Hot Streak and Cold Streak Programming Tools

Michael D. Shah
Tufts University
mshah08@cs.tufts.edu

Abstract

In a variety of professional sports if an athlete completes many throws in a row, scores multiple baskets in a row, or throws many strikes in a row—they are considered to be on a hot streak. Often programmers have a feeling of being "in the zone" when their code is compiling and they are achieving the results they have intended at a high frequency. On the contrary, programmers can enter a "cold streak" if their code is not compiling, addition code changes break a system, or progress cannot otherwise be made. To remedy this situation, we would like to have a system that can help create more "hot streaks" and mitigate "cold streaks" in order to improve a programmer's productivity and reduce programmer errors while using a given programming language.

1. Introduction

Our proposed idea is a system that builds off of compilers diagnostic error messages so that it can be adapted to existing compilers today. We would encourage this system to work with compilers that have standardized error messages generated with tools like Merr [3]. In addition, our system can be combined with user supplied bug patterns—similar to those found in systems such as findBugs [1]. If an error message is supplied, then the programmer must correct this statement by adding one of three items around the error: an assertion, a contract, or a unit test. These three constructs are well-defined and supported directly by many programming languages or with third-party libraries. We believe these tests can be programmer directed, or generated with tools such as DART to automatically generate tests [2].

1.1 Metrics

In order to have a system measure how well a programmer is doing, metrics must be defined that can correlate to how successful their coding session has been. We admit that these metrics may not be perfect, but it will remain a part of the system that can be refined in further work. Our evaluation takes place at a unit level, meaning we are only looking at small parts of an entire system (i.e. a programmer's "hot or cold streak" takes place in only a few blocks of code for the duration of several hours as opposed to an entire code repository over several weeks or months).

1.2 Compiler Errors

All programmers will generate compiler errors, so we do not want to penalize them immediately after each compiler error (that could also potentially break their concentration). Instead, the system needs to tally the context in which error messages occur. In addition to the context in which an error message occurs, we will also want to document how the programmer chooses to try to prevent future errors: assertion, contract, or unit test (or potentially using thereom solvers or other methods in future systems). Table 1 displays example error messages and what information can be parsed from them. Table 2 provides an example of how we tally error messages, and how a cold streak can be identified for a block of code.

2. Implementation

The process of implementing such a system is as follows:

1. Compiler error message is parsed with relevant information.
2. Programmer chooses to implement an assertion, contracts, or unit tests.
3. Program is run again, and our table checks to see if test passes.
4. If test passes continue adding code.
5. If test does not pass, and too many tests have failed with the same context, the programmer is in a "cold streak" and must work out of the "cold streak" as detailed in section 3.

We hope that our system can additionally be complimented with software engineering tools such as:

Code Completion: Build off of code completion systems to include templates for generating unit tests, assertions, and contracts.

Visualizations: Provide a visualization tool that shows code that has been reported by the compiler as not able to compile or previously generated error messages.

Profiling Tools: Extend our system beyond semantic errors, and be able to test for performance in space and time complexity.

3. Working out of a Cold Streak

If a programmer is in a cold streak, we believe there should be methods to help them work back into a successful rhythm. We provide two ideas that are controversial (the second idea in particular) in that they are indirect ways of solving a problem.

3.1 Documentation

We want to force the programmer to document the code they are writing. However, in order for this to be useful we believe we must parse the code they are commenting. If they are creating comments that make reference to the code they are trying to fix, we believe they are heading in the right direction. We believe programming languages that have built-in documentation support can help reduce bugs. Examples include C# with its heavy use of XML[5]. We are also encouraged to see that mismatches between documentation and code can be a source of error here, and will look forward to more results following the work by Rubio- González and Liblit [6].

3.2 Refactor Code

We can also force the programmer to take a block of code that is not working, and make them rewrite it using different programming constructs. Often there is more than one way to complete a task, and a rewrite of an algorithm can be checked to see if the result is semantically the same with some degree of success (several solutions are discussed in [4]). Note that we are not trying to make robust systems, but rather systems that work as intended, even at the cost of performance. If a programmer's goal is to write the same code two different ways and they are unsuccessful, then they should not add to the codebase and refine their algorithm first. We also understand that it may not be possible to write code in two different ways, or it is a tedious process. We will investigate more ideas for solving edge cases that could occur in our system.

When a programmer has performed one of two of these tasks, they can once again be able to work in a section of code. It is not our systems job to lock out sections of code, but rather to inform programmers that they need to understand various parts of a their source code before modifying it. We believe if they do not know how to test code for correctness, this is an indicator that they will not understand how to build correct code.

4. Conclusion and Discussion

We have provided a new idea about a system that would force programmers to take a step back from programming if they are in a cold streak. On the flipside, programmers can have confidence in their coding session if they are not frequently being notified by our system. We believe there are many in the programming and software engineering industry working on productivity tools that can be useful. We also hope that our tool brings to attention the need for programming languages to become more tightly integrated with tools that can increase test coverage.

5. References

[1] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '04). ACM, New York, NY, USA, 132-136.

[2] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). ACM, New York, NY, USA, 213-223.

[3] Clinton L. Jeffery. 2003. Generating LR syntax error messages from examples. ACM Trans. Program. Lang. Syst. 25, 5 (September 2003), 631-640.

[4] Robert Metzger and Zhaofang Wen Automatic Algorithm Recognition And Replacement: A New Approach to Program Optimization

[5] "C# Language Specification." C# Language Specification. Microsoft, 2012. Web. 04 May 2013. <http://msdn.microsoft.com/en-us/library/ms228593.aspx>.

[6] Cindy Rubio-González and Ben Liblit. 2010. Expect the unexpected: error code mismatches between documentation and the real world. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '10). ACM, New York, NY, USA, 73-80.

Error Message	Parsed Information	Error Context	Fix Applied	Error Resolved
Undefined reference to myClass::foo()'	Foo is not implemented	Syntax Error	Contract	No
Warning: 'value' is used uninitialized in this function	Value is uninitialized in function bar	Variable	Assertion	Yes
Warning: 'value' is used uninitialized in this function	Value is uninitialized in function bar	Variable	Contract	Yes

Table 1 –The Error Message is the compiler error or warning returned to the user. The Parsed information is additional information given to the programmer. The Error Context classifies the type of error message into some pre-defined category. The Fix Applied is stored for the programmer to see how they attempted to mitigate the problem. If the Error is resolved in the next run of the program it changes to 'yes', otherwise it remains no.

Error Context	Assertion	Contract	Unit Test
Syntax Error	0/0	1/1	0/0
Variable	1/0	1/0	0/0

Table 2 – Our rows represent all of the categories of errors (the Context from table 1). The columns represent how the user attempted to fix this error. The first tally represents the total number of occurrences of a specific Error Context and fix applied. The second tally in each cell represents the number of times this error was not resolved (e.g. for syntax errors we have found 1 error that was attempted to be resolved by a contract, and 1 of those errors remains unresolved as observed in Table 1). We have not performed user studies, but any ratio that is less than 100% could be considered a cold streak, and thus force our programmer to refactor code or document because of lack of comprehension of the job they are trying to achieve.