Distributed Black-Box Software Testing Using Negative Selection

Ramin Rahnamoun[1]

[1] Computer Engineering Department, Azad University-Tehran Central Branch, Tehran, Iran. Email: ram.rahnamoon@iauctb.ac.ir

Abstract

In the software development process, testing is one of the most human intensive steps. Many researchers try to automate test case generation to reduce the manual labor of this step. Negative selection is a famous algorithm in the field of Artificial Immune System (AIS) and many different applications has been developed using its idea. In this paper we have designed a new algorithm based on negative selection for breeding test cases. Our approach, belongs to the category of black-box software testing. Moreover, this algorithm can be implemented in a distributed model. Two well-known case studies from software testing benchmarks is selected and results show the efficiency of this algorithm.

Keywords: Artificial Immune System, black-Box Testing, Negative Selection, Code Coverage

1. Introduction

In the process of software development, testing is one of the most complex and human intensive steps. Software testers concentrate on generating proper test cases. As software systems grow and embed in improvement processes of large organizations, the cost of software failures continues to escalate. Therefore, software companies must pay much more money and time to develop trustworthy products. Automated test generation is a process of reducing cost and time in such situations.

This paper uses negative selection algorithm from Artificial Immune System (AIS) to design a black-box software test case generator. Black-box testing is a method that does not need source code of software and thus it is especially useful for vulnerability detection in closed source environments. Different algorithms (such as Genetic Algorithms) has been applied for test case generation in white-box testing, but few algorithms has been designed based on black-box approach.

The main idea of this paper is focused on normal interaction between user and software. Users of any software have a normal interaction, so if an abnormal request is submitted to the system, the probability of finding an exception would increase. This concept can be expressed by the code coverage of software. In our approach, normal behavior of users with system, traverse a set of paths in flow graph. The remaining path is called abnormality code coverage and may also include abnormal behaviors.

This new algorithm tries to find such paths, those paths that normal inputs cannot cover. To obtain this, we use negative selection algorithm from Artificial Immune System (AIS). At first, a record log is created from normal interaction of the users with software. It is possible to generate a distributed log files, because every user can be logged separately and a merged log can be created from them. Negative selection generates detectors (input values) that are assumed as abnormality. This input set can be fed to software and the results might show the probable exceptions.

In the following section, related works about automated test case generation and different approaches of negative selection is reviewed. Section 3 describes the basic ideas of AIS with a special focus on negative selection. Section 4 discusses software testing and general methods of test case generation. The main idea of this paper is described in section 5. In section 6 and 7 case studies of our work is explained with experimental results. In the last section, our opinions about this work and also future works are discussed.

2. Background and Related Works

Software testing has a wide history in the field of software engineering. This paper is only focused on automated test case generation, so we only review this background. In static structure analysis of programs, symbolic execution has made an especial attention between researchers. This idea is related to King [1][2] in 70's decade, but has been recently used in new research works [3][4]. Dynamic test data generation has a wide range of methods. Random search generate randomly input cases and watch the results of them for exceptional events. Miller [5] and later Korel [6] has extended this method by redefinition of program conditions. Applying meta-heuristic search methods is also popular. Simulated annealing [7] is an example of such methods, but evolutionary algorithms has made much more attention in this area of research. Genetic Algorithms [8][9] is the main method among such algorithms. According to [10], evolutionary structural test generation can be classified into coverage-oriented [11] and structure-oriented approaches. Moreover, the later approach sub-classified to control-oriented [12], branch-distance-oriented [13] and combined [14] methods. Most of the mentioned methods belong to white-box approach. In contrary, black-box methods has limited implementation. This method in practice is useful for vulnerability detection [15], but it is also applicable in the field of software testing. Random and mutation testing can also be considered in this category. Modelling software in Z notation has also been used as a black-box method [16].

Artificial Immune System was used in various applications, from computer security [17][18] and Intrusion Detection Systems (IDS) [19][20], to clustering [21] and machine learning [22]. Negative selection algorithm introduced in [23] and then extended by new researchers [24].

3. Artificial Immune System and Negative Selection

Vertebrate immune system has different intelligent behaviors and one of the most important one is self/non-self-discrimination. This system can discriminate 10^{16} self-antigens from any other antigens, a feature that many artificial systems cannot simulate. Moreover, learning, induction, and memory are other interesting features of this system. Artificial Immune System (AIS) is a general paradigm that covers all of these features in computational models. Among various mechanisms in the immune system that are explored for AIS, negative selection, immune network model and clonal selection are still the most discussed models [24].

