# When to Let the Developer Guide: Trade-offs Between Open and Guided Test Amplification

Carolin Brandt, Danyao Wang, Andy Zaidman
*Delft University of Technology*
c.e.brandt@tudelft.nl, wangdanyaoa@gmail.com, a.e.zaidman@tudelft.nl

***Abstract*—Test amplification generates new tests by mutating existing, developer-written tests and keeping those tests that improve the coverage of the test suite. Current amplification tools focus on starting from a specific test and propose coverage improvements all over a software project, requiring considerable effort from the software engineer to understand and evaluate the different tests when deciding whether to include a test in the maintained test suite. In this paper, we propose a novel approach that lets the developer take charge and guide the test amplification process towards a specific branch they would like to test in a control flow graph visualization. We evaluate whether simple modifications to the automatic process that incorporate the guidance make the test amplification more effective at covering targeted branches. In a user study and semi-structured interviews we compare our user-guided test amplification approach to the state-of-the-art open test amplification approach. While our participants prefer the guided approach, we uncover several trade-offs that influence which approach is the better choice, largely depending on the use case of the developer.**

***Index Terms*—Software Testing, Test Amplification, Automated Test Code Modification, User-centric Design, Human-Automation Interaction**

## I. Introduction

Software testing is one of the central activities in the software development lifecycle [1]. One part of this are developer tests, i.e., small automated programs that software developers write to check that their code behaves as they intend and prevent it from breaking in the future [2]. While developer testing is widely seen as valuable, it is also a tedious and time-consuming activity [3]. One automated approach to relief developers of this manual effort is *test amplification*. Test amplification mutates existing, developer-written tests to explore new behavior of the code under test [4]. Previous studies have shown that it can provide valuable tests to developers [5]–[7], but at the cost of long runtimes [5], [7] and effort for the developers to understand the behavior and impact of the amplified tests [7]–[9]. Let us illustrate this with an example:

> Masha, a software developer, is working on a new feature of their software project, that requires small changes in their existing code. Before submitting a patch, she needs tests that cover all her new code, so she decides to use test amplification to generate them automatically. She picks an existing test from the class she worked on and asks the tool to create new tests based on it. After a while the tool reports back and proposes several tests to her. Unfortunately, the class did not have a high test coverage, so she has to sift through quite a few tests spending time to understand what code they cover and realize it is not the code she is concerned with. Even for the tests that target her code, she has to switch between several methods under test and every time recall what behavior this method should have, so she can judge whether the generated test is correct.

Our hypothesis is that these understandability issues are in part rooted in the disconnect between the present point of interest of a developer in the code base, and the dispersed coverage contributions amplified tests are providing, i.e., they need to rebuild the task context [10]. To bridge this disconnect, we propose to involve the software developer more tightly in the test amplification process. Ideally, they can convey what piece of code they are interested in to test and then the test amplification presents only those tests that are relevant for the focus of the developer.

In this paper, we propose a novel approach of user-guided test amplification. Starting from a method in their code base, the developer can initiate the test amplification and choose in a visualized control-flow graph which branch of the method should be tested. The test amplification is then directed to call this method specifically, and generates a variety of tests for it. It measures the tests' branch coverage and presents all tests that cover the intended branch to the developer, using the same control-flow graph visualization to help the developer understand how the test executes the method under test.

We conduct a technical case study and a user study to understand the impact and potential use of user-guided test amplification.[1] In both studies we compare it to the existing test amplification approach [6], [7], which we will call *open test amplification* for a clearer distinction. With our technical case study on 31 classes from two open source projects, we investigate whether our simple changes in the guided amplification process are indeed effective at producing a higher ratio of tests for the targeted branch, and whether to guidance enables us to cover more branches overall in a project. Our findings from this study answer our first research question:

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032)

[1]We follow the empirical standard for engineering research: https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=EngineeringResearch

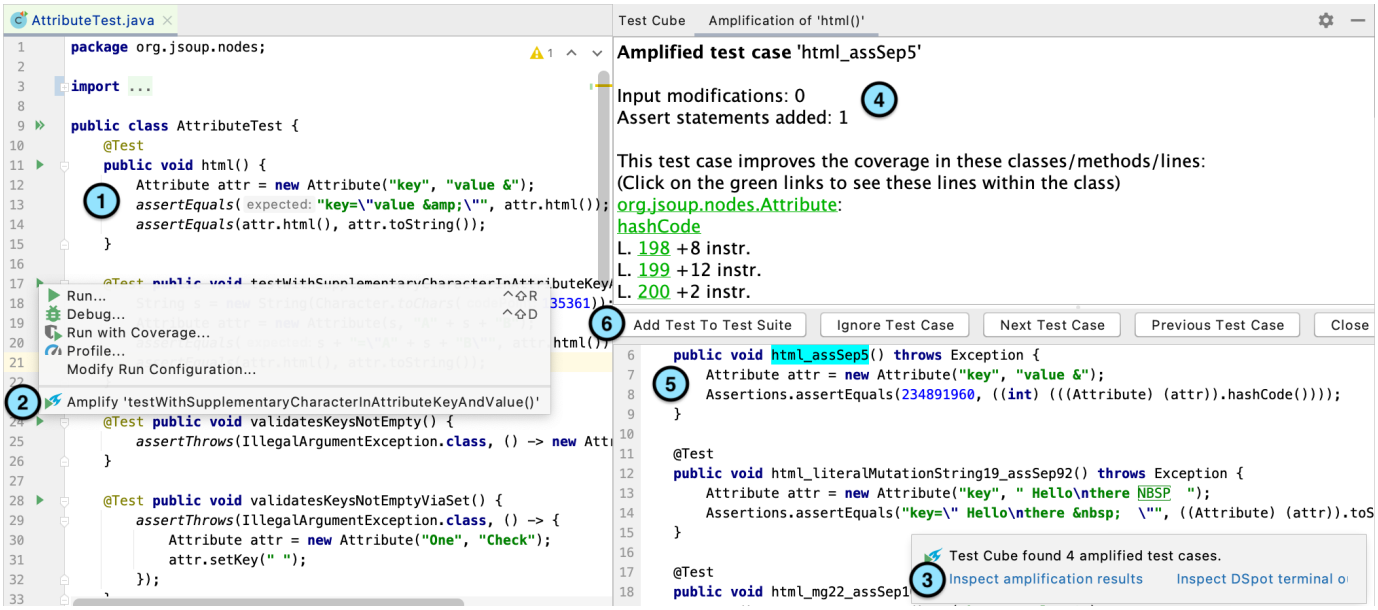


Fig. 1: Interaction with Brandt and Zaidman's test exploration IDE plugin for open test amplification [7].

