08351 Executive summary
Evolutionary Test Generation
— Dagstuhl Seminar —

Holger Schlingloff[1], Tanja E.J. Vos[2] and Joachim Wegener[3]

[1] Fraunhofer Institut - Berlin, Germany
hs@informatik.hu-berlin.de
[2] Universidad Politècnica de Valencia, Spain
tvos@dsic.upv.es
[3] Berner & Mattner Systemtechnik - Berlin, Germany
joachim.wegener@berner-mattner.com

Abstract. From September 24th to September 29th 2008 the Dagstuhl Seminar 08351 "Evolutionary Test Generation " was held in Schloss Dagstuhl – Leibniz Center for Informatics. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. This paper contains an executive summary of the seminar and the open problems that were found.

Keywords. Software-testing, evolutionary algorithms, meta-heuristic search

1 Overview of the seminar

The "Evolutionary Test Generation" Dagstuhl seminar that was held from September 24th to September 29th 2008. The organisation of the seminar was initiated by the EvoTest project, a project funded by the European Commission under the contract number IST-33472.

The goal of our seminar was to bring together researchers from the software testing and evolutionary algorithms communities for the discussion of problems and challenges in evolutionary test generation. This goal has been satisfactorily met and has led to a comprehensive list of open problems and challenges identified and discussed during the seminar. This list in described in Section 3 of this executive summary.

The seminar has been attended by 33 people: 30 were researchers from all over the world working on evolutionary testing, test generation and/or evolutionary computing; 3 were industrial participants with experience and feedback from real-life challenges were present: Microsoft, IBM and Berner& Mattner. The abstract collection indicates the talks that were given by the participants.

2 Brief introduction to evolutionary testing

Systematic testing is the most widely used method to ensure that a program meets its specification. The effectiveness of testing for quality assurance largely

Dagstuhl Seminar Proceedings 08351
Evolutionary Test Generation
http://drops.dagstuhl.de/opus/volltexte/2009/2022

depends on the chosen test suite. Currently, test suites are constructed either manually or semi-automatically from the program code or program specification. For large systems, however, manual test case construction is tedious and error-prone, whereas semi-automatic procedures often achieve only insufficient coverage. Therefore, new methods for the automated generation of "good" test suites are necessary.

Evolutionary adaptive search techniques offer a promising perspective for this problem. Genetic algorithms have been investigated for complex search problems in various fields. Their basic principles are selection, mutation, and recombination. These principles can be beneficially applied to the automated generation and optimisation of test suites, both from code (white-box testing) and specification (black-box testing). However, to make this approach successful in practice, a lot of problems remain to be solved: the question of adequate testing objectives, coverage and reliability measures, representation issues for test cases and test suites, seeding, recombination and mutation strategies, and others.

2.1 Evolutionary Algorithms

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and DarwinŠs theory of biological evolution. They are characterized by an iterative procedure and work parallel on a number of potential solutions for a population of individuals. Permissible solution values for the variables of the optimization problem are encoded in each individual.

The fundamental concept of evolutionary algorithms is to evolve successive generations of increasingly better combinations of those parameters that significantly affect the overall performance of a design. Starting with a selection of good individuals, the evolutionary algorithm tries to achieve the optimum solution by random exchange of information between increasingly fit samples (recombination) and introduction of a probability of independent random change (mutation). The adaptation of the evolutionary algorithm is achieved by selection and reinsertion procedures based on fitness. Selection procedures control which individuals are selected for reproduction, depending on the individualsŠ fitness values. The reinsertion strategy determines how many and which individuals are taken from the parent and the offspring population to form the next generation.

The fitness value is a numerical value that expresses the performance of an individual with regard to the current optimum, so that different individuals can be compared. The notion of fitness is fundamental to the application of evolutionary algorithms; the degree of success in using them may depend critically on the definition of a fitness that changes neither too rapidly nor too slowly with the design parameters. The fitness function must guarantee that individuals can be differentiated according to their suitability for solving the optimization problem.