Artificial negative selection is a model that is developed based on a natural process in thymus for T-cells. Immature T-cells in thymus is assembled based on an especial rearrangement of genetic codes. Then, thymus eliminates any T-cells that interact with self-antigens. The models that are developed based on this natural process are called negative selection and applied to various real world applications. Different researchers try to modify classical model, but the major characteristic of negative selection still remains [23]. This algorithm consists of two stages. In generation stage, based on a random process non-self-antigens (detectors) are generated and similar detectors with self-samples will be eliminated. So, producing a set of self-samples is essential in this stage. In detection stage, every new sample is compared with detectors and those having similarities with detectors will be recognized as non-self [24].

Negative selection algorithm has the following essential points that must be discussed in each implementation [23][25]:

– *Data representation of samples and detectors*:

Self and non-self-entities of each real world application have different structures. So it must be modelled in an especial form, for example binary or real-valued vectors representation.

– *Matching rule*:

In both phases of the algorithm, we need a mechanism to calculate similarities between inputs, detectors and self-set. This matching rule is resembled to any detection, classification and recognition algorithms.

– *Detector generation mechanism*:

In classical form an exhaustive (random) search is used, but deterministic algorithms is also implemented.

4. Black-Box Software Testing

Software testing is a critical element of software quality assurance. But testing cannot show

the absence of defects, it can only show that software defects are present [26]. However the design of test cases for a software is a challenging problem. The total input space of an operational software is so large, and therefore it is not possible to test every input values. A perfect test case is the one that find most errors with a minimum amount of time and effort [27].

Although different references in software testing use distinct classification of testing approaches [27][28], but white-box and black-box testing approaches are more popular. According to this view, black-box approach concentrates on functional behavior of software in the operational environments while white-box approach looks at internal structure of software modules [28].

Black-box, data-driven, or input/output data-driven is a famous testing strategy. In this approach, testers do not concentrate about internal behavior or structure of software. Instead, they concentrate on finding circumstances in which the program does not behave according to its specifications [15]. Black-box testing has different advantages. This approach can be applicable, even if the source code is not available. In vulnerability detection, this ability has a critical role. Also in web-based applications that source code changes rapidly, this method can be used easily. Software in operation may face some errors that cannot be detected by trying the approaches based on the structure of source code, but black-box testing may find such defects. Simplicity of this approach is an important feature. White-box testing needs to analyze the details of software and this process will be more difficult in large scale source code, but in black-box approach we do not need any information about details of program structure. This approach can be reused easily for testing other software therefore reproducibility is another interesting feature of it. Beside of these abilities, the challenging problem is determining when to stop testing and how much effective the testing has been [15].

5. A New Approach Based on Negative Selection

In this section we first define some important keywords in software testing and negative selection and then we explain the framework based on our algorithm with them.

5.1. Formal Definitions

– *Input vector*: An input vector is a vector $v = (x_1; x_2 \ldots ; x_k)$ of input variables to program P. The domain of input variable x_i, $1 \le i \le k$ is the set of all possible values for x_i denoted by Dx_i. The input domain of P is the cross product $DP = Dx_1 \times Dx_2 \times \ldots \times Dx_k$. A program input x is a single point in the k-dimensional input space DP, $x \in DP$. So the domain of input vector for real functions is very large, thus it is not possible to evaluate each member of DP in order to find probable bugs.

– *Control flow graph*: A control flow graph for a program P is a directed graph $CFG = (N, E, s, e)$, where N is a set of nodes, E is a set of edges, and s and e are respective unique entry and exit nodes. Each node $n \in N$ is a statement in the program, with each edge, $e = (n_i, n_j) \in E$, representing a transfer of control from node n_i to node n_j [10].

– *Code coverage*: In software testing, code coverage is a set of nodes $C \in N$ of CFG which is covered when a set T of test cases is executed by P. Code coverage is normally shown by percent and is equal to $\frac{\|C\|}{\|N\|} \times 100$. Normality samples cover the normality set of nodes $Norm \in N$. Considering our test case set T covering k nodes which are not in Norm, we take one step further and define.

$$Abnormality\ Code\ Coverage = \frac{k}{\|N\| - \|Norm\|} \times 100.$$

– *Euclidean distance*: If α and β are two input values of an input variable x_d then $dist(\alpha, \beta) = \|\alpha\text{-}\beta\|$.

