Journal of Software Engineering and Applications, 2015, 8, 603-616
Published Online December 2015 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2015.812057

Automatic Test Data Generation for Java Card Applications Using Genetic Algorithm

Saher Manaseer[1], Warif Manasir[1], Mohammad Alshraideh[1], Nabil Abu Hashish[2], Omar Adwan[3]

[1]Department of Computer Science, The University of Jordan, Amman, Jordan
[2]Al Israa University, Amman, Jordan
[3]Department of Computer Information Systems, The University of Jordan, Amman, Jordan
Email: saher@ju.edu.jo, warefalmanaseer@gmail.com, mshridah@ju.edu.jo, nabil.mosa@iu.edu.jo, adwanoy@ju.edu.jo

Received 23 November 2015; accepted 21 December 2015; published 24 December 2015

Abstract

The main objective of software testing is to have the highest likelihood of finding the most faults with a minimum amount of time and effort. Genetic Algorithm (GA) has been successfully used by researchers in software testing to automatically generate test data. In this paper, a GA is applied using branch coverage criterion to generate the least possible set of test data to test JSC applications. Results show that applying GA achieves better performance in terms of average number of test data generations, execution time, and percentage of branch coverage.

Keywords

Software Testing, Genetic Algorithm, Java Smart Card

1. Introduction

In recent years, software testing is becoming more essential in the software development industry, and it is a vital component of software engineering. Indeed, software testing is a broad term encircling a variety of activities along the development cycle and beyond aimed at different goals. Software testing represents 40% of software development budget [1]. Software testing has become more difficult because of the vast array of programming languages, operating systems, and hardware platforms that have evolved in the last decades [2]. In practice, testing cannot be exhaustive and some approaches to test selection must be used [3].

Software testing is the process of executing a program with inputs and observing the results. The aim of software testing is to generate a minimal number of test data such that it detects as many faults as possible [4]. Test data generation techniques attempt to find a program input that will satisfy the testing requirement. The tester chooses test data inputs in order to achieve some given structural coverage criteria. The automation of test data generation is an important step in the reduction of the cost of software development and maintenance [5]. GA is a leading technique of evolutionary software testing for generating test data to examine branch, statement and path coverage [6].

GA is well applied in software testing but it has never been used in testing of JSC applications. Smart cards store and process information through electronic circuits embedded on board [7]. A smart card can be intelligent, *i.e*. offers reading, writing, and calculating capability, or a memory card which offers information storage; only Java-based smart cards are capable of running Java programs [8]. JSC programs are called applets. Multiple applets can reside on the same card at the same time, and can be updated dynamically [9]. However, due to limitation in memory resources and computing power in the smart card, not all the language features of the Java programming language are supported on the JSC [7].

In this paper, we will apply GA to automatically generate test data to test JSC applications. The goal is to test JSC applications to reveal as many faults as possible, with a least possible number of test data using GA according to branch coverage criterion.

The rest of the paper is organized as follows: Section 2 presents the fundamentals of GA. Section 3 gives an overview of JSC. Section 4 describes the literature review on JSC software testing. Section 5 presents the proposed methodology and experiments. Section 6 evaluates the results and concludes the paper. Section 7 gives an overview of our future work.

2. Genetic Algorithms

The GA created by John Holland in the 1970's at the University of Michigan (USA), is an evolutionary algorithms inspired by biological evolution principles such as natural selection and genetic inheritance [10]. Evolutionary computing techniques attempt to simulate the biological process on the computer in order to solve problems in many applications with great complexity.

Holland simulated the methods used when biological systems adapt to their environment in computer software models to solve optimization problems [11]. In the context of software testing, the basic idea is to search the search space for input that satisfies the testing criteria.

The possible solutions to a problem being solved are represented by a population of chromosomes. Each chromosome is made of genes, and the value of a gene can be represented in binary, numerical or string of characters depending on the problem to be solved. GA uses three operators on its population which are described below:

- **Selection**

The selection for reproduction is the first operator applied on the population. Here, the selection operator chooses two individuals from a generation to become parents for the reproduction process to produce an offspring for the next generation. Individuals are chosen based on their fitness; the fitness function measures the suitability of a chromosome to survive in an environment. Before being included in the next generation, the chromosome must undergo the crossover and mutation operations which will be discussed briefly. Selection is a key factor of affecting the performance of evolutionary algorithms [12].

- **Crossover**

This operator combines two chromosomes to produce a new offspring. With the idea that new individuals will be closer to a global optimum, the best genes of the parent chromosome are combined to produce an offspring that is better than the parent [13]. Crossover occurs during evolution according to a predefined crossover probability. Crossover is of different types. It can be at one point crossover, double point crossover, or uniform crossover.

