Journal of Computer Languages 78 (2024) 101255

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

A test amplification bot for PHARO/SMALLTALK

Mehdrad Abdi [a,*], Henrique Rocha [b], Alexandre Bergel [c], Serge Demeyer [d]

[a] Nokia Bell NV, Antwerp, Belgium
[b] Loyola University Maryland, Baltimore, MD, USA
[c] RelationalAI, Switzerland
[d] University of Antwerp, Antwerp, Belgium

ARTICLE INFO

Dataset link: https://github.com/hscrocha/TestAmplificationBotReproductionPackage

Keywords:
Test amplification
Mutation testing
Continuous integration
Crash recovery
Pharo smalltalk

ABSTRACT

Test amplification exploits the knowledge embedded in an existing test suite to strengthen it. A typical test amplification technique transforms the initial tests into additional test methods that increase the mutation coverage. Although past research demonstrated the benefits, additional steps need to be taken to incorporate test amplifiers in the everyday workflow of developers. This paper describes a proof-of-concept bot integrating SMALL-AMP with GITHUB-ACTIONS. The bot decides for itself which tests to amplify and does so within a limited time budget. To integrate the bot into the GITHUB-ACTIONS workflow, we incorporate three special-purpose features: (i) prioritization (to fit the process within a given time budget), (ii) sharding (to split lengthy tests into smaller chunks), and (iii) sandboxing (to make the amplifier crash-resilient). We evaluate our approach by installing the proof-of-concept extension of SMALL-AMP on five open-source projects deployed on GitHub. Our results show that a test amplification bot is feasible at a project level by integrating it into the build system. Moreover, we quantify the impact of prioritization, sharding, and sandboxing so that other test amplifiers may benefit from these special-purpose features. Our proof-of-concept demonstrates that the entry barrier for adopting test amplification can be significantly lowered.

1. Introduction

Unit testing is writing small pieces of executable code to exercise the program's units and ensure they work as intended. Even though writing these unit tests is initially a tedious process, it prevents the system under test from regressing in the long term. A common way to evaluate the strength of a test suite is to measure code coverage or mutation coverage [1]. Since manually covering all corner cases of a program is a challenging task, automated test generation [2–4] and test amplification [5–11] tools were investigated to create stronger test suites. These tools analyze the program under test and produce new test methods that permanently increase coverage if merged into the codebase.

SMALL-AMP [12] is the state-of-the-art test amplification tool in the PHARO/SMALLTALK ecosystem. It extends DSPOT (see [5]) by bringing test amplification to dynamically typed languages. Both tools work as a recommender system that synthesizes new test methods and presents them to developers, which then decide whether these tests are worthwhile to be merged into the codebase. Qualitative studies on DSPOT and SMALL-AMP illustrate that developers value the generated tests and accept the corresponding pull requests [6,12].

Despite the promising results of test amplification tools, their practical application is still questionable. Past research shows that test amplification tools are cumbersome [13]. Not only are they complex and hard to configure, but their execution time is unpredictable and sometimes even unacceptable. For instance, considering test amplifiers employing mutation testing, the amplification of some test classes requires 5+ hours in DSPOT [6], 5+ hours in DCI [7], 2+ hours in SMALL-AMP [12], and 3+ hours in AMPYFIER [11].

⇒ *Although test amplification tools emerged to support developers, the complexity and long execution times hinder their adoption.*

Brandt and Zaidman [14] employ a lighter version of DSPOT in an IDE and introduce developer-centric test amplification. Because of the time cost consideration, they restrict test amplification to increase the instruction coverage and skip amplifying the mutation coverage. This confirms that developers' workstations are unsuitable for comprehensive mutation-based test generation: it is impossible to provide instantaneous feedback.

⇒ *Executing mutation-based test amplifiers on a developer workstation is seldom feasible due to the computational overhead.*

* Corresponding author.
E-mail addresses: newmrd@gmail.com (M. Abdi), henrique.rocha@gmail.com (H. Rocha), alexandre.bergel@me.com (A. Bergel), serge.demeyer@uantwerpen.be (S. Demeyer).

https://doi.org/10.1016/j.cola.2023.101255
Received 11 September 2023; Received in revised form 20 November 2023; Accepted 4 December 2023
Available online 11 December 2023

In contrast, the work by Campos et al. [15] and Danglot et al. [7] employ *continuous integration servers* to exploit automated tests. The former integrates EvoSuite [2] (a test generation tool for Java) within a continuous integration setting to optimize the test generation. The latter runs a variation of DSpot (named DCI) to detect the behavioral changes on each commit in continuous integration.

⇒ *Continuous integration servers, running on powerful servers configured in build farms, open up possibilities for improved test synthesis.*

A long-term possibility is to delegate the tedious task to a *software engineering bot*. Following this vision, the bot autonomously amplifies selected tests, integrates the synthesized tests into a separate branch, and, upon successful completion, opens a pull request to strengthen the test suite on the main branch. This entire process requires no manual intervention from a software engineer. Subsequent code reviews treat the pull request with the strengthened test suite just like any other submitted by a human team member.

⇒ *The ultimate vision for test synthesis in the context of* GitHub-Actions *is a test amplification bot that opens pull requests for autonomously strengthening the test suite.*

One issue preventing fully autonomous test synthesis is that mutations in the code may result in system crashes [10]. Especially in live systems such as Pharo, system crashes corrupt the system image beyond repair. If a crash happens, the system must revert back to a state where the system is known to be pure.

⇒ *A crash-resilient test synthesis process is a necessary prerequisite for a test amplification bot.*

In this paper, we explore the feasibility of a test amplification bot. We present a proof-of-concept tool that integrates Small-Amp with GitHub-Actions to automatically strengthen the existing test suite within a limited time budget. To this end, our proof-of-concept incorporates three special-purpose features:

- (i) prioritization (to fit the process within a given time budget),
- (ii) sharding (to split lengthy tests into smaller chunks),
- (iii) and sandboxing (to make the amplifier crash-resilient).

We evaluated our approach by installing the proof-of-concept extension of Small-Amp on five open-source Pharo projects deployed on GitHub. Our results show that autonomous test amplification is feasible at a project level by integrating it into the build system. Moreover, we quantify the impact of sharding, prioritization, and sandboxing so that other test amplifiers may benefit from these special-purpose features. Our experiments show that prioritization has better performance (up to a 34% increase), crashes occurred in about 17% of the cases, and are restored successfully by the sandboxing mechanism, and sharding allowed for large classes to fit into our time budget but came at a cost of 30% more duplicated mutants.[1] Additionally, our new time budget-aware process was able to finish the amplification in an acceptable period of 30 to 90 min.

The remainder of the paper is organized as follows. Section 2 provides the necessary background to understand the challenges of test amplification bots. Section 3 explains how we integrate Small-Amp with GitHub-Actions, and details the sharding, prioritization, and crash recovery features. Section 4 presents the quantitative results of the evaluation of five projects. Section 5 enumerates the threats to validity. Section 6 provides an overview of the related work that inspired this proof-of-concept. Finally, we summarize the main conclusions in Section 7.

[1] Duplicated mutants refer to identical instances of simulated code errors introduced more than once during the testing process, potentially leading to redundancy in assessing the effectiveness of a test suite.

2. Test amplification

Modern software repositories contain a considerable amount of tests. These tests are written mostly by developers who have deep knowledge and understanding of the program. The main idea in *test amplification* [16] is exploiting this valuable resource of knowledge to improve the test suite.

In Small-Amp [12], the state-of-the-art test amplifier in the Pharo ecosystem, this improvement is achieved by synthesizing new test methods that permanently increase the mutation coverage when merged into the codebase. Small-Amp is a replication of DSpot [6] in the dynamic language of Pharo [17,18].

