Gil Guday & Britt Henderson
gilg & hendeb96

SIAT: Simultaneous Alternative Implementation Testing

At some points during the course of programming a project, it is advantageous to try out different implementations of specific elements. This can range in scope from something simple such as using a different library's function call in a specific method, to something complex such as restructuring a project entirely. This can also range in purpose, such as trying to identify where a bug is causing unexpected behavior, to optimizing an algorithm for time and/or memory constraints. The concept that we are proposing serves as a means of addressing most of the range of needs described, and automating the process as much as possible to make it easier and streamlined from the perspective of the programmer.

Typically, there are two different approaches that we are familiar with; one is to continuously comment out and/or replace blocks of code throughout the course of development to test out different implementations, and the other is working with version control software such as git and creating different branches/commits for different variations. Both approaches don't quite meet in the middle well in terms of scope. For small issues and comparisons that don't require a lot of rewriting and maintenance, temporarily commenting out and replacing code can work adequately, but this approach does not scale well, and it is easy for code to become cluttered, messy, and unorganized especially in the event that multiple different points in the code have alternative implementations which might be tied to one another. Working with version control software is great for this latter case when the scope and scale of changes is large, but it's also not very feasible to explore more than a small limited number of alternatives at a time using this approach without creating a large number of commits/branches, which leads to a messy repository that is difficult to track through later. It also doesn't allow for much simultaneous exploration since it would require comparison across commits, which can be a difficult prospect as far as the git setups and rollbacks that may be required.

The limitations of our idea are that it most likely will be an extension to a testing framework such as Junit rather than a standalone, and for multiple complex implementation alternatives, especially in the case of multi-class or multi-method alternatives, it can be hard to infer how to automate the restructuring in ways that make sense. It is also unclear at the moment how to handle issues such as dependencies and inheritance that may change between implementations.

Because there are multiple uses for this type of project, it could be a beneficial tool anywhere from students working on assignments trying to optimize path finding algorithms or hunting for a bug hiding somewhere in a convoluted stack trace, all the way up to optimizing large scale machine learning problems for time or space.

The current approach we are thinking of is to give the programmer access to a scratch file(s) where they can simply dump different methods, classes, etc., then extend Junit framework to find this scratch file and extract all the code chunks from it and match where they could be viable alternatives, primarily through names and signatures. Junit would then conduct tests with different options depending on user specifications; the tests would run across all viable variations of the overall implementation, and report back which variations failed, succeeded, and how fast/memory intensive they were, according to which information the programmer was most interested in.

The biggest challenge that we face is that neither of us are very familiar with the specifics of Junit and how to extend its functionality, so there is a risk that doing so is nigh impossible and we will need to

rethink our approach to target text replacement in the programmer's target code files rather than the cleaner approach of dealing with it through Junit.