Live Programming By Example

Mark Santolucito
Computer Science
Yale University
mark.santolucito@yale.edu

William T. Hallahan
Computer Science
Yale University
william.hallahan@yale.edu

Ruzica Piskac
Computer Science
Yale University
ruzica.piskac@yale.edu

ABSTRACT

Live programming is a novel approach for programming practice. Programmers are given real-time feedback when writing code, traditionally via a graphical user interface. Despite live programming's practical values, such as providing an easier overview of code and better understanding of its structure, it is not yet widely used. In this work, we extend live programming to general purpose code editors, which allows for live programming to be used by programmers, and provides new interfaces for understanding and changing the functionality of code. To achieve this we extended a fully-featured IDE with the ability to show input/output examples of code execution, as the programmer is writing code. Furthermore, we integrate programming by example (PBE) synthesis into our tool by allowing the user to change the shown output, and have the code update automatically. Our goal is to use live programming to give novice programmers a new way to interact and understand programming, as well as being a useful development tool for more advanced programmers.

CHI'19 Extended Abstracts, May 4–9, 2019, Glasgow, Scotland Uk

Figure 1: Code is written in the left hand panel, while examples are shown in the right hand panel.

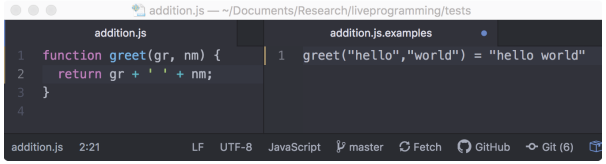

Figure 2: When the code is modified, the examples update in real time. Here, the user has added a space to the output, by editing the code.

Figure 3: The user can also modify the output examples, to repair the code. Here, the user has added a exclamation point to the end of the example's output, resulting in new code that appends an exclamation point. The old code is preserved in a comment.

INTRODUCTION

Traditionally, writing a program is a relatively static process: a programmer writes some code and, after a successful compilation, can observe and inspect its behavior. If the code does not actually implement the programmer's intentions, they can correct the program and repeat the process.

Of course, this cycle often extends over a long period of time. Not all bugs are discovered immediately, and old code often has to be updated to suit new purposes. Unfortunately, documentation is often incorrect or out of date, leaving the best resource for developers as the code itself [10]. In fact, it is estimated that half of a programmers time is spent just *comprehending* previously written code [4].

The live programming paradigm advocates a more dynamic programming cycle that allows the programmer to inspect and understand the code as it is written. As code is written, the user interface gives real-time feedback. While existing live programming environments [2, 3, 16] focus on programs with graphical or auditory output, we focus on general-purpose live programming.

We seek to give programmers immediate feedback as they write code, an understanding of code that was written in the past, and a way to avoid manually writing code altogether. We accomplish these goals through input/output examples. We use *live debugging* to show realtime feedback via changing outputs to fixed inputs as a function is modified. We also leverage recent advances in *programming by example* (PBE) to offer automated repairs based on input/output examples. Programming by example can be a useful technique for novice programmers, for example in a classroom setting [15].

Both live debugging and PBE are also beneficial when *later* modifying the code. As opposed to traditional documentation, the live debugging examples update automatically, and thus will never fall out of date. PBE simplifies modifying legacy code, by allowing programmers to simply demonstrate new behavior. Of course, the programmer must be careful to preserve desired existing behavior, but this can be done by comparing the old and new code, rather than having to write new code manually.

As an implementation, we developed a Javascript live coding plugin for the Atom text editor [7].

LIVE CODING PLUGIN

As shown in Figures 1 to 3, our live programming methodology relies on two panels. Programmers write code in one panel. The other panel displays input/output examples, which are used for both live debugging, and programming by example.

Live Debugging

We introduce *live debugging* as a technique to show realtime feedback as programmers write code. Programmers can specify an arbitrary number of function inputs. As users write code, we continually run the code on the given inputs.By observing changes in the input-output pairs, the user receives immediate feedback about whether the code is correct without actually analyzing it in detail.

