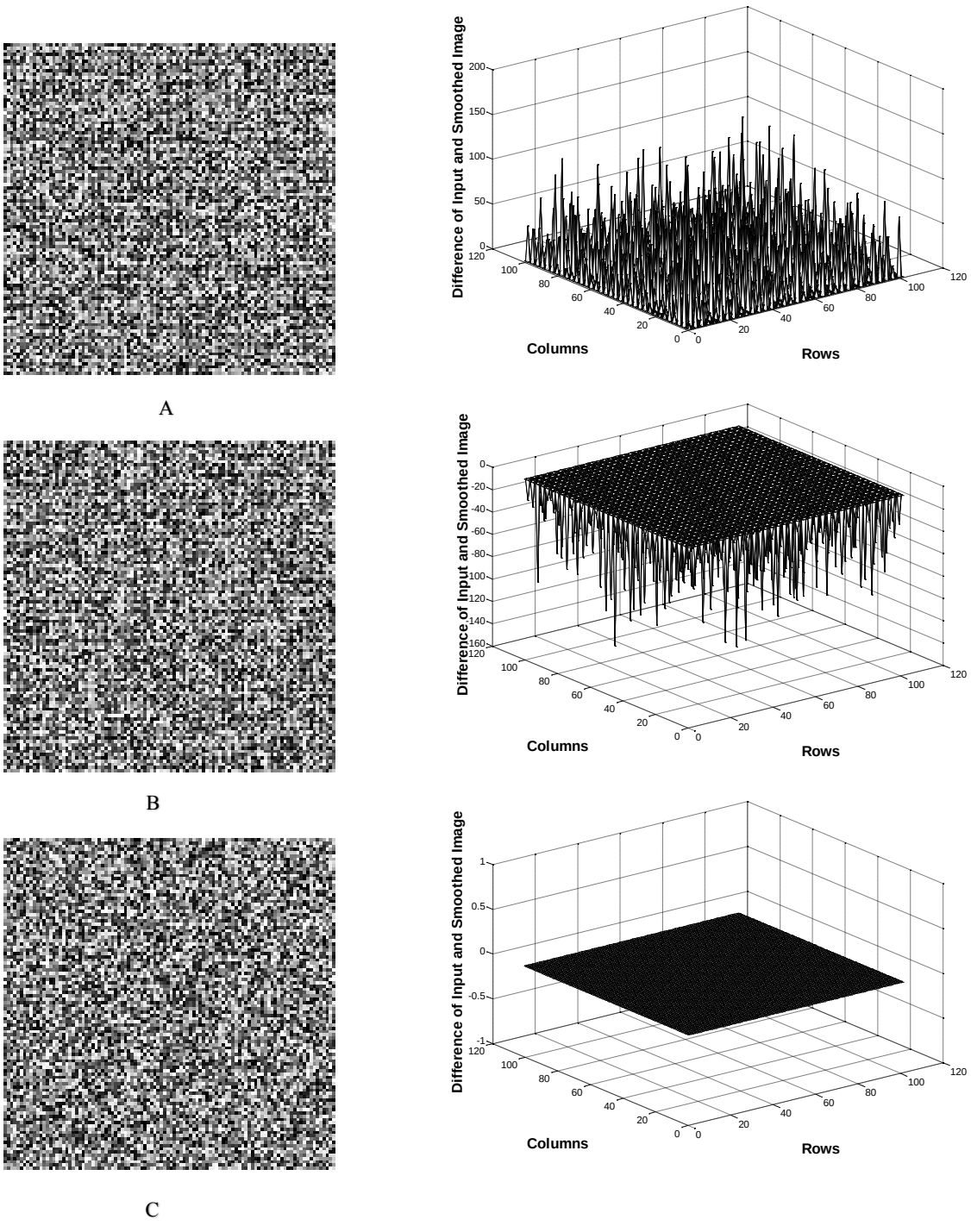

Figure 5 Test Images and Evaluation of Robust Filtering Application

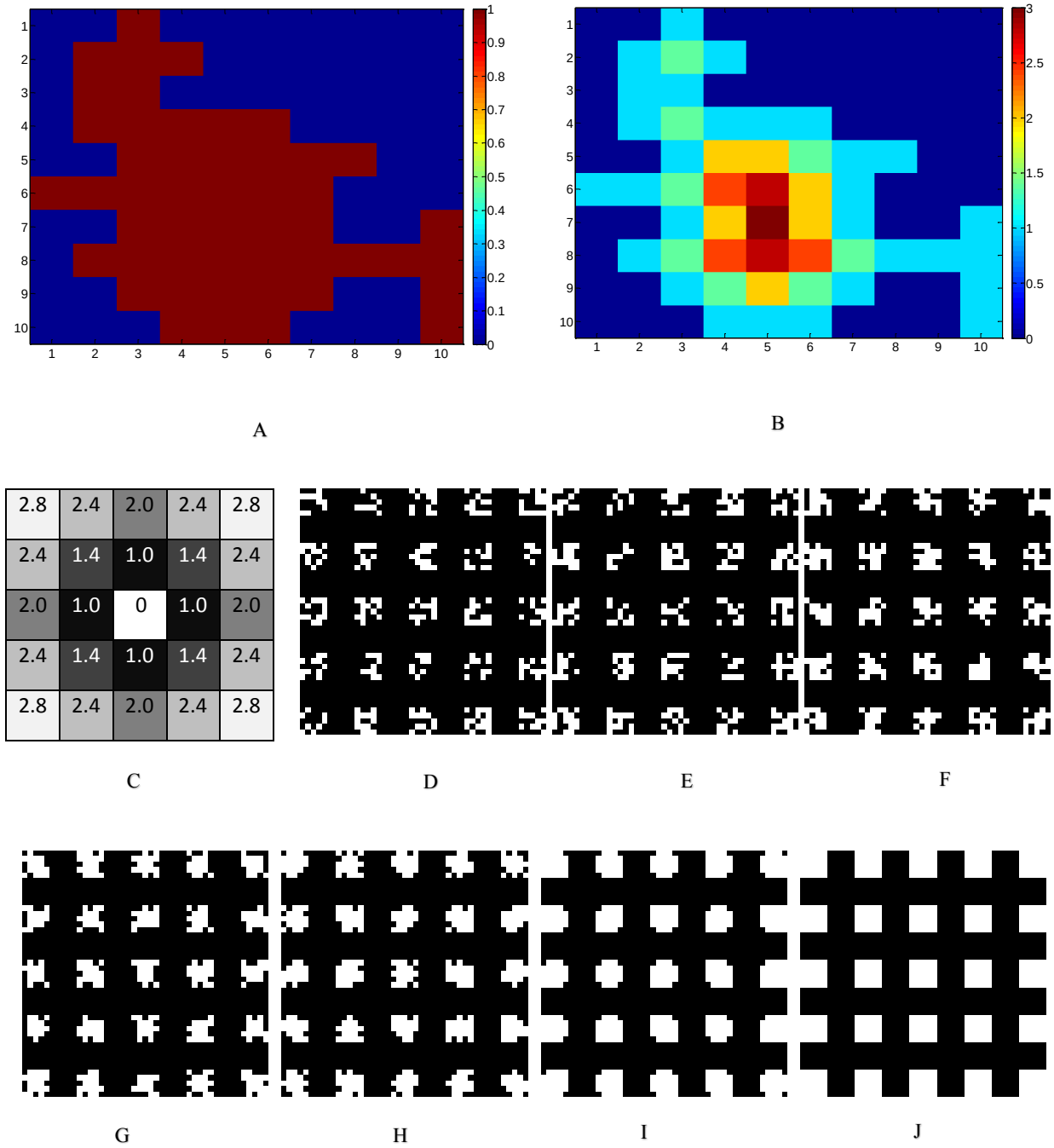

Figure 6 Distance Transform (Qausi) and Test Images Generated

table of Qausi Distance to compute distance transform. There are 7 different paths in the program for the given distances. Figure 6 (A) shows a hand crafted 10x10 test image and Figure 6 (B) shows its distance transform image using different color map. The distance is scaled from 0 to 2.8. The pixels having distance greater than 2.8 are classified as 3. Figure 6 (D) to 6 (J) shows the seven test images generated for each path. In each image different possible combinations of the solution for path constraint are generated randomly by IMSUIT. When these images are given as input to the program under test, the resulting image have a single value of the distance computed. In this way we can generate different size of images by using IMSUIT whereas the manual generation of such images is a very tedious and difficult task.