- **Mutation**

The aim of this operator is to maintain diversity among generation of one population chromosomes to the next. In mutation, one or more gene values are altered, according to a predefined mutation probability, from its initial value, thus resulting in a new gene value added to the gene pool. As a result, the mutation avoids the solution to fall into local optima of the search space. De Jong showed that mutation rates can have a destructive characteristic. If the mutation rate is too high, search is like a random search, and if too low the search might get stuck at local minima [14].

3. Java Smart Card

Smart cards currently exist for a vast verity of applications. A smart card is a secure, efficient and cost effective computational device of an embedded system that comprises of a microprocessor, memory modules (RAM, ROM, and EEPROM), serial input/output interfaces and data bus. The operating system of the chip is contained in ROM and the applications are stored in the EEPROM [15].

JSC is an open standard from Sun Microsystems for a smart card development platform. JSC brings the benefits of the Java technology to the world of smart cards. Smart cards created using the JSC platform have Java applets stored on them. The applets can be added to or changed after the card is issued. Each applet has unique Applet Identification (AID).

Java-based smart cards store data on an integrated microprocessor chip. Applets are loaded into the memory of the microprocessor and run by the Java Card Virtual Machine (JCVM). JSC enables multiple application programs to be installed and coexist independently [9].

The JSC technology supports a limited subset of Java functionalities to develop applets that run on smart cards. JSC does not support long, double, character, string, and float data types, multi-dimensional arrays, and threads. The supported features include byte, short and Boolean data types in addition to one-dimensional arrays [16].

Smart cards are deployed in a wide range of industries to support access, identity, payment and other services. An example of JSC application is the electronic purse payment application. The smart card carries a monetary value to allow the card holder to issue transactions [1].

Another example of smart card application in the governmental field is the e-Passport issued by the United States Government Printing Office and the Department of Homeland Security. The e-passport contains a small embedded integrated circuit that stores the same data a regular passport hold digitally [17].

As in typical communications, data packages are interchanged following set of protocols, JSC uses data packages for communication. Data packages used in smart card communications are called Application Program Data Units (APDU). APDU allow communication between the card application and the client via commands and response messages [18].

4. Literature Review

Software testing has become more difficult because of the vast array of programming languages, operating systems, and hardware platforms that have evolved in the last decades [2]. Software quality is the central concern of software engineering. Testing is the single most widely used approach to ensuring software quality [8].

Most research concentration was on testing JSC applets using models. Model-based testing can be easily introduced to the development process of the smart card, automatic test generation process saves 30% of labor when modeling task is included compared with manual testing. However, model-based testing is limited to functional testing [19].

Martin and Bousquet (2001) proposed a solution for JSC applet validation. In order to perform applet validation, the authors used a conformance testing approach that is black box testing [20].

Automatic test data are generated from the specifications and test purposes of the application. The specifications are expressed with a UML model, and then automatically translated into a Labeled Transition System (LTS). After that, the authors used the Test Generation with Verification (TGV) tool to automatically produce test data from the LTS. The strategy followed here is to test each function for every normal use and every possible misuse. Results show that the proposed approach by the authors offer high confidence in the application conformity regarding its UML specification [20].

Van Weelden, *et al*. (2005) showed that automated, formal, specification-based testing of smart card applets is feasible, and that errors can be detected [21].

Bouquet, *et al*. (2005) focused on functional testing based on formal models of functional specifications of the software under test to automatically generate test data. Functional testing aims at ensuring the correctness of operations and their conformance to the functional requirements. Unfortunately, formal methods demand real effort in order to formalize the specifications of the smart card applications [19].

Most of the testing conducted for smart cards and automatic test case generation are model-based testing. Model-based testing requires additional cost to construct the model and the test case specifications [22]. None of the applied automatic test data generation tools used heuristic search techniques. Although, heuristic search techniques have proved their strength in the software testing field especially GA, but they were never introduced to the JSC world.

5. Methodology and Experiments

In this section, we will explain the details steps of applying GA to automatically generate test data for JSC applications to achieve branch coverage with minimal test data. This approach is considered the first to use GA to test JSC applets. JSC applets are used in vital areas in our lives so observing the execution of the applets to validate whether they behave as intended and identify faults is an essential process that must be considered, especially because the JSC applet structure is complex. As software systems become more complex and embedded in industries, the cost of failure becomes more severe [23]. Such challenges can be faced by GA due to its capabilities of testing complex software. GA has been successfully applied in the area of software testing.

5.1. Experimental Settings

The following sets of parameters were considered for test data generation using GA.

Fitness function: the fitness value for an individual solution is computed according to Korel's Local Distance (LD) function [24]. The predicate distance is calculated according to Korel's Local distance function in **Table 1**, and each branch predicate is transformed to the equivalent predicate provided in the table. A predicate has only two outcomes, either it evaluates to TRUE or FALSE. A branch is traversed only if the predicate is evaluated to TRUE and not traversed if the predicate is FALSE. Korel assumed that a FALSE branch is greater than zero and a TRUE branch is less than or equal zero [24]. K is the smallest positive constant in the domain (*i.e.* 1 in the case of integer domain).