Figure 1 provides an overview of a typical procedure for evolutionary algorithms. First, a population of guesses on the solution of a problem is initialized,

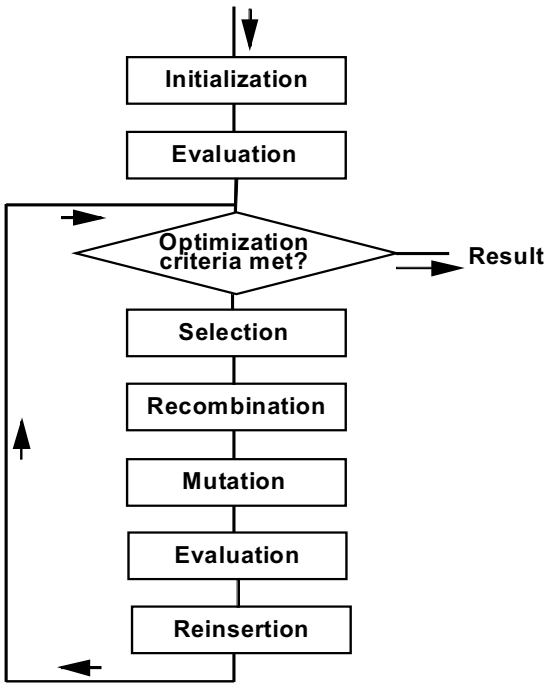

Fig. 1. Evolutionary algorithms

usually at random. Each individual within the population is evaluated by calculating its fitness. This will usually result in a spread of solutions ranging in fitness from very poor to good. The remainder of the algorithm is iterated until the optimum is achieved, or another stopping condition is fulfilled. Pairs of individuals are selected from the population according to the pre-defined selection strategy, and combined in some way to produce a new guess analogously to biological reproduction. Combinations of algorithms are many and varied. Additionally, mutation is applied. The new individuals are evaluated for their fitness, and survivors into the next generation are chosen from parents and offspring, often according to fitness. It is important, however, to maintain diversity in the population to prevent premature convergence to a sub-optimal solution.

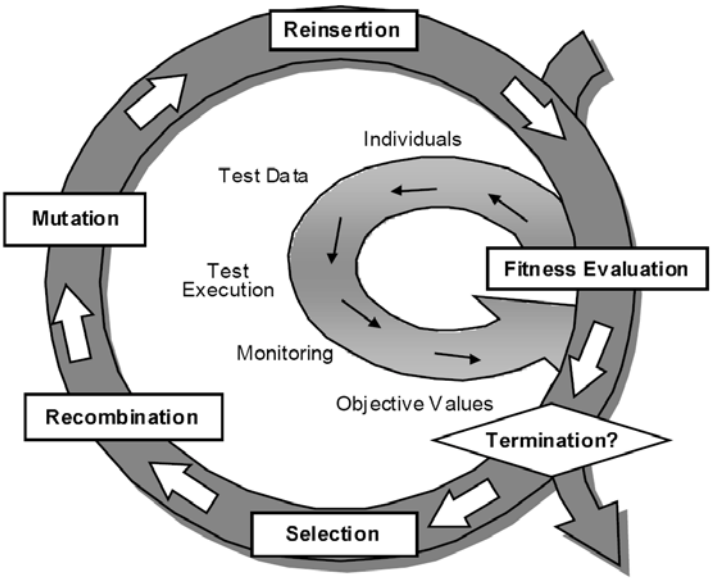

Fig. 2. Evolutionary testing

2.2 Software testing

In order to automate software tests with the aid of evolutionary algorithms, the test aim must itself be transformed into an optimization task. For this, a numeric representation of the test aim is necessary, from which a suitable fitness function for the evaluation of the generated test data can be derived. Depending on which test aim is pursued, different fitness functions emerge for test data evaluation. If an appropriate fitness function can be defined, then the Evolutionary Test proceeds as follows.

The initial population is usually generated at random. In principle, if test data has been obtained by a previous systematic test, this could also be used as initial population. The Evolutionary Test could thus benefit from the tester's knowledge of the system under test. Each individual of the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the fitness value is determined for the corresponding individual. Next, population members are selected with regard to their fitness and subjected to combination and mutation processes to generate new offspring. It is important to ensure that the test data generated is in the input domain of the test object. Offspring individuals are then also evaluated by executing the corresponding test data. Combining offspring and parent individuals, according to the survival procedures laid down, forms a new population. From here on, this process repeats itself, starting with selection, until the test objective is fulfilled or another given stopping condition is reached (see Figure 2).

