EvoSuite: On The Challenges of Test Case Generation in the Real World

Gordon Fraser
University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

Andrea Arcuri
Certus Software V&V Center at Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—Test case generation is an important but tedious task, such that researchers have devised many different prototypes that aim to automate it. As these are research prototypes, they are usually only evaluated on a few hand-selected case studies, such that despite great results there remains the question of usability in the "real world". EVOSUITE is such a research prototype, which automatically generates unit test suites for classes written in the Java programming language. In our ongoing endeavour to achieve real-world usability, we recently passed the milestone success of applying EVOSUITE on hundred projects randomly selected from the SourceForge open source platform. This paper discusses the technical challenges that a testing tool like EVOSUITE needs to address when handling Java classes coming from real-world open source projects, and when producing JUnit test suites intended for real users.**

***Keywords*-test case generation; search-based testing; testing classes; search-based software engineering**

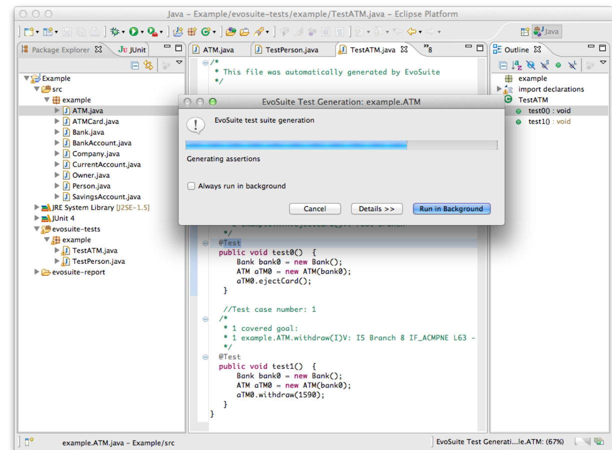

Figure 1. EVOSUITE is mainly used through the command line for large scale experiments, but it can also be used as an Eclipse plugin.

I. INTRODUCTION

Software testing is an essential part of any software development process. Because it is a difficult and error-prone task, automation is desirable. Several different techniques to automatically generate test cases have been proposed and evaluated, resulting in a wealth of research papers. Even though in software engineering research it is now common practice to perform a thorough evaluation of any newly proposed technique, researchers are seldom fortunate enough to have the time and resources to develop a full-fledged tool. Consequently, most empirical studies are performed using brittle prototypes and small sets of hand-selected case studies for which these prototypes are optimized [7].

EVOSUITE [4] is a research prototype that uses search-based techniques to derive unit tests for Java classes. It has its roots in the μTEST prototype [9], which in the initial study "survived" an experiment on two open source libraries – one of the largest studies on mutation testing at the time. Since then, EVOSUITE has been used to address several research questions in software testing, such as seeding [6], tuning [2], or bloat control [5]. Year after year, we gradually worked on larger case studies, encountering new and unexpected problems whenever new code was used for experiments. Despite the size of later studies (e.g., 1,741 classes [8]), these case studies still suffered from the problem that, because they were hand-selected and the tool optimized to work on them, results are difficult to generalize.

To overcome this problem, we randomly selected 100 open source projects from Sourceforge, one of the largest platforms for open source software on the Internet [7]. The only constraint on projects was for them to be compilable in order to run test cases. Given the sometimes surprising behaviour of the Java compiler, weird language constructs, and unexpected restrictions of the virtual machine, it required a major engineering effort to simply make EVOSUITE run on these projects, which we call the SF100 corpus of classes [7]. The experiments revealed to our surprise that, despite years of engineering effort, our tool is still missing essential features in order to work on "real" code. To the best of our knowledge, those essential features are not discussed in the literature, such that other prototypes are likely to suffer from the same problems we face with EVOSUITE.

In this paper, we describe the challenges EVOSUITE had to take in order to reach its current level of maturity. These challenges are not only related to the struggle against the surprises the Java language offers, but also to producing a tool that is usable for large scale research experiments, yet is also usable by practitioners. However, the SF100 experience shows us that there are many further challenges ahead before our research prototype is finally a completed tool.