- Coding: binary string.
- Selection method: tournament selection.
- Single point crossover.
- Mutation probability: 0.05.
- Stopping criteria: fitness value equals 0 or number of generation's equals 700.

5.2. Evaluation Parameters

The performance of the GA to automatically generate test data for JSC applets was assessed by test data generation time, average number of generations and coverage target that is branch coverage. The average values were calculated after running the algorithm ten times for every program unit, this experiment was done for five times, every time with different population size. The populations sizes considered are 30, 50, 70, 90 and 110. After each execution, we recorded the average number of generations and the average execution time in addition to the coverage percentage achieved. It is useful to have the smallest average number of generations because it means that GA generates required test data with the small number of generations which is required.

Table 1. The Korel's Distance Function.

Branch Predicate	Branch Function
$A = B$	$ABS(A - B)$
$A \neq B$	$K - ABS(A - B)$
$A < B$	$(A - B) + K$
$A \leq B$	$(A - B)$
$A > B$	$(B - A)+K$
$A \geq B$	$(B - A)$
X OR Y	MIN (Distance(X), Distance(Y))
X AND Y	MAX (Distance(X), Distance(Y))

5.3. Programs under Test

We applied the algorithm to eight JSC programs, these programs are tested for the first time using GA, and these programs are described in **Table 2**, where LOC stands for lines of code in each program. The size of the programs is different from 59 lines to 4277 lines.

- **Passport Applet**

The passport applet is an open source, card side implementation of the Java Machine Readable Travel Documents (JMRTD) that follows the International Civil Aviation Organization (ICAO) standards. The smart card chip holds the biometric information of the passport holder thus provides security and protection against identity theft [17].

This program consists of 12 classes each class is responsible for a specific functionality such as processing APDU's, initializing the applet, encryption and decryption, scanning tags, and other instructions. The source code of the applets is available at (http://sourceforge.net/).

The nature of the code was diverse; different data types, structures (simple and compound predicates) exist, as well as different nesting levels of conditional statements. For example, **Figure 1** shows calcLc From Padded Data ( ) method, this method computes the actual length of a data block as byte value.

- **CoolKey Applet**

CoolKey Applet generates cryptographic keys on the card and allows external keys to be inserted onto the

Table 2. Programs under test.

Program Name	LOC	Number of Branches
Passport	3548	280
Network Connection Tracker	413	50
OATH	810	154
PKI	1738	220
RSACrypto	147	22
Calculator	182	48
CoolKey	4277	310
HelloWorld	59	6

```
public static byte calcLcFromPaddedData(short[]
apdu, short offset, short length){
        if(length>=0){
      if((apdu[(short)(offset + length)] & 0xff)!=
0x80){
                    return (byte)(length & 0xff);
                }
                else{
                    return (byte)(length & 0xff);
                }
        }
        return 0;
}
```

Figure 1. CalcLc From Padded Data ( ) method.

card. These keys can be used in encryption and decryption operations after proper user authentication. When a new key is created and the user plugs it in for the first time, the key is automatically supported with certificates and unique PIN. The source code is available at (https://github.com).

In this applet there are five main classes with a total of 310 branches. The applet contains all combinations of nested if-statements, switch statements, for loops, do-while, and calls to other methods inside if-statements. In spite of the complexity of the branches, the test data generated by the algorithm achieved 100% coverage.

- **Network Connection Tracker Applet**

This applet keeps track of the account information for a wireless device connecting to a network service. The device has a local area network and can operate remotely. The applet provides a number of functionalities via specific commands such that you can add credits to the account and inquiry the amount of available credits. The source code of the applet is available at (https://kenai.com).

In this program there are 50 branches, where there are nested if-statements and switch statements, with simple and composite predicates. Although the structure of this program is not as complex as the previously discussed counterparts, full branch coverage was not achieved. Only 98% of the branches were covered at all population sizes specified.

Figure 2 below provides the code with the uncovered branch. The else branch of the if-statement on line 12 was not traversed because the algorithm could not find the test data that covers this branch. Unfortunately the algorithm reached the maximum number of generations without any improvement on the generated solutions. The main problem in this section of code was the incorrect handling of boundary values; a value out of range exception was thrown because the value of INACTIVE_AREA is out of the range of its data type (*i.e.* short) that allows it to traverse the else branch on line 12.

- **Calculator Applet**

This program is a JSC calculator, the instructions available by this calculator are the ASCII characters of the keypad keys: "0" – "9", * "-", * "x", ":", "=". The applet has a simple structure of if-statements and switch statements. Most of the conditions in this program used the "==" operator and logical "||" operator. The source code of the applet is available at (http://www.codeproject.com/).