As Javascript is an interpreted language, running syntactically correct code is fairly straightforward. Unfortunately, the process of editing code often involves that code being in a malformed, syntactically incorrect state. Thus, we only update the displayed output when the code is, in fact, syntactically valid. When the user closes the file or editor, we write the examples to a metadata file. We reload the examples when the file is opened again.

Programming by Example

Our framework also allows for *programming by example* [5, 6, 9, 11]. PBE is a synthesis technique that automatically generates programs that coincide with given input/output examples. An example is specified as a tuple of input and output values. Given a set $S = \{(i_1, o_1), \ldots, (i_n, o_n)\}$ of input/output examples, the goal is to automatically derive a program P such that for every j, $P(i_j) = o_i$. The success and impact of this line of work can be seen from the fact that some of this technology ships as part of the popular Flash Fill feature in Excel 2013 [8].

When a user modifies an examples output, we update the code to reflect the change. To synthesize code, we make use of CVC4's Syntax-guided synthesis (or SyGuS) algorithm [14]. SyGuS is an approach that performs an enumerative search over the space of possible programs, based on a given grammar. We draw possible grammatical elements from the existing function implementation and the provided examples. This both helps ensure that the newly generated code does not stray too far from the programmers original implementation, and helps constraint the space CVC4 has to search over. If we fail to find a solution to the SyGus problem, we can iteratively increase the size of the grammar to include elements not present the user-provided code.

Implementation

We have implemented our live programming methodology as a plugin for Javascript programming in the Atom text editor [7]. To demonstrate the key ideas, our implementation supports live debugging for programs manipulating strings. We are in the process of extending this support to other datatypes, and we see no significant theoretical obstacles to doing so.

There have been many systems from the program synthesis community that build custom editors for live programming [12] or support synthesis for domain-specific languages invented by the researchers [13]. A key contribution in our implementation is embedding live programming by example into a language (Javascript) and an editor (Atom) that has a large userbase. By implementing our tool in this way, we hope to learn how users interact with live programming by example in the wild. We can collect logs of synthesis tasks requested by users of the tool to contribute new synthesis benchmarks (for example to the SyGuS competition set [1]) that more accurately reflect the synthesis tasks that users need.

CONCLUSION

By combining live debugging and programming by example, our methodology offers programmers a useful work environment. Live debugging offers rapid feedback as code is written and modified. When the user encounters unexpected output, they have two options. The user can go back to the code, detect the source of the error, and correct it manually. However, they can also adjust that output value directly, and rely on programming by example to ensure the program gives the expected output.

Acknowledgments. This work was supported in part by NSF grants CCF-1302327, CCF-1715387, and CCF-1553168.

REFERENCES

[1] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. SyGuS-Comp 2017: Results and Analysis. *arXiv preprint arXiv:1711.11438* (2017).

[2] Andrew R Brown and Andrew Sorensen. 2009. Interacting with generative music through live coding. *Contemporary Music Review* 28, 1 (2009), 17–29.

[3] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices* 51, 6 (2016), 341–354.

[4] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[5] Allen Cypher. 1991. EAGER: programming repetitive tasks by example. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 33–39.

[6] A. Cypher and D.C. Halbert. 1993. *Watch what I Do: Programming by Demonstration.* MIT Press.

[7] GitHub. 2018. Atom - The hackable text editor. https://github.com/atom/atom.

[8] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*. 317–330.

[9] Sumit Gulwani. 2012. Synthesis from Examples: Interaction Models and Algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (2012). Invited talk paper.

[10] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. ACM, 492–501.

[11] H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann Publishers.

[12] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276497

[13] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. 2018. Live Functional Programming with Typed Holes. *arXiv preprint arXiv:1805.00155* (2018).

[14] Andrew Reynolds and Cesare Tinelli. 2017. SyGuS Techniques in the Core of an SMT Solver. *arXiv preprint arXiv:1711.10641* (2017).

[15] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. ACM, New York, NY, USA, 2951–2958. https://doi.org/10.1145/3027063.3053187

[16] Bret Victor. 2012. Learnable Programming : designing a programming system for understanding programs. (2012). Available at http://worrydream.com/LearnableProgramming/.