

# RESEARCH STATEMENT

Lingxiao Jiang  
jiangl@cs.ucdavis.edu

Software systems are becoming more and more sophisticated, complex, and ubiquitous, and their dependability is highly critical. There are abundant deep, challenging research issues, ranging from design to implementation, prototyping to production, and development to maintenance, that need to be studied for improving software quality and developer productivity. The central goal of my research is to develop techniques, tools, and methodologies that can help develop highly dependable software more productively.

I am interested in software engineering, programming languages, and systems in general. To tackle dependability and productivity problems, I like to not only develop and refine techniques within these areas, such as program analysis, and automated testing and debugging, but also adapt techniques from other areas, such as data mining and machine learning. The variety and abundance of open source software also provide rich opportunities and excellent test-beds for studying many research questions that interest me. Specifically, I seek answers to four questions: (1) How to effectively detect different types of software defects? (2) How to leverage prior knowledge to avoid similar defects? (3) How to specify and extract good design and programming patterns? (4) How to represent and retrieve such knowledge to ease future development?

My research has focused on analyzing existing, large code bases for program understanding, maintenance, and defect detection, mainly targeting the first two questions<sup>1</sup>. I will continue this line of research with a three-pronged approach: (1) *investigate* what types of defects can occur during the development process, (2) *develop* novel program analysis techniques, tools, and infrastructures for preventing and detecting defects, and (3) *design* domain specific formalisms, languages, or language extensions for capturing and retrieving the essence of defects. Much of my research is interdisciplinary, incorporating ideas and techniques from software engineering, programming languages, systems, data mining, and algorithms. A preferred setting for my work is a computer science department that encourages interdisciplinary collaborations among all these areas and supports first-class research and instruction in software engineering.

## I Code Clone Analysis

I believe there is much hidden knowledge in open source software that can be valuable for improving software quality and productivity. For example, if we want to implement a new feature and know there exists code with similar functionalities in other projects, either reusing the existing code or simply using it for reference will likely shorten development time. For another example, if we know a software fault may occur within a particular context, we can quickly locate all similar defects.

To uncover the hidden knowledge, a requisite step is to create an effective code search tool that can scale to the whole open source world, which contains hundreds of billions, even trillions of lines of code. An important aspect of code search is to find copied-and-pasted code or code that looks similar (*a.k.a.* code clones). I developed an efficient and accurate tree-based clone detection tool, DECKARD [ICSE'07], scalable to billions of lines of code. With algorithms drawn from Information Retrieval and Computational Geometry, the work provides a general, efficient framework for measuring similarity among large sets of structural data, such as abstract syntax trees and program dependency graphs.

On the other hand, inconsistent code changes may occur during software evolution and lead to defects that are difficult to detect with traditional techniques. To help address this problem, I proposed a *context-based approach* that formalizes defects in the context of code clones as syntactic inconsistencies among clone instances [ESEC/FSE'07]. My empirical study showed that this approach can detect many defects of different classes and complements other bug detection techniques. The study also confirmed that developers need tool support during code cloning to prevent introducing incidental defects.

---

<sup>1</sup>I could not have done all the work without the support from my advisor, Zhendong Su, and other collaborators. I use the first-person herein for brevity.

Further, code with similar behavior (*a.k.a semantic clones*) is of particular interest for developers, while previous work mainly targeted syntactic clones. Although definitions for semantic equivalence exist, there has been no study even for such a simple question as how commonly semantic clones occur in software projects. Answers to this question may enable many potential applications. For example, if a functionality is frequently re-implemented, it may be worthwhile to extract it into a shared library; if a component pool contains different implementations of a functionality (*e.g.*, space *vs.* time efficiency, code size *vs.* complexity), a semantic-aware component recommendation system may suggest the best one to use under particular circumstances. I utilized random testing to mine likely semantically equivalent code fragments: candidate code fragments and their input/output variables are automatically extracted from subject programs, and random inputs are generated to quickly partition the code fragments based on their outputs for the given inputs. I conducted a preliminary evaluation of the idea on the Linux kernel and detected many semantic clones that are syntactically different. Further evaluations are being carried out to stress test the algorithm, and more importantly, to study the characteristics of semantic clones so that we can understand how and why they occur in the development process and how we can benefit from them.

## II Testing and Debugging

Another important aspect for software quality and developer productivity is to develop more cost-effective approaches for testing and debugging. Similar to the potentials of open source software provided by numerous participants, I believe in the potentials of the mass for improving the effectiveness of testing and debugging, especially for defects difficult to be detected by existing in-house techniques and tools. By collecting sparse information about each execution from a large number of users, developers could accumulate insightful information about program failures and infer possible root causes more easily with data mining.

Based on such execution profiles, I applied machine learning algorithms to identify *bug predictors*, and developed a specialized depth-first graph search that translates bug predictors into *faulty control flow paths* that link many failure-related code segments together to provide informative contexts so that it may be easier for developers to understand failures and look for root causes [ASE'07].