II. WHOLE TEST SUITE GENERATION

EVOSUITE is a search-based testing (SBST) tool that uses a Genetic Algorithm (GA) to produce test suites (sets of test cases) achieving highest possible code coverage. The GA starts with an initial population of randomly generated test suites, and successively evolves these test suites by applying selection, crossover and mutation, until a solution has been found (100% coverage) or a stopping condition is reached (e.g., timeout). A key novelty of EVOSUITE over previous SBST tools is that it does not attempt to produce one test case at a time for distinct coverage goals, but that it evolves entire test suites targeting all coverage goals at the same time. This has several advantages: Infeasible coverage goals do not impair the search, there is no coincidental coverage, and there is no question on the order in which coverage goals should be addressed. We have shown [8] that this approach can outperform the classical approach of targeting individual coverage goals. During the search, the number of test cases in a test suite and the length of the individual test cases can vary, creating additional challenges such as bloat [5].

III. ASSERTION GENERATION

Automatically generated test cases can be used to find violations of partial specification. For example, the most generic specification of correct behaviour would be that a program should not crash, and so undeclared exceptions thrown by a test case likely indicate faults. However, if a class under test does not crash, there remain two important questions: 1) Is the class functionally correct? 2) What do the test cases need to check in order to ensure that future versions of the class preserve the current behaviour? To solve these questions, EVOSUITE produces test cases with *assertions*. A test assertion is a predicate that compares some aspect of the observed behaviour against the intended behaviour. In the absence of a complete specification the intended behaviour is not known. However, EVOSUITE can pinpoint those assertions that check important variables, and the locations where to check these variables; this is achieved using mutation testing. Details of this approach can be found in [9]. To find faults in the current version of the class under test the developer needs to inspect and verify each of these assertions.

IV. USABILITY CHALLENGES

When test cases are generated, even if failures are automatically found, the user still need to actually look at the generated JUnit files. This is done not only to check if the failures are indeed symptoms of real faults (and not maybe just some null pointer exception due to a violated implicit precondition), but also to check if the captured behavior (e.g., the test assertions in the JUnit files generated by EVOSUITE) is correct. Even if one would accept a test case and its assertions without questioning, e.g., when building a regression test suite, then there still remains the problem that once a test case fails, the user will need to step through the JUnit test during debugging. Therefore, it is extremely important that the JUnit files generated by EVOSUITE are easy to read and understand.

Such a *readability* goal could be considered as a further objective to optimize besides the usual system under test (SUT) code coverage criteria. Unfortunately, as it is difficult to numerically quantify how readable a JUnit test case is, it is not possible yet to cast this problem as a search problem. In this section, we discuss some of the approaches we employed in EVOSUITE to address the readability problem, although we have not yet carried out a empirical study with human subjects to assess and quantify their effectiveness.

A. Test Minimization

The first step to improve readability is to generate test suites that are *small*. EVOSUITE includes the total length of a test suite as secondary optimization goal, such that the search prefers smaller test suites. However, as the stopping conditions are based on resources (e.g., number of fitness evaluations or maximum time) or coverage (stop as soon as 100% coverage is achieved) the resulting test suites are not optimal with respect to their size. EVOSUITE therefore minimizes test suites as a post-processing step.

In our initial experiments [3] EVOSUITE tried to achieve a globally minimal test suite by removing all unnecessary statements. Although this turned out to be very effective, the minimal number of statements in a test suite is often achieved with a small number of fairly long test cases. However, this does not match the intention of unit tests, which are supposed to be small and fast. EVOSUITE therefore now applies a different minimization strategy: Given the resulting test suite, EVOSUITE iterates over all individual coverage goals. For each coverage goal, it randomly selects one test out of the resulting test suite that covers this goal, and then minimizes this test case with respect to the coverage goal. If a goal is already covered by a previous minimized test case, then no new test case is added for it. This strategy may not achieve the minimal number of statements in total, but results in much more readable test cases.

During the minimization EVOSUITE iterates over the coverage goals sorted in order of appearance in the source code. This way, the order of the test cases in the JUnit test suite matches the order of methods in the class under test.

B. Value Minimization

The second minimization strategy applied by EVOSUITE does not involve the test case lengths, but rather the values contained in these test cases. Because of its inherent randomness, test cases produced by EVOSUITE will contain random numbers and random strings, as long as they achieve the goal of satisfying coverage. However, large random numbers might distract from the actual purpose of a test case. Therefore, EVOSUITE applies a binary search for each

declared number x between x and 0, as is also done by Pex [10]. Furthermore, EVOSUITE tries to simplify strings by attempting to remove every character. After this minimization, all values are either 0/“”, or they represent a value that is necessary in order to achieve a coverage goal.