The third 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 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 pixels. Whereas the output of second image is all white pixels. Violation of the test oracle points to a bug in the function. The fourth function smooth threshold binarizes the input image and also eliminate noise, which causes problems in character recognition. This function uses three different bands of pixel values to binarize input image. It 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 fifth function is OCR Alphabets which searches alphabet patterns of 25x25 pixels in the given image. This function has 27 paths and 27 images are generated for each path. The sixth 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 seventh 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. 7 shows the input images generated for alphabets, numbers and special characters. Fig. 8 shows

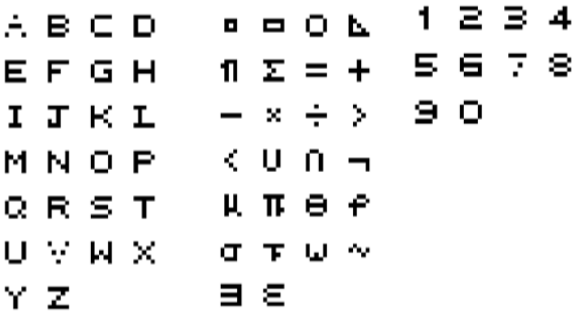

Figure 7 Test Image Generated for OCR

Figure 8 Binarized Image

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, test images are required covering all paths which can be difficult in some cases. Using IMSUIT we can generate test images and also we can control variation in the test images for testing purposes and analyzing the performance of the program under test.

