Tianyi Zhang | Research Statement

Writing code is the most versatile and powerful way to leverage computational power. Over the past decade, many new programming interfaces (e.g., programming languages, frameworks, libraries) have been designed to meet the needs in various domains. In the meantime, there has also been an ever-increasing number and variety of people who need or want to code, from computational scientists to financial analysts to data journalists. The need is pressing on both sides: While programmers are grappling with the constantly evolving landscape of programming interfaces, non-programmers are facing significant learning barriers of writing basic programs.

My research in *Software Engineering* and *Human-Computer Interaction* seeks to help both programmers and non-programmers write code with higher quality, greater consistency, and less effort. As programmers often rely on examples to learn and use new programming interfaces, I have built interactive systems that help them build a bird's-eye-view user experience over the abundance of online examples [1, 2, 3, 4]. By unveiling what others have or have not done in similar contexts, these systems help a programmer distill data-driven insights and make more informed decisions in different tasks such as API learning [1] and neural network design [4]. After writing their code, programmers can further check the syntactic and behavioral consistency of their code through interactive code review [5] and differential testing [6]. On the other hand, to dismantle coding barriers for non-programmers, I have developed new program synthesis techniques that automatically generate code from user-provided examples. Unlike traditional program synthesis, these techniques provide (1) *enriched feedback loops* for ambiguity resolution [7] and (2) *greater algorithmic transparency* for users to build more accurate mental models and provide strategic guidance in challenging tasks that a synthesizer cannot solve alone [8].

Learning Programming with a Bird's-Eye View of Online Code Examples

As millions of tutorials, Q&A discussions, and open-source repositories are made available online, the Internet has become the "go-to person" for both professional and novice programmers. While programmers are awash with online coding resources, sifting through many possible solutions and finding the right one for their unique combination of tasks and circumstances remains challenging. Searching online can result in tutorials and examples with varying quality and relevance, which is time-consuming to navigate and assess. As programmers may not start with clear goals in mind, it is almost impossible to search for the unknowns.

To overcome the limits in search-based methods, I have been on a quest to **build bird's-eye-view user experiences over the abundance of online code examples**, as illustrated in Figure 3. I have developed a series of interactive systems for navigating, assessing, and adapting relevant examples to fit programmers' own needs. EXAMPLORE [1] aligns hundreds of API usage examples and registers the distinct API usage patterns in an API skeleton with their distributions, so developers can simultaneously compare all these examples and decide which pattern suits their usage scenario the best (Figure 2). EXAMPLECHECK [2] prompts users with potential API usage mistakes in a code example and renders alternative API usage with statistical evidence such as how many other developers also follow the same pattern on GitHub. EXAMPLESTACK [3] guides users to adapt a code example by summarizing the adaptations performed by other GitHub developers in an interactive code template. While these three systems focus on low-level implementation details, EXAMPLENET [4] visualizes the high-level design choices in neural network models from GitHub. Specifically, EXAMPLENET aggregates neural network architectures into a single Sankey diagram based on layer types and positions. It also renders the distribution over hyperparameter settings. By summarizing the gist of many relevant examples in a bird's-eye view, all these systems allow users to answer questions that were prohibitively time-consuming to answer in the past, such as "what are all possible alternatives of using an API?" and "what are the common and uncommon model structures and hyperparamters in image segmentation models?" This in turn promotes re-evaluation of their own understandings and brings in more awareness of alternatives and unknowns, resulting in better and more consistent choices when designing and implementing software.

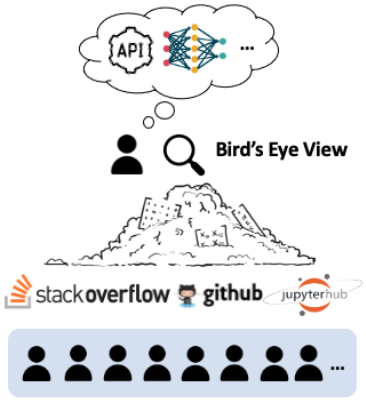

Figure 1: Build bird's-eye-view user experiences over large-scale code examples shared by others.