– *Hamming distance*: Assuming b_i is the *ith* bit of the every input value b, The hamming distance between two input values α and β of input variable x_d is defined as $dist(\alpha, \beta) = \sum_{i=1}^{size}(\alpha_i \oplus \beta_i)$, where size is the bit-size of input variables and $\oplus$ is the logical Exclusive OR representative.

– *R-chunk matching*: An input value α of an input variable x_d with the size of N is said to be the r-chunk match of input value β of the same input variable if all bits of α matches the bits of β in the window(s) specified by w. That is if we define wz as the representative of w where the bits in the window are 1 and the one outside are 0 then the sum $\sum_{i=1}^{size}((\alpha_i \odot wz_i)(\beta_i \odot wz_i))$ where $\odot$ is logical *And* representative should be zero.

5.2. General Framework

Fig.1 shows the activity diagram of our framework. A log recorder module should be installed on every machine that has the software which users interact with them normally. This module produces a log file from every machine and these distributed logs will be merged into a central log file.

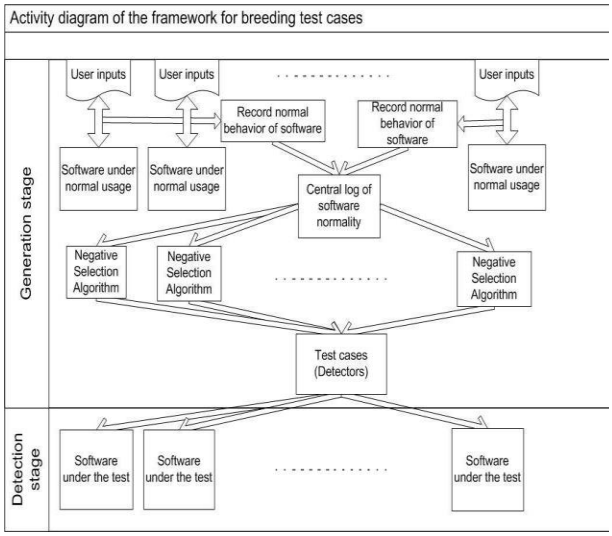

Fig.1. General Framework for breeding test cases based on negative selection

This log file represents normal behavior of the software. Now, the negative selection algorithm generates abnormality detectors (in the terminology of testing, they are test cases). The process of negative selection can be distributed on different machines via classification of random generator outputs to reduce run time. In this case the generated test cases can also be merged into a central test case set. This set is then sent to every machine that runs the software under the test mode.

The negative selection algorithm has a central role in this framework. In section 3, we described the main characteristics of negative selection however this algorithm can be implemented in different ways. For every application, the details needed for this algorithm must be carefully defined. In the next section different versions of this algorithm is discussed with the generated experimental results.

6. Case Study

For case studies we decided to choose two completely different cases. The triangle [30] problem is chosen because of its input space which are integers and in the other hand sreadhex [31] input space is a string of characters with two related integer. Both TriTyp and sreadhex are famous software testing problems which are used to test the effectiveness of new methods introduced in software testing.

6.1. Triangle Problem

The classic triangle problem which is also known as TriTyp is one of the most famous benchmarks for software testing. TriTyp simply tries to classify a triangle by the given 3 values of its sides into three categories, scalene, isosceles and equilateral. Sides are normally of the type signed short which is 16 bit in the current 32 bit systems, thus the problem space of TriTyp is pretty huge (2^{48} different values) in contrast with its limited inputs. It may seem that TriTyp is a simple problem for software testing but in fact it isn't. Considering the size of the problem space there are only few inputs which satisfy some of the branches of TriTyp. For example there are $2^{15} - 1$ equilateral triangles, so the mathematical expectation of choosing 3 random short values which would result in the branch which processes the equilateral triangles is $\frac{2^{15}-1}{2^{48}} \simeq \frac{1}{2^{33}}$. This behavior of TriTyp makes it perfect for evaluation and comparison of different methods in software testing. In our approach scalene, isosceles and equilateral triangles are considered to be from normality space. The samples is gathered with our little program from students also supports our assumption. The abnormality space is also categorized into 17 categories concerning zero and negative values and invalid triangles. This categorization helps us distinguish which matching would do better for this problem and also gives us a metric for comparison with another software testing methods like Random Testing.