C. Constant Inlining

Internally, EVOSUITE uses a representation where each statement in a sequence defines one variable. This means that each constant value is defined in a separate statement, and if a primitive value is needed as a parameter then such a variable is used; potentially the same variable is used at different places. To further shorten tests and to avoid confusion through variable reuse, EVOSUITE inlines all constants before writing JUnit test code.

D. Variable Naming

Finally, the choice of variable names in the JUnit test code has an important effect on the readability. EVOSUITE originally used a simple naming convention where each statement defined a variable `var`n with successive values of n. However, to make it more apparent what type a variable has and how variables interact, EVOSUITE now uses a naming convention where variables are named using the classname in camel case and lower caps first letter, following by the number of the variable. For example, integers are named `int0`, `int1`, etc., while an instance of the class `DateTime` would for example be `dateTime0`.

V. ENGINEERING CHALLENGES

Generating and executing test cases for real world software is challenging. Furthermore, as probably anyone who has developed a research tool targeting the Java language can confirm, the language, compiler, and virtual machine sometimes offer unexpected and surprising behaviour which needs to be accepted when trying to develop a universal tool. In this section, we summarize some of our efforts to handle the challenges faced when developing EVOSUITE for Java.

A. Environment Problem

Real-world software often interacts with its environment. For example, the SUT can read and write files, open TCP connections with remote hosts, start GUI windows, react to mouse/keyboard events, etc. If those cases are not properly taken care of, the user might receive unwelcome surprises, like the testing tool calling the SUT with inputs that delete files randomly from the file system! An often observed behaviour, when running EVOSUITE on classes with I/O, is that the search leads to creation of files with random strings as filenames; after a couple of minutes of running EVOSUITE there can be thousands of such files.

To run EVOSUITE on SF100, using a Java “security manager” is thus simply compulsory. When performing potentially unsafe actions, Java code automatically asks the currently active security manager for permission. EVOSUITE uses a custom security manager, which is activated during test execution. This security manager allows all permissions to the threads spawned by EVOSUITE, whereas the thread running the test cases, and any thread spawned by the SUT, are treated specially. In the default configuration, the custom security manager only allows I/O for classloaders (identified by inspecting the stack trace), as well as a number of selected safe or essential operations (e.g., reading properties, loading libraries, reflection, and handling charsets and fonts). As the security framework in Java was designed to handle *applets* (programs downloaded from Internet and run inside a browser), there are several operations that are potentially malicious for an applet (e.g., stealing user information), but that would be harmless when a program is tested with EVOSUITE. Therefore, the security manager in EVOSUITE likely is overprotective, but identifying which operations are safe for testing tools is still an open research question [7].

B. Non-determinism and Undesired Behaviour

There are several operations allowed in Java with unwanted side-effects for test generation. An important call that should never be allowed in client code is a call to `System.exit`, which would potentially shut down EVOSUITE. Calls to `System.exit` are therefore replaced during class loading with calls to a static helper method that raises a custom exception, which allows EVOSUITE to detect when a test case has attempted to call `System.exit`.

A less obvious problem is presented by calls to `System.currentTimeMillis` or `Random.nextInt`: Such functions are not unsafe, but they may cause non-deterministic behaviour. In particular, if EVOSUITE adds an assertion based on a value that is influenced by a random number of the current system time, then such an assertion would fail on re-execution of the test. Therefore, EVOSUITE replaces these functions with custom helper methods which return incrementing integer numbers.

C. Java Peculiarities

In Java, there is a limit (64 kilobytes) to the size a method in a class can have. A class which does not satisfy those limits will simply fail to be loaded in the JVM. This is a problem for tools that instrument the source/bytecode of the SUT (e.g., to monitor the test case executions), as such instrumentation could exceed those limits. This is a particularly dire issue for testability transformations and mutation testing, as they often add a lot of extra branches in the SUT. This is a problem we actually faced for quite a few of the classes in the SF100 corpus. EVOSUITE therefore uses a configurable upper bound on the number of mutations that can be applied in a single method during mutation testing (with an empirically determined default value of 800).