In addition to constructing such static information for bug localization, it is also useful to provide developers more dynamic information, such as failure-inducing inputs and actual failure traces, for debugging. Combined with my experience on language-based techniques, such as static analysis, symbolic execution, directed random testing, and model checking, I proposed the concept of *profile-guided program simplification* [FSE'08] that reduces code complexity but preserves failure paths in programs to scale up the capabilities of existing testing and analysis techniques. The idea is a generic, effective way to combine many benefits of both user profiling techniques and in-house testing and analysis techniques; it naturally focuses developers' attention more on failures that are experienced more often by users and help reduce triage costs.

## III Program Analysis

Along the line of improving software quality and developer productivity, I am also actively interested in program analysis techniques capable of finding hard-to-spot defects efficiently. In particular, I developed *Osprey* [ICSE'06], a non-standard, constraint-based type-system that facilitates validation of measurement units of abelian group nature (multiplicative, commutative, and associative). With a small language extension that allows unit annotations for variables, the type system automatically performs dimensional analysis that is carried out manually and routinely in scientific programs for detecting misuses of units.

I am also interested in developing program analysis techniques (*e.g.*, type inference, data-flow analysis, and abstract interpretation) for the purposes of program understanding besides defect detection. For example, I experimentally used these techniques to automatically infer pre-/post-conditions to construct code summaries. However, we know that the scalability and effectiveness of these techniques often rely on both code complexity and the actual properties defined for analysis. I am experimenting with new ways that

combine clone analysis techniques with program analyses for better results. For example, in our study of clone-related defects, the accuracy of the bug reports could be significantly improved if more semantic information, such as the path constraints that reach the clone instances or the data flows within the clones, could be incorporated to predict defects. Also, commonalities or differences among clone instances can provoke specific questions about the properties of the code interesting to developers, and program analysis techniques can be well suited to answer such questions. Complementing each other, these two classes of techniques can be essential candidates that may help to understand code behavior, extract programming patterns, construct program summaries, and lay a foundation for semantic-aware code/component recommendation systems.

## IV Future

I will continue and extend my line of research on software quality and developer productivity to address the four questions listed earlier. I believe in and will develop specialized techniques that can effectively attack problems with domain specific knowledge; I also believe in and will carry out research on general approaches that provide unified frameworks and solutions for many problem domains.

In the short term, I will further enhance the capability and scalability of my clone detection tools (both syntactic and semantic-based), investigate commonly occurring clone patterns in large scale, and construct a knowledge base of such patterns. All open source projects in different programming languages may collectively reach trillions of lines of code. To study such a large code base requires techniques and tools that can (1) scale; (2) detect clones, especially semantic ones, across language boundaries; (3) extract clone patterns and code summaries; and (4) aggregate commonly used patterns into shared libraries for reuse. In addition to the analyses working directly at the code level, I will also investigate analyses at higher levels, such as design patterns and software architectures, as well as analyses at meta-levels, such as error-proneness and time/space complexity of code. High-level and meta-level characteristics will help developers to understand and reuse existing code more efficiently, and help them to prevent similar pitfalls from happening again.

In the long run, my investigations will be stepping stones for *search-assisted programming*, where a code search engine is integrated with development environments and answers many of developers' questions, ranging from syntax to semantics, from simple code reuse to overall design choices. For example, what is the standard way to use this particular API; How to perform a regular task, such as sending a file through the HTTP protocol; Which design pattern is best suited for this particular situation; How to organize these particular components, *etc.* In addition to programming, all these studies will also help enhance *search-assisted testing and debugging*. For example, are bugs that previously occurred under similar contexts also possible here; Is there a known test case for similar code usable for this code; Is there any known defect that has similar symptoms as this one; Is there a better way to make this code use less memory. Although this *search-assisted paradigm* is still far from the dream that software requirements can be magically translated into functioning implementations, it is a realistic goal that utilizes historical information to help developers to make more intelligent choices and fewer errors. The success of this paradigm will rely on how well we distill and organize useful information from large existing code bases and their associated repositories and on how well such information can be integrated into software engineering processes. Similar to token-level word auto-completion provided by existing development environments, a foreseeable feature of the search-assisted paradigm is to provide auto suggestions and validations on higher level information based on the surrounding contexts, for example, to suggest the best scenarios to use a given API or check whether the API is used correctly, and to check the complexity of the code or suggest a better optimized version.

I am interested in developing both general and specific techniques, tools, and methodologies for improving software quality and developer productivity. I will take the approach I outlined above and focuses on a unique combination of program analysis, pattern discovery, and knowledge reuse to lead its progress. I believe the line of research will cut across many areas besides software engineering, such as programming languages, systems, data mining, and algorithms and theories, and facilitate healthy collaboration with my future colleagues. I am excited to take the opportunity to stay on the forefront of the field, address all potential challenges, and promote the advance of software engineering and interdisciplinary research.