6.2. Triangle Problem Test Case Generation

Three detectors each concerning the value for sides should be chosen for each attempt and the problem reduces to finding the highest abnormality coverage with fewer attempts. Two metrics can be taken into account when it is coming to comparison of different matching or methods, the code coverage and the number of attempts for reaching certain code coverage. The following specific matching functions used for elimination of generated detectors showing a certain amount of matching with normality:

– *Euclidean distance*: This simple matching simply rejects all the values within the range of normality, with the gathered samples any integer within the range of [6, 2000] would be rejected as a detector.

– *Hamming distance*: For 16 bit values the threshold parameter of Hamming distance seems to work fine with 2 as the value.

– *R-chunk matching*: The bit stream ddddd111111111dd is used for this matching with d representing for don't cares. With this specific bit stream the r parameter of r-chunk would actually be 9.

Also two heuristics are used trying to lower the number of attempts. These heuristics are general heuristics concerning boundary values. The first heuristic simply takes a value from the set {0, 1,−1, $2^{15} - 1$,-2^{15}} half of the times and the other one from {0, $2^{15} - 1$,-2^{15}} and the values around them. Adding

these heuristics to the classic negative selection algorithm shows a great efficiency.

This will be discussed more in detail in the results section.

6.3. Sreadhex

As for our second case study we decided to use a routine which works mostly with characters and related integers, something which happens in the majority of software or routines. This makes this case study more like a real-world example of how negative selection can effectively be used to reduce the number of test cases while having a fair amount of abnormality coverage. This would indeed reduce costs in many different systems. sreadhex is a C routine aimed to manipulate hexadecimals out of a string of representing characters, a similar Java program named PackHexChar is also discussed in [29]. The routine is called with 5 arguments but only 3 of them are interesting for us. The first parameter *str* holds the input string of characters which are representation of hexadecimals, Each two characters together meaning two hexadecimal together represent the 8 bits of a byte. The second parameter *rlen* holds the number of bytes which should be read from *str*. If the number of valid characters in the first *rlen* characters of *str* and our third parameter is even then we will have a complete set of bytes, otherwise we should save the remaining nibble in our third parameter odd which is a pointer to integer. This allows continuous calls to sreadhex without worrying about the length of valid characters in the first *rlen* characters of *str*. We also categorized the abnormality space of sreadhex into 8 distinct categories. This categorization is concerned with invalid *rlen* and odd values and the invalid characters in *str*.

6.4. Detectors and Test Case Generation in sreadhex

Considering our categorization, we decided to choose four kinds of detectors: byte detectors concerning the validity of hexadecimal representatives and three integer detector of type signed char concerning string length, one for controlling *dlen* and one for our odd parameter. Except the detector which controls the string length, all the other three have a direct influence on the abnormality space coverage. The following matching function is used to determine if the detector can be accepted:

– *Euclidean distance*: With the selected normality all the detectors with values inside the character range of [0...f] or [48, 102] are going to be rejected.

This issue will be discussed more about this matching in the next section.

– *Hamming distance*: The threshold parameter is somehow very limited in this case due to few numbers of bits in detectors. The results are gathered with the same threshold value as before.

– *R-chunk matching*: The bit-stream 11dd11dd is used for checking against the normality, with this bit-stream the two higher bits of each nibble become important.

7. Results

The most time consuming part of negative selection algorithms is detector generation. After detector generation each set of detectors are considered for execution. The corresponding results including abnormality code coverage with each try as a metric are gathered via our driver program. It is only good for comparing results and calculating the efficiency of each method. In the real world however we cannot have such information about code coverage. This is a general limitation of Black-Box software testing.

In *triangle problem* the number of generated detectors in each of approaches including our classic random generation of detectors and our two heuristics could be found in Table 1.

Table.1
Total Number of Detectors Generated in Triangle Problem
H1 and H2 are our two heuristics. The maximum try for reaching full abnormality code coverage was 450000.

Matching	Euclidean	Hamming	R-chunk
Classic	450000	450000	450000
H1	1098	450000	525
H2	1749	450000	642

Table 2 is concerned with specific code coverage regarding the number of attempts needed to be made with random detector generation approach while Table 3 and 4 covers the results with the help of our heuristic functions.

Table 2 also includes random selection result which is another black-box approach. From Table 2, it is clear that r-chunk detectors are showing a great efficiency when higher abnormality code coverage is desired but whenever time becomes a more important factor Euclidean detectors are doing better.

Table.2
Abnormality code coverage with random generation of detectors.
The maximum coverage of 82.35% reached, *na* means the coverage is not reached with the maximum try of 150000 attempts with each attempt containing 3 detectors.

