On the Idea of Using Nature-Inspired Metaphors to Improve Software Testing*

Francisca Emanuelle Vieira[1], Francisco Martins[1], Rafael Silva[1]
Ronaldo Menezes[2,1]**, and Márcio Braga[3,1]

[1] NATUS Project, IVIA
Fortaleza, Ceará, Brazil
{martins.filho,rafael.silva,emanuelle.vieira}@ivia.com.br
[2] Computer Sciences, Florida Tech
Melbourne, Florida, USA
rmenezes@cs.fit.edu
[3] Software Engineering, IVIA
Fortaleza, Ceará, Brazil
marcio.braga@ivia.com.br

Abstract. The number of software defects found in software applications today costs users and companies billions of dollars annually. In general, these defects occur due to an inadequate software development process that does not give the necessary importance to testing. Another contributor to these costs is the lack of adequate automated tools that can find "bugs" that would not otherwise be verified by experts. This paper looks at the combinatorial characteristics of the problem of testing – tools essentially search among all test cases for those that are promising (find existing bugs in the application) – and the effect that abstractions inspired by nature, such as genetic algorithms and swarm intelligence, may have in the construction of more "intelligent" testing tools. The paper argues that these abstractions may be used to construct automated tools that are more powerful, less biased, and able to incorporate expert knowledge while maintaining the ability to discover new, never-thought-of software defects.

1 Introduction

Software applications consist of a sequence of well defined instructions intended for execution in a computer. The exact sequence of instructions that is executed depends on several factors but primarily on the input values and the order these values are presented to the application. The majority of software testing tools (and programmers) perform functionality testing where the test cases are generated from real data. This approach is necessary but not sufficient. The choice of test cases is generally driven by the experts' concept of what is important to be tested. Still, software testers struggle to

* This research is funded by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil, under grant number 551761/2005-9
** Corresponding Author

Please use the following format when citing this chapter:
Vieira, Francisca Emanuelle, Martins, Francisco, Silva, Rafael, Menezes, Ronaldo, Braga, Marcio, 2006, in IFIP International Federation for Information Processing, Volume 204, Artificial Intelligence Applications and Innovations, eds. Maglogiannis, I., Karpouzis, K., Bramer, M., (Boston: Springer), pp. 541–548

answer questions such as: *what test cases should be executed?* and *What sequence of operations should be placed in a test case?* What we all would like to answer is: *all of them*. However, this is an impossibility for medium to large systems for the number of possible sequences is very large, making the problem intractable – and humans have a limited capability in understanding complex systems [1]. To make matters worse, Dijkstra [6] observed that the number of bugs in an application has a direct relation to the logic implemented rather than just the input values. He also stated that software testing can be used to show the presence of bugs but never the absence.

What is true is that the discipline of practical software testing suffers from a lack of respect from other computer science fields. It is important to make sure that what is meant by *practical* software testing. In general there are two basic approaches to test software. The first, more accepted in the academia, consists of using formal specification to design an application and then use theorem provers to demonstrate the application's properties. This approach is very strict but not often used given that the majority of software developers and designers are not prepared to deal with the strictness of formal specifications. As if this was not sufficient, the breadth of formal specification methods do not encompass all the functionality needed in today's applications.

The second approach consists of the traditional software engineering models (eg. waterfall, spiral, prototyping) that have a specific phase for testing that generally occurs after the application has been implemented. Granted: modifications to these traditional models have been proposed to incorporate testing in every phase of the software development, or for instance to minimize mistakes while developing with methodologies such as extreme programming [2]. However, despite all the claims of these techniques being effective, the truth here is that current approaches are insufficient to appropriately test software, thus causing the current status of the field which clearly seems to be loosing the battle of providing users with reliable software.

The point to be made is not that software testing is a non-scientific, not-deserving-attention field but rather that current approaches are not sufficient to convince academics and practitioners that the field needs to be taken seriously. From a financial point of view, it only makes sense to look at software testing with more attention given that low estimates, put the cost of software testing at around 40% of the overall cost of development [11]. Ergo what is needed is a scientific method that allows testing to be carried out effectively without the need for the rigorous approach that exist in formal methods. This paper discusses the use of nature-inspired metaphors in a process to improve the *status quo*of automated software testing.

2 Software Testing

Software testing is a main area of research in the Software Engineering community and is even subject of independent study in many universities. The process of software development is far from being trivial. The complexity of the new applications asks for a development process with strict quality control at every phase of the process.

