Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation

José Campos[1] Andrea Arcuri[2] Gordon Fraser[1] Rui Abreu[3]

[1]Department of Computer Science, University of Sheffield, UK

[2]Certus Software V&V Center Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

[3]Faculty of Engineering, University of Porto Porto, Portugal

ABSTRACT

In object oriented software development, automated unit test generation tools typically target one class at a time. A class, however, is usually part of a software project consisting of more than one class, and these are subject to changes over time. This context of a class offers significant potential to improve test generation for individual classes. In this paper, we introduce *Continuous Test Generation* (CTG), which includes automated unit test generation during continuous integration (i.e., infrastructure that regularly builds and tests software projects). CTG offers several benefits: First, it answers the question of how much time to spend on each class in a project. Second, it helps to decide in which order to test them. Finally, it answers the question of which classes should be subjected to test generation in the first place. We have implemented CTG using the EVOSUITE unit test generation tool, and performed experiments using eight of the most popular open source projects available on GitHub, ten randomly selected projects from the SF100 corpus, and five industrial projects. Our experiments demonstrate improvements of up to +58% for branch coverage and up to +69% for thrown undeclared exceptions, while reducing the time spent on test generation by up to +83%.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

General Terms. Algorithms, Experimentation, Reliability

Keywords. Unit testing, automated test generation, continuous testing, continuous integration

1. INTRODUCTION

Research in software testing has resulted in advanced unit test generation tools such as EVOSUITE [7] or Pex [33]. Even though these tools make it feasible for developers to apply automated test generation on an individual class during development, testing an entire project consisting of many classes in an interactive development environment is still problematic: Systematic unit test generation is usually too computationally *expensive* to be used by developers on entire projects. Thus, most unit test generation tools are based on the scenario that each class in a project is considered a unit and tested independently.

In practice, unit test generation may not always be performed on an individual basis. For instance, in industry there are often requirements on the minimum level of code coverage that needs to be achieved in a software project, meaning that test generation may need to be applied to *all* classes. As the software project evolves, involving code changes in multiple sites, test generation may be repeated to maintain and improve the degree of unit testing. Yet another scenario is that an automated test case generation tool might be applied to all classes when introduced for the first time in a legacy project. If the tool does not work convincingly well in such a case, then likely the tool will not be adopted.

By considering a software project and its evolution as a whole, rather than each class independently, there is the potential to use the context information for improving unit test generation:

- When generating test cases for a set of classes, it would be sub-optimal to use the same amount of computational resources for all of them, especially when there are at the same time both trivial classes (e.g., only having get and set methods) and complex classes full of non-linear predicates.
- Test suites generated for one class could be used to help the test data generation for other classes, for example using different types of seeding strategies [9].
- Finally, test suites generated for one revision of a project can be helpful in producing new test cases for a new revision.

An attractive way to exploit this potential lies in using *continuous integration* [6]: In continuous integration, a software project is hosted on a controlled version repository (e.g., SVN or Git) and, at each commit, the project is built and the regression test suites are run to verify that the new added code does not break anything in the application. Continuous integration is typically run on powerful servers, and can often resort to build farms or cloud-based infrastructure to speed up the build process for large projects. This opens doors for automated test generation tools, in order to enhance the typically manually generated regression test suites with automatically generated test cases, and it allows the test generation tools to exploit the advantages offered when testing a project as a whole.

In this paper, we introduce *Continuous Test Generation (CTG)*, which enhances continuous integration with automated test generation. This integration raises many questions on how to test the classes in a software project: For instance, in which order should they be tested, how much time to spend on each class, and which information can be carried over from the tests of one class to another? To provide first answers to some of these questions, we have implemented CTG as an extension to the EVOSUITE test generation tool and performed experiments on a range of different software projects. In detail, the contributions of this paper are as follows:

ASE'14, September 15-19, 2014, Västerås, Sweden

- We introduce the problem of generating unit tests for whole projects, and discuss in details many of its aspects.
- We describe different strategies of scheduling the order in which classes are tested to improve the performance.
- We propose a technique to incrementally test the units in a software project, leading to overall higher code coverage while reducing the time spent on test generation.
- We present a rigorous empirical study on 10 open source projects from the SF100 corpus, eight of the most popular projects on GitHub, and five industrial projects supporting the viability and usefulness of our presented techniques.
- All the presented techniques have been implemented as an extension of the EvoSuite test generation tool, which is freely available for researchers and practitioners at: www.evosuite.org.

Our experiments demonstrate that, by intelligently using the information provided when viewing a software project as a whole, the techniques presented in this paper can lead to improvements of up to +58% for branch coverage and up to +69% for thrown undeclared exceptions. At the same time, applying this test generation incrementally not only improves the test effectiveness, but also saves time — by up to +83%. However, our experiments also point out important areas of future research on CTG: There is potential to further improve budget allocation techniques, and although we provide evidence that seeding can be beneficial, we also observe cases where this is not the case, thus calling for more intelligent techniques to apply seeding.

2. BACKGROUND

This section provides a brief overview of the key concepts that are relevant to this paper.

2.1 Continuous Integration and Testing

The roots of continuous integration [6] can be traced back to the Extreme Programming methodology. One of the main objectives of continuous integration is to reduce the problems of "integration hell", i.e., different engineers working on the same code base at the same time, such that their changes have to be merged together. One approach to deal with such problems is to use controlled version repositories (e.g., SVN or Git) and to commit changes on a daily basis, instead of waiting days or weeks. At each new code commit, a remote server system can build the application automatically to see if there are any code conflicts. Furthermore, at each new build, the available regression test suites can be run to see if any new features or bug fixes break existing functionality; developers responsible for new failures can be automatically notified.

Continuous integration is widely adopted in industry, and several different systems are available for practitioners. The most popular ones include the open source projects Jenkins[1], CruiseControl[2], Apache Continuum[3], Oracle's Hudson[4] and Bamboo from Atlassian[5]. The functionalities of those continuous integration systems can typically be extended with plugins. For example, at the time of writing this paper, Jenkins had more than 600 plugins, including plugins that measure and visualise code coverage of tests.

Besides running regression test suites on dedicated continuous integration servers, these suites could also be automatically run in the background on the development machines by the IDE (e.g., Eclipse). The idea would be to provide feedback to the developers as soon as possible, while they are still editing code. Some authors call this approach *continuous testing* [27, 28].

In principle, continuous testing does not need to be restricted to regression test suites. If automated oracles are available (e.g., formal post-conditions and class invariants), then a test case generation tool can be run continuously 24/7, and can report to the developers as soon as a specification condition is violated. One such form of "continuous" testing is for example discussed in [22].

2.2 Automated Unit Test Generation

Unit test suites are commonly used by developers to check the correctness and completeness of the code they wrote, and to guard it against future regression faults. Such test suites are commonly hand-written, often even before the classes they are testing are implemented (the so called "Test Driven Development"). To support developers in this task, researchers have devised methods to automatically generate unit tests. Some approaches assume the existence of a formal model of the class (e.g., UML [23]), and many other popular approaches require only source code.

To generate unit tests from source code, the simplest approach is to do so *randomly* [2]. This approach can produce large numbers of tests in a short time, and the main intended usage is to exercise generic object contracts [5, 25] or code contracts [20]. Approaches based on symbolic execution have been popularized by *Dynamic Symbolic Execution (DSE)* [18], which efficiently explores paths from a given entry function. For example, a popular variant to apply DSE is to manually write a parameterized unit test as an entry function, and then to explore the paths through a program by deriving values for the parameters of the test [33]. Generating unit tests that resemble manually written tests (i.e., few short test cases with high coverage) is commonly done using *Search-based Software Testing (SBST)*. When applying SBST for unit test generation, efficient meta-heuristic search algorithms such as genetic algorithms are used to evolve sequences of method calls with respect to code coverage and other criteria [34].

Automatically generated test cases could be added to existing regression suites and be run within continuous integration. For this purpose there are also commercial tools like AgitarOne[6] that can be integrated into build environments (e.g., Maven), such that new test suites are generated with each new build of a system. Such tools will face the same problem we are addressing in this paper.