- **RSACrypto Applet**

The RSA cryptosystem is the most widely-used public key cryptography algorithm invented by Ron Rivest, Adi Shamir, and Len Adleman [25]. The RSA algorithm can be used to encrypt messages and digital signatures.

The RSACrypto JSC applet encrypts and decrypts data blocks of at most 128 bytes long using RSA keys which are generated off-card and uploaded to the card. The source code of the applet is available at (https://www.cs.ru.nl).

```
1.    private void timeTick(APDU apdu) {
2.    byte[] buffer = apdu.getBuffer();
3.    byte numBytes = (buffer[ISO7816.OFFSET_LC]);
4.    byte byteRead = (byte) (apdu.setIncomingAndReceive());
5.    if ((numBytes != 2) || (byteRead != 2)) {
6.    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
7.    }
8.    short newAreaCode=(short) (buffer[ISO7816.OFFSET_CDATA+1] &
0x00FF);
9.    if (newAreaCode != INACTIVE_AREA) {
10.   activeAreaCode[0] = newAreaCode;
11.   }
12.   else {
13.   resetConnection();
14.   ISOException.throwIt(SW_NO_NETWORK);
15.   }
```

Figure 2. Timetick () method.

- **HelloWorld Applet**

The JSC HelloWorld applet is the simplest applet that can be written, it outputs "HelloWorld" to the off-card application after receiving a specific APDU. This applet is the smallest applet tested by the algorithm, we wanted to test variety of programs that differ in functionality, structure and most important the number of branches. This program has six branches with combination of if-statements and switch statements. The applet source code is available at (https://kenai.com).

- **OATH Applet**

OATH (Open Authentication) is an open specification for One-Time-Passwords (OTP) developed by the Initiative for Open Authentication. It includes public, open specifications for event based authentication and time-based authentication using encryption techniques. OATH is capable of generating an event-based OTP that is triggered by a button press. In addition to event-based OTP, a time-based OTP is generated automatically every 30 seconds.

The OATH applet is designed for use on JSC to provide a one-time password generation service that conforms to the OATH specifications. The OATH applet implements a PIN user authentication, triple-DES encryption and decryption, and a secure hashing generation. This project implements the card functionality used on the YubiKey Neo device that is sold by Yubico [26]. The source code is available at (https://github.com).

In this program there are 154 branches in addition to a complex structure of nested if-statements and for loops as well as switch statement. Simple and complex predicates exist a lot in this program.

- **PKI Applet**

Public Key Infrastructure (PKI) is an architecture that supports secure digital communication by issuing digital certificates. PKI is based on public and private keys to encrypt and digitally sign information; it offers a high level of authentication for users.

As with web browsers, web servers, and many other types of hardware and software, PKI standards support smart cards [27].

This applet was one of the complex applets to test, it has 220 branches. The structure of the code was complex as well, there exists many nested if-statements, switch statements, while loops, do-while, and for loops all together. The source code is available at (http://sourceforge.net/).

5.4. Experimental Results

This section presents the results of the conducted experiments and provides a discussion of the results. **Figure 3**