6. Discussion

Synthetic image generation is a research tool which is extensively used in image processing testing and evaluation [20, 21]. A synthetic database allows the economical generation of images, which represent scenarios that are difficult or impossible to obtain in real data and provides a direct control of image parameters [22]. Synthetic images are also used for testing of image processing applications [23 24]. However, it is difficult

to produce test images and evaluate their test results. An intuitive way to generate a test image, is to randomly select independent values of each pixel called random binary image model [24]. For a binary image, the pixel values are selected with some probability for black and white pixels. Boolean model [25] is introduced by random points into which random grains are translated. The number of random points follows a Poisson distribution.

To test image operations, Mayer [23] applied models from stochastic geometry for random input generation and used statistical heuristics to compare the results. In image processing and such other complex input systems the exact output of a program under test cannot be verified. Despite of its limitations the statistical oracle can be used to verify some statistical characteristics of the actual test results. In case of a failure, there are multiple test cases required to identify the bug. The scheme is useful to generate random tests for basic image operations such as dilation and erosion but the limitation of the approach is that statistical oracle requires a couple of test cases to decide a pass or fail. Furthermore, the approach cannot check all the characteristics of the output and only works if statements about the statistical distribution of the test result is possible, which limits the applicability of the approach. In his extended work, Mayer [24] presents assessment of models that can be used for random test data generation and presented a method to generate additional test cases. The validation is performed using mutation analysis for a concrete image processing operation.

The techniques discussed are for specific algorithms and their use is limited to binary images as they are using binary models. IMSUIT is extending testing goals to gray scale images and the programs where arithmetic, and logical operations are applied on neighboring windows. It allows automatic generation of test images and different random combinations of the solutions of path constraint.

7. 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 real programs. 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 constraint solver based on random number generation. The path constraints generated for image processing systems comprise of pixels as symbolic variables and their values ranges from 0 to 255 for 8-bit images. This fact makes possible the use of a simple solver. The result shows that it has successfully created test images for each program path.

References

1. Beizer, Boris. "Software testing techniques. 1990." New York, ISBN: 0-442-20672-0.
2. King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.
3. 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.
4. 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.
5. Korel, Bogdan. "Automated software test data generation." Software Engineering, IEEE Transactions on 16.8 (1990): 870-879.
6. 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.
7. Clarke, Lori A. "A system to generate test data and symbolically execute programs." Software Engineering, IEEE Transactions on 3 (1976): 215-222.
8. Cadar, Cristian, et al. "Symbolic execution for software testing in practice: preliminary assessment." Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011
9. Li, Yi, et al. "Symbolic posium on Principles of programming languages. ACM, 2014.
10. Clarke, Lori A., and Debra J. Richardson. "Applications of symbolic evaluation." Journal of Systems and Software 5.1 (1985): 15-35.
11. 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.

12. Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.
13. Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.
14. Larson, Eric, and Todd Austin. "High coverage detection of input-related security faculties." Proceedings of the 12th conference on USENIX Security Symposium-Volume 12. USENIX Association, 2003.
15. Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.
16. Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.
17. Sen, Koushik, and Gul A. Agha. "Concolic testing of multithreaded programs and its application to testing security protocols." (2006).
18. Haralick, R. M. "Some neighborhood operators." Real-Time Parallel Computing. Springer US, 1981. 11-35.
19. Haralick, Robert M., Stanley R. Sternberg, and Xinhua Zhuang. "Image analysis using mathematical morphology." Pattern Analysis and Machine Intelligence, IEEE Transactions on 4 (1987): 532-550.
20. Cappelli, Raffaele, Dario Maio, and Davide Maltoni. "Synthetic fingerprint-database generation." Pattern Recognition, 2002. Proceedings. 16th International Conference on. Vol. 3. IEEE, 2002.
21. Prakosa, Adityo, et al. "Generation of synthetic but visually realistic time series of cardiac images combining a biophysical model and clinical images." Medical Imaging, IEEE Transactions on 32.1 (2013): 99-109.
22. Thomson, Chris J., Thomas T. Steck, and Ken G. Krebaum. "Synthetic Image Generation for Automatic Target Recognizer Evaluation." 1988 Orlando Technical Symposium. International Society for Optics and Photonics, 1988.
23. Mayer, Johannes. "On Testing Image Processing Applications with Statistical Methods." Software Engineering. 2005.
24. Mayer, Johannes, and Ralph Guderlei. "On random testing of image processing applications." Quality Software, 2006. QSIC 2006. Sixth International Conference on. IEEE, 2006.
25. Chiu, Sung Nok, et al. Stochastic geometry and its applications. John Wiley & Sons, 2013.APA