Continuous test generation is closely related to *test suite augmentation*: Test suite augmentation is an approach to test generation that considers code changes and their effects on past test suites. Some test suite augmentation techniques aim to restore code coverage in test suites after changes by producing new tests for new behaviour (e.g. [37]), while other approaches explicitly try to exercise changed code to reveal differences induced by the changes (e.g., [24, 29, 31]); we are also working on extending EvoSuite in this direction [30]. Although test suite augmentation is an obvious application of CTG, there are differences: First, CTG answers the question of *how to implement* test suite augmentation (e.g., how to allocate the computational budget to individual classes). Second, while CTG can benefit from information about changes, it can also be applied *without* any software changes. Third, CTG is not tied to an individual coverage criterion; for example, one could apply CTG such that once coverage of one criterion is saturated, test generation can target a different, more rigorous criterion. Finally, the implementation as part of continuous integration makes it possible to automatically notify developers of any faults found by automated

[1] http://jenkins-ci.org, accessed 03/2014.
[2] http://cruisecontrol.sourceforge.net, accessed 03/2014.
[3] http://continuum.apache.org, accessed 03/2014.
[4] http://hudson-ci.org, accessed 03/2014.
[5] http://atlassian.com/software/bamboo, accessed 03/2014.
[6] http://www.agitar.com/pdf/AgitarOneJUnitGeneratorDatasheet.pdf, accessed 03/2014.

oracles such as assertions or code contracts. Some of the potential benefits of performing test suite augmentation continuously have also been identified in the context of software product-lines [35].

2.3 The EvoSuite Unit Test Generation Tool

In this paper, we use the EVOSUITE [7] tool for automatic unit test suite generation for Java programs. EVOSUITE works at Java bytecode level (so it can also be used on third-party systems with no available source code), and it is fully automated: it does not require manually written test drivers or parameterized unit tests. For example, when EVOSUITE is used from its Eclipse plugin, a user just needs to select a class, and tests are generated with a mouse-click.

EVOSUITE implements a hybrid approach that combines the best of SBST and DSE [16] to generate unit test suites for individual Java classes. It uses a genetic algorithm in which it evolves whole test suites, which has been shown to be more efficient at achieving code coverage than generating tests individually [13, 14]. Depending on the search properties, DSE is adaptively used to satisfy coverage goals that are difficult for SBST.

Once unit tests with high code coverage are generated, EVOSUITE applies various post-processing steps to improve readability (e.g., minimizing) and adds test assertions that capture the current behaviour of the tested classes. To select the most effective assertions, EVOSUITE uses mutation analysis [15]. EVOSUITE can generate test suites covering different kinds of coverage criteria, like for example weak and strong mutation testing [14], and it can also aim at triggering undeclared exceptions [10]. EVOSUITE can be integrated into a programmer's development environment with its Eclipse plugin, or it can be used on the command line.

The whole test suite generation [13, 14] approach implemented by EVOSUITE removes the need to select an order in which to target individual coverage goals, and it avoids the need to distribute the search-budget among the individual coverage goals. The problem of choosing an order and distributing the test generation budget between coverage goals is similar to the problem of scheduling and distributing test generation budget between classes of a project. However, there are subtle differences that prevent an easy, direct application of the whole test suite approach at project level.

On one hand, when a test case is executed on a Class Under Test (CUT), we can easily collect information on all the branching predicates it executes. On the other hand, when generating tests for a specific CUT A, most of the other CUTs will not be executed. Even if a class B is called by A, test cases generated for A would not be directly useful as unit tests for B (e.g., all methods called would belong to A) unless post-processing is applied. Furthermore, there might be scalability issues when keeping in memory whole test suites for all CUTs in a project, all at the same time.

3. TESTING WHOLE PROJECTS

Test generation is a complex problem, therefore the longer an automated test generation tool is allowed to run on an individual class, the better the results. For example, given more time, a search-based approach will be able to run for more iterations, and a tool based on DSE can explore more paths. However, the available time budget is usually limited and needs to be distributed among all individual classes of a given software project. The problem addressed in this section can thus be summarized at high level as follows:

> *Given a project X, consisting of n units, and a time budget b, how to best use an automated unit test generation tool to maximize code coverage and failure detection on X within the time limit b?*

The values for n and b will be specific to the projects on which test generation is applied. In our experiments, the values for n range from 1 to 412, with an average of 90. Estimating what b will look like is more challenging, and at the moment we can only rely on the feedback of how our industrial partners think they will use EVOSUITE on whole projects. However, it is clear that already on a project of a few hundred classes, running EVOSUITE with a minimum of just a few minutes per CUT might take hours. Therefore, what constitutes a reasonable value for b will depend on the particular industrial scenario.

If EVOSUITE is run on developer machines, then running EVOSUITE on a whole project at each code commit might not be a feasible option. However, it could be run after the last code commit of the day until the day after. For example, on a week day, assuming a work schedule from 9 a.m. to 5 p.m., it could mean running EVOSUITE for 16 hours, and 64 hours on weekends. Given a modern multicore PC, EVOSUITE could even be run on a whole project during the day, in a similar way as done with regression suites in continuous testing [27,28]; but that could have side effects of slowing down the PC during coding and possible noise issues that might be caused by the CPU working at 100%. An alternative scenario would be a remote continuous integration system serving several applications/departments within a company. Here, the available budget b would depend on the build schedule and on the number of projects for which the continuous integration server is used. Some companies also use larger build-farms or cloud-infrastructure for continuous integration, which would allow for larger values of b, or more frequent runs of EVOSUITE.

The simplest, naïve approach to target a whole project is to divide the budget b equally among the n classes, and then apply a tool like EVOSUITE independently on each for b/n minutes (assuming no parallel runs on different CPUs/cores). In this paper, we call this *simple* strategy, and it is the strategy we have used in past empirical studies of EVOSUITE (e.g., [8]). However, this simple strategy may not yield optimal results. In the rest of this section, we describe different aspects of targeting whole projects that can be addressed to improve upon the *simple* strategy. Note that in principle test generation for a class can be finished before the allocated budget is used up (i.e., once 100% coverage is achieved). In this case, the time saved on such a class could be distributed on the remaining classes; that is, the schedule could be adapted dynamically during runtime. For our initial experiments we optimised for coverage and exceptions [10], where no test generation run would end prematurely. However, we will consider such optimisations as future work.

3.1 Budget Allocation

In the *simple* approach, each n CUT gets an equal share of the time budget b. If there are k CPUs/cores that can be used in parallel (or a distributed network of computers), then the actual amount of available computational resources is $k \times b$. For example, assuming a four core PC and a 10 minute budget, then a tool like EVOSUITE could run on 40 CUTs for one minute per CUT. However, such a resource allocation would not distinguish between trivial and complex classes requiring more resources to be fully covered. This budget allocation can be modeled as an optimization problem.

Assume n CUTs, each taking a share of the total $k \times b$ budget, with b expressed as number of minutes. Assume a testing tool that, when applied on a CUT c for z minutes, obtains performance response $t(c, z) = y$, which could be calculated as the obtained code coverage and/or number of triggered failures in the CUT c. If the tool is randomized (e.g., a typical case in search-based and dynamic symbolic execution tools like EVOSUITE), then y is a random variable. Let $|Z| = n$ be the vector of allocated budgets for each CUT, and $|Y_Z| = n$ the vector of performance responses $t(c, z)$ calcu-

lated once Z is chosen and the automated testing tool is run on each of the n CUTs for the given time budgets in Z. Assume a performance measure f on the entire project that should be maximized (or minimized). For example, if y represents code coverage, one could be interested in the average $f(Z) = \frac{\sum_{y \in Y_Z}}{n}$ of all of the CUTs. Under these conditions, maximizing $f(Z)$ could be represented as a search problem in which the solution space is represented by the vector Z, under two constraints: first, their total budget should not exceed the total, i.e., $\sum_{z_i \in Z} z_i \leq k \times b$, and, second, it should be feasible to find a "schedule" in which those n "jobs" can be run on k CPUs/cores within b minutes. A trivial consequence of this latter constraint is that no value in Z can be higher than b.