2.1 Automation of Testing Procedures

In the last years, software testing has been considered one of the most important procedures to assure software quality. Software engineering has created many techniques to derive a set of test cases. However, a main theme to many researchers and practitioners in the area has been the creation of automated testing tools that are reasonably autonomous with the objective of assuring that software budgets remain within the estimates.

An automated testing tool, would execute the work of a tester, but with an advantage, it is cheaper – most of the cost of testing comes from personnel and the longer working hours they require to test the applications. Research in technologies such as the ones inspired in Artificial Intelligence and their integration in software testing appears to be the correct way to move forward. Artificial Intelligence can naturally aid the the process of testing by augmenting the ability of a software to discover defects that would not likely to be found by experts due to their bias.

2.2 Testing as an Optimization Problem

It is fair to say that applications will never be free of defects – problems exist even if all the development steps are taken correctly and the requirement analysis done thoroughly. Software testing aims at eliminating the majority of these problems via a systematic process that identifies components of the software prone to errors.

The identification of components with defects is not trivial given the size of the search space: all the possible execution sequences of the software. This is an unbounded space; there is no systematic process that can explore all the search space in a finite amount of time. As Dijkstra [6] said

> *Program testing can be used to show the presence of bugs, but never to show their absence.*

Hence, the discussion whether software testing can be used as a verification tool is subject of discussion in the testing community since the publications of the seminal work on verification edited by Boyer and Strother-Moore [4]. Independently of the answer to the discussion about the validity of testing as verification, and the arguments in favor software testing such as [7], the truth is that software testing is the *de facto* industry standard for validation and verification of software.

The development of automated testing tools is in unison with the industry standard. Industry has a growing need for tools that are efficient (in their execution) and effective (in finding bugs). Nature has provided us with several models that that when implemented are efficient and effective in dealing with complex problems. This paper discusses the direction that can be followed in a research that combines nature-inspired algorithms, namely genetic algorithms and swarm intelligence, in the design of automated tools for software testing.

3 Nature-Inspired Models in Software Testing

Our argument that software testing can be seen as an optimization problem gives us the opportunity to look at techniques that have been successfully used in the optimization arena and their usefulness to software testing.

3.1 Genetic Algorithms

Genetic Algorithms (GAs) was proposed by Holland in 1975 [8] as a metaheuristic to combinatorial problems. It was inspired by the process of natural selection as proposed by Darwin [5]. GAs have been successfully used in a plethora of practical applications. The main steps involved in using GAs are:

Codification: Solutions for problems need to be coded as a genotype that can evaluated based on some fitness value. There must be a total order on the fitness values to allow solutions to be compared against each other.

Population Creation: A group of (normally) random-generated individuals are created to compose a population of fixed size. As part of this step, one needs to look at the issue of diversity in the population. The more diverse the population, the more likely it is that good genetic sequences (part of the genotype) are present in the population.

Individual Selection: Given the fitness of each individual, as calculated by the fitness function, the algorithm can stochastically select candidates of the population to be used in the next step (mating).

Mating: Mating requires the selection of two or more individuals. Mating can occur in various ways but the most common is point-crossover in which a "cut" point is randomly selected and new individuals are generated from combinations of the genotypes' parts. For instance, with a 1-point-crossover and two individuals, one can generate 2 new individuals combining the first part one parent with the second part of the other, and vice-versa. The decision to mate is also stochastic – individuals may be selected and not mate.

Mutation: Mutation is a process that allows the introduction of new genetic material in the population. Although mutation is an important step, it is normally desirable with less frequency. Hence GAs set the mutation rate of algorithms to very low probabilities. A larger than necessary mutation rate may had a undesirable side-effect to the algorithm convergence.

Understanding the steps above are essential in understanding what GAs can offer to software testing. As explained earlier, software testing consists of a difficult search for solutions in an open search-space. GA is a mechanism to explore this space in a consist, impartial, and systematic way. A GA approach may remove the partiality of currently software testing techniques that rely on the knowledge of a software-testing expert. Section 4 describes the use of GAs in testing.

3.2 Swarm Intelligence

In nature, we find examples of intelligence being revealed at the group level in species with limited cognitive capacity, such as ants, bees and termites. This type of intelligence is called Swarm Intelligence (SI) and it has been extensively studied [3, 9]. SI is an area of research that has shown efficiency in dealing with distributed problems where the solution space of the problem is large. In SI, agents are not aware of the entire problem but are only programmed to do simple actions that (maybe) contribute to the solution of the problem. The joint action of these agents and their interactions causes the emergence of a solution to the problem.