3 Results of the discussions: Open problems and future challenges

The future challenges identified at the Dagstuhl seminar have been categorized as follows:

- Theoretical foundations
- Search Technique improvements
- New testing objectives
- Tool environment/testing infrastructure
- New application areas

3.1 Theoretical foundations

Evolutionary testing uses meta-heuristic search algorithms. The advantage of meta-heuristic search is that is it widely applicable to problems that are infeasible by analytic approaches. All one has to do, is to come up with a representation and objective function. This can make one think that evolutionary computation is some kind of golden hammer that can be applied to attack any problem. However finding good representations and fitness functions can be very hard, and sometimes one cannot find one at all. We lack a good theoretical foundation to tell us *which problems can be solved using evolutionary computation, and for which it is unsuitable?* This problem is experienced by the entire field of search based software engineering. There are many search algorithms available and it is not clear which technique is best for a certain problem/search space. The choice is often made somewhat ad hoc, based on experience or by trying an arbitrary selection of search algorithms. To tackle this problem Harman [Har07] calls for a more concerted effort to characterise the difficulty of the software engineering problems for which search already produced good results. This characterisation will help to determine the most suitable search technique to apply.

A step further would be the development of a *benchmarking suite* consisting of different SUTs and testing-objectives. Such a suite would allow for much better development of experiments, providing more thorough comparison of different testing techniques, evolutionary as well as others. It would allow us to gain insight in the strengths and weaknesses of each technique. This is a valuable tool for industry to help them to make a well-founded decision on which tool to apply. Furthermore it would drive further research and development of techniques.

Another very important element in the theoretical foundation is an assessment of *what is the quality of the evolutionary testing results?* How good are the generated tests compared to tests derived using other techniques or developed manually by a tester? Furthermore, figures are needed to assess the reliability of the test results. Such assessments are necessary to which extent evolutionary testing could be used as a substitute for manual testing and to which extent as an addition to manual tests. The previously said is especially important for dependable systems.

A general problem with evolutionary computation is: when to stop the search. Usually this is decided by a bound on the number of generations or by analysing the testing progress and by analyzing the search progress. But again reliability is an interesting aspect: *when we stop the test, how sure can we be that, in the next generation, no more errors will be found?*

3.2 Search Technique improvements

Many approaches that look promising in the lab are inapplicable in the field, because they do not scale up. However, making a solution scalable is easier said than done. There are a couple of areas where work is needed:

Parallel computing. A great advantage of evolutionary computing is that it is naturally parallelizable. Fitness evaluations for individuals can easily be performed in parallel, with hardly any overhead. Search algorithms in general and SBSE in particular; therefore offer a 'killer application' for the emergent paradigm of ubiquitous user-level parallel computing. Grid-computing is the subject of a great number of EU-projects. There is a great opportunity to team up with these projects and apply the technologies developed in that area on evolutionary testing.

Combining search techniques. Another way of increasing efficiency is to improve the search techniques. Other search techniques may perform better than genetic algorithms. For example a recent study [WWW07] shows that particle swarm optimisations outperform traditional GA's for many instances of structural testing. Another promising approach is the use of hybrid search techniques. Research is needed to find out what (combination of) search technique(s) is best for which categories of test-objectives.

Multi-objective approaches also provide an opportunity for improving the efficiency of evolutionary testing. By targeting multiple test-objectives at the same time, the value obtained from the expensive process of executing the subject under test can be maximized. A recent case study [LHM07] shows promising results in this direction. The study investigates the performance of multi-objective GA for the twin objectives of achieving branch coverage and maximizing dynamic memory allocation. The results show that multi-objective evolutionary algorithms are suitable for this problem, and the way in which a Pareto optimal search can yield insights into the tradeoffs between the two simultaneous objectives. Evolutionary algorithms are a very powerful tool for many problems. However to obtain the best performance out of them it is crucial that their parameters are well-tuned to the problem to which they are applied. For this you need to be an expert in the area of evolutionary algorithms. This is unfortunate as testers usually have very little knowledge about evolutionary computation.

Static parameter tuning. A solution would be that the testing tool automatically tunes the parameters. One approach would be to do this statically, based on the characteristics of the SUT. These could be obtained for example from the tester; as a tester has a lot of knowledge on the SUT. In the case

of white box testing this information can also stem from (static) analysis of the SUT.

Dynamic parameter tuning. Another approach is to let the search algorithm tune itself, based on how well the proceeding. In this way the search could automatically adapt to the search-landscape. This approach is seems very promising for search problems that have many different sub-goals. It could very well be that the parameter settings that are good for one sub goal are completely ineffective for another.

Testability transformations. For structural testing, it is possible to remove certain code constructs that cause problems for evolutionary search by applying transformations. This approach is taken, for example, when removing flag variables. Flag variables introduce large plateaus in the search space, effectively deteriorating a guided search into a random search. Recent work [HHL+07] presents an algorithm for removing loop-assigned flags; a special case of flags that conventional flag removal algorithms could not handle. The algorithm substitutes the definition/uses of a flag with two helper variables in order to enable the calculation of a smooth fitness landscape. Other problematic constructs may also be dealt with using this strategy. Research is needed in this area.