**RQ1:** How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

In our user study, 12 developers apply both approaches to two classes and we interview them about their experiences. From this, we learn how they perceive each technique and their considerations when comparing them to each other. Our observations address our second research question:

**RQ2:** How do developers perceive guided test amplification (compared to open test amplification)?

Our two evaluation studies show that user-guided test amplification does deliver on the intended goals of making the test amplification process more effective and the coverage of the amplified tests easier to understand. However, the studies also show that the user-guided version of test amplification is not always better. From the participant's explanations during the interviews we learned that user-guided test amplification is closer to the real-life process of developing and testing new code where the developer focuses on a specific feature, writing code and tests for it. On the other hand, open test amplification is more suited when focusing on improving the test suite for an already existing code base, as it connects new tests clearer to the already existing tests. This is one example of the trade-offs between open and user-guided test amplification that our studies make apparent. We discuss all trade-offs we encountered to help the reader understand the strengths and weaknesses of both approaches, and to help developers choose which approach fits best to their goals and workflow.

## II. Test Amplification

In this section, we introduce the concept of (open) test amplification, which is realized in the state-of-the-art test amplification tool for Java called DSpot [6].

The aim of *test amplification* is to generate new tests by leveraging the knowledge in existing, human-written tests [4]. These new tests improve the existing test suite with respect to a defined engineering goal, e.g., structural coverage or mutation score. Our work is based on Brandt and Zaidman's proposal of *developer-centric test amplification*, which focuses on generating short and easy-to-understand tests to be included into the developer's maintained code base [7].

A central part of Brandt and Zaidman's proposal is to combine the automatic test amplification with a *test exploration tool* that guides the developer's interaction with the test amplification. Fig. 1 illustrates the workflow with their prototype in form of an IDE plugin. The developer starts by selecting an original test to be the basis for the amplification ① and requesting the plugin to amplify that test ②. When the amplification finishes, it notifies the developer ③ that they can start exploring the generated tests. The exploration tool presents to the developer the additional coverage that an amplified test provides ④, the code of the test ⑤, and action buttons to easily add the test into the test suite or browse through the list of amplified tests ⑥.

The automated process behind test amplification (see the upper half of Fig. 3) starts from an original test which comes from the existing, manually written test suite of a software project. We mutate the input phase of the test with several amplification operators: changing literal values slightly or replacing them with random values, as well as adding, duplicating or removing method calls to the objects under test. The old assertions are replaced by new ones which use the current behavior of the system as the oracle. Then, we execute all new tests and measure their instruction coverage. The tool selects all tests that cover new instructions compared

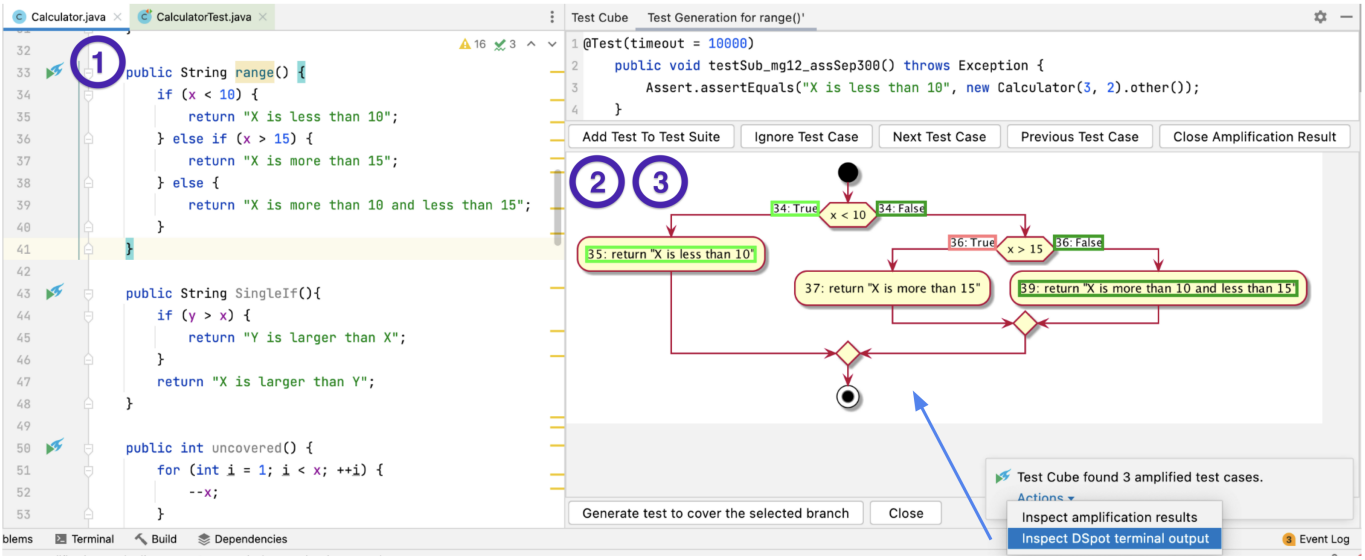


Fig. 2: Interaction with user-guided test amplification.

to the existing test suite and presents them to the developer.

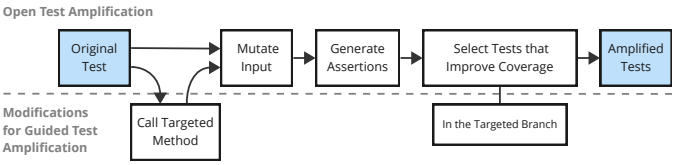


Fig. 3: The automated process behind open test amplification and modifications to it for guided test amplification.