Coverage%	5.88	23.52	47	70.58	82.35
Euclidean	1	7	15	4154	122435
Hamming	1	7	15	4235	*na*
R-chunk	1	5	21	3909	76059
Random	1	7	15	4284	131573

Table.3
Abnormality code coverage with the influence of first heuristic
na means the coverage is not reached with the maximum try of 150000 attempts with each attempt containing 3 detectors. The table is started with 23.52% since before that the coverage is fast.

Coverage%	23.5	47.05	76.4	94.11	100
Euclidean	4	9	28	46	3266
Hamming	4	9	*na*	*na*	*na*
R-chunk	4	9	22	39	175

Table.4
Abnormality code coverage with the influence of second heuristic.

Coverage%	23.5	47.05	76.4	94.11	100
Euclidean	4	11	31	70	483
Hamming	4	14	*na*	*na*	*Na*
R-chunk	4	14	27	79	214

Hamming distance is not desired when input variables are integers and have a direct influence on CFG. Hamming detectors could not reach a higher abnormality code coverage in comparison to other methods, because detectors with 0 value would be rejected and in our categorization it plays an important role.

Both r-chunk and Euclidean detectors dominate the classic random selection methods. From Table 3 it is also clear that a little amount of information about the software can make a huge difference in efficiency, and it normally is the case.

However it is always a good idea to check the boundary values and the points around them in input vector space. Our second heuristic shows the same result while being a little less efficient checking a little more space around boundary values.

The number of detectors in each attempt is variable in the case of sreadhex. Results which can be seen in Fig.2 shows a fast growth in the abnormality code coverage. The dominance of r-chunk detectors over the classic random selection is clear. The same conclusion can be made in sreadhex case: when higher abnormality code coverage is desired, r-chunk detectors would be the desired choice. The Euclidean detectors are clearly not a good choice when dealing with the string of characters. They never reached 100% coverage because some characters in the abnormality space are rejected. Hamming detectors are also showing good results after certain coverage. The r-chunk detectors in both cases have shown good results in comparison to other type of detectors and the classic random selection.

8. Conclusion and Future Works

Black-box software testing is a method that can be used for automated test case generation without the need of source code. Breeding test cases has a practical importance of reducing the manual labor in testing process. In this paper, we used negative selection algorithm that is inspired from vertebrate immune system to generate proper test cases for the software under test. A distributed framework is also designed to reduce the running time of test case generation.

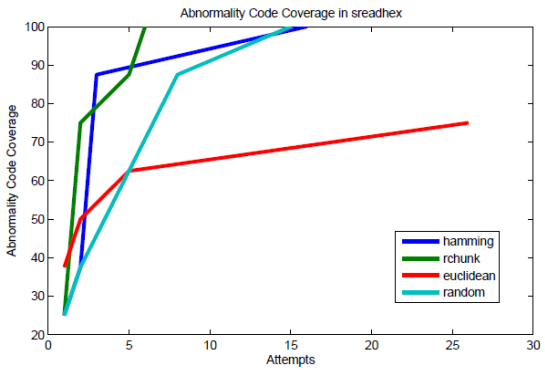

Fig.2. Abnormality code coverage regarding the number of attempts in sreadhex. The details of results at the beginning of plot are more important thus the Fig. is not plotted in its full domain.

The internal detail of negative selection is discussed and different variations are compared in two case studies.

This new distributed algorithm can also be used in new different areas. We believe that our approach can be used to develop a new vulnerability detection method. Vulnerabilities which endanger the security of software can be more devastating than the normal bugs. The reader should have noticed that our approach is only implemented on simple programs without internal states. Many of practical software have internal states and in such situations the term of normality must be redefined. In these scenarios, the user's session with the system can be considered as an instance of normal behavior. So the definition of test cases must also be redefined. This new concept can be used in web applications and object oriented unit testing.

References

[1] J. King, "A New Approach to Program Testing", In Proceedings of the International Conference on Reliable Software ACM Press, pp.228-233, 1975.

[2] J. King, "Symbolic Execution and Program Testing", Communications of the ACM, Vol.19, Iss.7, pp.385-394, 1976.

[3] S. Khurshid, C. S. Pasareanu, W. Visser: Generalized symbolic execution for model checking and testing, In Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp.553-568, April 2003.

[4] K. L. McMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[5] W. Miller, D. Spooner, "Automatic Generation of Floating-Point Test Data". IEEE Trans. On Software Engineering, Vol.2, No.3, pp.223-226, 1976.