Given this optimization problem definition, any optimization/search algorithm (e.g., genetic algorithms) could be used to address it. However, there are several open challenges with this approach, like for example:

- The optimization process has to be quick, as any time spent on it would be taken from the budget $k \times b$ for test generation.
- The budget allocation optimization has to be done before generating any test case for any CUT, but the values $t(c, z) = y$ are only obtained *after* executing the testing tool and the test cases are run. There is hence the need to obtain an estimate function t', as t cannot be used. This t' could be for example obtained with machine learning algorithms [21], trained and released as part of the testing tool. A further approach could also be to execute some few test cases, and use the gathered experience to predict the complexity of the CUT for future test case generation efforts.
- Even if it is possible to obtain a near perfect estimate function $t' \simeq t$, one major challenge is that its output should not represent a single, concrete value y, but rather the probability distribution of such a random variable. For example, if the response is measured as code coverage, a possibility could be that the output of $t'(c, z)$ is represented by a $|R| = 101$ vector, where each element represents the probability P of y obtaining such a code coverage value (with 1% interval precision), i.e. $R[i] = P(y == i\%)$, where $\sum r \in R = 1$. Based on how R is defined (could even be a single value representing a statistics of the random variable, like mean and median), there can be different ways to define the performance measure $f(Z)$ on the entire project.

After having described the budget allocation problem in general, in this paper we present a first attempt to address it. We start our investigation of addressing whole projects with a simple to implement technique. First, each CUT will have a minimum amount of the time budget, e.g., $z \geq 1$ (i.e., one minute). Then the remaining budget $(k \times b) - (n \times 1)$ can be distributed among the n CUTs proportionally to their number of branches (but still under the constraint $z \leq b$). In other words, we can estimate the difficulty of a CUT by counting its number of branches. This is an easy way to distinguish a trivial from a complex CUT. Although counting the number of branches is a coarse measure, it can already provide good results (as we will show in the empirical study in this paper). It is conceivable that more sophisticated metrics such as, for example, cyclomatic complexity, may lead to improved budget distribution, and we will investigate this in future research.

Having a minimum amount of time per CUT (e.g., $z \geq 1$) is independent of whether a smart budget allocation is used. For example, if we only have one core and budget $b = 5$ minutes, it would make no sense to run EvoSuite on a project with thousands of CUTs, as only a few milliseconds would be available on average per CUT. In such cases, it would be more practical to just run EvoSuite on a subset of the classes (e.g., five) such that there is enough time (e.g., one minute) for each of those CUTs to get some usable result. Ensuring that all classes are tested would then require allocating the budget to different classes in successive runs of EvoSuite in the following days (Section 4.1 will present some more ideas on how to use historical data).

3.2 Seeding Strategies

After allocating the time budget Z for each of the n CUTs, the test data generation (e.g., using EvoSuite) on each of those n CUTs will be done in a certain order (e.g., alphabetically or randomly), assuming $n > k$ (i.e., more CUTs than possible parallel runs). This means that when we start to generate test cases for a CUT c, we will usually have already finished generating test suites for some other CUTs in that project, and these test suites can be useful in generating tests for c. Furthermore, there might be information available from past EvoSuite runs on the same project. This information can be exploited for *seeding*.

In general, with seeding we mean any technique that exploits previous knowledge to help solve a testing problem at hand. For example, in SBST existing test cases can be used when generating the initial population of a genetic algorithm [38], or can be included when instantiating objects [9]. Seeding is also useful in a DSE context, in particular to overcome the problem of creating complex objects [32], and the use of seeding in test suite augmentation is established for SBST and DSE-based augmentation approaches [37].

In order to make it possible to exploit information from different CUTs within a run of EvoSuite on a whole project, one needs to *sort* the execution of the n CUTs in a way that, when a class c can use test cases from another class c', then c' should be executed (i.e., generated test for) before c (and if test execution for c' is currently running, then postpone the one of c till c' is finished, but only if meanwhile another class c'' can be generated tests for). For example, if a CUT A takes as input an object of type B, then to cover A we might need B set in a specific way. For example:

```
public class A {
  public void foo(B b) {
    if (b.isProperlyConfigured()) {
      ... // target
    } } }
```

Using the test cases generated for B can give us a pool of interesting instances of B. To cover the target branch in $A.foo$, one could just rely on traditional SBST approaches to generate an appropriate instance of B. But, if in CTG we first generate test suites for B, then we can exploit those tests for seeding in A. For example, each time we need to generate an input for $A.foo$, with a certain probability (e.g., 50%) we can rather use a randomly selected instance from the seeded pool, which could speed up the search.

4. Continuous Test Generation (CTG)

So far, we have discussed generating unit tests for all classes in a project. However, projects evolve over time: classes are added, deleted, and changed, and automated test generation can be invoked regularly during continuous integration, by extending it to CTG. CTG can exploit all the historical data from the previous runs to improve the effectiveness of the test generation.

There are two main ways in which CTG can exploit such historical data: First, we can improve the *budget allocation*, as newly introduced classes should be prioritized over old classes that have been extensively tested by CTG in previous runs. Second, the test cases generated in the previous runs can be directly used for *seeding* instead of regenerating tests for each class from scratch at every CTG run on a new software version.

4.1 Budget Allocation with Historical Data

The *Budget* allocation described in Section 3.1 only takes into account the complexity of a CUT. However, there are several factors that influence the need to do automated test generation when it is invoked repeatedly. Usually, a commit of a set of changes only adds/modifies a few classes of a project. If a class has been changed, more time should be spent on testing it. First, modified source code is more prone to be faulty than unchanged source code [19]. Second, the modifications are likely to invalidate old tests that need to be replaced, or add new behaviour for which new tests are required [26]. If a class has *not* been changed, invoking automated test generation can still be useful if it can help to augment the existing test suite. However, once the test generator has reached a maximum level of coverage and cannot further improve it for a given class, invoking it again will simply waste resources.

For example, suppose a project X has two classes: a "simple" one S, and a "difficult" one D. Assume that, by applying the *Budget* allocation (Section 3.1), the budget allocated for D is twice as much than for S, i.e. $z_D = 2 \times z_S$. Now, further suppose that only S has been changed since the last commit; in this case, we would like to increase the time spent on testing S, even though it is a simple one. For this, we first use an underlying basic budget assignment (e.g., *Budget* or *Budget & Seeding*), and then multiple by a factor $h > 1$, such that the budget for S becomes $z_S = h \times \frac{1}{2} \times z_D$. Thus, if $h = 2$ (which is the value we use in the experiments reported in this paper), then the *modified* simple class S will receive the same amount of time as the *unchanged* difficult class D.

Given an overall maximum budget (see Section 3.1), the total budget should not exceed this maximum, even in the face of changed classes. That is, $\sum_{z_i \in Z} z_i \leq k \times b$; however, it will happen that adding a multiplication factor h for new/modified classes results in the total budget exceeding this maximum. As test generation will be invoked regularly in this scenario, it is not imperative that all classes are tested, especially the ones that have not been modified. So, one can apply a strategy to skip the testing of some unchanged classes in the current CTG execution. To do that, we rank classes according to their complexity and the fact of whether they were modified, and then select the maximum number of classes such that the total budget $k \times b$ is not exceeded.

For classes that have not been changed, at some point we may decide to stop invoking the test generator on them. A possible way to do this is to monitor the progress achieved by the test generator: If 100% coverage has been achieved, then generating more tests for the same criterion will not be possible. If less than 100% coverage has been achieved, then we can invoke test generation again. However, if after several invokations the test generator does not succeed in increasing the coverage, we can assume that all coverage goals that the test generator can feasibly cover have been reached. In the context of this paper, we look at the last three runs of the test generator, and if there has been no improvement for the last three runs, then we stop testing a class until it is changed again.

4.2 Seeding Previous Test Suites

When repeatedly applying test generation to the same classes, the results of the previous test generation run can be used as a starting point for the new run. This is another instance of seeding, as described in Section 3.2. There are different ways how a previous result can be integrated into a new run of a genetic algorithm. For example, in previous work [9] where the goal was to improve upon manually written tests, we re-used the existing test cases by modifying the search operators of EvoSuite such that whenever a new test case was generated, it was based on an existing test case with a certain probability. Xu et al. considered the reuse of test cases during test suite augmentation for DSE [37] or search-based and hybrid techniques [36], by using the old tests as starting population of the next test generation run; in this approach the success of the augmentation depends strongly on the previous tests.