The interaction and the underlying amplification process starts from a developer-selected original test, randomly mutates it and keeps all new tests that cover new instructions anywhere in the project under test. We coin it *open test amplification* as it openly looks for any new tests that could be valuable for a project.

Previous studies on open test amplification showed that with this approach, it is difficult for the users to connect the test to the code under test it covers [7], [9]. Also, not all uncovered code is equally important to be tested in the opinion of the developers [7]. The original proposers of the approach had to take several design decisions that limit the power of the amplification, in order to make it fast enough to be interactively used [7]. To address these shortcomings, we propose to let the developer take the lead and guide the test amplification towards the code they find relevant to be tested.

## III. User-Guided Test Amplification

To speed up the process of finding new tests and make it easier for the developer to understand the context of the generated tests, we propose to let the developer direct the test amplification to the specific code they want to test. We call this approach *user-guided test amplification* and build it upon the developer-centric implementation of DSpot [6], [7].

The developer starts by selecting a method in the code under test which they would like to test (see ① in Fig. 2). Then, the test exploration tool presents them with a control flow graph of that method, similar to the graph shown at ②. The graph shows the execution structure of the method through boxes for each statement and condition, connected with arrows. The arrows annotated with "True" or "False" represent branches in the control flow of the method, letting the developer see the different scenarios that might need testing. We compute the existing test coverage for the method and highlight the branches that are already covered in green, and those that are not covered in red. The developer can select the branch that they would like to cover and start the test amplification. The tool automatically looks for the corresponding test class and picks the first—often most simple—test as the original test for the amplification. If no corresponing test class or test can be found, the tool prompts the user to create a test and invoke the amplification again. When inspecting the result, the test exploration tool reuses the same control flow graph to show the developer the additional coverage that the amplified test provides ③. The developer can then decide whether to add the test to the test suite or to continue exploring the other tests or invoke the tool again for other branches.

We add two simple modifications to the underlying automated test amplification process to incorporate the guidance provided by the developer. The lower half of Fig. 3 illustrates the modifications we make to the open test amplification process. As the first modification to the input of the original test, we call the method selected by the developer with randomly generated values for the parameters. When an object is needed, DSpot looks for a public constructor and uses it with random values to initialize the object. Then we continue by randomly mutating the test input as with open test amplification. All produced tests that cover the branch selected by the developer

are selected as results to be presented to the developer.

We intentionally make simple modifications and largely rely on the amplification operators available in the base tool DSpot, e.g., the random generation of parameter values for object initialization. Our aim is to see whether such simple changes can already be effective to improve test amplification before considering more complex and runtime-impacting alternatives.

## IV. Evaluation

To evaluate our proposed user-guided test amplification, we conduct two comparative studies: a technical case study and a user study. Our first goal is to judge the effectiveness of our technical changes to the test amplification process: does the guidance lead to a larger proportion of the generated tests covering the targeted branch compared to using open test amplification (**RQ1**)? The second goal is to elicit the opinions of developers on interacting with user-guided and open test amplification (**RQ2**).

**RQ1:** How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

**RQ2:** How do developers perceive guided test amplification (compared to open test amplification)?

To answer **RQ1** we conduct a technical case study, where we apply both approaches to generate tests for 100 branches sampled from 31 classes of two open source projects. We analyze the ratio of amplified tests fulfilling our coverage goals to determine which approach is more effective. To answer **RQ2** we perform a user study with 12 developers that apply both open and guided test amplification to test two classes. Then we interview each participant to elicit their impression of each approach and how they compare to each other.

### A. Design Technical Case Study

In our technical case study, we sample code branches from two open source projects and apply both guided and open test amplification to try to cover them. We measure how many branches can be covered at all by each approach, and what percentage of the amplified tests generated in one run cover the targeted branch.

We select two open source projects as study objects: Javapoet [2], a library to generate java source files, and Stream-lib [3], a library for summarizing data in streams. An important selection criterion was the traceability from code to tests: in both projects we can identify the matching test class for a class, because they adhere to consistent naming conventions. To select the targeted methods under test, we pick all classes with a clearly identified test class and from these classes select all public, non-static and non-abstract methods, which are the methods that can be called by DSpot's amplification operators. Taking all branches from the selected methods under test (160 from Javapoet, 264 from Stream-lib), we randomly sampled 100 branches per project. From their matching test class, we take the first test as the original test method for the amplification.

We run both guided and open test amplification for each of the sampled branches, limiting the number of produced tests to 200 per run. Next, we collect all resulting tests as well as their coverage information. Per project, we calculated the ratio of covered branches over the sampled branches (Equation (1)).

$$ratio\ covered\ branches = \frac{\#\ branches\ covered}{\#\ branches\ sampled} \quad (1)$$

We calculate for each approach per project the average ratio of successful tests (Equation (2)) over all runs. The ratio of successful tests looks at how many of the returned amplified tests do indeed cover the targeted branch.

$$ratio\ successful\ tests = \frac{\#\ tests\ covering\ branch}{\#\ tests\ returned} \quad (2)$$

### B. Results Technical Case Study

Table I shows the calculated effectiveness of guided and open test amplification in comparison. We see that the guided test amplification can cover more branches in both projects, but the difference is small, and neither approach can cover more than 41% of the sampled branches. This shows that guiding the test amplification by explicitly calling the method that contains the targeted branch is only marginally helpful in covering a larger variety of branches of a project.