Search space size reduction. Another way to improve efficiency is to use knowledge about the subject under test to improve the fitness function; effectively reducing the complexity of the search space. For example knowledge on value ranges could be used to set parameters of the search, such as e.g. step size for variation of integers, doubles, etc. Another example is the seeding of test data with literals extracted from the program code. Such strategies could result in a very significant search space reduction.

There are many ways to uncover information about the system under test. The models and specifications (on system, software, design or component level) could be analysed for information that can be used to improve the test or the search.

The source code of the subject under test is also valuable source of knowledge on the system under test. Static analysis techniques can be applied to provide information that is useful increase efficiency and effectiveness of the evolutionary test. For example static analysis can be used to determine which input-variables are relevant to the search. The irrelevant variables can be left out making the input-domain of the fitness function smaller. The bounds of variables or the control flow are other examples of knowledge that can be used by the fitness function to guide the search. Abstract interpretation may be employed to provide equivalence partitions. Such a partition is a range of value for which the SUT behaves the same. In the search one needs to sample only a single element of the partition to cover to whole range, greatly reducing the search space. Symbolic execution may be employed to compute the path conditions for a sub-goal. The open question is how and which results of the static analysis research we can utilize to improve evolutionary testing.

Another analysis technique is concolic execution, which combines static analysis with concrete execution. During the evolutionary search the subject under test is executed many times, so why not collect data on these concrete executions and let the search learn from them.

Monitoring the search progress and the testing progress in order to adapt and optimize the search or the objective function might also be an interesting research area.

Searching for multiple sub-goals at the same time. The current approaches for structural testing apply a separate search for each sub-goal. It often happens that the separate searches are solving (almost) the same problem over and over again because several sub-goals have a lot in common. For example the paths to sub-goals deeper in the control-flow-graph may share a long common prefix. It would be more efficient if the search starts off solving the easy sub-goals at the start the control-flow-graph and gradually work its way down towards the deeper more complex sub-goals. A way this might be achieved is by having a dynamic fitness function that rewards individuals on how close they are to still uncovered sub-goals.

Other interesting questions: what can we learn about the system under test from the execution of a huge number of test data? Is testing the only thing or could we achieve results for other software engineering activities from that?

3.3 New testing objectives

Previous work on evolutionary testing mainly focuses on structural test-objectives, such as branch-coverage. Although the topic of branch-coverage is extensively researched, there are still many points for improvements:

- dealing with internal states
- dealing with predicates containing complex types, such as strings, dates, structs, arrays.
- dealing with loops, especially data dependencies between values calculated in loops and used outside the loop.
- how to improve the calculation of the fitness function for combined conditions (and, or, etc.)

Evolutionary testing can be applied for many other testing-objectives, both structural as well as functional ones. Research is needed to develop an appropriate representation and fitness-function for each new testing-objectives. Below we describe a number of possible testing-objectives. For some it is clear how to implement them, for others the required representations and fitness function yet unclear and thus open research topics. Run-time error testing: examples of run-time errors are: overflow, division by zero, memory leaks. For testing run-time errors the objective is to find inputs that trigger such an error. It should be possible to tackle this area by extending the existing approaches to structural testing. For example to test for memory leaks the fitness function should favour test-inputs on which the subject under test uses more memory. It should be

possible to tackle this area by extending the existing approaches to structural testing.

Testing interactive systems: for this the test-input is a sequence of user-actions, such as keystrokes and mouse clicks. The system under test can be tested for example for responsiveness. The fitness function should then for example favour combinations of user actions that take a long time to complete. Another objective could be the coverage of different user actions in various combinations.

Integration testing: a system usually consists of a number of more or less independent parts (e.g. functions, components, services) that are assembled to form the complete system. These parts should of course be tested in isolation, however there are also problems that only occur when integrated. What should the test goals and corresponding fitness functions is an open question.

Testing parallel, multithreaded systems: testing such systems is hard, especially finding bugs that only occur with certain interleavings of the processes or threads of the system under test. The need for testing becomes ever larger, systems get more and more complex and multi-processor computers are getting more common. An objective for testing such systems is for example trying to find deadlock situations. The fitness function should somehow favour executions that are close to a deadlock. Open questions are how the best represent interleaving executions of the system under test, and how to measure the "closeness" to deadlock.

Testing non deterministic systems. Another open question is if evolutionary testing could be used to test systems which are based on non-deterministic technologies, e.g. systems based on learning algorithms such as neural nets, randomized algorithms etc.