The approach we took in the context of CTG is to first check which of the previous tests still compile against the new version of a CUT. For example, if from version p_n to p_{n+1} a signature (i.e., name or parameters) of a method, or a class name is modified, test cases may no longer compile and therefore are not candidates to be included in the next test suite. One the other hand, tests that still compile on the new version of the CUT can be used for seeding. We insert such a suite as one individual into the initial population of the new genetic algorithm, thus essentially applying a form of elitism between different invocations of the genetic algorithm.

5. EMPIRICAL STUDY

We have implemented the techniques described in this paper as an extension of the EvoSuite unit test generation tool. This section contains an empirical evaluation of the different strategies. In particular, we aim at answering the following research questions:

RQ1: What are the effects of smart *Budget* allocation?

RQ2: What are the effects of *Seeding* strategies?

RQ3: How does combining *Seeding* strategies with smart *Budget* allocation fare?

RQ4: What are the effects of using CTG for test generation?

RQ5: What are the effects of *History*-based selection and *Budget* allocation on the total time of test generation?

5.1 Experimental Setup

To answer the research questions, we performed two different types of experiments: The first one aims to identify the effects of optimizations based on testing whole projects; the second experiment considers the scenario of testing projects over time.

5.1.1 Subject Selection

We used three different sources for case study projects: First, as an unbiased sample, we randomly selected ten projects from the SF100 corpus of classes [8] (which consists of 100 projects randomly selected from SourceForge); this results in a total of 279 classes. Second, we used five industrial software projects (1307 classes in total) provided by one of our industrial collaborators. Due to confidentiality restrictions, we can only provide limited information on the industrial software.

To simulate evolution with CTG over several versions, we required projects with version history. Because it is complicated to obtain a full version history of compiled software versions for each project in the two previous sets (due to different repository systems and compilation methods), we additionally considered the top 15 most popular projects on GitHub. We had to discard some of these projects: 1) For some (e.g., `JUnit`, `JNA`) there were problems with EvoSuite (e.g., EvoSuite uses `JUnit` and thus cannot be applied to the `JUnit` source code without modifications). 2) Some projects (`Jedis`, `MongoDB Java Driver`) require a server to run, which is not supported by EvoSuite yet. 3) We were unable to compile the version of `RxJava` last cloned (10 March, 2014). 4) Class files of the `Rootbeer GPU Compiler` project belong to the "`org.trifort.rootbeer`" package, however they are incorrectly compiled as "`edu.syr.pcpratts`" package. 5) Finally, we removed `Twitter4J`, the largest project of the 15 most popular projects, as our experimental setup would not have allowed to finish the experiments in time. This leaves eight projects (475 classes in total) with version history for experimentation.

5.1.2 Experiment Procedure

For each open source project of the SF100 corpus and industrial project, we ran EVOSUITE with four different strategies: *Simple*, smart *Budget* allocation (Section 3.1), *Seeding* strategy (Section 3.2), and a combination of the latter two (i.e, smart *Budget* and *Seeding* strategy at the same time, *Budget & Seeding*). For the open source projects from GitHub we ran EVOSUITE with the same four strategies, but also with another strategy, a *History* strategy (Section 4.1) which used seeding of previous test suites (Section 4.2).

When running EVOSUITE on a whole project, there is the question of how long to run it. This depends on the available computational resources and how EVOSUITE will be used in practice (e.g., during the day while coding, or over the weekend). In this paper, due to the high cost of running the experiments, we could not consider all these different scenarios. So, we decided for one setting per case study that could resemble a reasonable scenario. In particular, for all the case studies we allowed an amount of time proportional to the number of classes in each project, i.e., three minutes per CUT. For the smart *Budget* allocation, we allowed a minimum amount of time $z \geq 1$ minute (see Section 3.1).

Unlike the other strategies, the *History* strategy requires different versions of the same project. As considering the full history would not be feasible, we limited the experiments to the last 100 commits of each project, i.e., we considered the latest 100 consecutive commits of each project. Note, one of the eight projects only has 65 commits in its entire history.

For the experiments, we configured *History* to use the *Budget* allocation as baseline because the average branch coverage on the first set of experiments (10 projects randomly selected from SF100 corpus) achieved an highest relative improvement on that approach. The maximum time for test generation was calculated for *History* for each commit in the same way as for other strategies proportional to the number of CUTs in the project (three minutes per CUT).

On the open source projects from SF100, each experiment was repeated 50 times with different random seeds to take into account the randomness of the algorithms. As we applied *History* with a time window of 100 commits to the GitHub projects, we only ran EVOSUITE five times on these eight projects. On the industrial systems we were only able to do a single run.

Running experiments on real industrial case studies presents many challenges, and that is one of the reasons why they are less common in the software engineering literature. Even if it was not possible to run those experiments as rigorously as in case of the open source software, they do provide extra valuable information to support the validity of our results.

5.1.3 Measurements

As primary measurement of success of test generation we use branch coverage. However, branch coverage is only one possible measure to quantify the usefulness of an automatically generated test suite [4, 17]. In the presence of automated oracles (e.g., formal specifications like pre/post-conditions), one would also want to see if any fault has been found. Unfortunately, automated oracles are usually unavailable. One could look at program crashes, but that is usually not feasible for unit testing. However, at unit level it is possible to see if any exception has been thrown in a method of the CUT, and then check whether that exception is declared as part of the method signature (i.e., using the Java keyword `throws`).

As a second measurement we used undeclared exceptions. If an exception is declared as part of a method signature, then throwing such an exception during execution would be part of normal, expected behaviour. On the other hand, finding an undeclared exception would point to a unit level bug. Such a bug might not be critical (e.g., impossible to throw by the user through the application interfaces like a GUI), and could even simply point to "implicit" preconditions. For example, some exceptions might be considered as normal if a method gets the wrong inputs (e.g., a null object) but, then, the developers might simply fail to write a proper method signature. This is the case when an exception is explicitly thrown with the keyword `throw`, but then it is missing from the signature.

Whether a thrown exception represents a real fault is an import question for automated unit testing. In particular, it is important to develop techniques to filter out "uninteresting" exceptions that likely are just due to violated implicit preconditions. However, regardless of how many of these exceptions are caused by real bugs, a technique that finds more of these exceptions would be better. For this reason, tools like EVOSUITE not only try to maximize code coverage, but also the number of unique, undeclared thrown exceptions for each method in the CUTs, and experiments have shown that this can reveal reals faults [10].

For the first set of experiments the overall time per project was fixed. In the second set of experiments on CTG we also look at the time spent on test generation.

5.1.4 Analysis Procedure

The experiments carried out in this paper are very different than previous uses of EVOSUITE. In previous empirical studies, each CUT was targeted independently from the other CUTs in the same project. That was to represent scenarios in which EVOSUITE is used by practitioners on the classes they are currently developing. On the other hand, here when targeting whole projects there are dependencies: e.g., in the smart *Budget* allocation, the amount of time given to each CUT depends also on the number of branches of the other CUTs. When there are dependencies, analyzing the results of each CUT separately might be misleading. For example, how to define what is the branch coverage on a whole project?

Assume a project P composed of $|P| = n$ CUTs, where the project can be represented as a vector $P = \{c_1, c_2, \ldots, c_n\}$. Assume that each CUT c has a number of testing targets $\gamma(c)$, of which $k(c)$ are actually covered by applying the analyzed testing tool. Because tools like EVOSUITE are randomized, the scalar $k(c)$ value will be represented by a statistics (e.g., the mean) on a sample of runs (e.g., 50) with different random seeds. For example, if the tool was run r times, in which each time we obtained a number of covered targets $k_i(c)$, then $k(c) = \frac{\sum_{i=1}^{i=r} k_i(c)}{r}$. If we want to know what is the coverage for a CUT c, then we can use $cov(c) = \frac{k(c)}{\gamma(c)}$, i.e., number of covered targets divided by the number of targets, which is what usually done in the literature. But what would be the coverage on P? A typical approach is to calculate the average of those coverage values averaged over all the r runs:

$$avg(P) = \frac{1}{n} \sum_{c \in P} \frac{k(c)}{\gamma(c)} \, .$$

However, in this case, all the CUTs have the same weight. The coverage on a trivially small CUT would be as important as the coverage of a large, complex CUT. An alternative approach would be to consider the absolute coverage on the project per run:

$$\mu(P_i) = \frac{\sum_{c \in P} k_i(c)}{\sum_{c \in P} \gamma(c)} \, ,$$

and, with that, consider the average on all the r runs:

$$\mu(P) = \frac{1}{r} \sum_{i=1}^{i=r} \mu(P_i) \, .$$

Table 1: Branch coverage results for the 10 open source projects randomly selected from SF100 corpus.