TABLE I: Ratio of covered branches (see Equation (1))).

| | Javapoet | Stream-lib |
|---|---|---|
| Open Test Amplification | 23% | 35% |
| Guided Test Amplification | 32% | 41% |

To understand why many branches could not be covered by either test amplification approach, we manually inspected the branches that could not be covered. A core reason for not covering a branch was that the objects under test or the target method parameters are not initialized with the right values. In some cases, this came from the amplification tool not supporting the parameter's type, e.g., for a class without a public constructor. Then, the tool sets the parameters to null or empty values, which lead to exceptions when trying to generate assertions. We saw that Javapoet's classes have more methods whose parameter types are classes without public constructors, while Stream-lib mostly works with simple data types for the parameters. As the amplification tool's implementation does not support initializing classes without public constructors, this could explain why the amplification is more effective on Stream-lib than on Javapoet. Similarly, generating tests for faults or locations that require complex input objects is challenging for search-based tools like EvoSuite [11].

We investigated whether the choice of the original test impacts the ability to cover a certain branch. For this, we sampled ten branches that were not covered by the amplification and also not the existing test suites. Then, we amplified all tests in

[2]https://github.com/square/javapoet

[3]https://github.com/addthis/stream-lib

the corresponding test class, but still could not generate tests that cover the sampled branches. This shows that selecting different initial tests likely does not impact how effective the test amplification is at covering the sampled branches. The earlier mentioned likely cause for not covering the branches, not being able to generate the right initialization for the objects under test, seems to not be solved by selecting different initial tests.

Table II shows how many of the tests generated in one run of guided and open test amplification cover the targeted branch. While the ratio of tests that successfully cover the targeted branch with open test amplification is only 24% for Javapoet and 45% for Stream-lib, for guided test amplification this ratio is 70% for both projects. These results show that the guided test amplification is substantially more likely to produce tests that cover the targeted branch. This indicates, that the simple guidance we implemented into the guided test amplification—calling the method containing the targeted branch—is indeed effective at guiding the test amplification towards our target. Therefore, using guided test amplification enables us to set the amplification to generate fewer tests, while still having a good chance at receiving a test that covers the targeted branch.

TABLE II: Average ratio of successful tests, which cover the targeted branch (see Equation (2)).

| | Javapoet | Stream-lib |
|---|---|---|
| Open Test Amplification | 24% | 45% |
| Guided Test Amplification | 70% | 70% |

Looking how the ratio of successful tests is distributed over the sampled, targeted branches (Fig. 4), we see clear differences between the projects. While in Javapoet the distributions are dense and the higher effectiveness of guided test amplification is clearly visible, for the Stream-lib project the ratio of tests successfully covering the targeted branch differs much more significantly from branch to branch. One possible explanation for this difference is that number of methods in Javapoet's classes is higher than in Stream-lib. This means that it benefits more from the modification in guided test amplification that explicitly calls the method under test before the further input mutation.

### C. Answer to RQ1: How effective does guided test amplification generate tests for targeted branches (compared to open test amplification)?

Summarizing the results of our technical case study, we can see that **guided test amplification is more effective than open test amplification** when covering a specific targeted branch. However, both approaches fail to cover the majority of the sampled branches and depending on the project there can be a large variety in the ratio of generated tests covering the targeted for both approaches. We will discuss and interpret these observations together with the insights from our user study in Section V.

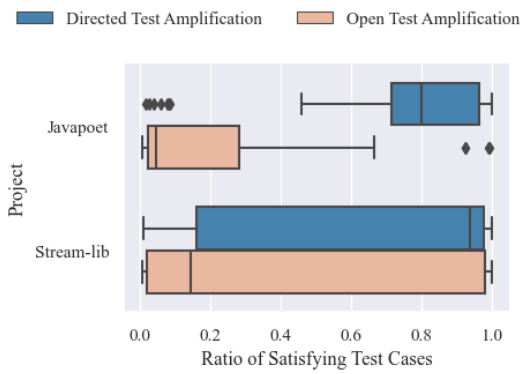


Fig. 4: Distribution of the ratio of successful tests (see Equation (2)).

### D. Design User Study

Our central ideas for guided test amplification were motivated by the interaction with the user: the developer initiates the test amplification and guides it towards a method and branch, reducing the search space for new tests. In addition, this should help the developer understand and review the generated tests, because they already built up the necessary mental task context of the method unter test [10]. To elicit the opinions of developers on the use of guided test amplification in comparison to open test amplification, we conduct a study.

The user study starts with a questionnaire collecting demographic information and informed consent from each participant. Then, the participants are introduced to the concept of test amplification and asked to generate tests for two classes with similar complexity taken from the open source project Stream-lib. Each developer applied both open and guided test amplification, and we equally shuffled the order of the approaches and which class they test according to the four groups in Table III. After the participant solved both tasks, we conduct a semi-structured interview. Guided by a list of closed questions (see Figs. 5 and 6) we ask the participants to reflect on their experience with the open and guided test amplification, to compare both approaches and to express their overall impression of the amplified tests.

We conducted the study fully remotely in sessions of 60 to 90 minutes. We recruited 12 participants through convenience sampling in our professional networks and on social media. You can find the complete tasks and questionnaires in our online appendix [12]. Our study design was approved by our local ethics review board.

### E. Results User Study

From the demographic questionnaire, we learn that we have a relatively young population of 12 participants with a development experience of one to three years (7), four to six years (4) and seven to nine years (1). Two of the participants had used an automatic test generation tool before. Their main programming languages were Python (6), Java (4), or C++ (3),

TABLE III: Task ordering for our participant groups.

| Group | First Task | Second Task |
|---|---|---|
| 1 | User-Guided Test Amplification *StreamSummary* | Open Test Amplification *ConcurrentStreamSummary* |
| 2 | User-Guided Test Amplification *ConcurrentStreamSummary* | Open Test Amplification *StreamSummary* |
| 3 | Open Test Amplification *StreamSummary* | User-Guided Test Amplification *ConcurrentStreamSummary* |
| 4 | Open Test Amplification *ConcurrentStreamSummary* | User-Guided Test Amplification *StreamSummary* |

and they mainly identified as working in general software development (4), research (2) or data and analytics (2).