Building bird's-eye views over massive code-related data from the Internet requires to address both technical challenges (e.g., noisy online code, algorithmic scalability limitations) and cognitive constraints of human programmers. My research follows a structured framework to address these challenges from three aspects:

#1. Develop proper abstractions based on real user needs. For each system, I have conducted a rigorous need-finding study and then designed an abstraction that highlights the semantic gist of online code to support the information needs of users. For example, in the task of API learning and usage [2, 1], I have designed *structured API call sequences* that only retain the temporal ordering of API calls as well as relevant control structures and guard conditions. Irrelevant program statements that have no data or control dependency to an API of interest are pruned by backward and forward slicing. Variable names are canonicalized by types, and expressions are canonicalized by semantic equivalence using an SMT solver. Having a proper abstraction not only lifts the semantic gist from the underlying noisy data to a level of salience, but also reduces computational complexity and increases mining accuracy. By abstracting raw code to structured API call sequences, the precision and recall of API usage mining are improved by 15% and 10% with over 4X speedup [2].

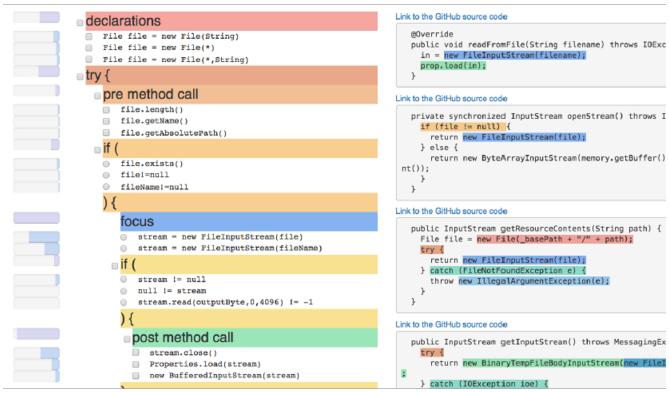

Figure 2: Examplore shows the distribution of distinct API usage features in 100 code examples [1].

#2. Build infrastructures that scale to large code corpora. Sophisticated program analysis such as program slicing is known to be unscalable. My collaborators and I have built a distributed software mining infrastructure that compiles the analysis code to map-reduce jobs and then deploys them to a cluster of seven machines. This infrastructure performs call graph analysis and program slicing and extracts structured API call sequences from 380K GitHub projects within up to 15 min (10 min on average, Chapter 5.4 in [9]). It is the first method that scales sophisticated program analysis to hundreds of thousands of GitHub projects at an unprecedented level.

#3. Design aggregate views with on-demand zoom in and filtering. It is cognitively demanding to compare and contrast many examples. Following Shneiderman's mantra—*overview first, zoom and filter, then details-on-demand*, I have designed different aggregate views to help users easily understand the essence of many examples and identify what can vary. Users are allowed to filter the underlying code corpus or zoom in to a finer granularity via interaction. For example, ExampleStack [3] automatically infers *a code template with holes* based on the unchanged parts of many adapted examples. When clicking on a hole, distinct code adaptations in that particular location are rendered in a drop-down menu, along with their frequencies.

Broader Impacts Based on this line of research, I have helped draft a grant proposal on data-driven programming interface design and usage, which has recently been awarded $500K from NSF. This research has also led to a recent collaboration with Facebook, through which we developed a bird's-eye view over code examples returned by Facebook's code search engine [10]. It has been integrated internally and used by thousands of Facebook developers. My research also has gained traction outside of CS. In an on-going collaboration, I have been helping psychiatrists at Massachusetts General Hospital get a bird's-eye view over the health records of their patients. I have built an interface that visualizes various health trajectories in a timeline view, through which psychiatrists could recognize variations among patients and make better treatment selection in the future.

Writing Consistent Code with Interactive Code Review and Differential Testing

I have also developed interactive techniques to help developers ensure the way they write code is consistent across their own code bases or with similar code in the wild. Critics helps developers identify syntactic inconsistencies among similar code during peer code reviews [5]. It allows developers to interactively construct a code template by paramaterizing code elements (e.g., variable names, types) that may vary in similar locations. Then it automatically summarizes similar code based on the template and detects syntactic inconsistencies. Later, I helped Aishwarya, a junior PhD student I mentored, develop Alice to reduce the manual effort of template construction by automatically inferring a parameterized template via active learning [11].