2.1. Amplification algorithm

Small-Amp (as well as DSpot) iterates all test methods in a test class and applies the following operations: input amplification, assertion amplification, and selection by mutation score. To illustrate these operations, we are going to use a test for a Circular Queue (Listing 1).

```
CircularQueueTest >> testEnqueueDequeue
 | cq |
 cq := CircularQueue new.
 cq enqueue: 11.
 cq enqueue: 22.
 cq enqueue: 33.
 self assert: (cq size) equals: 3.
 self assert: (cq dequeue) equals: 11.
 self assert: (cq dequeue) equals: 22.
```

Listing 1: Circular Queue Test Example

- *Input amplification* transforms the original test method using a set of input amplifiers to generate new versions of the test method. Usually, some of these transformed versions of the test method bring the program under test to an untested state or take a different execution path from the original test method. However, the original test method usually contains some assertion statements to verify the intended state. Since these assertion statements are no longer valid in the transformed versions, Small-Amp removes the original statements before the transformation. Listing 2 shows one example after our original test method (Listing 1) goes through input amplification. More specifically, the assertions were removed from the original test, the called methods and constants changed, and in this case, a new temporary variable was used.

```
CircularQueueTest >> testEnqueueDequeue_amp_A1
 | cq tmp_Ne5QRJZLBV1 |
 cq := CircularQueue new.
 cq enqueue: 22.
 cq enqueue: 0.
 tmp_Ne5QRJZLBV1 := cq dequeue.
 cq enqueue: 11.
```

Listing 2: Circular Queue Test Example

- *Assertion amplification* regenerates appropriate assertions to verify the actual state of the program by manipulating the generated test method and inserting observer statements. In this step, the test method is executed, and Small-Amp extracts the actual state of the program using object inspection. Small-Amp generates new assertion statements using these extracted states and adds them in place of observer statements. The new assertions should all pass for the version of the code they were generated for. Listing 3 shows the previous test method that was input amplified (Listing 2) after it has new assertions generated by the Assertion amplification operation. We can see that Small-Amp creates many new assertions and uses new methods in such assertions (in this case,`class`, `isEmpty`, and `isFull`).

```
CircularQueueTest >> testEnqueueDequeue_amp_A1
 | cq tmp_Ne5QRJZLBV1 |
 cq := CircularQueue new.
 self assert: cq class equals: CircularQueue.
 self deny: cq isFull.
 self assert: cq isEmpty.
 self assert: cq size equals: 0.
 cq enqueue: 22.
 self deny: cq isFull.
 self deny: cq isEmpty.
 self assert: cq size equals: 1.
 cq enqueue: 0.
 self deny: cq isFull.
 self deny: cq isEmpty.
 self assert: cq size equals: 2.
 tmp_Ne5QRJZLBV1 := cq dequeue.
 self deny: cq isFull.
 self deny: cq isEmpty.
 self assert: cq size equals: 1.
 self assert: tmp_Ne5QRJZLBV1 equals: 22.
 cq enqueue: 11.
 self deny: cq isFull.
 self deny: cq isEmpty.
 self assert: cq size equals: 2
```

Listing 3: Circular Queue Test Example

- *Selection by mutation score.* Up to this step, we have new versions of the original test method that were transformed by input amplifier and equipped with new assertion statements by assertion amplifier. In this step, mutation testing is run on the program under test using these generated test methods. Test methods that increase the mutation coverage by killing new mutants are kept, and the remaining test methods are discarded. Small-Amp relies on Mutalk [19], a test amplification platform for Pharo.

More details and examples on the Test Amplification Algorithm are presented in our previous work [12] and the first author's Ph.D. thesis [20].

2.2. Challenges for test amplification

The previous section gives the result of a simple example of test amplification. However, in practice, producing amplified tests is a long and laborious activity. In this section, we identify the main challenges faced by test amplification tools, Small-Amp in particular, to be more practical and incorporate them into the daily workflow of developers.

- *Using test amplification tools is cumbersome.* In addition to writing code and tests, developers are usually busy with other activities like meetings, bug fixing, emails, networking, learning, documentation, helping others, administration tasks, and others [21–23]. Test amplification tools are complicated and hard to configure, and using them requires deep knowledge about different topics like mutation testing [13]. If we expect developers to run the tool on their workstations, each developer would need to deal with some extra tedious tasks.
- *Current test amplification tools do not support time budget management.* Test amplification execution time varies from test class to test class, and estimating it in advance is difficult; amplification tools usually have long execution times and need considerable processing resources. It is inconvenient for developers to employ these tools in their workstations, dedicating the entirety of their resources to test amplification and waiting hours or days for a test amplification run to completion. Setting a time limit is necessary for such a long process. On the other hand, the current test amplification tools lack a mechanism to prioritize their tasks to gain the maximum benefit when running on a time budget.
- *Test amplification in live systems is more challenging.* Small-Amp amplifies programs written in Pharo/Smalltalk which is a so-called *live* programming environment [24]. Pharo offers the notion of *liveness* which greatly impacts how developers work [25]. The system always offers an accessible evaluation of a source code instead of the classical edit-compile-run cycle, and as a consequence, the live programming environment allows for nearly instantaneous feedback to developers instead of forcing them to wait for the program to recompile [26].
  Ducasse et al. [27] identify the challenges of supporting automated testing tools in Pharo, and they mention executing destructive methods in random testing as a challenge and emphasize the need for sandboxing. During the amplification process, Small-Amp works with two different kinds of mutations: the mutation on the production code (mutation testing), and the mutation on test methods (input amplification); and each mutation applies a random change to the code. Executing such random code in a live system introduces two major challenges:
  - *Random code easily leads to infinite loops/recursions and deadlocks.* Worse, it is possible to call critical methods (terminating the virtual machine and unloading class), leaving the live system in an unsafe state. Consequently, an image crash or freeze is more prone to happen during amplification [10]. In a live system, a crash means we no longer have the amplification state in which the previously amplified results were stored.
  - *Random code may pollute the internal state of a system, resulting in flaky tests [28,29].* For example, suppose an object is cached in a class variable (a static variable in Java parlance). Developers expect this cached value to be immutable, but it may be altered unexpectedly during mutation testing. As a result, all tests depending on this cached value would fail after the mutation testing while passing before. This pollution would remain in the live system forever and may cause side effects on the generated tests.

⇒ **A Test Amplification Bot** may alleviate these challenges.

- Instead of asking developers to run a completely configured tool on a desktop computer, we can embed the tool in a fully autonomous process on the continuous integration servers.
- Instead of running the tool until completion, no matter how long it takes, we can change the base algorithm to run in a given time budget and optimize accordingly.
- Instead of restarting the process after a crash in a fresh unpolluted state, we can run the tool in a sandbox environment to be able to circumvent (even reproduce) the crash.

3. Design of the proof-of-concept

In this paper, we explore the feasibility of a test amplification bot. We present a proof-of-concept tool that integrates Small-Amp with GitHub-Actions to fully automatically strengthen the existing test suite within a limited time budget.

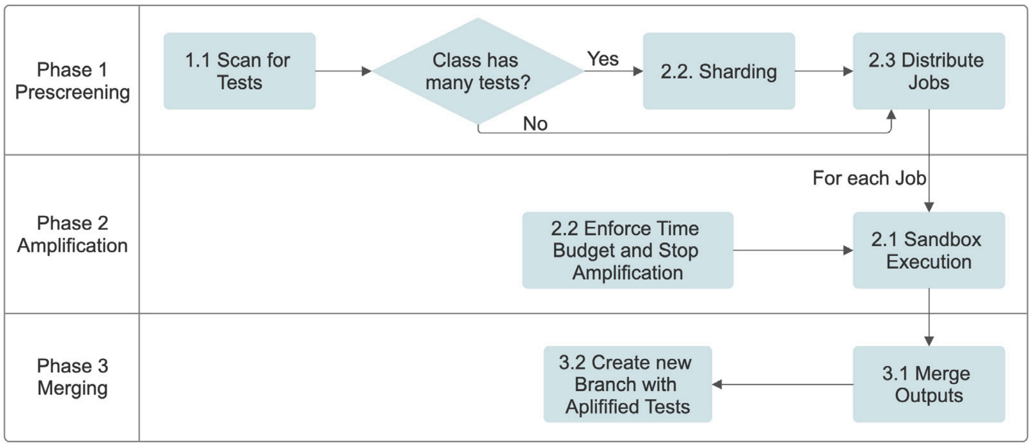

Fig. 1. Diagram showing the phases and main processes for the GitHub-Actions integration.

3.1. Small-Amp *and Pharo*

Why small-amp? In principle, we could have chosen any test amplification tool for our proof-of-concept. We decided to focus on tools used in dynamic languages given the popularity of such languages among practitioners. For instance, JavaScript was the most popular language on StackOverflow.[2] At the time of the research, we were not aware of any test amplification tool for JavaScript. Therefore, we chose SmallAmp [12] because it was a recent tool for the dynamic language Pharo Smalltalk. Moreover, Pharo presents more challenges due to its live programming environment which we deemed interesting to investigate.