*1) Guided Test Amplification:* Looking at the feedback regarding the guided test amplification, presented in Fig. 5, the participants strongly agree that the control flow graph showing the coverage of the target method is easy to understand (Q1). When asked whether the information provided is valuable, the participants strongly agree (Q2) and point out that the primary value is in visualizing the code structure and coverage, especially when the complexity of the method under test is high. Question (Q3) centers around whether the control flow graph effectively lets the participants convey their expectation of what to cover to the amplification. On average the participants agree to this, pointing out that it also helps identify all scenarios that are possible when calling the method under test.

They agree that the same visualization is also easy to understand when it comes to showing the coverage of an amplified test (Q4), and helps to select which amplified test to keep and add into the test suite (Q5). In this selection process, the visualization was especially helpful when the amplified tests provided diverse coverage contributions in methods with many branching points. Two participants were neutral about using the control flow graph to select a test, pointing to that they only want to cover the previously selected branch and rather focus on the code of the amplified test instead when selecting or add the test without further inspection.

*2) Open Test Amplification:* When it comes to the open test amplification, our study participants are more divided, but on average agree that the text-based instruction coverage explanation is easy to understand (Q6, Q7) and provides useful information (Q8). The main complaints were that listing each occurrence of new instruction coverage was too detailed and that the connection between the test and the covered instructions was not clear even with the provided hyperlinks. The participants that were positive found the class and method names informative and liked that the hyperlinks let them locate the code under test conveniently. We asked whether the provided information about the amplification mutations in the test (Q9) and the additional coverage (Q10) helped the developers select which test to keep. The participants on average agreed that the additional coverage is helpful to select which test to keep (Q10). However, they criticized that they could not see the existing coverage to judge if a line in the code under test is already covered or not. One participant also thought out loud about whether the provided coverage is actually important coverage.

*3) Both Approaches Compared:* After discussing each amplification approach separately with our participants, we asked several questions to compare both approaches (see Fig. 6). Directly asked whether the instruction coverage of open test amplification or the branch coverage of guided test amplification is easier to understand, all participants prefer the branch coverage (Q13). The participants found it easier to map the branch coverage to the source code structure. Some were also not familiar with the concept of instruction coverage and struggled to identify the single instructions in a line of code. Most participants prefer the visualized control flow graph over representing coverage as highlights in the editor (Q14). Using the visualization they did not need to read the source code of the method under test.

We asked the developers to reflect which approach helps them more during test generation (Q16) and they were divided between the two approaches. Seven participants prefer the guided test amplification as it is closer to writing tests in real-life scenarios, where they focus on specific features to cover. Two participants prefer open test amplification: one proposes to use it early in the test creation process to cover as much code as possible, the other focuses on connecting a new test with the existing ones it is based on, which is clearer during open test amplification. Three participants were neutral and voted to combine the two approaches. When they do not have a specific coverage goal they would use open test amplification, while they would choose the guided test amplification when they aim for more control over each tests' coverage.

Regarding selecting which resulting test to incorporate into the test suite, the participants mainly prefer the guided test amplification (Q15). The ten participants voting for guided test amplification mention that when writing tests they usually have a specific feature in the code they want to cover, which they can achieve by guiding the test amplification. One participant prefers open test amplification as they focus on covering the whole project as much as possible and want to compare the different tests based on their total contributed coverage. One participant is neutral and would use both approaches depending on the situation.

Finally, we asked about their overall impression of the amplified test, which was positive (Q11, Fig. 5). The participants on average strongly agree that they would use test amplification again (Q12) and gave a variety of suggestions on how to improve the tools for both test amplification approaches. One aspect they noted positively is that the tool clearly indicates when it could not generate a test for a selected branch, which made these situations less negative in the participants' opinion.

### *F. Answer to RQ2: How do developers perceive guided test amplification (compared to open test amplification)?*

Looking at all the results of our user study, we see that a majority of our **participants prefer the user-guided test amplification** approach (Q16) because it **fits better into the typical situation they create tests in**: when they want to test a specific location in their code. Factors contributing to

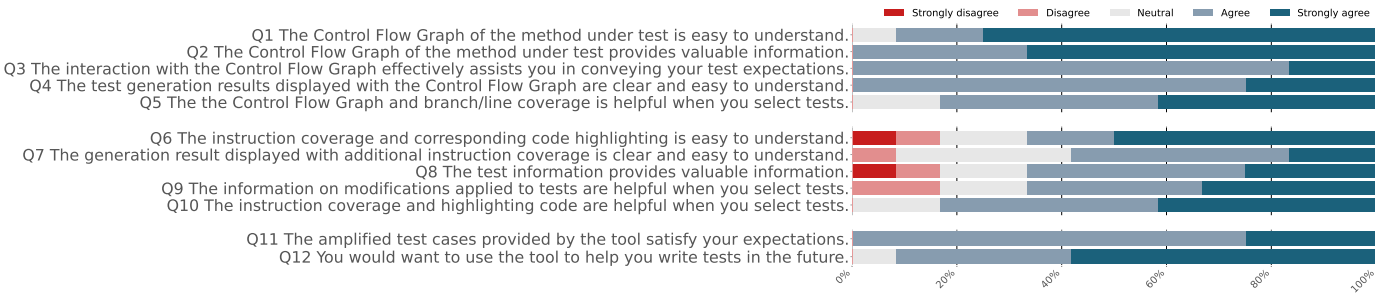


Fig. 5: Participant answers on each of the two amplification approaches and test amplification in general.

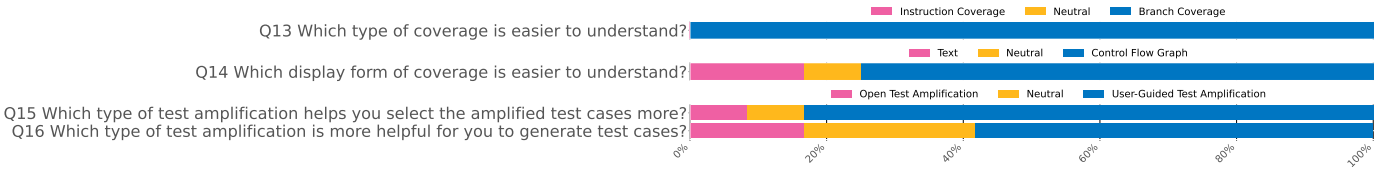


Fig. 6: Participant answers on comparing user-guided and open test amplification.

this judgement are that all participants found branch coverage easier to understand than instruction coverage (Q13), and most preferred the structure-revealing control-flow graph visualization over the more precise textual representation of additional coverage (Q14). This preference for user-guided test amplification is also supported by the overall more positive ratings in the detailed questions about the approach (Q1-5), compared to the detailed questions about open test amplification (Q6-10).