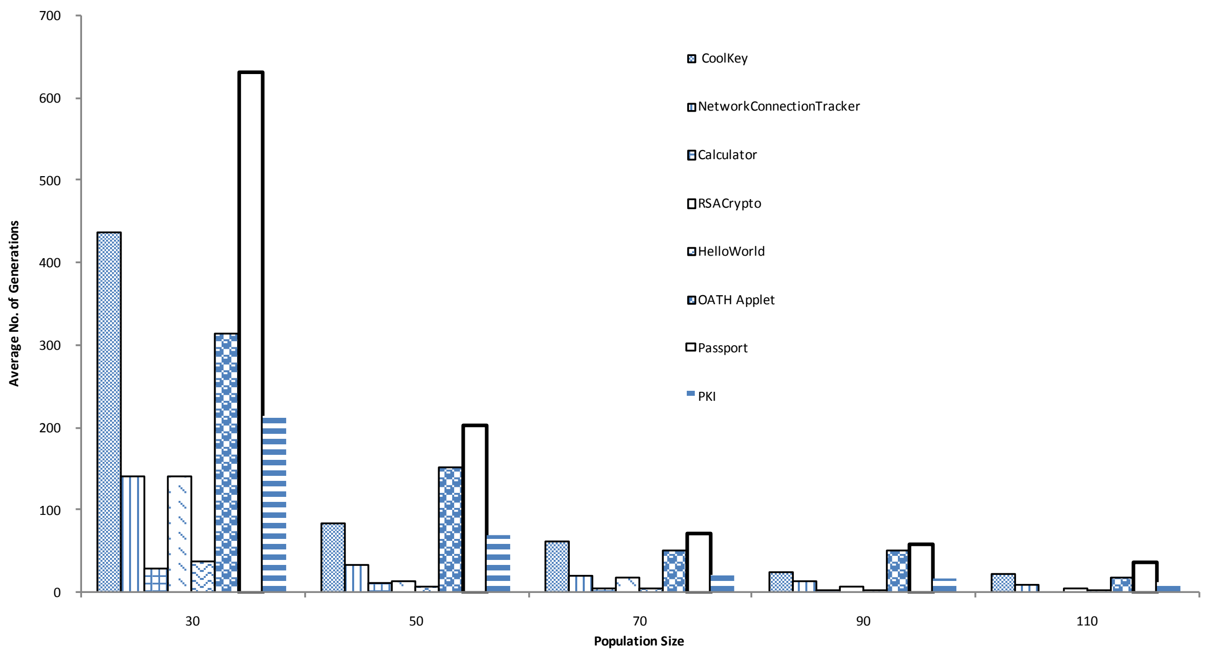

Figure 3. Average no. of generations for software under test.

presents the average number of generations generated for each applet as the population size increases. **Figure 4** presents the time consumed by the algorithm to generate the required test data.

It is clear from **Figure 3** that the average number of generation decreases as the population size increases same thing in **Figure 4** where the execution time decreases as the population size increases. This is due to the fact that the probability of finding optimal solution increases as the sample size of candidate solutions from the search space increases leading to a less number of generations and consequently execution time since the optimal solution would be near. Moreover, increasing the population size increases the accuracy of the GA because the greater the population size is the greater the chance that the population contains a chromosome representing the optimal solution [5]. The GA can generates the required test data with a small number of generations and less time since it selects the individuals with the best fitness such that it accelerates the process of searching and consequently reduces the time required to find the right individual.

5.5. Performance Evaluation

The experiments in this paper address eight different JSC programs to test the proposed algorithm, each program with different number of branches and unique characteristics.

Three evaluation parameters are used, average number of generations, execution time, and percentage of coverage that is branch coverage. The behavior of each parameter is monitored against the dynamic change of population size. Each parameter was used to study the behavior of the GA when applied to JSC programs.

Starting with the first parameter, the average number of generations, the general behavior of this parameter was decreasing as population size increases. This is justified by the fact that the greater the population size the greater the chance that the population will contain a chromosome representing the optimal solution, the probability of finding optimal solution increases as the sample size of candidate solutions from the search space increases. The GA can generate the required test data with a small number of generations to reach the test target in all programs under test as the population size increase. Therefore, this behavior is reflected on the time of searching for the optimal solution throughout the produced generations.

In terms of coverage, out of eight JSC programs under test, seven programs achieved 100% branch coverage at different population sizes. It can be noticed that the percentage of branch coverage increases as the population size increases. This is due to the fact that more numbers of candidate solutions increases the probability of traversing a new branch.

The experimental results were satisfactory; the GA was capable of automatically generating test data to achieve 100% branch coverage for most of the JSC programs under test. The algorithm was able to handle different code structures with simple and complex predicates; however some performance degradation occurred

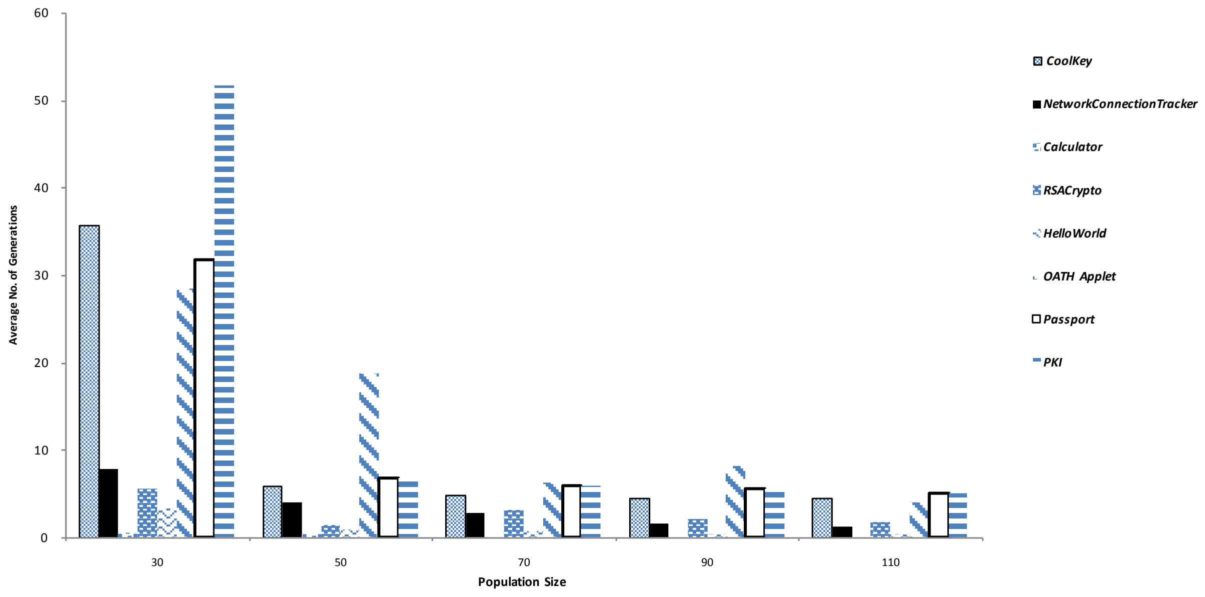

Figure 4. Execution time for software under test.

when dealing with complex predicates. In these cases, the performance of the algorithm was relatively different. For example, the execution time increased, and the average number of generations increased as well.

