Incremental Just-In-Time Test Generation in Lock-Step with Code Development

Carolin Brandt
Delft University of Technology
c.e.brandt@tudelft.nl

***Abstract*—State-of-the-art test generation strategies employ advanced analyses of the code under test and powerful optimization algorithms to generate automatic test cases for software systems. As these techniques require a large amount of computational power, they are often limited to generating tests after the code under test is already written. However, today's broad education about the importance of software testing lets developers strive to create test cases directly with new code they are contributing.**

To support these developers, we want to develop an incremental just-in-time test generation tool that works in close proximity to the development of the code under test. Whenever the developer creates a new class or functionality, the tool automatically proposes a matching test case. When the developer finishes implementing a new condition, the tool automatically recommends an additional test case that tests the code which was just added. The generated test cases are closely based on the existing test cases in the project with small, incremental changes to test the new lines of code.

To realize such a just-in-time test generation tool we have to tackle many challenges: Detecting the completion of a test-worthy condition, generating a fitting test case in a short time on the developer's machine, or effectively communicating the value of the new test case to the developer. With the participants of the SMILESENG Summer School we want discuss our new idea, brainstorm on the challenges that this research opens up and identify possible approaches to tackle them.

***Index Terms*—Software Testing, Automatic Test Generation, Test-Guided Development, Developer-Centric Design**

I. Introduction

To illustrate our idea of just-in-time test generation, let us introduce this anecdotal use case: Think of Jada, a software engineer in a large software development company. Her team is working on an app for public transport trips and tickets. The transport provider decided to introduce a new promotion: In the summer months of 2022, each monthly pass will cost only nine euros. Today, Jada's task is to adapt the ticket selection algorithm to propose the new ticket whenever the normal cost of a trip would be more than nine euros. Jada opens the code for the ticket selection component and adds a new condition comparing the trip price to the promotional ticket price. After finishing this new edge case, a notification pops up in the corner of her editor:

> "Do you want to add a test that a summer ticket is proposed when it is cheaper than the normal fare?"

As their project policy requests all new code to be fully tested, she is relieved to not have to write a test from scratch. She selects "Inspect Test" and the editor opens on the test class of the ticket selection component. The new test is already added to the source code and a green indicator shows her that the test is passing. Jada reviews the new test case and is pleased that here addition seems to work as she intended: An expensive route—initialized just as in the other tests—is passed to the selection, which then returns the summer ticket instead of the normal ticket. She accepts the new test case and commits it together with her changes. When creating the pull request, she can be confident that all changes are already covered by the test suite. And that mostly automated, thanks to the just-in-time test generator!

This is one use case we envision for our new technology. The just-in-time test generator closely follows the software developer's actions, identifies testable, test-worthy and finished scenarios, quickly generates matching test cases in the background, and immediately presents these test cases to the developer. The developer inspects the new test cases, modifies them where they see fit and takes them over into their maintained test suite.

Placing the test generation so close to the code development, provides advantages in several known challenges of automatic test generation:

- It narrows the search space by focusing the generation efforts on the just modified code.
- It makes it easier for developers to understand the behavior and coverage impact of the new test case, as they are still in the mental context of the code under test [1].
- By generating the oracle through executing the just modified code, the developer receives immediate feedback on the actual behavior of their code and whether it matches with the behavior they intended.
- This approach widens the application area of automatic test generation to the initial development of code and directly supports developers that aim to write test cases in conjunction with their production code (test-guided [2] or iterative test-last development [3])

II. What We Can Build Upon: Related Work

The idea of just-in-time test generation is closely related *test suite augmentation* [4]: adding new test cases to an existing test suite in order to improve its code coverage. Code-to-test traceability approaches [5], [6] can identify test cases that execute code close to the just modified code and with the help of symbolic execution and similar techniques we can

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

modify these base test cases to exercise the new scenario [7]. Powerful *search-based test generation* tools like EvoSuite [8] are already able to generate whole test suites from scratch. To apply them to just-in-time test generation, one would need to investigate how existing test cases could be used effectively as an initial population and how well the limited search space can improve the runtime/generation quality towards interactive performance. *Test amplification* [9] generates new test cases by mutating the input stage of the test case and generating assertions matching the new test behavior. In a previous study, we investigated the interaction of software developers with automated test amplification [1]. We saw that it is important to give the developer control over the interaction, provide them with the information necessary to judge the test cases and effectively communicate the impact that the generated test case will have on the quality of their test suite. Taking into account the results of further user studies of test generation tools [10]–[12], we conjecture that a strong focus on the design of the user interaction is crucial for our just-in-time test generation approach to be successful.