What is interesting about Pharo? Pharo is a Smalltalk-based object-oriented dynamic-typed language. Pharo also includes a programming environment, integrated with development tools, a run-time virtual machine, and live debugging features. Pharo is not “file-based” as programmers work directly in an Image which is a live environment that stores the code, the states manipulated by the code, and the current execution [18].

As a simple analogy, we can think of the Pharo image as an Operating System and IDE rolled into one container that becomes a live programming environment. This *liveness* offers more challenges for test amplification (as we previously explained in Section 2.2) which we consider interesting to explore in this research. Moreover, if our proof of concept works for a more challenging scenario, it will be possible to adapt it to simpler situations.

3.2. *Integration with* GitHub-Actions

Why github-actions? We adopted the GitHub-Actions build system as a suitable platform to build a proof of concept automated test amplification tool for several reasons. (1) A build system can be configured once and used by all contributors in a project or even multiple projects. (2) A build system can trigger the test amplification based on relevant events like each pull request, scheduled like running per week, or manually when needed. (3) A build system executes on the Continuous Integration Servers, freeing developers’ machines from the computation. (4) GitHub-Actions defines a language for defining workflows and which allows for parallelization. (5) Build system has become more popular in recent years [30–32]. (6) Most well-known open-source Pharo projects are hosted on GitHub, and GitHub-Actions is freely available for open-source projects [33].

How does github-actions *work?* GitHub-Actions is based on workflows, and each workflow contains one or more jobs that can be run in parallel or sequential. Each job starts a new operating system instance in a virtual machine or container and performs some steps. Each step may run a terminal command or use a private or public custom action [34]. Workflows can be triggered by predefined events like when a new code is pushed, merged, or based on a schedule. By default, the return value from a workflow run is only the state of success or failure. However, GitHub-Actions supports creating artifacts to persist additional data [35]. GitHub-Actions also allows defining reusable workflows [36], which facilitate workflow maintenance on the users’ side.

Small-amp *integration to* github-actions. For integrating Small-Amp, we define a reusable workflow,[3] and also a GitHub-Actions custom action[4] to setup a Pharo instance and run Small-Amp in it. Developers in user projects need to define a workflow that calls the reusable workflow and pass some main configuration parameters. Some of the essential parameters required to be configured by the users are the number of parallel jobs and the project loading parameters. If the workflow is triggered by a push or pull request, the test amplification tool considers all changes in the commit; but if triggered manually or by schedule, it amplifies the entire project or the specified classes.

The workflow contains three sequential phases (Fig. 1). Each phase is composed of a job or a set of similar jobs that run in parallel. Since each job starts on a clean operating system, Pharo is installed first, and then Small-Amp and the project-under-test are loaded in Pharo.

The first phase is *prescreening*, which consists of a single job with the following steps:

- 1.1 Small-Amp scans all defined test classes in the project and attempts to detect the class under test by its default heuristic. (Details are in [12]).
- 1.2 If a class contains too many test methods, the test optimization will perform poorly. Small-Amp, therefore, shuffles its test methods and breaks them into smaller temporary test classes.
- 1.3 Small-Amp assigns test classes to different job identifiers to be distributed over jobs in the next phase.

[2] https://survey.stackoverflow.co/2022/#technology-most-popular-technologies.

[3] http://github.com/mabdi/small-amp/blob/master/.github/workflows/SmallAmpCI.yml.

[4] http://github.com/mabdi/smallamp-action.

The second phase is *amplification*, which consists of multiple parallel jobs. Each job iterates over its assigned test classes and performs the following steps:

- 2.1 It creates a sandbox for each test class. The amplification tool is executed within a sandbox to make it *crash resilient* (see Section 3.5).
- 2.2 It enforces a maximum time budget for each test class (like 15 min) to ensure that the amplification terminates in a predictable time (see Section 3.3).

The final phase is *merging*, in which a single job collects all output files from the amplification jobs, merges them, and exports the tool's outputs as *artifacts*. All amplified tests are committed into a new git branch, the *amplification branch*. The new branch is pushed onto the repository and a pull request is sent from the amplification branch to the main branch. In order to facilitate code reviewing and individual selection from the pull request (cherry-picking), each commit only includes a single test method. Developers can load the code submitted in the amplification branch into their IDE and use debuggers and editors to polish the tests. GitHub's web interface can be used for quick corrections.

In that sense, the proof-of-concept decides for itself which tests to amplify, incorporates the synthesized tests in a separate branch, executes the strengthened test suite and —if all steps pass— opens a pull request with the strengthened test suite onto the main branch. All this happens without any intervention of a software engineer. Subsequent code reviewing then needs to handle the pull request with the strengthened test suite just like it would handle a pull request by a human team member.

3.3. Test-method prioritization

Since the execution time of the test amplification tool varies from test class to test class, we set a time limit on each amplification process to make it more practical. The test amplification algorithm introduced in DSpot and Small-Amp does not provide time budget management. In this section, we extend the Small-Amp algorithm by proposing a test-method prioritization heuristic to increase the algorithm's efficiency when executed within a limited time budget.

This heuristic is based on the two intuitions (i) a test method covering more live mutants has a better chance of killing them; (ii) a test method killing mutants in the immediate method under test has a better focus. The first intuition is inspired by the reach–infect–propagate–reveal principle: a good test must reach the mutant before it can kill it [37]. The second intuition is based on the idea of a *focal method under test*, the fine-grained primary target exercised by a unit test [38]. Therefore, we first count the number of live mutants in all covered methods by a test and compute a mutant coverage score for each test method. Next, we filter out the immediate mutants, i.e. mutants injected in methods one step away in the call graph.

Finally, based on these scores, we calculate a weight for each test method and select one of them using the roulette wheel method [39]. We select an individual randomly in a roulette wheel selection, but the probability of this selection corresponds to its weight. The benefit of using this selection mechanism is to increase diversity in the selected methods by giving a chance to less-favored test methods of being selected.

Setting scores and weights. We suppose that we have a function μ that returns the number of live mutants (a_k) in each method under test (m_k):

$$\mu = \{m_1 \mapsto a_1, m_2 \mapsto a_2, \dots, m_n \mapsto a_n\}$$

In addition, we have a directed graph $G = (V, E)$ for the method invocations. The vertices correspond to all test methods (T) and methods under test (M). There is also a directed edge from node v to node v' if v invokes v'.

$$V = T \cup M$$
$$E = \{v \to v' | v, v' \in V \wedge v' \text{ is invoked from } v\}$$

We define the coverage set of the test method t as the set of all methods under test covered by t:

$$C_t = \{m | m \in M \wedge \exists\ p = (t \to \cdots \to m) \in \mathcal{P}(G)\}$$

In this relation, $\mathcal{P}(G)$ is the set of all paths in the graph G, and p is a path starting from t and ending in m. Similarly, we define the immediate coverage set (path length is 1) as:

$$I_t = \{m | m \in M \wedge \exists\ p = (t \to m) \in \mathcal{P}(G)\}$$

Now, the scoring function s is:

$$s(t) = \alpha + \beta \sum_{m \in I_t} \mu(m) + \gamma \sum_{m \in C_t - I_t} \mu(m)$$

The first part of this equation is the scoring offset. If $\alpha = 0$, all test methods not covering any mutant are excluded from the amplification process. The second part of the equation is the immediate coverage score. As a result, we expect the mutants in these methods to be killed faster than deeper mutants. The third part of the equation is the coverage score. In this part, we consider all remaining mutants in other covered methods.

The variables α, β and γ are tuning parameters: For a default value, we choose $\alpha = 1$ to prevent excluding the test methods with no mutant coverage because these tests may be able to kill new mutants after some transformations. Since we prefer to prioritize the mutants in instantly covered methods, so we choose $\beta = 3, \gamma = 1$.

After calculating the score for each test method, we set a weight for each test method as:

$$w(t) = \frac{s(t)}{S}\ ;\text{ where } S = \sum_{t \in T} s(t)$$

These weights drive the roulette wheel selection of the test method to be amplified. The scores and weights need to be updated in each cycle because the number of live mutants in the methods changes after each test amplification loop. Recalculating the weights does not have much overhead because the coverage graph does not need to be regenerated each time. We only need to update the μ function and recalculate $s(t)$ and $w(t)$ for all remaining tests.

To summarize, our prioritization assigns weights to test methods. The methods scoring higher weights will be favored when the test amplification selects a test method to amplify.