5.6. Java Card Applets Investigation

Testing JSC programs using GA revealed several aspects that must be considered when programming JSC programs.

One of the limitations is that JSC programs intensively use byte data type which limits the search space and throws an exception because the maximum value for byte data type is 127. This is an issue because the values of parameters specified are larger than 127 that is incorrect handling of boundary values. For example, in **Figure 5** the data type of p1p2 is byte but when it is compared to 0xdead an Input Mismatch Exception specifically "value out of range error" is thrown because the range of 0xdead exceeds 127. However, the test was performed on a modified version of the code.

Another limitation is that many variables are defined but not initialized, and they are used before initialization. For example, in **Figure 6** the Boolean variable found was used before initialization this led to an error and consequently the entire block of code is not reachable.

To sum up, many exceptions were thrown in the programs mostly Null Pointer Exceptions and Array Index out of Bound Exception, as well as different logical problems that were found similar to the ones we discussed above. This means that precise development and evaluation of JSC applets must be considered in order for it to function as expected. JSC applets must be tested using powerful testing techniques such as the GA that was successfully able to generate test data that revealed errors, forced exceptions to be thrown, and most importantly highlighted unreachable branches in the programs that were tested.

The test process used the following general flow shown in **Figure 7**. Moreover, a sample of the GA used is also presented in **Figure 8**.

6. Result Evaluation and Conclusion

In software development life cycle, software testing is considered as one of the most critical phases. The efficiency of a software test is directly related to code coverage. In turn, code coverage is greatly influenced by the test data, so providing efficient techniques to automatically generate test data is a key step.

A GA based on theory of natural selection is used to automatically generate test data to test JSC applets. The overall aim is to use the GA as search technique in order to find the required test data according to branch criteria to test JSC programs.

The experimental results show that branch coverage is achieved such that all test targets in all programs under test are reached, except for one program. The algorithm cannot find the test data that covers one of the branches of the Network Connection Tracker program. This means that the coverage percentage achieved is 99%.

In summary, we analyze the performance of the GA based on the average number of generations, execution time, and percentage of branch coverage. We measure the behavior of those parameters while changing the population's size, we start with initial population of size 30 then we increase it to 50, 70, 90 and 110. The GA shows good results in searching the search space for test data for every JSC program we tested.

The experiments show that the average number of generations decreases as the population size increases. It

```
byte ins = buf[ISO7816.OFFSET_INS];
byte p1p2= buffer[OFFSET_CLA];
if (ins == RESET_INS) {
      if (p1p2 == (byte)0xdead) {
handleReset();
            }
   }
```

Figure 5. Example of incorrect handling of boundary values.

```
          public void destroyObject(short type, short id, booleansecure,shortcurr,shortprev)
            {boolean found;
                 if(found) {
                short curr = obj_list_head;
                 short prev = -1;
                    found=false;
                if ( !found &&curr != -1 ) {
                   if(mem.getShort(curr,(short)OBJ_H_CLASS)==type                          &&mem.getShort(curr,
          (short)OBJ_H_ID)== id){
                                 found = true;
                            } else {
                                 prev = curr;
                                 curr = mem.getShort(curr, (short)0);
                            }
                       }
                       if(found) {
                            if(prev != -1)
            mem.setShort(prev, (short)0, mem.getShort(curr, (short)0));
                            else
                                 obj_list_head = mem.getShort(curr, (short)0);
                            if(secure) {
                                 Util.arrayFillNonAtomic(mem.getBuffer(),
                                      (short)(curr + OBJ_H_DATA),
                                      mem.getShort(curr, (short)OBJ_H_SIZE),
                                      (byte)0);
                            }
                            mem.free(curr);
                                       }
            }
```

Figure 6. Example of uninitialized variables.

has been clarified that such behavior occurs because the probability of finding optimal solution increases as the sample size of candidate solutions from the search space increases. Accordingly the second parameter, the execution time is affected. The execution time decreases or increases as the average number of generations decreases or increases.