Machine learning approaches are gaining popularity also in the area of automatic test generation, e.g. to generate assertion statements [13], [14]. Recently, neural code completion tools such as GitHub Copilot[1] are able to propose fully fledged implementations when triggered with a natural language method name. Nonetheless, it remains to be investigated how effective they are in generating useful test cases and which information needs to be encoded in the trigger to steer the generation towards the intended code under test.

III. Divide and Conquer: Steps to Tackle

On the way to fully-fledged just-in-time test generation we see several challenges to be addressed. These could be the basis for our discussion at the SMILESENG summer school, together with the following questions:

> Which further challenges do you see as part of realizing just-in-time test generation?
> What approaches should we explore to tackle them?
> What chances or caveats do you see with these approaches?

- **Cutting out a *test-worthy* condition / scenario** as a target for the test generation. Approaches could be detecting coherent edits made by the developer, and learning from single-concern commits or the coverage of existing test cases. Together with the following test generation, this should be a deterministic and transparent process to let the developer build trust and understanding in the capabilities of our tool.
- **Detecting the right moment to contact the developer** as well as giving them control to start and feed the tool themselves.
- **Rapid on-device generation** to enable interactive cooperation between developer and tool. To speed up the test generation, we propose to leverage *incrementality*: Building upon existing test cases by modifying them only slightly. In addition, we look at incremental compilation and building to speed up a run of a test generation tool, as well as explore ways to avoid the expensive executions of test cases to measure adequacy metrics such as mutation score or structural coverage.

[1]https://copilot.github.com/

IV. An Outlook into the Future

The possible applications of a just-in-time test generation tool and its components will be much wider than just adding test cases for new conditions in the code. We could propose immediate updates to no-longer-passing test cases before the developer reruns their tests after a change. We could determine no longer needed test cases after large code cleanups and propose suiting re-locations of test code after refactorings. Integrating test generation right within the developer's workflow lets them become familiar with the advantages test generation can offer and lets them gain trust in the capabilities of (partially) automated software engineering.

While just-in-time test generation is still many steps away, the development of each its subparts helps us strengthen and better understand the area of automatic test generation and its interaction with software developers. We are excited to present our idea to the SMILESENG Summer School participants and together discuss approaches to tackle it.

References

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

[2] M. Beller *et al.*, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Softw. Eng.*, vol. 45, no. 3, 2019.

[3] A. Santos *et al.*, "A family of experiments on test-driven development," *Empir. Softw. Eng.*, vol. 26, no. 3, 2021.

[4] R. Bloem *et al.*, "Automating test-suite augmentation," in *2014 14th Int. Conf. on Quality Softw.* IEEE, 2014.

[5] N. Aljawabrah *et al.*, "Understanding test-to-code traceability links: The need for a better visualizing model," in *Computational Science and Its Applications - 19th Int. Conf.* Springer, 2019.

[6] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *13th European Conf. on Softw. Maintenance and Reengineering*. IEEE Computer Society, 2009.

[7] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *18th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng.* ACM, 2010.

[8] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *m19th ACM SIGSOFT Symp. on the Foundations of Softw. Eng.* ACM, 2011.

[9] 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, 2019.

[10] M. M. Almasi *et al.*, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *39th Int. Conf. on Softw. Eng.: Softw. Eng. in Practice*. IEEE Computer Society, 2017.

[11] S. Panichella *et al.*, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *38th Int. Conf. on Softw. Eng.* ACM, 2016.

[12] J. M. Rojas *et al.*, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *2015 Int. Symp. on Softw. Testing and Analysis*. ACM, 2015.

[13] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *42nd Int. Conf. on Softw. Eng.* ACM, 2020.

[14] H. Yu *et al.*, "Automated assertion generation via information retrieval and its integration with deep learning," in *42th Int. Conf. on Softw. Eng.*, 2022.