For each project we report the branch coverage of the *Simple* strategy. For each of the other strategies, we report their branch coverage and the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *Simple* strategy. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold. Results on the open source case study are based on 50 runs per configuration.

Project	Classes	Simple	Budget			Seeding			Budget & Seeding		
		Coverage	Coverage	$\hat{A}_{12}$	Rel. Impr.	Coverage	$\hat{A}_{12}$	Rel. Impr.	Coverage	$\hat{A}_{12}$	Rel. Impr.
tullibee	18	39.1%	43.5%	**0.89**	+11.3%	39.6%	0.56	+1.1%	43.9%	**0.92**	+12.1%
a4j	45	62.5%	64.4%	**0.86**	+3.0%	55.3%	**0.00**	-11.5%	55.2%	**0.00**	-11.7%
gaj	10	66.5%	65.6%	0.46	-1.4%	67.5%	0.54	+1.5%	67.2%	0.53	+1.0%
rif	13	25.3%	25.0%	0.48	-1.4%	25.7%	0.58	+1.4%	24.8%	0.45	-2.0%
templateit	19	20.1%	24.6%	**0.97**	+22.4%	20.3%	0.53	+0.8%	24.9%	**0.97**	+23.7%
jnfe	51	38.7%	51.7%	**0.96**	+33.5%	43.9%	**0.64**	+13.4%	51.6%	**0.96**	+33.3%
sfmis	19	35.8%	46.7%	**1.00**	+30.6%	36.2%	0.55	+1.1%	46.3%	**0.99**	+29.3%
gfarcegestionfa	48	25.2%	33.4%	**0.96**	+32.5%	23.8%	0.43	-5.4%	33.1%	**0.95**	+31.5%
falselight	8	6.1%	6.2%	0.51	+2.0%	6.1%	0.50	0.0%	6.1%	0.50	0.0%
water-simulator	48	3.1%	3.8%	**0.75**	+19.1%	3.2%	0.53	+1.4%	4.0%	**0.78**	+27.2%

Table 2: Thrown exception results for the 10 open source projects randomly selected from SF100 corpus.

For each project we report the total number (i.e., sum of the averages over 50 runs for each CUT) of undeclared thrown exceptions of the *Simple* strategy. For each of the other strategies, we report their undeclared thrown exceptions and the effect sizes ($\hat{A}_{12}$ and relative ratio difference) compared to the *Simple* strategy. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold. Results on the open source case study are based on 50 runs per configuration.

Project	Classes	Simple	Budget			Seeding			Budget & Seeding		
		Exceptions	Exceptions	$\hat{A}_{12}$	Rel. Impr.	Exceptions	$\hat{A}_{12}$	Rel. Impr.	Exceptions	$\hat{A}_{12}$	Rel. Impr.
tullibee	18	23.46	29.36	**0.88**	+25.1%	23.46	0.49	0.0%	29.06	**0.92**	+23.8%
a4j	45	88.96	93.58	**0.77**	+5.1%	87.20	0.41	-2.0%	88.22	0.47	-0.9%
gaj	10	30.28	29.74	0.40	-1.8%	31.26	**0.64**	+3.2%	30.32	0.50	+0.1%
rif	13	10.66	9.60	**0.28**	-10.0%	11.10	0.59	+4.1%	9.44	**0.25**	-11.5%
templateit	19	18.48	31.14	**0.97**	+68.5%	19.05	0.56	+3.1%	30.66	**0.98**	+65.9%
jnfe	51	89.84	94.46	**0.90**	+5.1%	92.88	**0.64**	+3.3%	94.34	**0.89**	+5.0%
sfmis	19	30.58	35.02	**0.93**	+14.5%	31.24	0.59	+2.1%	35.08	**0.89**	+14.7%
gfarcegestionfa	48	53.70	51.10	**0.33**	-4.9%	51.66	0.41	-3.8%	50.60	**0.29**	-5.8%
falselight	8	1.52	1.42	0.45	-6.6%	1.58	0.53	+3.9%	1.42	0.45	-6.6%
water-simulator	48	45.74	43.10	**0.21**	-5.8%	45.88	0.52	+0.3%	43.46	**0.23**	-5.0%

With $\mu(P)$, we are actually calculating the average ratio of how many targets in total have been covered over the number of all possible targets. The statistics $avg(P)$ and $\mu(P)$ can lead to pretty different results. Considering the type of problem addressed in this paper, we argue that $\mu(P)$ is a more appropriate measure to analyze the data of our empirical analyses.

All the data from these empirical experiments have been statistically analysed following the guidelines in [1]. In particular, we used the Wilcoxon-Mann-Whitney U-test and the Vargha-Delaney $\hat{A}_{12}$ effect size. The Wilcoxon-Mann-Whitney U-test is used when algorithms (e.g., result data sets X and Y) are compared (in R this is done with $wilcox.test(X, Y)$). In our case, what is compared is the distribution of the values $\mu(P_i)$ for each project P. For the statistical tests, we consider a 95% confidence level.

Given a performance measure W (e.g., branch coverage), $\hat{A}_{xy}$ measures the probability that running algorithm x yields higher W values than running algorithm y. If the two algorithms are equivalent, then $\hat{A}_{xy} = 0.5$. This effect size is independent of the raw values of W, and it becomes a necessity when analyzing the data of large case studies involving artifacts with different difficulty and different orders of magnitude for W. E.g., $\hat{A}_{xy} = 0.7$ entails one would obtain better results 70% of the time with x.

Beside the standarized Vargha-Delaney $\hat{A}_{12}$ statistics, to provide more information we also considered the relative improvement ρ. Given two data sets X and Y, the relative average improvement will be defined as:

$$\rho(X, Y) = \frac{mean(X) - mean(Y)}{mean(Y)} .$$

5.2 Testing Whole Projects

The first set of experiments considers the effects of generating unit tests for whole projects. Table 1 shows the results of the experiments on the 10 open source projects randomly selected from SF100 corpus. The results in Table 1 are based on branch coverage. The *Simple* strategy is used as point of reference: the results on the other strategies (smart *Budget*, *Seeding* and their combination, *Budget & Seeding*) are presented relatively to the *Simple* strategy, on a per project basis. For each strategy compared to *Simple*, we report the $\hat{A}_{12}$ effect size, and also the relative improvement ρ.

Table 2 presents the results on the number of unique pairs exception/method for each CUT, grouped by project. For each run, we calculated the sum of all unique pairs on all CUTs in a project, and averaged these results over the 50 runs. In other words, Table 2 is structured in the same way as Table 1, with the only difference that the results are for found exceptions instead of branch coverage.

The results on the industrial experiments were analysed in the same way as the open source software results. Table 3 shows the results for branch coverage. However, due to confidentiality restrictions, no results on the `thrown` exceptions are reported.

The results in Table 1 clearly show that a smart *Budget* allocation significantly improves branch coverage. For example, for the project `sfmis` the branch coverage goes from 35.8% to 46.7% (a relative improvement of +30.6%). The $\hat{A}_{12} = 1$ means that, in *all* the 50 runs with smart *Budget* allocation the coverage was higher than in *all* the 50 runs with *Simple* strategy. However, there are two projects in which it seems it provides slightly worse results; in those cases, however, the results are not statistically significant.