Once Critics identifies syntactic inconsistencies, developers may want to examine how these inconsistencies affect the runtime behavior in similar code locations. I have developed Grafter, a test transplantation and differential testing framework that examines the behavioral differences between similar code [6]. Grafter automatically transplants code between similar locations so that they can be exercised by the same set of test cases. To do so, Grafter uses def-use analysis to expose the defacto interfaces of similar code. It then reconciles variations in their surrounding contexts based on heuristics. With Grafter, developers can compare the behavior of similar

code in test outcomes and intermediate program states.

Broader Impacts Huawei has adopted CRITICS to perform consistency checking on the API updates in their Android-based operating system. Every time Android is updated, Huawei has to go through a major refactoring on their own codebases to update the usage of Android APIs. Previously, Huawei developers have to manually conduct extensive code reviews to ensure that all API call sites are updated consistently. Now with the help of CRITICS, developers only need to create a code template of an API update, and CRITICS will then automatically summarize similar updates across the codebase and detect inconsistencies among them.

Democratizing Programming with Interactive and Interpretable Program Synthesis

Inductive program synthesis frees novices and end-users from learning new programming languages and tools by automatically generating code from high-level user specifications, such as input-output examples and demonstrations. A long-standing challenge in inductive program synthesis is the inherent ambiguity in user specifications, leading to plausible programs that only match user-intended behavior on given inputs but deviate on new inputs. To address this challenge, I have proposed a new interaction model called **interactive synthesis by augmented examples** [7]. This approach allows users to disambiguate their intent by specifying how different parts of an example should be generalized by a synthesizer via light-weight annotations (i.e., semantic augmentation). Furthermore, it reduces the cognitive load of understanding and validating a synthesized program by revealing how it behaves on new inputs, which then can be used as counterexamples for the next synthesis iteration (i.e., data augmentation). Specifically, I have implemented an automaton-based input generation method to explore the hypothetical input space and identify corner cases based on some program coverage criteria.

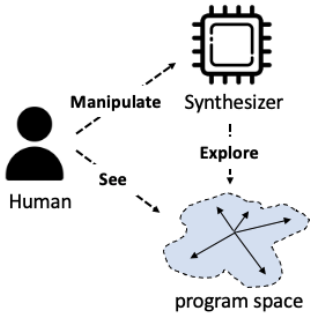

Figure 3: Enable users to "see" and "manipulate" the synthesis process.

In challenging tasks, program synthesizers often get stuck, failing to generate anything given a time that a user is willing to wait. Unfortunately, existing synthesizers follow a *black-box design*, providing no means for users to reason about such synthesis failures. Users' patience and faith in program synthesis are quickly exhausted by recurring failures and a lack of means to troubleshoot. To counter the lack of understanding and loss of trust, I have proposed **interpretable program synthesis**, which unveils the synthesis process and enables users to monitor and guide the synthesis process [8]. Interpretable synthesis renders to a user which search directions and what programs have been tried by a synthesizer, i.e., the explored program space. I have designed and evaluated three representations of the program space with different levels of fidelity. First, a live-updated line chart renders how many programs have been tried by the synthesizer over time and how many user-provided examples each of them satisfies. Second, a syntactically and semantically diverse set of program samples is drawn from the explored program space and shown to users. Third, a search tree organizes and visualizes all explored programs based on how they are derived from the DSL grammar, i.e., their derivation trees. A user study has shown that interpretable synthesis significantly improved participants' problem solving capability for challenging tasks and participants with higher engagement tendency or less expertise expressed a stronger preference towards the highest-fidelity representation, i.e., the search tree.

Broader Impacts As computation is woven into our everyday life, less than 1% of the world's population are professional developers. That is a lot of power locked in the hands of a few. New technologies such as interactive program synthesis have great potential of widening the demographic of people who do not have the traditional tech background to leverage the power of computation. Though not everyone has to learn how to build complex software, there is real value in automating mundane tasks such as matching phone numbers from a mass of unstructured text data. Programmer or not, understanding how the machines around us work is a valuable perspective when they are shaping our lives.