From the explanations of our participants we learned that they do not universally prefer user-guided test amplification over open test amplification, but that it depends on their use case, the information that they need to judge the amplified tests and the amount of control they want to have over the amplification process. The results of our technical study showed that the effectiveness of guided test amplification compared to open test amplification depends on the class structure in the code under test and the data types used as parameters. Taken together, we see that there are **trade-offs between the two approaches** that should be considered when choosing either to work with or to improve in future research. In Section V we collect these trade-offs and discuss the implications of them for practitioners and researchers.

### G. Threats to Validity

There are several threats to the validity of our two studies and their results. When it comes to *internal validity*, we mitigated the threats by switching the order of the two approaches (threat: learning effect) and which class each approach was applied to (threat: dissimilar classes) equally over the four randomly-assigned participant groups. The characteristics of the two projects and their classes in our technical study could dictate the outcome of our technical study. To mitigate this, we manually analyzed the classes and transparently discuss the impact of the number of methods per class and the complexity of the used data types on the effectiveness comparison of the test amplification approaches. To ensure the *confirmability* of our user study results, we focus on presenting the closed question ratings and support them with explanations staying as close as possible to the participants' formulations.

Regarding *construct validity*, the results of both studies are influenced by our prototype implementations. We used the same test amplification tool for both approaches, which is based on DSpot and limited to Java, with the only differences in implementation described in Section III. Another threat is whether we are measuring the effect of the different amplification approaches or the changed user interface (UI) from open to user-guided test amplification. We agree with the original creators of developer-centric test amplification [7] that a tool for developers and its UI can fundamentally not be developed or studied in isolation. To mitigate this threat, we ask separate questions about the information and the UI elements to our participants (Q1/2, Q4/5, Q6/10, Q13/14).

The *external validity* of the results from our technical study is threatened by the two projects selected for the case study. We observed that the complexity of the used data types and the number of methods in a class influence the effectiveness of the test amplification. Further studies on a larger variety of projects and classes are needed to demonstrate the generalizability of our findings. Another threat to the external validity of our user study is whether the participants experienced the whole variety of methods which to test with amplification. To mitigate this, we selected example classes with a varied complexity of methods and initial tests that cover some methods of the class fully, partially or not at all. In the user study we have participants from a range of different software domains, but no participant has more than ten years of development experience, making the results potentially not generalizable to very senior developers.

## V. Discussion and Implications for Practitioners and Researchers

With designing user-guided test amplification, we set out to improve the effectiveness of the process and the understandability of the produced tests. Our technical case study indicates that user-guided test amplification is indeed more effective, and the user study suggests that developers find its components more understandable than those of open test amplification. However, we also saw that the effectiveness of each approach varies per project and class, and that the developers might prefer different test amplification approaches depending on their current goal with testing. In this section, we will discuss a series of trade-offs that we identified based on our two studies and the design of both amplification techniques. Table IV gives an overview of these trade-offs, together with the source from which we take the answer for either technique.

The two amplification approaches **fit two complimentary use cases** for software developers. From the participants reflecting on which approach is more helpful to generate tests (Q16), we learned that the user-guided version is better suited when they write tests in conjunction with the production code, also called test-guided development [3], [13]. When their focus is to improve the test suite itself, e.g., to address technical test debt [14]–[17], open test amplification would be the better choice. This is because it connects an amplified test clearer to the original test from the test suite by pointing out the applied input modifications.

Open test amplification also informs the developer about the **coverage impact of an amplified test** across the whole project [7]. With the high prevalence of integration tests in JUnit test suites [18], [19], tests amplified from them can improve test coverage in several locations throughout a software project [9]. Because this scattered coverage information can be confusing [7] and partially irrelevant to developers [9], user-guided test amplification focuses only on the impact in the targeted method. In return, it can use the available room to convey the stronger metric of branch coverage in a simple and easy to understand visualization (Q14).

A previous study on the interaction of software developers with test amplification showed the importance of **managing the users' expectations** and making sure they align with what the tool can provide [7]. Open test amplification only proposes tests for locations it can actually cover, so it can easily fulfill the user's expectations for receiving tests. In our proposal of user-guided test amplification the developers can select any branch as a target, but as we saw in the technical study, more than half of the branches in our study projects could not be covered. This might disappoint the user and not meet their expectations. When the participants of our study encountered this, they however were positive about the fact that the tool clearly reported that it could not generate a test (participant reflection on Q12). To address the low success rate of guided test amplification, we would need to initialize the objects and parameters correctly to hit the targeted branch (manual inspection technical study). Advanced techniques like concolic execution [20]–[22], or search-based optimization [23] could address this. However, these can be expensive to compute.

When studying the **effectiveness** of test amplification in our technical study, we saw that guided test amplification produces a higher ratio of tests that successfully cover the targeted branch. This highly fits the use case of testing the developer's current focal method. In contrast, the more explorative search in the whole method space of a class under test that open test amplification performs is more effective when the goal is to improve the coverage across the whole class. Someone who uses guided test amplification for this would need to invoke it over and over again for each method in the class.

### A. Implications for Practitioners

Our evaluation of user-guided and open test amplification uncovered a set of trade-offs a software developer or their manager should consider when choosing which approach to apply. The main, reoccurring consideration is *why* someone wants to generate tests: (1) to improve the test suite itself (choose open test amplification), or (2) to get support for writing tests while working on a specific part of the production code (choose user-guided test amplification). Beyond this, our study also shows anecdotal evidence that when a code base contains many complex classes with private constructors, test amplification with our state-of-the-art tool will likely not be able to cover many branches.