The results in Table 2 are slightly different. Although the smart *Budget* allocation still provides significantly better results on a higher number of projects (statistically better in five out of 10; and equivalent results in two subjects), there are three cases in which results are statistically worse. In two of those latter cases, the branch coverage was statistically higher (Table 1), and our conjecture is that the way exceptions are included in EvoSuite's fit-

Table 3: Branch coverage results for the industrial case study.

For each project we report the branch coverage of the *Simple* strategy. For each of the other strategies, we report their branch coverage and the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *Simple* strategy. Results on the industrial case study are based on one single run.

Project	Classes	Simple	Budget			Seeding			Budget & Seeding		
		Coverage	Coverage	$\hat{A}_{12}$	Rel. Impr.	Coverage	$\hat{A}_{12}$	Rel. Impr.	Coverage	$\hat{A}_{12}$	Rel. Impr.
projectA	245	23.6%	28.3%	1.00	+19.8%	24.2%	1.00	+2.4%	28.8%	1.00	+21.9%
projectB	122	13.0%	21.9%	1.00	+67.6%	15.6%	1.00	+19.7%	21.2%	1.00	+62.2%
projectC	412	30.4%	41.3%	1.00	+35.8%	30.3%	0.00	-0.2%	41.5%	1.00	+36.4%
projectD	211	72.5%	87.9%	1.00	+21.2%	72.7%	1.00	+0.2%	86.0%	1.00	+18.5%
projectE	317	23.9%	28.5%	1.00	+19.0%	24.1%	1.00	+0.8%	28.8%	1.00	+20.1%

ness function (cf. [10]) means that test suites with higher coverage (as achieved by the *Budget* allocation) would be preferred over test suites with more exceptions. In this case, improving EvoSuite's optimization strategy (e.g., by using multi-objective optimization) may lead to better results with respect to both measurements. For the `rif` project (a framework for remote method invocation) the decrease in exceptions is not significant, but also the coverage is decreased insignificantly. In this case, it seems that the use of the number of branches is not a good proxy measurement of the test complexity. This suggests that further research on measurements other than branches as proxy for complexity would be important. On the other hand, we would like to highlight that for the `templateit` project the relative improvement was +68.5%.

RQ1: *Smart Budget allocation improves performance significantly in most of the cases.*

Regarding input *Seeding*, in Table 1 there is one case in which it gives statistically better results, but also one in which it gives statistically worse results. Regarding the number of `thrown` exceptions, there are two projects in which it gives statistically better results (Table 2). Unlike the *Budget* allocation, the usefulness of seeding will be highly dependent on the specific project under test. If there are many dependencies between classes and many branches depend on specific states of parameter objects, then *Seeding* is likely to achieve better results. If this is not the case, then the use of *Seeding* may adversely affect the search, e.g., by reducing the diversity, thus exhibiting lower overall coverage in some of the projects. However, note that the actual *Seeding* implemented in EvoSuite for these experiments is simplistic. Thus, a main conclusion from this result is that further research is necessary on *how* to best exploit this additional information during the search.

RQ2: *Input Seeding may improve performance, but there is a need for better seeding strategies to avoid negative effects.*

Finally, we analyse what happens when input *Seeding* is used together with the smart *Budget* allocation. For most projects, either performance improves by a little (compared to just using smart *Budget* allocation), or decreases by a little. Overall, when combined together, results are slightly worse than when just using the *Budget* allocation. This is in line with the conjecture that *Seeding* used naively can adversely affect results: Suppose that seeding on a particular class is bad (for example as is the case in the `a4j` project), then assigning significantly more time to such a class means that, compared to *Budget*, significantly more time will be wasted on misguided seeding attempts, and thus the relative performance will be worse. Note also that the overall result is strongly influenced by one particular project that is problematic for input *Seeding* (i.e., `a4j` with $\hat{A}_{12} = 0$ in Table 1). This further supports the need for smarter seeding strategies.

RQ3: *Seeding with Budget allocation improves performance, but Seeding strategies may negatively affect improvements achieved by Budget allocation.*

5.3 Continuous Test Generation

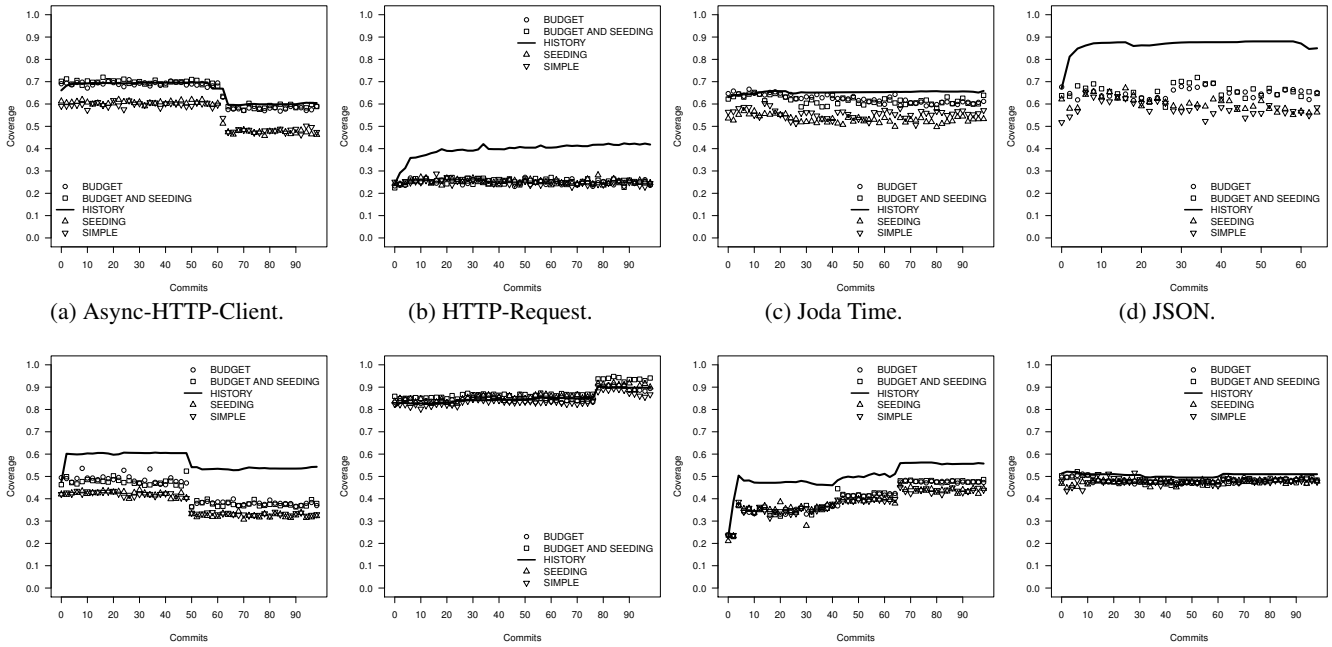
The second set of experiments considers the effects of CTG over time. Figure 1 plots the overall branch coverage achieved over the course of 100 commits. We denote the strategy that uses *Seeding* from previous test suites and allocation based on *History*. In most of the projects the higher coverage of the *History* strategy achieves clearly higher coverage, and this coverage gradually increases with each commit. The coverage increase is also confirmed when looking at the results throughout history; to this extent, Table 4 summarizes the results similarly to the previous experiment, and compares against the baseline strategies *Simple*, *Budget*, and *Budget & Seeding*. In all projects, the coverage was higher than using the *Simple* strategy (only on `SpringSide` is this result not significant). Compared to *Budget*, there is an increase in all projects (significant for four) but for `Scribe`, coverage is essentially the same. Compared to *Budget & Seeding*, there is a significant increase in five projects, and an insignificant increase in two projects. Interestingly, for the `Scribe` project *History* leads to significantly lower coverage (-2%) than *Budget & Seeding*. This shows that seeding of input values is very beneficial on `Scribe` (where 69% of the classes have dependencies on other classes in the project), and indeed on average the benefit of input *Seeding* is higher than the benefit of the *History* strategy. However, in principle *History* can also be combined with *Budget & Seeding*.

RQ4: *CTG achieves higher coverage than testing each project version individually, and coverage increases further over time.*

Figure 2 shows the time spent on test generation. Note that the strategies (*Simple*, *Seeding*, *Budget*, *Budget & Seeding*) were always configured to run with the same fixed amount of time. During the first call of CTG, the same amount of time was consumed for the *History* strategy, but during successive commits this time reduces gradually as fewer classes need further testing.