Changes in the algorithm. First of all, we update the *input amplification* and *assertion amplification* steps in Small-Amp to make them time budget aware: If the time limit is attained, new test inputs are not input/assertion amplified, and all the currently amplified instances are returned. We also added a test method selection based on the weight assignment heuristic, and the roulette wheel method described earlier in this section. We present the updated time budget aware algorithm in Algorithm 1 (main changes highlighted in blue).

In the new algorithm, initially, we run mutation testing to calculate the live mutants (ALV). Then, we execute assertion amplification on all test methods to kill those mutants that can be killed only by expanding the assertion statements. Since a single assertion amplification is faster than the combination of input amplification and assertion amplification, this step does not need any selection based on the scores. We remove the newly killed mutants from ALV and select a random test to be amplified using the roulette wheel method (line 5). The main amplification loop runs on the selected test method t (lines 8 to 13). Then, the method is removed from the list of all test methods to be amplified T (line 14), and the weights are recalculated based on the current live mutants. Then, a random test from the remaining

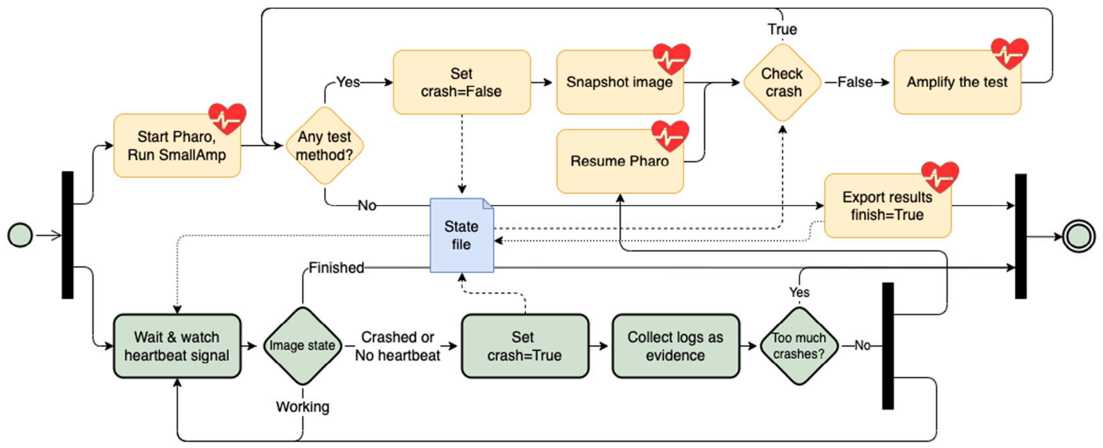

Fig. 2. Activity diagram for a self-aware test amplification in a live system. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```
input  : class-under-test CUT
input  : set of test methods T
input  : hyperparameters {N_iteration}
output: set of amplified test methods ATM
1  $ALV \leftarrow$ mutationTesting($CUT, T$);
2  $U \leftarrow$ amplifyAssertions($T$);
3  $ATM \leftarrow \{x \in U|$ x improves mutation score$\}$;
4  $ALV \leftarrow ALV - \{x \in ALV|$ x is killed in ATM$\}$;
5  $t \leftarrow$ rouletteWheel($CUT, T, ALV$);
6  while $t \neq null$ do
7      $V = \{t\}$;
8      for $i \leftarrow 0$ to $N_{iteration}$ do
9          $V \leftarrow$ amplifyInputs($V$);
10         $U \leftarrow$ amplifyAssertions($V$);
11         $ATM \leftarrow ATM \cup \{x \in U|$ x improves mutation score$\}$;
12         $ALV \leftarrow ALV - \{x \in ALV|$ x is killed in ATM$\}$;
13     end
14     $T \leftarrow T - \{t\}$;
15     $t \leftarrow$ rouletteWheel($CUT, T, ALV$);
16 end
17 return $ATM$
```

Algorithm 1: Updates in amplification algorithm to support time budget management

unamplified tests is selected considering their weight (line 15). If all test methods are visited or the time budget is due, the roulette wheel returns a null value, and then the final amplified test methods (ATM) are returned.

For identifying pollution (Section 2.2), we run the test class after the early mutation testing (Algorithm 1 line 1). If the test is green, we assume the state is not polluted and continue the algorithm.

3.4. *Sharding*

While experimenting on real projects with the time budgets, we witnessed that the number of test methods skipped in some classes was unacceptable because they include tens (sometimes hundreds) of test methods. Methods were skipped due to a consequence of our roulette wheel selection, which favors methods with higher priority. Using a variable time budget based on the test method numbers does not solve the problem entirely because amplifying some of these large classes may need more time than the jobs' allowed time (6 h in GitHub-Actions). Therefore, as a solution to decrease the number of skipped methods, we split long test classes into smaller temporary test classes. We call this action *Sharding*.

We use a threshold (default 15 test methods) as a sharding factor. We chose 15 based on our experience, considering it a suitable threshold for the analyzed projects. If a class contains more test methods than the defined threshold, Small-Amp shuffles its test methods and distributes them into smaller temporary test classes. In parallel jobs, all jobs must produce the same shards, so a shared randomization seed is required. We derive this seed from the *workflow run id* in our proof-of-concept.

Predicting the number of jobs. The following formula estimates the minimum number of parallel jobs required for a successful test amplification:

$$J_{min} = \lceil \frac{b \times \sum_{c \in TC} \lceil \frac{t_c}{S} \rceil}{M} \rceil$$

where J_{min} is the minimum number of jobs required, b is the time budget per class, M is the maximum allowed execution time for each job by platform, t_c is the number of tests in the class c, S is the sharding factor, and finally, TC is the set of test classes to be amplified.

Therefore, the expression $\sum_{c \in TC} \lceil \frac{t_c}{S} \rceil$ shows the number of classes after sharding. We expect that all shards finish in the defined budget (b), so the fraction's numerator calculates the minimum time needed to amplify all tests in a single job. However, we suppose that the build system defines an execution limit on each job (M). Dividing the time needed to amplify all classes by the maximum job execution achieves the minimum number of required jobs.

3.5. *Crash resilience*

Small-Amp is designed to run within a Pharo image, representing the complete state of the live system. Besides the system under test, the tool and its necessary components (the compiler, test runner, mutation testing framework, ...) run in the same Pharo image and therefore the same memory space. This architecture introduces a serious risk: if a crash happens, the whole Pharo process is lost, including the crashed component as well as the Small-Amp core.

A reliable test amplification tool running in a live environment, like Pharo, should be able to recover from these crashes without losing the entire state of the amplification process. Without a crash recovery mechanism, integration into build systems is impossible because any crash in Small-Amp will fail the entire workflow.

We enumerate some common reasons for an unexpected termination:

- *Killed by the operating system.* The Operating System may kill the Pharo process with an *Out of memory* error. This issue commonly happens in the mutation testing step when the process creates infinite recursion because of the injected fault.
- *Pharo process crashes.* The Pharo process is terminated unexpectedly with errors like *Segmentation fault* or *Assertion failed*. For example, in one case (github.com/ObjectProfile/Roassal3/issues/142) the Pharo process crashes because one of its native libraries aborted.
- *Pharo process freezes or waits forever.* Pharo freezes because it executes a mutated test method that enters a deadlock and waits forever (github.com/pharo-project/pharo/issues/6754), (github.com/feenkcom/gtoolkit/issues/1454). Note that similar problems also happen in other tools such as DSpot (github.com/STAMP-project/dspot/issues/994).
- *Unwanted process termination.* A mutated test calls a critical method in the system API. For example in one case (github.com/pharo-project/pharo-launcher/issues/454), during input amplification, a method call is added that snapshots the image and exits with return code 0.

Consistent with previous studies [10,40,41], these crashes are interesting from a reliability perspective, and the pieces of code that broke Pharo can be used to reproduce the crash. So, besides recovering from the crashes, we collect sufficient information to allow developers to reproduce them.

How to recover from crashes? We use the application heartbeats [42] to make test amplification self-aware [43] by detecting crashes and recovering from them. To this aim, we use a separate process (called the *runner script*) that initiates the amplification process in Pharo and watches its status. Fig. 2 shows the activity diagram for describing how these two processes interact to detect the crashes and recover from them. The components on top, which are colored yellow and have a light border, are steps executed in the Pharo image by Small-Amp. The components on the bottom, which are colored green and have a bold border, are steps executed in the runner script.

The runner script runs a Pharo process (child process) and initiates Small-Amp. Small-Amp regularly creates heartbeat signals by updating the content of a heartbeat file to notify that it is still functioning. The runner script also periodically watches the update time of the heartbeat file to make sure that the child is alive and works as expected.

Before starting to amplify a test method, Small-Amp sets a flag `crash` to `False` in a shared file (state file) and snapshots the current state of the Pharo image. Then, it loads the state file and checks the value of the `crash` flag. If it is the same stored value (`False`), it starts to amplify the test method.

If a crash happens while amplifying a test method, the Pharo image terminates unexpectedly or ceases to produce heartbeat signals. In this case, the runner notifies that there is a problem in the child process. It kills the child process if it is still running, then updates the state file by setting the `crash` flag value to `True`. Then it collects the available logs, including the last generated test method causing the crash. Finally, it resumes the Pharo image and watches its status again.