In insects colonies, the indirect communication through stigmergy is very common. In this type of communication, the agent behavior modifies the environment, which, in a feedback loop, influences the behavior of the insects. For example, ants use pheromone to indicate paths to food sources while other ants use this information to stochastically decide their own paths.

There are several metaphors in swarm intelligence that may be used to improve the automation of software testing. In particular, we could use an approach based on alarm pheromone [10] to ensure that a set of test cases is tested by distributed tools. This could take the form of ants (representing testing tools) that explore a terrain (space of test cases to be executed) making test cases already tested less attractive to the ants. The emergent behavior of such action is that ants would be spread across the space of cases. At the same time, one can apply a negative feedback in test cases that have been executed a long time ago so that they gradually become more attractive to the ants.

4 Nature-Inspired Test Automation

The obvious question is whether the approaches described above can yield good test cases and good testing tools. In this section we discuss the use of the ideas and how they can be combined for form an interesting automated tool.

The occurrence of a defect in a software application is directly proportional to the level of inconsistency the state of the application is in. This means that when an error occurs in an application it was not necessarily the last operation, ℓ, executed in the application that caused the defect; it may have been a sequence of previously executed operations that caused the application to be in an inconsistent state by the time ℓ was executed. Our premise is that an application has a state of inconsistency ϕ_i which is non-decreasing as operations ,ℓ_i, are executed. Or in other words, for any sequence of operations, $\phi_i \leq \phi_{i+1}$. This means that every transition t from operation ℓ_i to ℓ_{i+1}, written as $t(\ell_i \rightarrow \ell_{i+1})$, adds to the state of inconsistency of the application. In these lines, the goal of software testing is to minimize $t(\ell_j \rightarrow \ell_k), \forall j, k$.

Given the definition above, we can assign values to each possible transition in a software system to represent the contribution of that transition to the degree of inconsistency of the application. Clearly, this is a biased approach that should be avoided

in practice – we require an expert to define how problematic a particular transition is. Still, we are using this here because the bias does not affect what we are trying to demonstrate: that approaches such as GA can be used to evolve good test cases. Table 1 shows the representation of the transition values.

Table 1. Representation of the assigned values for inconsistency added by each transition. For instance $t(\ell_2 \rightarrow \ell_3) = v_{3,2}$.

	ℓ_1	ℓ_2	ℓ_3	$\cdots$	ℓ_n
ℓ_1	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$\cdots$	$v_{1,n}$
ℓ_2	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$\cdots$	$v_{2,n}$
ℓ_3	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$\cdots$	$v_{3,n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
ℓ_n	$v_{n,1}$	$v_{n,2}$	$v_{n,3}$	$\cdots$	$v_{n,n}$

Given a transition table as the one in Table 1 we can define a fitness function, f_s, for a test case (sequence of instructions) $s = \ell_1, \ell_2, ..., \ell_k$ as $f_s = \sum_i^{k-1} t(\ell_i \rightarrow \ell_{i+1})$. This indicates that the larger the value of f_s the better the sequence is considered given that it is more likely to take the application to an inconsistent state. GA can now be used to evolve a population of sequences of a fixed size into an individual that is close to a maximum value for f_s. The resulting sequence is generated as part of an automated process in which experts do not play a role in the sequence (except for the definition of the transition table).

What is important here is that this is just a (perhaps naïve) example of a fitness function. Functions based on other factors may be used to generate a test case. In fact, in a more elaborate use of GAs, the transition table itself can be updated in a feedback loop based on the result of the execution of the test case (as described later in Figure 1). That is, it may be possible to initialize the transition table with the same value for all cells (or random values for each cell) and upon the execution of a sequence that did not cause problem the values for all transitions that are part of the sequence can be decreased by a certain percentage causing the table to slowly converge to a configuration where the values do really indicate the contribution each transition is adding to the inconsistency of the application.

SI (as described in Section 3.2) may also be used to generate test sequences based on a structure similar to Table 1. Ants (agents) may be programmed to traverse a transition table. The agents would make transitions less important (via negative feedback) each time a transition from one operation to another is traversed. Similarly, each time a transition presents problem, its value may be positively reinforced to influence agents to generate test cases including that transition.