RQ5: *CTG reduces the time needed to maximise the code coverage of unit test suites for entire projects.*

Let us now look at some of the examples in more detail. `Async-HTTP-Client` exhibits two interesting events during its history (see Figure 1a): From the first commit until commit number 63 all strategies have a constant coverage value. At commit 64, 20 classes were changed and three new classes were added. Although this affected the coverage of *History* and also other strategies, *History* only increase its time for test generation briefly from 18 minutes at commit 63, to 30 minutes on commit 64, on average (compared to 219 minutes for a full test generation run). Figure 2a further shows a large increase of the test generation time at commit 93, although the coverage does not visibly change. In this commit, several classes were changed at that time, but only *cosmetic* changes happen to the source code (commit message "Format with 140 chars lines"). As EvoSuite apparently had already reached its maximum possible coverage on these classes, no further increase was achieved. We can observe



(a) Async-HTTP-Client. (b) HTTP-Request. (c) Joda Time. (d) JSON.

(e) JSoup. (f) Scribe. (g) Spark. (h) SpringSide.

Figure 1: Branch coverage results over the course of 100 commits for the GitHub open source case study.

Table 4: Coverage over time.

For each project we report the "time coverage": the average branch coverage over all classes in a project version, averaged over all 100 commits. These time coverages are averaged out of the five repeated experiments. We compare the "*History*" strategy with the "*Simple*", "*Budget*", and "*Budget & Seeding*" ones, and report the effect sizes ($\hat{A}_{12}$ over the five runs and relative average improvement). Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold. The number of classes is not a constant, as it can change at each revision. We hence report the total number of unique classes throughout the 100 commits, and, in brackets, the number of classes at the first and last commits.

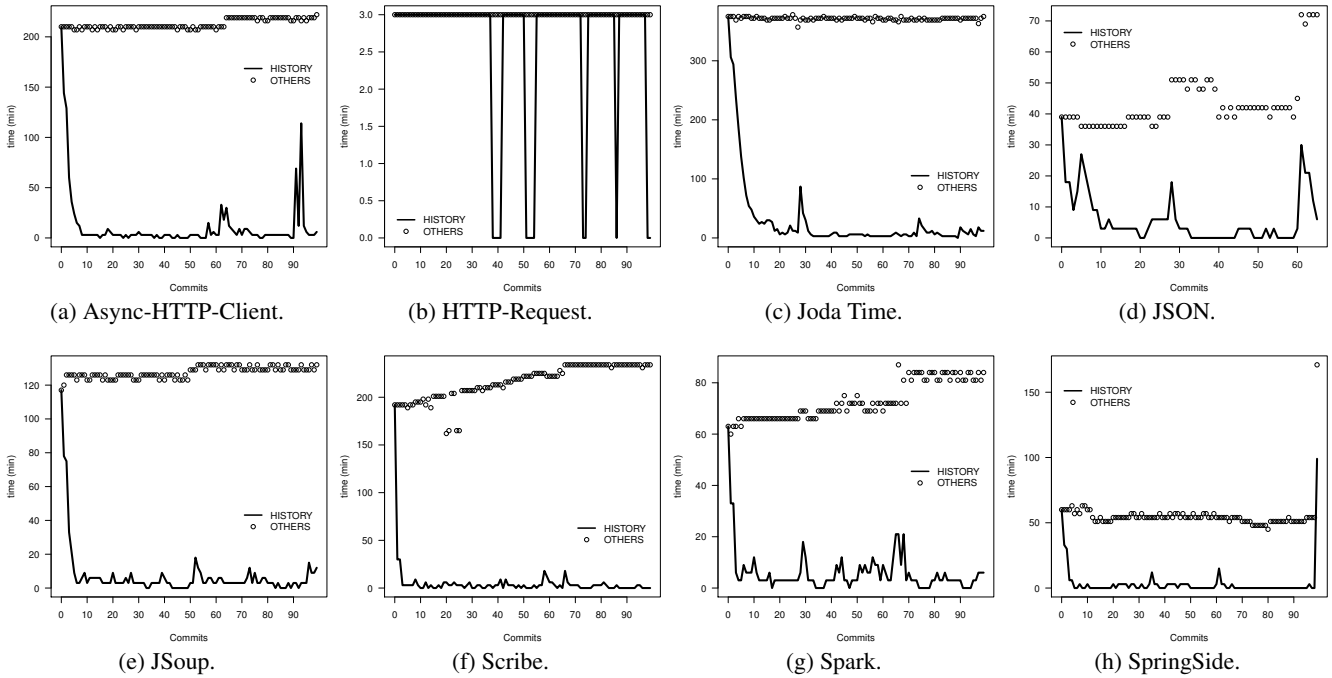
Project	Classes	Simple Coverage	Budget Coverage	Budget & Seeding Coverage	History Coverage	$\hat{A}_s$	Rel. Impr.	$\hat{A}_b$	Rel. Impr.	$\hat{A}_{b\&s}$	Rel. Impr.
HTTP-Request	1 [1:1]	0.25	0.24	0.25	0.39	**1.00**	+57.97%	**1.00**	+58.69%	**1.00**	+56.77%
JodaTime	135 [133:132]	0.55	0.62	0.61	0.65	**1.00**	+17.82%	0.80	+4.85%	**0.92**	+6.06%
JSon	37 [16:25]	0.58	0.64	0.65	0.86	**1.00**	+49.19%	**1.00**	+33.72%	**1.00**	+32.02%
JSoup	45 [41:45]	0.37	0.43	0.42	0.56	**1.00**	+51.18%	**1.00**	+31.74%	**1.00**	+33.73%
Scribe	79 [65:78]	0.83	0.85	0.87	0.85	**1.00**	+1.76%	0.48	+0.02%	**0.00**	-2.41%
Spark	34 [21:30]	0.38	0.40	0.40	0.50	**1.00**	+31.38%	**1.00**	+25.39%	**1.00**	+24.32%
Async-HTTP-Client	81 [71:75]	0.55	0.64	0.65	0.65	**1.00**	+18.90%	0.80	+1.32%	0.80	+0.14%
SpringSide	63 [23:60]	0.47	0.47	0.47	0.50	0.60	+5.90%	0.60	+5.62%	0.60	+6.13%

similar behaviour in the plots of `JSoup` (Figure 1e), where a major change occurred at commit 50 with the introduction of a new class (`org.jsoup.parser.TokeniserState`), which adds 774 new branches to the 2,594 previously existing branches.

The `HTTP-Request` subject reveals a nice increase over time, although the time plot (Figure 2b) shows only small improvement (13% less time in total). This is because this project consists only of a single class. Consequently, most commits will change that particular class, leading to it being tested more. In the commits where the class was not tested, no source code changes were performed (e.g., only test classes or other project files were changed, not source code). Thus, `HTTP-Request` is a good example to illustrate how using previous test suites for seeding gradually improves test suites over time, independently of the time spent on the class. Because this project has only one class, the *Seeding* strategy has similar results (on average) to the *Simple* strategy. A similar behaviour can also be observed for `JSON` (see Figure 1d), where *History* leads to a good increase in coverage over time. There is a slight bump in the coverage plot at commit 61 (Figure 2d), where 13 new classes were added to the project.

`JodaTime`, `Scribe`, and `SpringSide` are examples of projects with only a small increase in coverage (Figures 1c, 1f and 1h, respectively). Although these projects differ in size, it seems that their classes are all relatively easy for EvoSuite, such that additional time or the seeding has no further beneficial effect. For example, 72% of the classes in `SpringSide` have less than 17 branches. However, in all three cases the reduction in test generation time is very large (Figures 2c, 2f and 2h respectively).