[6] B. Korel, "Automated Software Test Data Generation", IEEE Trans. On Software Engineering, Vol.16, No8, pp.870-879,

1990.

[7] N. Tracey, J. Clark, K. Mander, J. McDermid, "An automated framework for structural test-data generation", In Proceedings of the International Conference on Automated Software Engineering, pp.285-288, IEEE Computer Society Press, Hawaii, USA, 1998.

[8] D. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley, 1989.

[9] M. Mitchell, "An Introduction to Genetic Algorithms", MIT Press, Cambridge, MA, 1996.

[10] P. McMinn, "Search-Based Test Data Generation, A survey", Journal on Software Testing, Verification and Reliability, Vol.14, No.2, pp.105-156, June 2004.

[11] A. Watkins, "The Automatic Generation of Test Data Using Genetic Algorithms", In Proceedings of the Fourth Software Quality Conference, pp.300-309, 1995.

[12] R. Pargas, M. Harrold, and R. Peck, "Test-Data Generation Using Genetic Algorithms", Software Testing, Verification and Reliability, Vol.9, No.4, pp.263-282, 1999.

[13] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, K. Karapoulios, "Application of Genetic Algorithms to Software Testing", In 5th International Conference on Software Engineering and its Applications, pp.625-636, Toulouse, France, 1992.

[14] J. Wegener, A. Baresel, H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing", Information and Software Technology, Vol.43, No.14, pp.841-854, 2001.

[15] M. Sutton, A. Greene, P. Amini, "Fuzzing: Brute Force Vulnerability Discovery", 2006.

[16] B. Jones, H. Sthamer, X. Yang, and D. Eyres, "The Automatic Generation of Software Test Data Sets Using Adaptive Search Techniques", In Proceedings of the 3rd International Conference on Software Quality Management, pp.435-444, Seville, Spain, 1995.

[17] S. Forrest, S. Hofmeyr, A. Somayaji, "Computer Immunology", Communications of the ACM, Vol.40, No.10, pp.88-96, 1997.

[18] J. O. Kephart, G. B. Sorkin,W. C. Arnold, D.M. Chess, G. J. Tesauro, S. R.White, "Biologically Inspired Defences Against Computer Viruses", Machine Learning and Data Mining: Method and Applications, R. S. (Ed) Michalski, I. Bratko, M. Kubat, John-Wiley & Son, pp.313-334, 1997.

[19] Jung Won Kim, "Integrating Artificial Immune Algorithms for Intrusion Detection", PhD Thesis, Department of Computer Science, University College London, July 30, 2002.

[20] F. Gonzalez, "A Study of Artificial Immune Systems Applied to Anomaly Detection", PhD Thesis, Division of Computer Science, University of Memphis, Memphis, TN 38152, May 2003.

[21] J. Timmis, "Artificial Immune Systems: A Novel Data Analysis Technique Inspired by the Immune Network Theory", PhD Thesis, Department of Computer Science, University of Wales, Aberystwyth, 2001.

[22] T. Knight, "MARIA: A Multilayered Unsupervised Machine Learning Algorithm Based on the Vertebrate Immune System", PhD Thesis, The University of Kent at Canterbury, 2005.

[23] S. Forrest, A. Perelson, L. Allen, R. Allen, Cherukuri, "Self-nonself Discrimination in a Computer", In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, pp.202-212, Los Alamitos, CA. IEEE Computer Society Press, 1994.

[24] Z. Ji, D. Dasgupta, "Revisiting Negative Selection Algorithms", Evolutionary Computation, Vol.15, No,2, pp.223-251, 2007.

[25] F. Esponda, S. Forrest, P. Helman, "A Formal Framework for Positive and Negative Detection Schemes", pp.357-373, IEEE Systems, Man, and Cybernetics Society, 2003.

[26] R. G. Pressman, "Software Engineering, a Practitioner's approach", McGraw-Hill, 2005.

[27] B. Beizer, "Software Testing Techniques", Second Edition, The Coriolis Group, 1990.

[28] G. J. Myers, "The Art of Software Testing", John Wiley & Sons, 2004.

[29] L. C. Briand, Y. Labiche, Z. Bawar, "Using Machine Learning to Refine Black-Box Test Specifications and Test Suites", Technical Report, Carleton University, 2007.

[30] TriTyp source code, http://www.irisa.fr/lande/gotlieb/resources/Mutants/trityp.c

[31] Marick B., "The Craft of Software Testing", Prentice Hall, 1995.