Take into account the cost of the test activities subsequent to test case generation and try to minimise the overall test-cost. For example for coverage testing reduce the number of test cases or avoiding test cases that take a long time to execute, while maintaining the same amount of coverage. More generally this could lead to a kind of multi-objective search, maximizing test effectiveness but reducing cost.

3.4 Tool environment/testing infrastructure

Evolutionary testing still does not have a successful take-up in industrial practice. One way to improve this is to increase the support of evolutionary testing within different languages, development and testing tools.

New programming languages. The main reason for this is that there is insufficient support of different programming languages. Even for C, evolutionary testing technology is usually not entirely applicable because only a subset of the programming language is supported. Problematic language

constructs that are not entirely supported are for example recursion, pointers and complex data types. For industrial applicability the tool environment should support all, or at least most language features.

New programming paradigms. Supporting other programming paradigms poses a great challenge for evolutionary testing. Progress in this area has been made on testing object-oriented programs [WW06]. In his work, Wappler shows that evolutionary testing is in principle applicable to the testing of object-oriented programs. Other programming paradigms such as logic programming or functional programming remain largely unexplored.

Hidden input interfaces. For many functions the input interface is simply its formal parameters plus the global variables that it uses. However sometimes it is not that simple. For example the embedded systems in the automotive industry often communicate through busses and ports. Hence the input is just port on which messages are coming in. Static analysis may be employed to some extent to discover what ports are used and what kind of data is expected by the function under test. Also the tester may be asked to specify what the input ports are and what types of values may be sent through them.

Eclipse. The tool developed in the EvoTest project will be integrated in Eclipse, an open source IDE that is used by many developers in industry and academia. Many other EU project also integrate their tools into Eclipse, making collaboration between projects easier. Furthermore, tools existing in the Eclipse community should be used to allow sufficient progress in the development of the EvoTest tool environment.

Visualisation can provide a user with important insights. There are aspects of visualisation:

- Visualisation of testing progress, for example how much was tested, testing effort, test coverage, reliability figures.
- Visualisation of search progress, e.g. how does the search perform, potential for better results when continuing, identify potentials for improving the search, search space landscape.

Important questions are, which data is useful for a user of the tool, and how to display them in a concise manner. Displaying the amount of coverage for a small piece of code is easy, one can simply colour the covered code in the editor, or display a coloured control flow graph of the code. However for a large system consisting of many lines of code different techniques need to be developed. Also the visualisation of the search landscape is a challenge. The space usually has many dimensions, making it hard to display in a concise manner in 2-D.

3.5 New application areas

Moving up the life cycle. Thus far testing is applied late in the life cycle of software development. The EvoTest-project mainly focuses on testing source code and completed systems. It is best to start with testing as early in the life

cycle as possible. Testing could also be applied during the modelling phase. More formally defined and executable models gain in popularity and could be tested for inconsistencies and other anomalies. There is an opportunity for collaboration with the EU-funded MODELPLEX project to advance in this area. Another possible use of models is to generate test-sets that cover the model. Such test-sets can then be used to test whether the developed system adheres to the specification defined by the model.

Heterogeneous complex systems. Evolutionary testing could help to test heterogeneous complex systems that are created by integrating many different services and components through the Internet to generate new applications or user functionalities. These systems result in a very heterogeneous software architecture and design in which many different programming languages are involved. Since these systems are difficult to test with traditional testing techniques, evolutionary testing might be a solution here.

Reproducibility of "non-reproducible" faults. Often a user or a tester runs into a fault and writes a reports with the steps to reproduce. However when the developer follows these steps the fault does not occur. As a result a developer is not able to find and fix the bug, and the reported fault is marked as not-reproducible. Often this is caused by slight variations between the configuration of the development environment and the users/testers environment. Evolutionary search might be employed to find the configurations exhibiting the fault. Another cause for non-reproducibility is concurrency. A program might behave well for most interleavings, only in some exceptional ones the fault appears. In this case the search might try to find test inputs that have a higher probability of exhibiting the fault, making it easier for the developer to locate the source of the bug.

References

Har07. Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

HHL+07. Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164, New York, NY, USA, 2007. ACM.

LHM07. Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105, New York, NY, USA, 2007. ACM.

WW06. Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using A hybrid evolutionary algorithm. In Gary G. Yen, Lipo Wang, Piero Bonissone, and Simon M. Lucas, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 3193–3200, Vancouver, 6-21 July 2006. IEEE Press.

WWW07. Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1121–1128, New York, NY, USA, 2007. ACM.