Finally, `Spark` shows interesting behaviour where *all* approaches lead to increased coverage over the course of time (Figure 1g). This is because during the observed time window of 100 commits the project was heavily refactored. For example, some complex classes were converted into several simpler classes, increasing the time spent for non-History based strategies (Figure 2g), up to a maximum of 84 minutes on the last commit. This project also illustrates nicely why applying seeding blindly does not automatically lead to better results: For example, at commit 30 there are only 9 out of 25 classes that actually have dependencies, and many of the dependencies are on the class `ResponseWrapper` — which EvoSuite struggles to cover. As a consequence, there is no improvement when using seeding. This suggests that there is not a single optimal seeding strategy, but that seeding needs to take external factors such as dependencies and achieved coverage into account.



(a) Async-HTTP-Client. (b) HTTP-Request. (c) Joda Time. (d) JSON.

(e) JSoup. (f) Scribe. (g) Spark. (h) SpringSide.

Figure 2: Time spent on test generation for the GitHub open source case study over the course of 100 commits.

5.4 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. Furthermore, randomized algorithms are affected by chance. To cope with this problem, we repeated each experiment (50 times for the SF100 experiments and five times for the GitHub experiments) and followed rigorous statistical procedures to evaluate their results.

To cope with possible threats to *external validity*, the SF100 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge [8]. From SF100, 10 projects were randomly chosen. Although the use of SF100 provides high confidence in the possibility to generalize our results to other open source software as well, we also included on our experiments some of the most popular Java projects from GitHub.

Because open source software represents only one face of software development, in this paper we also used five industrial systems. However, the selection of those systems was constrained by the industrial partners we collaborate with. Results on these systems might not generalize to other industrial systems.

The techniques presented in this paper have been implemented in a prototype that is based on the EvoSuite tool, but any other tool that can automatically handle the subjects of our empirical study could be used. We chose EvoSuite because it is a fully automated tool, and recent competitions for JUnit generation tools [3, 11, 12] suggest that it represents the state of the art.

To allow reproducibility of the results (apart from the industrial case study), all 18 subjects and EvoSuite are freely available from our webpage at `www.evosuite.org`.

6. CONCLUSIONS

In this paper, the scope of unit test generation tools like EvoSuite is extended: Rather than testing classes in isolation, we consider whole projects in the context of continuous integration. This permits many possible optimizations, and our EvoSuite-based prototype provides CTG strategies targeted at exploiting complexity and/or dependencies among the classes in the same project. To validate these strategies, we carried out a rigorous evaluation on a range of different open source and industrial projects, totalling 2061 classes. The experiments overall confirm significant improvements on the test data generation: up to +58% for branch coverage and up to +69% for thrown undeclared exceptions, while reducing the time spent on test generation by up to +83%.

Our prototype at this point is only a proof of concept, and there remains much potential for further improvements:

- **Seeding** could be improved by making it more adaptive to the problem at hand, for example by using the abundant information made available through CTG.
- **Historical data** offers potential for optimizations, for example by using fault prediction models.
- **Different coverage criteria** could be used, for example starting with simpler criteria such as statement coverage, and slowly building up to more thorough criteria such as mutation testing or entropy [4].
- **Differential testing generation** could lead to better regression tests than using standard code coverage criteria.
- **Different testing scenarios** such as integration testing could benefit from CTG as well.

Although our immediate objective in our current experiments lies in improving the quality of generated test suites, we believe that the use of CTG could also have more far reaching implications. For example, regular runs of CTG will reveal testability problems in code, and may thus lead to improved code and design. The use of CTG offers great incentive to include assertions or code contracts, which would be automatically and regularly exercised.

Acknowledgements

This work is supported by a Google Focused Research Award on "Test Amplification", the EPSRC project "EXOGEN" (EP/K030353/1), the Norwegian Research Council, and the FCT project PTDC/EIA-CCO/116796/2010.

7. REFERENCES

[1] A. Arcuri and L. Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)*, 24(3):219–250, 2014.

[2] A. Arcuri, M. Z. Iqbal, and L. Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277, Mar. 2012.

[3] S. Bauersfeld, T. Vos, K. Lakhotia, S. Poulding, and N. Condori. Unit testing tool competition. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 414–420, March 2013.

[4] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-Based Test Generation for Improved Fault Localization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2013, pages 257–267, New York, NY, USA, 2013. ACM.

[5] C. Csallner and Y. Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[6] M. Fowler and M. Foemmel. Continuous Integration. *(Thought-Works)* `http://www.thoughtworks.com/ContinuousIntegration.pdf`, 2006.

[7] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *ACM SIGSOFT European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[8] G. Fraser and A. Arcuri. Sound Empirical Evidence in Software Testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, ICSE '12, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[9] G. Fraser and A. Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, ICST '12, pages 121–130, Washington, DC, USA, 2012. IEEE Computer Society.

[10] G. Fraser and A. Arcuri. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Software Engineering (EMSE)*, pages 1–29, 2013.

[11] G. Fraser and A. Arcuri. EvoSuite at the SBST 2013 Tool Competition. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, ICSTW '13, pages 406–409, Washington, DC, USA, 2013. IEEE Computer Society.

[12] G. Fraser and A. Arcuri. EvoSuite at the Second Unit Testing Tool Competition. In *Fittest Workshop*, 2013.

[13] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, Feb. 2013.

[14] G. Fraser and A. Arcuri. Achieving Scalable Mutation-based Generation of Whole Test Suites. *Empirical Software Engineering (EMSE)*, pages 1–30, 2014.

[15] G. Fraser and A. Zeller. Mutation-driven Generation of Unit Tests and Oracles. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[16] J. Galeotti, G. Fraser, and A. Arcuri. Improving Search-based Test Suite Generation with Dynamic Symbolic Execution. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE '13, pages 360–369, Nov 2013.

[17] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing Non-adequate Test Suites Using Coverage Criteria. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[19] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering (TSE)*, 26(7):653–661, July 2000.

[20] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu. Automatic Testing of Object-Oriented Software. In *Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 114–129, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[22] C. D. Nguyen, A. Perini, P. Tonella, and F. B. Kessler. Automated Continuous Testing of MultiAgent Systems. In *Fifth European Workshop on Multi-Agent Systems (EUMAS)*, 2007.

[23] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

[24] A. Orso and T. Xie. BERT: BEhavioral Regression Testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, WODA '08, pages 36–42, New York, NY, USA, 2008. ACM.

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[26] L. S. Pinto, S. Sinha, and A. Orso. Understanding Myths and Realities of Test-suite Evolution. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, FSE '12, pages 33:1–33:11, New York, NY, USA, 2012. ACM.

[27] D. Saff and M. D. Ernst. Reducing Wasted Development Time via Continuous Testing. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE '03, pages 281–, Washington, DC, USA, 2003. IEEE Computer Society.

[28] D. Saff and M. D. Ernst. An Experimental Evaluation of Continuous Testing During Development. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA '04, pages 76–85, New York, NY, USA, 2004. ACM.

[29] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-Suite Augmentation for Evolving Software. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '08, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[30] S. Shamshiri, G. Fraser, P. Mcminn, and A. Orso.

Search-Based Propagation of Regression Faults in Automated Regression Testing. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, ICSTW '13, pages 396–399, Washington, DC, USA, 2013. IEEE Computer Society.

[31] K. Taneja and T. Xie. DiffGen: Automated Regression Unit-Test Generation. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '08, pages 407–410, Washington, DC, USA, 2008. IEEE Computer Society.

[32] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. DyGen: Automatic Generation of High-coverage Tests via Mining Gigabytes of Dynamic Traces. In *International Conference on Tests and Proofs*, TAP '10, pages 77–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[33] N. Tillmann and J. De Halleux. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] P. Tonella. Evolutionary Testing of Classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[35] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel. Continuous Test Suite Augmentation in Software Product Lines. In *International Software Product Line Conference*, SPLC '13, pages 52–61, New York, NY, USA, 2013. ACM.

[36] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A Hybrid Directed Test Suite Augmentation Technique. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, ISSRE '11, pages 150–159, Washington, DC, USA, 2011. IEEE Computer Society.

[37] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed Test Suite Augmentation: Techniques and Tradeoffs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, FSE '10, pages 257–266, New York, NY, USA, 2010. ACM.

[38] S. Yoo and M. Harman. Test Data Regeneration: Generating New Test Data from Existing Test Data. *Software Testing, Verification and Reliability (STVR)*, 22(3):171–201, May 2012.