Pharo is a live system; when we snapshot its state, the next time we start the image, it will continue from the snapshot point. In our case, it resumes from the decision activity labeled `Check crash` (Fig. 2), not from the beginning of the Small-Amp algorithm. After recovering from a crash, Small-Amp loads the state file and checks the value of the `crash` flag. Since the runner has flipped its value, it concludes that a crash may happen if it continues to amplify the current test method. Therefore, it skips this test method and continues to amplify other methods.

The runner also keeps track of the number of recovered crashes. If it is more than a fair number `MAX_CRASH` (default is 10), it understands that there is a severe problem in amplifying this class, so it stops trying.

If all test methods are amplified in the Pharo process, it finalizes and exports the results. Then it updates a flag `finish` in the state file and exits the process. The runner realizes that the child process has exited with a `SUCCESS` return code. It checks the `finish` flag; if it is set correctly, it stops watching and finishes the process; otherwise, it considers this termination as a crash. As explained earlier, we do not trust the return value because the exit API can be called indirectly during test amplification.

4. Evaluation

To evaluate whether a test amplification bot is indeed feasible and to quantify the impact of prioritization, sharding, and sandboxing, we formulate the following research questions.

RQ1 – Is it possible to fully automatically amplify a test suite using GitHub-Actions? This is the primary research question for this feasibility study. To evaluate whether a test amplification bot is feasible, we install the proof-of-concept extension of Small-Amp on five open-source Pharo projects deployed on GitHub. We collect quantitative evidence on the execution times when run on the GitHub platform.

RQ2 – How does the prioritization heuristic affect the test amplification performance? To quantify the impact of the test prioritization, we compare the number of killed mutants with or without test prioritization and the execution time with and without the time budget.

RQ3 – How many duplicated mutants are created after sharding? The sharding step splits large test classes (more than 15 test methods) to avoid that the optimization step is forced to ignore relevant test methods. However, the same mutant may then be killed by more than one of the shards, thus resulting in duplicated mutants. Duplicated mutants designate wasted computations in the test amplification algorithm and, hence should be minimized.

RQ4 – Does sandboxing circumvent crashes? To illustrate the necessity of sandboxing, we count how often the sandbox recovered from (a) a system freeze, (b) a system crash, and (c) a polluted image.

4.1. Dataset

We used the GitHub search API to sort the Pharo projects based on the number of stars. Then we discarded the system projects and projects without a smalltalkCI configuration file (`.smalltalk.ston`). *smalltalkCI* is a framework to integrate the Pharo projects with continuous integration platforms. We prefer the projects to include `.smalltalk.ston` file because it contains all project-wise configurations, as well as shows that the project is CI-friendly. After filtering, we selected the five projects with the most stars. We had to limit the evaluation to five projects because running Small-Amp at the project level takes considerable time, and we had to repeat the evaluation multiple times. The final selected projects (with links to the project original repository and our fork used in this evaluation), their number of stars, and a short description are shown in Table 1.

Table 2 shows the descriptive statistics for the test classes in these projects. In this evaluation, we exclude the test classes without any passing (green) test method or test classes with 100% mutation coverage. We also consider test classes with more than 15 test classes as large test classes and break them down into shards.

4.2. Experimental design and results

We forked all projects in our dataset in GitHub and set up the test amplification GitHub-Actions workflow. In setting the default values for the workflow we exploited the content of `.smalltalk.ston` for loading the project, and the default values defined by Abdi et al. [12] for running Small-Amp ($N_{maxInput} = 10$, $N_{iteration} = 3$). Since they report that the majority of executions are finished in less than 6 min, we set the time budget to 12 min to have some leeway. We consider a crash if the heartbeat file is not updated for 4 min. We used eight parallel jobs

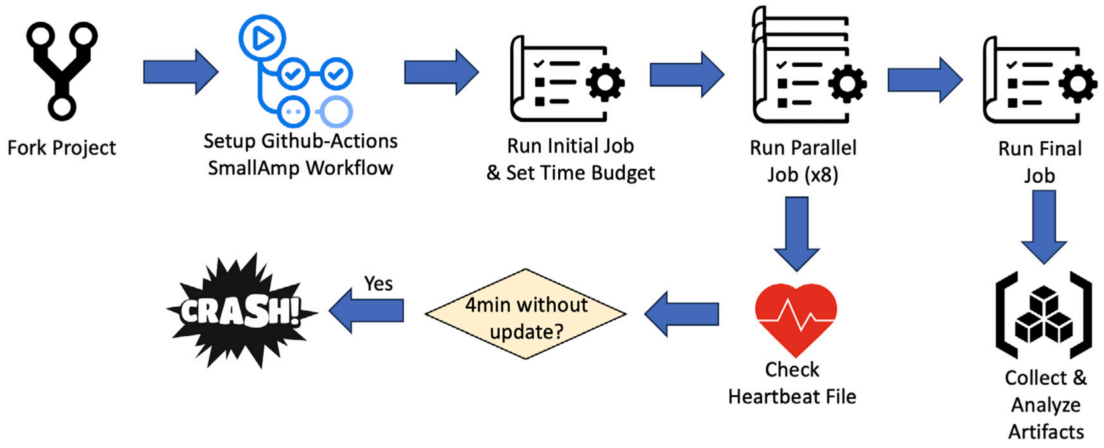
Fork Project
Setup Github-Actions SmallAmp Workflow
Run Initial Job & Set Time Budget
Run Parallel Job (x8)
Run Final Job
Check Heartbeat File
4min without update?
Yes
CRASH!
Collect & Analyze Artifacts

Fig. 3. Diagram showing the evaluation process to collect the test artifacts for each workflow.

Table 1
Dataset composed of 5 Pharo projects from GitHub.

Project	Stars	Description
Seaside	389	Web-application Framework
PolyMath	148	Scientific Computing for Pharo
NovaStelo	113	Block-style programming environment
Moose	103	Platform for software and data analysis
Zinc	69	HTTP networking protocol framework

Table 2
Descriptive statistics for the test classes.

	Seaside	PolyMath	NovaStelo	Moose	Zinc
# before sharding	82	68	50	8	35
# large test classes	10	3	7	0	2
# after sharding	99	87	82	8	40
# with 100% coverage	29	3	16	0	2
# without any green test	0	0	3	0	0
# Classes to be amplified	70	84	63	8	38

for each workflow, plus one initial and one finalizing job, totaling ten jobs for each workflow run. Since GitHub-Actions offers up to 20 jobs to run simultaneously for free accounts and open source projects at the time of writing this paper, this number of jobs is acceptable for open source projects. GitHub also allows each job to run for a maximum of 6 hours [33]. Fig. 3 shows a simplified diagram of the described process.

Then we manually ran the workflow six times. We enabled the test method prioritization mechanism in the first three runs and disabled it in the subsequent three runs. We opted for three repetitions for each experiment to account for the non-deterministic nature of the test-amplification algorithm, aiming to evaluate the potential impact of multiple executions on the results. We collected the generated artifacts and analyzed them to answer our research questions. To conclude, we manually ran 30 workflows in total (6 for each of the five projects), resulting in 300 jobs on GitHub servers.

Table 3 shows the results from this analysis. In each run, 263 test classes are executed (row 1) in which between 209 to 221 cases are finished (row 2), and 28 to 34 cases are unfinished (row 4). In 92 to 105 cases, Small-Amp is able to successfully amplify the test class (row 6) and generate 165 to 223 new test methods (row 9). In 17 to 21 cases, the time budget was in effect (row 10), so 116 to 148 test methods were skipped during amplification (row 11). The execution for each run, the sum of all projects, takes between 4:19 to 5 h (row 12). We have included the artifacts and workflow run logs in the reproduction package.[5]

4.3. RQ1: Is it possible to fully automatically amplify a test suite using github-actions?

Table 3 shows that for the 263 test classes, the test amplification finished successfully in 209–221 cases. The maximum execution time for an entire project was about 90 min for *Seaside*, while the minimum time was around 27 min for *Moose*. Since each workflow can run up to 6 h in GitHub-Actions, these values show that there is room for optimizing the configurations. This may be done by reducing the number of parallel jobs from 8 to a lower value; by increasing the Small-Amp parameters ($N_{maxInput}$, $N_{iteration}$); and by increasing the time budget. This illustrates that fine-tuning the configuration is needed when a test amplification bot is adopted for a given project but there is sufficient room to do so.

In addition, if we consider the execution time per project in all 6 runs, we see that the results are similar and do not vary a lot. This similarity confirms that setting a time budget makes the execution time of test amplification indeed more predictable.

Answer to RQ1: Our proof-of-concept demonstrates that integrating a test amplification tool within a continuous integration server allows for a fully autonomous process. Moreover, the time budget allows for doing so in an acceptable period of 30 to 90 min in our analysis.

4.4. RQ2: How does the prioritization heuristic affect the test amplification performance?

By design, if the test amplification hits the time limit, it kills fewer mutants. However, the test prioritization should dampen this effect. We expect more killed mutants with test prioritization than without. On the other hand, when the time limit is not reached, the impact of test prioritization should be negligible. We, therefore, compare the value of the increased kill count based on their timeout status in two configurations (prioritization enabled in the first three runs and disabled in the next three). The value of increased kills is calculated as follows:

$$100 \times \frac{\#killed\ mutants\ with\ prioritization}{\#killed\ mutants\ without\ prioritization}$$

[5] http://github.com/mabdi/SmallAmp-evaluations.

Table 3
The result of the quantitative analysis.

		Prioritization Enabled			No Prioritization		
		#1	#2	#3	#1	#2	#3
1	# All test classes executions	263	263	263	263	263	263
2	# Finished executions	221	216	217	213	209	216
3	# Image pollution	14	18	17	16	23	18
4	# Unfinished	28	29	29	34	31	29
5	# Recovered freezing/crashes	37 (16.7%)	37 (17.2%)	37 (17%)	43 (20.1%)	44 (21%)	37 (17.1%)
6	# Executions having improvement	105	98	97	92	96	102
7	% Executions having improvement	47.51%	45.37%	44.70%	43.19%	45.93%	47.22%
8	# Test methods	1805	1766	1757	1703	1677	1762
9	# Generated tests	223	213	194	165	166	199
10	# All mutants in finished cases	9758	9670	9713	9094	9029	8761
11	# Mutants live original	3984	3957	3972	3814	3315	3292
12	# Mutants killed original	5774	5713	5741	5280	5779	5469
13	# Newly killed mutants	561	561	533	483	499	538
14	% Increased kills	9.71%	9.81%	9.28%	9.14%	8.63%	9.83%
15	# Mutants killed in Large test classes	198	176	167	115	144	157
16	# Duplicated killed mutants in Large classes	56	56	48	34	51	58
17	% Duplicated killed in Large classes	28.28%	31.82%	28.74%	29.57%	35.42%	36.94%
18	# Time budget finished	18	18	18	17	19	21
19	# Test methods skipped	148	133	123	118	116	119
20	Workflow duration: All	4:33:41	4:42:09	4:35:45	5:00:54	4:19:13	4:37:20
21	Seaside	1:28:14	1:32:34	1:24:47	1:31:14	1:28:06	1:25:23
22	PolyMath	1:04:57	1:07:08	1:06:43	1:12:21	1:10:11	1:07:19
23	NovaStelo	1:00:58	1:01:08	1:04:21	1:12:24	0:41:16	1:03:53
24	Moose	0:26:45	0:27:37	0:27:52	0:27:18	0:27:59	0:26:29
25	Zinc	0:32:47	0:33:42	0:32:02	0:37:37	0:31:41	0:34:16

Table 4
Comparison of the number of newly killed mutants when prioritization is enabled and disabled.

	Timeout	In time
Number of classes	7	156
Disabled (killed mutants)	44	1267
Enabled (killed mutants)	59	1276
Increase	**34.09%**	**0.70%**

Table 4 compares the number of newly killed mutants for these test classes and their increase.

As expected, we see the cases with prioritization have better performance (34.09%) when they run out of time. For the classes that finished within time, the increase in the killed mutants is negligible (0.70%).

Answer to RQ2: Test prioritization is an effective way to impose a predictable time budget on the test amplification. In those cases where the time limit is reached, test prioritization kills more mutants. When the time limit is not reached, test prioritization has negligible impact.

4.5. RQ3: How many duplicated mutants are created after sharding?

Sharding involves splitting classes into smaller units to run in parallel. Therefore, this process may lead to more duplicate mutants since parallel jobs working independently can result in multiple jobs addressing the same mutants, as each job lacks knowledge or access to others.

To quantify how much waste is induced by the sharding step, we calculate the number of duplicated mutants killed due to splitting overly large (more than 15 test methods) classes.

First of all, Table 2 illustrates that such large test classes actually exist, although it depends a lot on the project. The **Seaside** project has 10 large classes; hence the sharding increased the number of test classes from 82 to 99. The **Moose** project, on the other hand, had no large test classes.

For those cases with large test classes, we found 198, 176, 167, 115, 144, and 157 killed mutants (3, row 15). Consequently, the number of duplicated mutants in the shards is 56, 56, 48, 34, 51, and 58 (3, row 16). Therefore, about 28% to 37% of the killed mutants are duplicated when we employ sharding, which is considerable. Further research is warranted to see whether we can decrease these duplicates, for instance, by clustering the shards based on their coverage. Finding a balance between the sharding factor and the time budget (in our analysis, we used 15 and 12 min) may also decrease the number of duplications.

Answer to RQ3: Sharding allows running the test amplification in a given time budget, even with overly long test classes. However, it comes at a cost: in our analysis, we see around 30% duplications in the killed mutants when large test classes get split into distinct shards.

4.6. RQ4: Does sandboxing circumvent crashes?

To assess the effectiveness of sandboxing mechanism, we process the job logs in all six runs to collect quantitative evidence of crashes, freezes, and polluted images. In the finished execution, we see 37 to 44 classes recovered from a crash (Table 3, row 5). Overall, the crash-recovery mechanism recovered the amplification process 235 times in all six runs. After investigating the reasons for these crashes, we found that most cases (about 95%) are recovering from a system freeze. In one case, the crash is because of a *Segmentation fault* error.

Table 3 row 3 shows in 14 to 23 cases of classes, image-state pollution occurs; note that the numbers vary because of sharding. However, this only occurred in one of the projects (*NovaStelo*), where test classes are green before mutation testing and become red after. The *NovaStelo* project had 58 test classes at the time of our analysis, therefore, between 24.1% (14/58) and 39.6% (23/58) suffered from image-state pollution.

We conclude that image crashes, freezing, and state pollution is a phenomenon that occurs regularly during test amplification. When these are not appropriately handled, test amplification integration in continuous integration will fail. Nevertheless, the proposed crash-recovery mechanism allows Small-Amp to overcome the problems and skip the failing cases.

Answer to RQ4: Image crashes, freezing, and state pollution frequently happen when running the test amplification in Pharo. The results of the evaluation show that the proposed sandboxing mechanism is effective in overcoming these problems.

5. Threats to validity

Did we measure what was intended? (construct validity). We use quantitative metrics (the number of newly killed mutants, the number of recovered crashes, and execution time) to quantify the impact of a test amplification bot and the effect of prioritization, sharding, and sandboxing. These metrics are the right metrics to address the four research questions. However, the larger question is whether a test amplification bot would be adopted by a team of practicing developers. A qualitative study would be needed to evaluate whether the recommended tests add value, which is beyond the scope of this paper.

Are there unknown factors that might affect the outcome of the analyses? (internal validity). The test amplification tool, the prioritization mechanism, and the GitHub-Actions workflow include various parameters. For configuring Small-Amp, we used the parameters from Abdi et al. [12], in which the authors have not claimed that values are optimal. Similarly, the prioritization mechanism parameters (α, β, and γ in Section 3.3) are also configured by preliminary values based on authors' insights. We see this risk as a minimum because optimizing the parameters should lead to a better result, not invalidating the findings.

Moreover, the choice of 15 as a threshold for the methods in Sharding lacks a deeper evaluation. Although based on our experience, it was not entirely arbitrary, but we recognize the need for a more thorough study to validate the optimal threshold value. This aspect is left for future work.

For identifying pollution (Section 2.2), we run the test class after the early mutation testing (Algorithm 1 line 1). If the test is green, we assume the state is not polluted and continue the algorithm, otherwise, pollution has occurred. We reduce the impact of image pollution by using a fresh Pharo image for amplifying each test class, which stops propagating the possible pollution to the following process. However, there might be some pollution undetected by the tests. Further research is needed for more effective methods to detect image pollution, as there may be additional pollution scenarios than those reported in our results.

To what extent is it possible to generalize the findings? (external validity). The proof-of-concept test-amplification bot has been validated against 5 Pharo projects available on GitHub. These were active projects receiving a high number of stars and coming with a good amount of unit tests covering a lot of the code (cfr. 1 and 2). As such these were projects with a good maturity, hence the selected projects imply a positive bias. It is unclear at this stage whether test amplification bots would also work in the context of less mature projects with heavily fluctuating tests.

Concerning the test amplification bots, we expect the overall findings like the applicability of project-level test amplification (RQ1), the impact of test method prioritization (RQ2) and sharding (RQ3), and the relevance of the sandbox mechanism (RQ4) to be valid in other tools. However, we cannot make any claim about the numbers and other details to be valid for other ecosystems.