From the experiments, we conclude that JSC programs are complex and require being tested using powerful techniques such as GA because the power of using GAs lies in their ability to handle input data which may be of

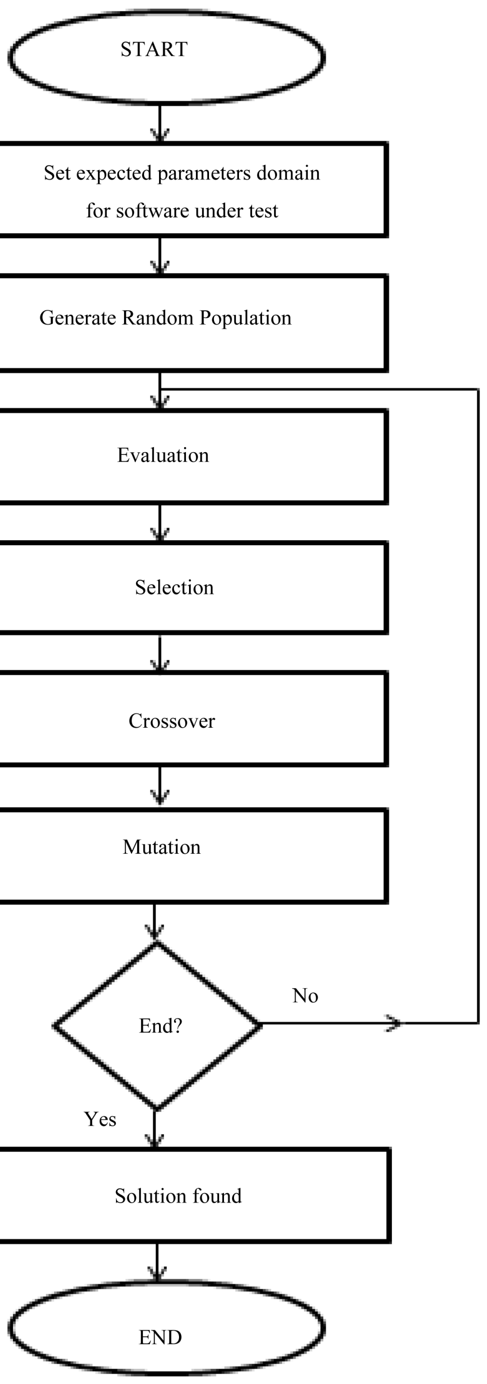
START
Set expected parameters domain for software under test
Generate Random Population
Evaluation
Selection
Crossover
Mutation
End?
No
Yes
Solution found
END

Figure 7. General test process.

complex structure, and predicates which may be complicated. As we have already mentioned, JSC applets are complex in structure and are implemented in important and highly demanding environments. Also, we highlight important issues that must be taken care of when implementing JSC programs; such as incorrect handling of boundary values.

The main advantage of using the GA as search technique is the strength of GAs; because the GA searches from a population of points rather than from a single point, thus reducing the probability of being stuck at a local

```
public class GA {
 private int NUM_CHROMOSOMES=30;
 private float MUTATE=(float)0.05;
 private Vector population;
int [] intArray= new int[NUM_CHROMOSOMES];
 double [] fitArray=new double [NUM_CHROMOSOMES];

 public GA(){
 generatePopulationRandomly();
 }
 private void generatePopulationRandomly(){
   population= new Vector();
   for(int i=0;i<NUM_CHROMOSOMES;i++){
 intrandomValue=(int)(Math.random()*MAX_NUMBER);
 population.add(new Chromosome(randomValue,MAX_POWER));
  }
 }
fitval= fitnessFuncofBranchData(intArray[t]);
           if(intNum==val){
                fitness=0;
               }

          else{
dest=(Math.abs(intNum-val));
          double k= Math.pow(1.001, dest);
          double k2=1/k;
          fitness=1-k2;
            }
  private Chromosome TournamentParentSelection(){
int index1= (int) (Math.random()*NUM_CHROMOSOMES);
int index2 = (int) (Math.random()*NUM_CHROMOSOMES);
intbestParentIndex;
       if(fitArray[index1]>fitArray[index2]){
  bestParentIndex=index1;
  }
  else{
```

Figure 8. GA example.

optimum, in addition to other advantages such as: it can be employed for a wide variety of optimization problems. GA is an effective global smart search method; it can solve efficiently the large space of complicated problems.

7. Future Work

The main aspect of the future work in this paper is to make a real-world prototyping of the algorithm on an actual JSC. Also, test JSC programs using different test coverage criteria such as path coverage in order to distinguish which is the most appropriate test coverage criterion for testing JSC.

References

[1] Stuber, G. (1996) The Electronic Purse: An Overview of Recent Developments and Policy Issues. Bank of Canada.

[2] Myers, G.J., Sandler, C. and Badgett, T. (2011) The Art of Software Testing. John Wiley & Sons, Hoboken.

[3] Sommerville, I. (2000) Software Engineering. Addison-Wesley, Harlow.

[4] Srivastava, P.R. and Kim, T.H. (2009) Application of Genetic Algorithm in Software Testing. *International Journal of Software Engineering and Its Applications*, **3**, 87-96.