The NATUS project explores SI for a different purpose. Once good test cases are generated using GAs, one still have to prioritize their execution. Note that this is not a question of just listing the cases and testing from the beginning of the list to the end

because this may not give us a diverse list of cases being tested (assuming that not all cases can be tested in the time frame available for testing). Basically, a space of test cases generated by the GA can be explored by ants (testing tools) where the higher the fitness value of the sequence the more likely it is to be executed. In a process opposite to ant foraging and inspired in alarm pheronomes [10], the ants can make locations in the environment (test-cases) less desirable as they select the cases. This makes the ants (testing tools) more inclined to choose tests that have not yet been executed. The obvious question here is why not choose from the list of test cases deterministically? This approach is possible and should be done if the number of test cases is small or one has enough time to execute all test cases. However, when testing large applications, it is not necessarily true that the execution based on a deterministic order will provide us with a subset of test cases that is diverse enough (cover most transitions). Also, one need to understand that important test cases (according to the fitness value) may be executed and indeed cause problems in the application. This test continues to be open for re-test and should continue to be considered according to its fitness value (or even with an increased fitness value). The use of positive feedback can be used to reinforce the fitness value of such test cases, thus attracting more ants to explore that case again.

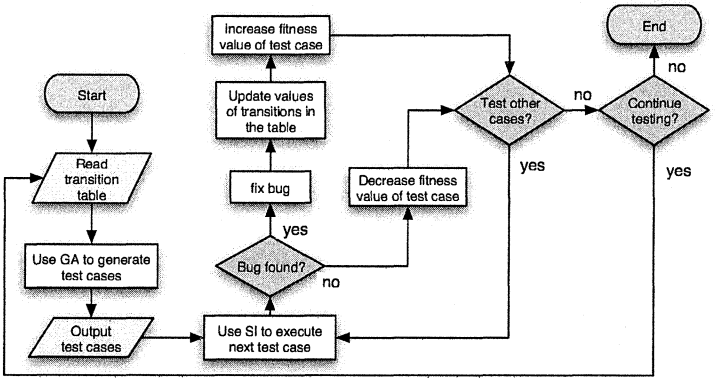

Fig. 1. A flow of a testing tool that utilizes the GA and SI approaches described earlier

5 Conclusion

This paper discussed automation in software testing and explored the idea that testing can be seen as an optimization problem. The paper argues that an efficient and effective software testing tool can be devised from GAs and SI. These are scientific approaches that have already been proved to work in problems where solutions need to be found in a large space of candidates.

Figure 1 summarizes all the automation process this paper proposed. In a nutshell, *(i)* a transition table is created (as in Table 1) which indicates a degree of inconsistency added by pairwise operations; *(ii)* a GA uses the table to generate test cases (sequence of operations) according to a fitness function; *(iii)* a SI algorithm explores the set of test cases (by executing them) and updates the transition table and the fitness value of the executed test depending on the occurrence of a bug in the test case – if a bug is found and the problem fixed, the transition which caused the problem has its degree of inconsistency decreased and the test case itself has its fitness value increased (as it is still attractive to testing tools). The execution of this process yields a fully automated testing process.

As part of the NATUS project, we are currently working on a prototype of a tool as described in Figure 1 – preliminary results look promising. The current results are for the GA part of the algorithm and are based on non-real-world values for the transition table. Our future work we propose to explore a real-world application and implement the SI part of the proposed automated process. On the SI front we are developing a model for the alarm pheromone which we intend to use as part of our proposed automation process.

References

1. A.-L. Barabási. *Linked: The New Science of Networks.* Perseus Publishing, 2002.
2. K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.
3. E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Santa Fe Institute Studies in the Sciences of Complexity Series. Oxford Press, July 1999.
4. R. Boyer and J. Strother-Moore, editors. *The Correctness Problem in Computer Science.* Academic Press, 1981.
5. C. Darwin. *On the Origin of Species: A facsimile of the first edition.* Harvard University Press, July 1975.
6. E. W. Dijkstra. Structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Notes on Structured Programming,* pages 1–82. Academic Press, 1972.
7. J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software,* pages 493–510, 1975.
8. J. H. Holland. *Adpatation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.
9. J. Kennedy and R. C. Eberhart. *Swarm Intelligence.* Morgan Kaufmann, 2001.
10. C. Lloyd. The alarm pheronones of social insects: A review. Technical report, Colorado State University, 2003.
11. C. E. Williams. Software testing and uml. In *Proceedings of the 10th International Symposium on Software Reliability Engineering,* Boca Raton, Florida, Nov. 1999. IEEE Press.