### B. Implications for Researchers

For researchers in the area of test amplification and generation, as well as developer-centric support tools, the insights from our study point to several new research directions.

Improving the effectiveness of guided test amplification asks for more advanced techniques to initialize objects to cover the targeted branch. Can we apply computationally expensive techniques while still providing an interactive user experience?

Could we actively ask the developer to help us with the initialization of objects that are hard to create? Here the question is whether they would know enough to provide a valuable initialization and whether the automation would still be worth it to use for the developer if they would have to contribute such substantial effort to the test generation.

Many decisions in the design of either test amplification approach are motivated by the required interactive speed. Would it be feasible to pre-generate tests in the background and then selectively present relevant ones to the developer when they request tests? A complication here is that current developer-test generation approaches like test amplification or search-based generation with EvoSuite [24], require the code under test to be available. However, we observed repeatedly in our user study that developers are looking for tests covering the code they just wrote a short while ago.

Why did the participants of our study prefer the control-flow graph visualization of the branch coverage over the bytecode instruction visualization of the line coverage? Based on our observations, we conjecture that the following aspect could influence this: (1) using a coverage metric that is embedded

TABLE IV: Trade-offs between user-guided and open test amplification.

| | User-Guided Test Amplification | Open Test Amplification |
|---|---|---|
| Fits use case | Writing production code & wanting tests for it [participant reflection on Q16] | Improving test suite and resolving technical debt [participant reflection on Q16, [7]] |
| Understand coverage contribution and test execution | In targeted method in detail [Q4, design user-guided test amplification] | Across the whole project [9] |
| Expectation of receiving tests | Might disappoint if targeted branch cannot be covered [Technical study] | Only proposes tests / additional coverage it can provide [design open test amplification [7]] |
| Runtime efficiency | More effective at providing tests for method of interest [Technical study] | Can provide larger coverage variety of tests for whole class [7] |

in the developer's mental structure of the code, (2) limiting the scope of the displayed code coverage to just the one method the developer is concerned about, and (3) presenting the existing coverage in conjunction with the additionally provided coverage, letting the developer grasp the differential impact a new amplified test makes.

## VI. Related Work

In this section, we discuss related work from the areas of directed and interactive test generation.

### A. Directed Test Generation

Search-Based Software Testing (SBST) uses search algorithms to automatically find tests that a variety test objectives captured in a fitness function [25]. SBST has been used to automate test generation for various test goals, such as maximizing structural coverage [26]–[30] and crash reproduction [23], [31], [32].

Test suite augmentation techniques are used to generate tests that target code changes that the existing test suite does not cover [33]. Xu et al. proposed several approaches for test augmentation using concolic testing [34], genetic algorithms [35], and a combined, hybrid approach [36], [37]. In their concolic approach, they find the source node of a changed branch and select existing tests that reach this source node. Then they explore different directions of path conditions to find new tests for the changed branch. Their genetic algorithm uses a fitness function that prefers the distance of a test's execution to the changed branch. In contrast to their approach, our test amplification focuses on all uncovered branches of a software, not just the recently changed ones. Further, our approach is simpler, as we only select a few initial tests and only amplify them with one evolutionary iteration.

Several researchers focused on generating targeted tests to support debugging. Ma et al. propose directed symbolic execution, using the distance to the target line as information to guide the symbolic execution [38]. Dinges et al. [39] combine symbolic execution, to find a suitable entry point to reach a target statement, with concolic execution and heuristics, to try to satisfy constraints too difficult for the symbolic execution. Our approach makes use of the existing tests as a basis for the amplification, and we do not use symbolic execution to reduce our computational costs.

### B. Interactive Test Generation

Several techniques are discussed to incorporate information provided by humans into the test generation process. Marculescu et al. proposed Interactive Search-Based Software Testing (ISBST) to involve domain specialists in test generation [40]. Their feedback adapts the fitness function during the search process by changing the relative importance of system quality attributes. The primary difference between their work and ours is that they involve domain specialists in the test generation, while we target software developers. They pointed out the importance of perfecting how automated test systems communicate with users and ensuring that results are understandable to the users when transferring ISBST to industry [41]. We address this in the design of our interface, visualizing information about the test amplification results to help the user's comprehension.

Murphy et al. propose to apply grammatical evolution into SBST and incorporate human expertise into the search [42]. They proposed that users can define the search space they want their tests to be created from by specifying a grammar. Ramírez et al. observed two key issues hindering the acceptance of automated tests by analyzing various studies that evaluated the effectiveness and acceptance of test generation tools [43]: the opacity of the generation process and the lack of cooperation with the tester. To address this, they incorporate the tester's subjective assessment of readability to compare tests with the same fitness in a search-based test generation process. Our work also addresses the concerns Ramírez et al. raised. We cooperate with testers and make the process transparent by letting testers express their branch coverage goal and guide the test generation. We also improve the understandability of tests by connecting the amplified tests with testers' coverage goals.

## VII. Conclusion and Future Work

The aim of user-guided test amplification was to ease the effort for software developers when understanding amplified tests, by letting them point the test generation to a specific target branch and then visualizing the resulting coverage leveraging a control flow graph of the method under test. Through our technical case study, we show that even simple modifications to the amplification process make guided test amplification more effective at generating tests for a

targeted branch. Our user study shows that developers prefer the interaction with user-guided test amplification, but that the choice for either technique is dependent on the current use case of the developer. From our studies and the design of both approaches, we identify and discuss four trade-offs that influence the choice between open and user-guided test amplification: (1) the current task and goal of the developer, (2) where the amplified test should provide coverage, (3) the ability to fulfill the user's expectation to receive a generated test, and (4) the available time for the test amplification.

Beyond the research implications we mentioned earlier, our work can be the basis for several future research directions:

We observed the developer's wishes to generate tests while they are working on a particular piece of code. While user-guided test amplification is a step in this direction, the next step would be to detect when a developer has finished a change, and automatically generate and propose a test for the code change to the developer.