[5] Pargas, R.P., Harrold, M.J. and Peck, R.R. (1999) Test-Data Generation Using Genetic Algorithms. *Software Testing Verification and Reliability*, **9**, 263-282. http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y

[6] McCart, J., Berndt, D. and Watkins, A. (2007) Using Genetic Algorithms for Software Testing: Performance Improvement Techniques. *Proceedings Americas Conference on Information Systems* (*AMCIS*), Colorado, 2007, 222.

[7] Coglio, A. (2003) Code Generation for High-Assurance Java Card applets. *Proceedings of* 3rd *NSA Conference on High Confidence Software and Systems*, Gold Coast, April 2003, 85-93.

[8] Tuteja, M. and Dubey, G. (2012) A Research Study on Importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models. *International Journal of Soft Computing*, **2**, 251

[9] Giorgio, Z. (2015) Understanding Java Card 2.0. http://www.javaworld.com/article/2076617/embedded-java/understanding-java-card-2-0.html

[10] Mitchell, M. (1996) An Introduction to Genetic Algorithms. MIT Press, Cambridge.

[11] Hermawanto, D. (2013) Genetic Algorithm for Solving Simple Mathematical Equality Problem. Cornell University Library, Computer Science, Neural and Evolutionary Computing, Indonesia, 1-10.

[12] Xie, H. and Zhang, M. (2009) Sampling Issues of Tournament Selection in Genetic Programming. School of Engineering and Computer Science, Victoria University of Wellington, Wellington.

[13] Spears, W.M. and Anand, V. (1991) A Study of Crossover Operators in Genetic Programming. Springer, Berlin Heidelberg, 409-418. http://dx.doi.org/10.1007/3-540-54563-8_104

[14] DeJong, K.A. (1975) Analysis of the Behavior of a Class of Genetic Adaptive Systems. Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

[15] El Farissi, I., Azizi, M., Lanet, J.L. and Moussaoui, M. (2013) Neural Network vs. Bayesian Network to Detect Java Card Mutants. *Agriculture and Agricultural Science Procedia*, **4**, 132-137. http://dx.doi.org/10.1016/j.aasri.2013.10.021

[16] Kindermann, R. (2009) Testing a Java Card Applet Using the LIME Interface Test Bench: A Case Study' Technical Report TKK-ICS-R18, Helsinki University of Technology, Department of Information and Computer Science, Espoo.

[17] Mostowski, W. and Poll, E. (2010) Electronic Passports in a Nutshell. Technical Report ICIS-R10004, Radboud University, Nijmegen. https://pms.cs.ru.nl/iris-diglib/src/getContent.php

[18] Vandewalle, J.J. and Vetillard, E. (1998) Developing Smart Card Based Applications Using Java Card. *Proceedings of the Third Smartcard Research and Advanced Application Conference*, Louvain-la-Neuve, 14-16 September 1998, 105-124.

[19] Bouquet, F., Legeard, B., Peureux, F. and Torreborre, E. (2005) Mastering Test Generation from Smart Card Software Formal Models. In: *Construction and Analysis of Safe*, *Secure*, *and Interoperable Smart Devices*, Springer, Berlin Heidelberg, 70-85. http://dx.doi.org/10.1007/978-3-540-30569-9_4

[20] Martin, H. and du Bousquet, L. (2001) Automatic Test Generation for Java Card Applets. In: *Java on Smart Cards*: *Programming and Security*, Springer, Berlin Heidelberg, 121-136. http://dx.doi.org/10.1007/3-540-45165-X_10

[21] VanWeelden, A., Oostdijk, M., Frantzen, L., Koopman, P. and Tretmans, J. (2005) On-the-Fly Formal Testing of a Smart Card Applet. In: *Security and Privacy in the Age of Ubiquitous Computing*, Springer, New York, 565-576. http://dx.doi.org/10.1007/0-387-25660-1_37

[22] Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S. and Scholl, K. (2003) Model-Based Test Case Generation for Smart Cards. *Electronic Notes in Theoretical Computer Science*, **80**, 170-184. http://dx.doi.org/10.1016/S1571-0661(04)80817-X

[23] Berndt, D.J. and Watkins, A. (2005) High Volume Software Testing Using Genetic Algorithms. *Proceedings of the* 38*th Annual Hawaii International Conference on System Sciences*, Big Island, 3-6 January 2005, 318b. http://dx.doi.org/10.1109/hicss.2005.296

[24] Korel, B. (1990) Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, **16**, 870-879. http://dx.doi.org/10.1109/32.57624

[25] Boneh, D. (1999) Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS*, **46**, 203-213.

[26] Willis, N. (2014) Smart Card Features on the YubiKey NEO. https://lwn.net/Articles/618888/

[27] Al-Khouri, A.M. (2012) PKI in Government Digital Identity Management Systems. *European Journal of ePractice*, **4**, 4-21.