Research Agenda

In future research, I will continue collaboration with researchers in SE, PL, and HCI, as well as engineers at tech companies, like Facebook and Microsoft, to build new programming interfaces and tools. In addition, I am excited to collaborate with researchers in Visualization, Robotics, and AI to build tools that enable rich, adaptive interaction among human, data, and intelligent systems.

Building bird's-eye-view experiences for domain experts. One group of domain experts I wish to help are API designers. I have conducted a need-finding study with 23 API designers and investigated how to help them make

more informed decisions based on API usage data from their API communities [12]. I plan to enact those ideas identified from the need-finding study and build new interfaces for data-driven API design. Furthermore, I intend to apply this idea to support decision making in domains outside of CS. During my postdoc at Harvard, I have been helping physician-scientists at Massachusetts General Hospital discover and reason about the disease and treatment patterns in large volumes of electronic health records. As I move forward, I would hope to also foster new collaborations with physician-scientists, computational biologists, and social scientists in the new school.

Synthesize increasingly complex programs. Program synthesis has been proven effective for generating small programs from small grammars. I am interested in pushing the boundaries to synthesize software that is bigger and more complex. In addition to continuing to build new interaction models and interfaces that elicit human expertise and guidance for synthesis, another promising direction I would like to pursue is to unify the abundance of online code data with program synthesis. Bigger and more complex code can be composed with (a) online code snippets as building blocks and (b) synthesis algorithms focusing on searching for the right pieces of code snippets and then generating glue code to stitch them together.

Support human-machine symbiosis in various tasks beyond writing code. Software engineering is more than just writing code. Human activities such as software testing and bug fixing heavily weigh in to build correct and reliable software. While fully automating these activities may still be challenging, having humans in the loop and co-pilot can be both more achievable and preferable by practitioners. Similar to how we combined human expertise with synthesis algorithms, I am interested in augmenting automated program repair techniques to provide more affordances for human developers to monitor, guide, and control the patch generation process.

I also hope to explore how to augment intelligent systems beyond SE and help a broader group of users to reason about autonomous system behavior. For example, I would like to collaborate with researchers in AI and Robotics to investigate how to render states and actions captured by a robot's policy and elicit rapid feedback, so human experts can recognize undesired states and refine the policy. This opens up many new opportunities that harness the relative strengths of humans and machines to accomplish what neither can achieve alone.

References

[1] E. L. Glassman*, T. Zhang*, B. Hartmann, and M. Kim, "Visualizing api usage examples at scale," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*, pp. 580:1–580:12, ACM, 2018. *The two lead authors contributed equally to the work as part of an equal collaboration between both institutions.

[2] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable? a study of api misuse on stack overflow," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pp. 1–12, 2018.

[3] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and supporting adaptation of online code examples," in *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, pp. 316–327, IEEE, 2019.

[4] L. Yan, E. L. Glassman, and T. Zhang, "Visualizing examples of deep neural networks at scale," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*, 22 pages, 2021.

[5] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE '15)*, vol. 1, pp. 111–122, IEEE, 2015.

[6] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, pp. 665–676, IEEE, 2017.

[7] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*, pp. 627–648, 2020.

[8] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. L. Glassman, "Interpretable program synthesis," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*, 24 pages, 2021.

[9] T. Zhang, *Leveraging Program Commonalities and Variations for Systematic Software Development and Maintenance*. PhD thesis, University of California, Los Angeles, USA, 2019.

[10] C. Barnaby, K. Sen, T. Zhang, E. Glassman, and S. Chandra, "Exempla gratis (eg): Code examples for free," in *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*, 12 pages, 2020.

[11] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim, "Active inductive logic programming for code search," in *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, pp. 292–303, IEEE, 2019.

[12] T. Zhang, B. Hartmann, M. Kim, and E. L. Glassman, "Enabling data-driven api design with community usage data: A need-finding study," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*, pp. 1–13, 2020.