D. Master/Slave Architecture

For a range of reasons, EvoSuite uses a master/slave architecture. A master process is responsible to spawn one or more slave processes, where the actual search for test data is done. Communication is carried out through TCP to avoid having EvoSuite depending on specific operating system signal messages (EvoSuite is meant to run on all the major operating systems, like Linux, Windows, Mac and Solaris) and to make it possible to run EvoSuite in parallel on several machines. Search algorithms can be easily parallelized (e.g., fitness function evaluations of a GA population done in parallel on several cores), and that leads to better results within the same amount of time. However, in the future such TCP communications will be replaced with RMI to make EvoSuite easier to code and maintain.

On one hand, if one is only interested in obtaining parallelism on the same multi-core machine, then the slaves could be simply run as threads in the same master process. On the other hand, in EvoSuite, and likely in any testing tool addressing real-world software, having the slaves running on separated processes was a necessity. One reason is that the code executed by the SUT could interfere with the master and the other slaves in many unexpected ways. Another reason is that, in this way, it was easier to address several complications of handling real-world software, as we will show throughout the rest of the paper, as for example muting the SUT output, spawning and stopping threads, memory consumption, etc.

E. Silencing the SUT

The SUT might print text, e.g., by direct calls to `System.out` or logging frameworks. This is a major issue, as such printings will end up mixed in the EvoSuite logs and console outputs, if not properly taken care of. In some extreme cases, this could even exhaust the available disk space on the host computer by generating extremely large log files, which would also lead to high computational overhead due to the I/O operations (this actually happened when we started experiments on SF100).

All the EvoSuite logging is done on the master process. All the printing in the slave processes is not redirected to any file/console, and this solves the problem. As further optimization, we replace the objects `System.out` and `System.err` with a `ByteArrayOutputStream` which is flushed after each test case execution. However, in the slave processes, we still want to get logging information from the EvoSuite framework. This is essential for debugging reasons. To achieve this goal, we used the *Logback*[1] logging framework. In particular, we made use of the `ch.qos.logback.classic.net.SocketAppender`. In other words, we configured the master process as a server opening a TCP port, and print logs when receiving log events from the slaves.

Interestingly, such solution was not enough. For example, the SUT could use the Logback framework itself, and so we would receive log events from the SUT as well! This is actually the case in some of the SF100 projects, which is a further argument to stress out the importance of large empirical studies on unbiased case studies. This issue was addressed by muting the root logger (i.e., *level="OFF"*), and then enabling loggers just for the `org.evosuite` package. In other words, the master process prints only the log events generated by a `org.evosuite` logger, whereas all the other events received by the slave processes are simply discarded.

F. Killing Threads

Unfortunate combinations of method calls or errors in the SUT can easily lead to long test execution times or even infinite loops. EvoSuite therefore uses a timeout (default value: 5000ms) after which it tries to terminate a test execution. However, there are situations where the SUT does not react to such termination requests. Furthermore, when we run a test case during the search, the SUT might spawn several extra threads. When a test case execution is finished (i.e., the EvoSuite driver has called the last method in the test case sequence), then we would like that all the spawn threads should be stopped as well. Otherwise, such threads could interfere with the following test cases that are going to be executed next. Furthermore, only a pre-defined number of threads can be present in a JVM at any given time (which usually depends on the operating system).

Unfortunately, Java does not support a reliable approach to stop threads. For example, the method `Thread.stop` is deprecated. The "recommended" approach is to signal the thread to be interrupted (e.g., by calling the method `Thread.interrupt`), but then it is up to the thread to check whether it was interrupted and stop if so. Unfortunately, threads spawn by the SUT could simple ignore those interrupt requests coming from EvoSuite.

EvoSuite instruments the SUT as well as *all* loaded classes such that, whenever execution passes a new line in the source code or enters a new method, it checks if the test execution should be terminated. If this is the case, then execution is yielded by throwing a custom timeout exception. However, this may not be sufficient: the execution can be stuck outside the instrumentable range of classes. Frequently occurring examples are calls to `java.lang.BigInteger`, which easily end up in extremely long lasting calculations of `gcd`, or new threads spawned by GUI components (e.g., the AWT event handler). As these classes are loaded by the Java bootclass-loader, they cannot be instrumented and will thus ignore EvoSuite's attempts to yield execution. If EvoSuite fails to join all spawned threads, it creates a new test execution thread in its

[1] http://logback.qos.ch, accessed 16.09.2012

thread pool, sets the execution priority of the old execution thread to low, and informs the execution tracer mechanism to ignore calls from this thread. The number of such threads EvoSuite allows to exist can be configured, and if that number is exceeded, then EvoSuite stops the search.