Is the result dependent on the tools? (conclusion validity). Our work depends on Small-Amp as the test amplification tool. In RQ2, we compare the results from Small-Amp in two different configurations to assess the impact of prioritization. So, we expect the differences in the results to be mainly because of the configuration, not the tool itself. Another critical factor is randomness, which we tried to diminish by repeating the experiment three times for each configuration on five different mature projects with a high number of test classes.

6. Related work

This work extends Small-Amp [12], test amplification in Pharo by integrating it into GitHub-Actions, providing a prioritization heuristic, sharding, and sandboxing. Test amplification for crash reproduction has also been reported in other papers [10,40,41]. To the best of our knowledge, sharding (i.e. splitting test cases to fit in a time budget) is a novel concept in test amplification. However, some works in parallel test case prioritization also split a test suite to fit a time budget (running different portions of the test suite in different machines) [44].

There are several works in the Pharo community related to our sandboxing solution: Polito et al. study the bootstrapping problem in Smalltalk and provide the Hazelnut model for bootstrapping reflective systems [45]. The work by Béra et al. introduces Sista, a fast snapshotting and restoring solution to increase the warmup time of Pharo images [46]. Epicea [47] records the changes in the code and some IDE events in logs, which can be used to recover the lost state.

We consider our extension of Small-Amp as a proof of concept for a test amplification bot. Relevant related work for test amplification bots is techniques for increasing the readability of the generated tests, i.e. intention revealing names [48] and removing the redundant statements [49].

Repairnator [50] on the other hand is a complementary attempt at a test automation bot, this time for automated program repair. Other complementary work on test automation bots is the work by Campos et al. [15]. They introduce Continuous Test Generation (CTG) by incorporating EvoSuite in a continuous integration setting. In a similar vein, Danglot et al. [7] investigated ways to exploit test amplification in a continuous integration setting.

7. Conclusion

In this paper, we argued that a test amplification bot may alleviate the challenges that prevent widespread adoption of test amplification tools. In this vision, a test amplifier decides for itself which tests to amplify, incorporate the synthesized tests in a separate branch, execute the strengthened test suite, and —if all steps pass— push the strengthened test suite onto the main branch. All without any intervention of a software engineer. To demonstrate the feasibility of this vision, we present a proof-of-concept tool that integrates Small-Amp with GitHub-Actions to automatically strengthen the existing test suite within a limited time budget.

We validated the proof-of-concept tool on five popular open-source Pharo projects. The results show that integrating a test amplification tool within a continuous integration server indeed allows for a fully autonomous process. Moreover, the time budget allows us to do so in an acceptable time span; of 30 to 90 min in our analysis. Test prioritization was able to improve the performance of the tool when the time budget was exceeded by up to 34%. Test sharding was needed to run the test amplification in a given time budget, even with overly long test classes. However, it comes at a cost, in our analysis, we see around 30% duplication in the killed mutants when large test classes get split into distinct shards. Last but not least, we demonstrated that sandboxing is an effective way to make the test amplification crash-resilient.

Even though a bot would facilitate the adoption of test amplification, more qualitative research is needed to make the results acceptable. This ranges from improving the readability of the generated test cases (intention revealing names, meaningful comments, . . .) to usability studies assessing the added value of the amplified tests.

Reproduction package

Supplementary resources are available in our reproduction package available at our GitHub Repository.[6]

CRediT authorship contribution statement

Mehrdad Abdi: Data curation, Investigation, Methodology, Validation, Writing – original draft, Conceptualization. **Henrique Rocha:** Investigation, Resources, Supervision, Writing – original draft, Writing – review & editing. **Alexandre Bergel:** Investigation, Methodology, Supervision, Validation, Writing – original draft, Writing – review & editing. **Serge Demeyer:** Conceptualization, Funding acquisition, Investigation, Methodology, Project administration, Supervision, Validation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Supplementary resources are available in our reproduction package available at our GitHub Repository.[7].

Acknowledgments

This work is supported by (a) the Fonds de la Recherche Scientifique-FNRS and the Fonds Wetenschappelijk Onderzoek - Vlaanderen (FWO) under EOS Project 30446992 SECO-ASSIST (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

References

[1] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, Mark Harman, Mutation testing advances: An analysis and survey, in: Advances in Computers, Adv. Comput. 112 (2019) 275–378.

[2] Gordon Fraser, Andrea Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011, pp. 416–419.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 75–84.

[4] Nikolai Tillmann, Jonathan de Halleux, Pex–white box test generation for. net, in: International Conference on Tests and Proofs, Springer, 2008, pp. 134–153.

[5] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, Dspot: Test amplification for automatic assessment of computational diversity, 2015, arXiv preprint arXiv:1503.05807.

[6] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, Martin Monperrus, Automatic test improvement with dspot: a study with ten mature open-source projects, Empir. Softw. Eng. 24 (4) (2019) 2603–2635.

[7] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, Benoit Baudry, An approach and benchmark to detect behavioral changes of commits in continuous integration, Empir. Softw. Eng. 25 (4) (2020) 2379–2415.

[8] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, Benoit Baudry, Suggestions on test suite improvements with automatic infection and propagation analysis, 2019, arXiv preprint arXiv:1909.04770.

[9] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, Test amplification in the pharo smalltalk ecosystem, in: Proceedings IWST 2019 International Workshop on Smalltalk Technologies, ESUG, 2019.

[10] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, Reproducible crashes: fuzzing pharo by mutating the test methods, in: International Workshop on Smalltalk Technologies, IWST, 2020.

[11] Ebert Schoofs, Mehrdad Abdi, Serge Demeyer, Ampyfier: Test amplification in python, J. Softw.: Evol. Process n/a (n/a) (2022) e2490.

[6] https://github.com/hscrocha/TestAmplificationBotReproductionPackage.

[7] https://github.com/hscrocha/TestAmplificationBotReproductionPackage

[12] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, Alexandre Bergel, Small-amp: Test amplification in a dynamically typed language, Empir. Softw. Eng. 27 (2022) 128.

[13] STAMPproject, Use cases validation report v3, 2019, [on line] https://github.com/STAMP-project/docs-forum/blob/master/docs/d57_uc_validation_report-final.pdf — last accessed on August 2023.

[14] Carolin Brandt, Andy Zaidman, Developer-centric test amplification the interplay between automatic generation and human exploration, 2021.

[15] José Campos, Andrea Arcuri, Gordon Fraser, Rui Abreu, Continuous test generation: Enhancing continuous integration with automated test generation, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 55–66.

[16] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, Benoit Baudry, A snowballing literature study on test amplification, J. Syst. Softw. 157 (2019) 110398.

[17] Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, Pharo By Example, Square Bracket Associates, c/o Oscar Nierstrasz, 2010.

[18] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, Jannik Laval, Deep Into Pharo, Lulu. com, 2013.

[19] Hernán Wilkinson, Nicolás Chillo, Gabriel Brunstein, Mutation testing, 2009, European Smalltalk User Group (ESUG 09). Brest, France. http://www.esug.org/data/ESUG2009/Friday/Mutation_Testing.pdf.

[20] Mehrdad Abdi, Toward Zero-touch Test Amplification (Ph.D. thesis), University of Antwerp, Antwerp, Belgium, 2022, https://repository.uantwerpen.be/docstore/d:irua:14939.

[21] André N. Meyer, Earl T. Barr, Christian Bird, Thomas Zimmermann, Today was a good day: The daily life of software developers, IEEE Trans. Softw. Eng. 47 (5) (2021) 863–880.

[22] Sven Amann, Sebastian Proksch, Sarah Nadi, Mira Mezini, A study of visual studio usage in practice, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, 2016, pp. 124–134.