The feedback on the coverage visualization showed that it helps developer to understand test coverage better. On the other hand, the expectations of the user guiding the amplification now requires more advanced test generation approaches that are already available in other tools. The next step, would be to disconnect the test generation tool from the interaction layer that proposes the tests to developers. This allows for more flexibility in choosing the test generation tool that is right for the job while still benefitting from the continued advancement in test communication.

## REFERENCES


[1] K. L. Beck, *Test-Driven Development - By Example*, ser. The Addison-Wesley signature series. Addison-Wesley, 2003.

[2] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.

[3] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.

[4] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *J. Syst. Softw.*, vol. 157, p. 110398, 2019.

[5] STAMP, "Use cases validation report v3," https://github.com/STAMP-project/docs-forum/blob/master/docs/, 2019.

[6] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with DSpot: A study with ten mature open-source projects," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 2603–2635, 2019.

[7] C. Brandt and A. Zaidman, "Developer-centric test amplification," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 96, 2022.

[8] S. Bihel and B. Baudry, "Adapting amplified unit tests for human comprehension," *KTH Internship Report*, 2018.

[9] C. Brandt and A. Zaidman, "How does this new developer test fit in? A visualization to understand amplified test cases," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022, pp. 17–28.

[10] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Softw. Qual. J.*, vol. 19, no. 1, pp. 5–34, Aug 2010.

[11] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE CS, 2017, pp. 263–272.

[12] Anonymous, "Online appendix for "when to let the developer guide: Trade-offs between open and guided test amplification"," https://doi.org/10.5281/zenodo.8074647, Jun. 2023.

[13] A. Santos, S. Vegas, O. Dieste, F. Uyaguari, A. Tosun, D. Fucci, B. Turhan, G. Scanniello, S. Romano, I. Karac, M. Kuhrmann, V. Mandic, R. Ramac, D. Pfahl, C. Engblom, J. Kyykka, K. Rungi, C. Palomeque, J. Spisak, M. Oivo, and N. Juristo, "A family of experiments on test-driven development," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 42, 2021.

[14] E. da S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *7th IEEE International Workshop on Managing Technical Debt (MTD)*. IEEE Computer Society, 2015, pp. 9–15.

[15] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[16] Z. Codabux and B. J. Williams, "Managing technical debt: An industrial case study," in *4th International Workshop on Managing Technical Debt (MTD)*. IEEE Computer Society, 2013, pp. 8–15.

[17] G. Samarthyam, M. Muralidharan, and R. K. Anna, "Understanding test debt," *Trends in Software Testing*, pp. 1–17, 2017.

[18] F. Trautsch, S. Herbold, and J. Grabowski, "Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects," *J. Syst. Softw.*, vol. 159, 2020.

[19] J. Van Geet and A. Zaidman, "A lightweight approach to determining the adequacy of tests as documentation," *Proc. PCODA*, vol. 6, pp. 21–26, 2006.

[20] K. Sen, "Concolic testing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 571–572.

[21] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java," in *First NASA Formal Methods Symposium (NFM)*, ser. NASA Conference Proceedings, vol. NASA/CP-2009-215407, 2009, pp. 121–125.

[22] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed unit test generation for C/C++ using concolic execution," in *35th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2013, pp. 132–141.

[23] P. Derakhshanfar, X. Devroey, and A. Zaidman, "Basic block coverage for search-based unit testing and crash reproduction," *CoRR*, vol. abs/2203.02337, 2022.

[24] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[25] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[26] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and 13th European Software Engineering Conference (ESEC)*. ACM, 2011, pp. 416–419.

[27] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *32nd IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 281–284.

[28] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: an open source tool for search based software testing of C programs," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 112–125, 2013.

[29] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce, "Using relative lines of code to guide automated test generation for python," *CoRR*, vol. abs/2103.07006, 2021.

[30] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2015, pp. 1–12.

[31] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, "Single-objective versus multi-objectivized optimization for evolutionary crash reproduction," in *10th International Symposium on Search-Based Software Engineering (SSBSE)*, ser. LNCS, vol. 11036. Springer, 2018, pp. 325–340.

[32] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for java," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1278–1282.

[33] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig, "Automating test-suite augmentation," in *14th International Conference on Quality Software*. IEEE, 2014, pp. 67–72.

[34] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *18th ACM SIGSOFT*

*International Symposium on Foundations of Software Engineering*. ACM, 2010, pp. 257–266.

[35] Z. Xu, M. B. Cohen, and G. Rothermel, "Factors affecting the use of genetic algorithms in test suite augmentation," in *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2010, pp. 1365–1372.

[36] Z. Xu, Y. Kim, M. Kim, and G. Rothermel, "A hybrid directed test suite augmentation technique," in *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 2011, pp. 150–159.

[37] Z. Xu, Y. Kim, M. Kim, M. B. Cohen, and G. Rothermel, "Directed test suite augmentation: An empirical investigation," *Softw. Test. Verification Reliab.*, vol. 25, no. 2, pp. 77–114, 2015.

[38] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *18th International Symposium on Static Analysis (SAS)*, ser. LNCS, vol. 6887. Springer, 2011, pp. 95–111.

[39] P. Dinges and G. A. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 31–36.

[40] B. Marculescu, R. Feldt, and R. Torkar, "A concept for an interactive search-based software testing system," in *4th International Symposium on Search Based Software Engineering (SSBSE)*, ser. LNCS, vol. 7515. Springer, 2012, pp. 273–278.

[41] B. Marculescu, R. Feldt, R. Torkar, and S. M. Poulding, "Transferring interactive search-based software testing to industry," *J. Syst. Softw.*, vol. 142, pp. 156–170, 2018.

[42] A. Murphy, T. Laurent, and A. Ventresque, "The case for grammatical evolution in test generation," in *Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2022, pp. 1946–1947.

[43] A. Ramírez, P. Delgado-Pérez, K. J. Valle-Gómez, I. Medina-Bulo, and J. R. Romero, "Interactivity in the generation of test cases with evolutionary computation," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 2395–2402.