G. Resource Limits

When running a Java application, there are limits to the amount of heap space the JVM will use. This can be for example set with the `-Xmx` option when a Java application is started. If this option is not set, a default value that can vary depending on the JVM version and operating system is used. The SUT might require a large amount of memory, but that information is not directly accessible in the bytecode of the SUT. As precaution, EvoSuite allocates a large amount of heap space, even if that is not going to be fully used.

When generating test cases, it is possible that extremely large amounts of memory are allocated, which may kill the EvoSuite slave processes due to out of memory exceptions. For example, assume that the SUT has a method that allocates an array whose size is based on an integer input parameter (e.g., a variable size vector whose buffer size is given as input in the constructor). If a testing tool generates the value two billions as input data (which is a valid input value), then just for that single method call the JVM would require two gigabytes of heap space! It is not only a matter of arrays, but any data structure that can grow in size. Even if one constrains the search to only small integer values (which would lead to several side effects, as for example making some feasible branches impossible to cover), that would not solve the problem as there can be many different reasons for which the SUT can increase its size.

In Java, it is possible to query how much heap space is free (e.g., there are several methods in the `java.lang.Runtime` class to query the state of the heap). A "brute force" approach to the memory problem in the SUT could be to instrument its bytecode such that, after each instruction, the memory is checked and the test case execution stopped if close to run out of memory. Unfortunately, such an approach would introduce a very large computational overhead to the execution of the test cases, which could severely hamper the search (i.e., less test cases will be evaluated). EvoSuite therefore checks the available memory after each fitness evaluation (i.e., test execution). If the available memory shrinks beneath a configurable threshold, then EvoSuite calls the garbage collector. If the garbage collector is not able to reclaim sufficient memory, then EvoSuite stops the search and gracefully shuts down, preserving the current result of the search.

For array allocations, EvoSuite uses additional instrumentation to prevent memory exhaustion: Before an array in the SUT is allocated, we check its size. If the size is above a configurable threshold (e.g., 10,000), then we stop that particular test case execution, and its fitness value will be penalized. This strategy is not free of possible negative side effects, but, for the time being, it is a good compromise that was necessary when using EvoSuite on SF100.

Not only the SUT can lead to memory issues, but also EvoSuite itself can lead to them. For example, the search might lead to generate test suite populations with many (long) test cases, and so consume all available memory. This is a problem that in Evolutionary Computation is generally known as *bloat*. We have implemented several techniques to handle such issue, and those are described in details in [5].

VI. Research Challenges

For a practitioner, likely the best way to run a test case generation tool is to have a plug-in to the IDE used to develop the SUT, as for example Eclipse, IntelJ and NetBeans. At the current moment, we are developing an EvoSuite plug-in to support Eclipse[2]. However, to run empirical studies on large case studies such as SF100 an IDE plug-in interface might be cumbersome. For this reason, EvoSuite can be run directly from the command line.

A. Parameters

Through the years, several new features have been added to EvoSuite, and each time we needed to evaluate whether those new features do indeed improve performance. All features/settings can be changed by command line using Java environment variables. For example, the crossover rate can be set with `-Dcrossover_rate=x`, where `x` is some numerical constant (e.g., 0.7). To prevent variables being ignored due to typos, all console inputs are validated (i.e., EvoSuite checks if those variable names point to existing variables) when EvoSuite starts.

A complex tool like EvoSuite has hundreds of parameters that can be set (i.e., "configuration" properties). Tuning them can be seen as a major challenge, although reasonable "default" values are often sufficient [2]. To study whether some features improve performance, we first need to define how performance is measured. Depending on the study, there are several "runtime" properties that can be considered, as for example branch coverage, test suite size, mutation score, number of executed statements, etc. When we carry out experiments, to analyze the results, we need to know which configuration property had non-default settings, and what were the outputs of the runtime properties of interest.