[23] Roberto Minelli, Andrea Mocci, Michele Lanza, I know what you did last summer - an investigation of how developers spend their time, in: 2015 IEEE 23rd International Conference on Program Comprehension, 2015, pp. 25–35.

[24] Juraj Kubelka, Romain Robbes, Alexandre Bergel, Live programming and software evolution: Questions during a programming change task, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), 2019, pp. 30–41.

[25] Steven L. Tanimoto, A perspective on the evolution of live programming, in: Proceedings of the 1st International Workshop on Live Programming, LIVE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 31–34, URL http://dl.acm.org/citation.cfm?id=2662726.2662735.

[26] Christopher Parnin, A history of live programming, 2013, URL http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/.

[27] Stéphane Ducasse, Manuel Oriol, Alexandre Bergel, Challenges to support automated random testing for dynamically typed languages, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '11, ACM, New York, NY, USA, 2011, pp. 9:1–9:6.

[28] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov, An empirical analysis of flaky tests, in: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, in: FSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, pp. 643–653.

[29] August Shi, Jonathan Bell, Darko Marinov, Mitigating the effects of flaky tests on mutation testing, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, in: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 112–122.

[30] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, Danny Dig, Usage, costs, and benefits of continuous integration in open-source projects, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, in: ASE 2016, Association for Computing Machinery, New York, NY, USA, 2016, pp. 426–437.

[31] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, Andy Zaidman, Continuous delivery practices in a large financial organization, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 519–528.

[32] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, Mark GJ van den Brand, Continuous integration in a social-coding world: Empirical evidence from GitHub, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 401–405.

[33] GitHub, GitHub actions usage limits, billing, and administration, 2022, [on line] https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration — last accessed In February 2022.

[34] GitHub, About custom actions, 2022, [on line] https://docs.github.com/en/actions/creating-actions/about-custom-actions — last accessed In May 2022.

[35] GitHub, Storing workflow data as artifacts, 2022, [on line] https://docs.github.com/en/actions/using-workflows/storing-workflow-data-as-artifacts — last accessed In May 2022.

[36] GitHub, Reusing workflows, 2022, [on line] https://docs.github.com/en/actions/using-workflows/reusing-workflows — last accessed In May 2022.

[37] Paul Ammann, Jeff Offutt, Introduction To Software Testing, Cambridge University Press, 2016.
[38] Mohammad Ghafari, Carlo Ghezzi, Konstantin Rubinov, Automatically identifying focal methods under test in unit test cases, in: Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on, IEEE, 2015, pp. 61–70.
[39] Wikipedia, Fitness proportionate selection, 2020, [on line] https://en.wikipedia.org/wiki/Fitness_proportionate_selection — last accessed In February 2022.
[40] Jeremias Rö*β*ler, Gordon Fraser, Andreas Zeller, Alessandro Orso, Isolating failure causes through test case generation, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, 2012, pp. 309–319.
[41] Jifeng Xuan, Xiaoyuan Xie, Martin Monperrus, Crash reproduction via test case mutation: let existing test cases help, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 910–913.
[42] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, Anant Agarwal, Application heartbeats for software performance and health, SIGPLAN Not. 45 (5) (2010) 347–348.
[43] Samuel Kounev, Peter Lewis, Kirstie L Bellman, Nelly Bencomo, Javier Camara, Ada Diaconescu, Lukas Esterle, Kurt Geihs, Holger Giese, Sebastian Götz, et al., The notion of self-aware computing, in: Self-Aware Computing Systems, Springer, 2017, pp. 3–16.
[44] Jianyi Zhou, Junjie Chen, Dan Hao, Parallel test prioritization, ACM Trans. Softw. Eng. Methodol. 31 (1) (2021).
[45] G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, B. van Ryseghem, Bootstrapping reflective systems: The case of pharo, Sci. Comput. Programm. 96 (2014) 141–155, Special issue on Advances in Smalltalk based Systems, URL https://www.sciencedirect.com/science/article/pii/S0167642313002797.
[46] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, Stéphane Ducasse, Sista: Saving optimized code in snapshots for fast start-up, in: Proceedings of the 14th International Conference on Managed Languages and Runtimes, in: ManLang 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–11.
[47] Martín Dias, Damien Cassou, Stéphane Ducasse, Representing code history with development environment events, in: IWST-2013-5th International Workshop on Smalltalk Technologies, 2013.
[48] Nienke Nijkamp, Carolin Brandt, Andy Zaidman, Naming amplified tests based on improved coverage, in: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021, pp. 237–241.
[49] Wessel Oosterbroek, Carolin Brandt, Andy Zaidman, Removing redundant statements in amplified test cases, in: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021, pp. 242–246.
[50] Simon Urli, Zhongxing Yu, Lionel Seinturier, Martin Monperrus, How to design a program repair bot? Insights from the repairnator project, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2018, pp. 95–104.

Mehrdad Abdi is a senior software test engineer at Nokia, Antwerp, Belgium. He obtained his Ph.D. from the University of Antwerp under the supervision of Prof. Serge Demeyer in 2022. In his Ph.D., he studied software testing, more specifically test amplification, in the context of ecosystems. Before his PhD, he was working as a software security engineer for 5 years. His main research/work interest is software testing.

Henrique Rocha since 2021 has been an assistant professor at Loyola University Maryland, USA. He completed his PhD in 2016 in ASERG (Applied Software Engineering Research Group) at UFMG (Federal University of Minas Gerais), Brazil. He was a post-doctoral researcher at Rmod Inria-Lille, France from 2017-2018, and at AnSyMo in the University of Antwerp, Belgium from 2019–2021. After working at Rmod, Henrique developed a newfound appreciation and respect for Smalltalk. His research interests involve Applied and Empirical Software Engineering, Software Testing and maintenance, and Blockchain-oriented Software engineering.

Alexandre Bergel is a Computer Scientist at RelationalAI, Switzerland. Until 2022, he was an Associate Professor and researcher at the University of Chile. Alexandre Bergel and his collaborators carry out research in software engineering. Alexandre Bergel has authored over 170 articles, published in international and peer-reviewed scientific forums, including the most competitive conferences and journals in the field of software engineering. Alexandre has participated in over 175 program committees of international events. Several of his research prototypes have been turned into products and adopted by major companies in the semiconductor industry, certification of critical software systems, and aerospace industry. Alexandre gave talks to prominent research institutes, including NASA JPL and the Deutsches Zentrum für Luft-und Raumfahrt (DLR). Alexandre is a member of the editorial board of Empirical Software Engineering. Alexandre authored 4 books on Pharo, Artificial Intelligence, and Data Visualization.

Serge Demeyer is a professor at the University of Antwerp and member of the AnSyMo (Antwerp System Modelling) research group. Serge Demeyer is the chair of the NEXOR interdisciplinary research consortium and an affiliated member of the Flanders Make Research Centre. In 2007 he received a "Best Teachers Award" from the Faculty of Sciences at the University of Antwerp and is still very active in all matters related to teaching quality. His main research interest concerns software evolution, more specifically how to strike the right balance between reliability (striving for perfection) and agility (optimizing for adaptability). He is an active member of the corresponding international research communities, serving in various conference organization and program committees. He has written a book entitled "Object-Oriented Reengineering" and edited a book on "Software Evolution". He also authored numerous peer reviewed articles, many of them in top conferences and journals. He has an h-index of 42 according to Google Scholar. Serge Demeyer completed his M.Sc. in 1987 and his PhD in 1996, both at the "Vrije Universiteit Brussel". After his PhD, he worked for three years in Switzerland, where he served as a technical co-ordinator of a European research project. Switzerland remains near and dear to his heart, witness the sabbatical leave during 2009–2010 at the University of Zürich in the research group SEAL. The rest of Europe is explored territory as well, with a sabbatical stay in Lille, France (INRIA-RMOD) and in Lund, Sweden (RISE-SICS and SERG).