B. Data Collection

To help analyze the results of the empirical studies, EvoSuite can generate comma separated variable (CSV) files. There is a column for each configuration/runtime property of interest, and each row contains the results of a run of EvoSuite on the SUT, i.e. the properties of the best

[2]Please refer to the EvoSuite website at http://www.evosuite.org for the current status of the plug-in

final test suite given as output to the users. When running EVOSUITE on an entire project, there is a row for each class. Although earlier versions of EVOSUITE dumped all available information in these CSV files, at some point with increasing case study size the memory footprint and I/O overhead became too big. Therefore, EVOSUITE has a configuration property that is used to define which configuration/runtime properties to save in the CSV files.

The choice of using CSV was to simplify data collection and analysis. Another option could had been to use a database, but that could make the installation of EVOSUITE more cumbersome. Furthermore, CSV can be directly read in R^3, which we use for all the statistical analyses and generation of tables/graphs. As EVOSUITE is based on randomized algorithms, the use of statistical tests is essential to properly analyze the results of the empirical studies [1].

C. *Running* EVOSUITE *Experiments on a Cluster*

To achieve sound empirical evidence, large case studies need to be used, and each experiment needs to be repeated several times with different seeds to take into account the randomness of EVOSUITE. Therefore, for all our empirical studies, we had to use a cluster of computers. Each run of EVOSUITE on a SUT is independent from the others, so it is pretty easy to parallelize the empirical analyses.

Whenever we prepare a new empirical study, we create a Python script that generates n shell scripts. Each shell script contains a sequence of calls to EVOSUITE with the different parameter settings and on different SUTs. Those shell scripts are the "jobs" that will be run on the cluster (e.g., on a IBM cluster, by using the command `qsub`). For example, if we want to run experiments on different settings of the crossover rate (e.g., three values, 0.2, 0.5 and 0.8) on the SF100 corpus, and we want 10 repetitions with different seeds per run, the Python script could generate $n = 10 \times 3 \times (9000/100) = 2700$ shell scripts, assuming a case study of roughly 9,000 classes (e.g., the SF100 corpus) and 100 SUTs per job. Note, as we mostly use time as stopping criterion for the search, each job will run for roughly the same amount of time, regardless of the complexity of the SUT.

Depending on the number of configurations to experiment with, one might want to subdivide the empirical study in a different number n of jobs. As a rule of thumb, we found it very useful to have different seeds in different jobs, and a number of jobs per seed that is roughly equivalent to the number of cores available in the cluster. Applied to the previous example, that would mean that those $n = 2700$ jobs would be fine for a cluster with roughly 270 available nodes. The reason for such a rule of thumb is simple: This way it is possible to run a "single seed" (i.e., all the different EVOSUITE configurations using the same random seed) by using all the available nodes. This was useful for debugging (e.g., running a single seed on entire case study to check for crashes and violations of EVOSUITE assertions) and when we needed to run more experiments (e.g., when time is available, it is usually good to have more repetitions to improve the power of the statistical tests [1]).

[3]http://www.r-project.org, accessed 16.09.2012

VII. CONCLUSIONS

Producing a research prototype that works on a fixed set of case study examples is hard work; producing a research prototype that works on a *large* number of case studies that are not manually selected is even harder. In this paper, we have summarized our efforts in making the EVOSUITE test generation tool usable on any *real* code, by *real* users. As our experiments on the SF100 corpus of classes show the engineering effort that has gone into EVOSUITE does pay off, yet there remain many challenges. First and foremost, the environmental dependencies are the primary reason for low code coverage on real code, and we are working on solutions for this and other problems.

To learn more about EVOSUITE, visit our Web site:

`http://www.evosuite.org`

Acknowledgments. This project has been funded a Google Focused Research Award on "Test Amplification". Andrea Arcuri is funded by the Norwegian Research Council.

REFERENCES

[1] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[2] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011, pp. 33–47.

[3] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. IEEE Computer Society, 2011, pp. 31–40.

[4] ——, "EvoSuite: Automatic test suite generation for object-oriented software." in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.

[5] ——, "It is not the length that matters, it is how you control it," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 150 – 159.

[6] ——, "The seed is strong: Seeding strategies in search-based software testing," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 121–130.

[7] ——, "Sound empirical evidence in software testing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188.

[8] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering (TSE)*, 2012.

[9] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 278–292, 2012.

[10] N. Tillmann and N. J. de Halleux, "Pex — white box test generation for .NET," in *International Conference on Tests And Proofs (TAP)*, 2008, pp. 134–253.