RESEARCH STATEMENT

Jooyong Yi (jooyong@comp.nus.edu.sg)
http://www.comp.nus.edu.sg/∼leejy/

My current research theme is, "make computers fix bugs". When debugging a program, human developers are often forced to think the way the computer operates, which is probably the main reason debugging is difficult and unproductive; developers often toil for hours and even days to modify just a few lines of buggy code. I think human developers should focus on what they are really good at—designing and building software at a high level, without being distracted by what they are not good at—debugging at a low level. The benefits of automated bug fixing does not end in making the lives of software developers easier. Software companies do not have to sacrifice software quality any more (currently, they often release software without fixing bugs due to the time pressure), since human-driven software development and machine-driven bug fixing can be performed hand in hand. Most importantly, customers of software will be less exposed to software vulnerabilities.

My current research on automated bug fixing is in line with my general research theme: providing developers with right tools to make software development easier and more productive. Apart from automated bug fixing tools, I have also built program verification environments in which developers can specify, develop, test and verify their code. I have identified and addressed various problems in performing each activity from specification to verification, and in supporting integration of multiple activities in one environment.

Note that almost all of my work is done in collaboration with students and colleagues, although this statement is written in the first person following the common convention.

Automated Program Repair

The once-futuristic idea of automated program repair (aka, automated bug fixing) is gradually becoming a reality through the efforts of pioneering researchers. The main innovation of my work lies in its semantic approach. Instead of randomly mutating the buggy program to find a repair, my repair tools analyze the semantics of the buggy program to extract repair constraints, and synthesize a repair that satisfies these extracted repair constraints. The main benefit of this semantic approach is that generated repairs are usually more precise than mutation-based approaches, since the semantics of the program is considered into the repair. While the scalability of the semantic approach was questioned previously, Angelix,[1] the latest repair tool I developed, is shown to be effective in fixing bugs of large-scale real-world software such as PHP and OpenSSL. Angelix is the only tool (as of the time of writing) that has successfully fixed the Heartbleed bug of OpenSSL automatically.

In Angelix, the problem of automated program repair is solved by solving the following three sub-problems in sequence.

1. **Where to fix:** First, fix locations, i.e., program locations to be repaired, are identified.
2. **What to fix to:** Second, a desired "angelic" program state at each fix location is identified. By changing the program state at each fix location into its corresponding angelic state, the bug of the program can be fixed.
3. **How to fix:** Lastly, a patch satisfying the identified angelic state is synthesized.

Note that each of these three tasks is a different search problem—a search for fix locations, angelic states, and patches, respectively. The novelty of my approach is that I solve each search problem efficiently via customized semantics-based techniques. For example, I search for a patch using a program synthesis technique customized to scale well regardless of the size of the program.

My ICSE 2016 paper describing Angelix receives a strongly positive review saying "probably the best program repair/patch generation paper I have ever read" and "the fact that the authors successfully applied it to HeartBleed is impressive".

Prior to Angelix, I also developed Directfix that generates a provably simplest repair — simplest in the sense that the structure of the original buggy program is preserved as much as possible. The simplicity of repairs is desirable because simple repairs are easy to read and validate. Also, experimental results show that simple repairs are less likely to change the correct behavior of the original version than more complex repairs.

[1] http://angelix.io/

Program Verification Environment

My efforts on this part of the work is centered around JML (Java Modeling Language). JML is a well-known behavioral specification language. Using JML, developers can express what they intend to develop, and check whether their code matches their intention. I have contributed to building verification environments such as JmlEclipse and OpenJML in which developers can specify, develop, test and verify their code. I have been an active member of the JML research community, and co-organized an NII Shonan meeting on JML in 2013.

One of my recent innovations is a novel methodology named "software change contract" that addresses the following problem. Software does not always evolve in a progressive way. For example, the Heartbleed bug is introduced only after OpenSSL version 1.0.1, and the previous versions of OpenSSL are safe from the bug. System administrators are faced with the dilemma of choosing between accepting a potentially dangerous patch and remaining exposed to known vulnerabilities. Regressive software changes happen when actual software changes do not match the changes developer intend to make. My work on "software change contract" makes it possible for developers to express intended changes as a change contract, and mechanically check whether there is any discrepancy between intended changes and actual software changes. I have designed a language for software change contract and implemented its supporting tools to find the violations of change contracts, verify the absence of such violations, and infer change contracts from actual code changes.

Apart from software change contracts, I also invented a new language feature for specification languages to make writing a specification easier, developed an efficient verification technique that can reason about complex specifications (e.g., specifications about data structures), and designed a new architecture of a verification environment for easier tool maintenance.

Future Work

The idea of "making computers complete uncompleted tasks" (bug fixing is one such example) has the potential to change the landscape of how software is developed and maintained. Currently, developers are spending significantly more time on implementing, testing, debugging, maintaining software than on designing and specifying software. In fact, software is hardly properly designed and specified; design and specification are considered too expensive! However, given the feasibility of automated program repair, which one would be more expensive between manually fixing bugs ceaselessly until software support is stopped, and making computers repair the software on their own whenever necessary, under the guidance of the provided specifications? In the future, software that is "designed by humans and manufactured/maintained by machines" will be the new normal.

I want to go beyond automated program repair which still involves human developers in the loop to validate generated patches. I believe that software systems can evolve on their own; they can learn by mistakes and adjust themselves accordingly; they can also adapt on their own to new environments encountered as smart devices move around. In this *self-evolving software framework*, human developers only lay down the foundation of the software, and software grows on their own as it is exposed to various different usage scenarios and environments. The more software is used, the more robust and secure software will be, if software can evolve on their own.

Basic ingredients to support self-evolving software are already available. Key techniques necessary for self-evolving software are automated program repair, program specification (to guide program repair), runtime program monitoring (to detect program errors), and program verification (to prevent undesirable accidental software changes). I have worked on all of these areas, and gained expertise.

Apart from innovating software development/maintenance methodology, I also see a good opportunity to revolutionize how education is delivered. The revolution has already started with the advent of online education; billions of people around the world now can easily access high-quality lectures that were previously available only to mostly privileged students. The problem arises from the huge scale of online education. Most prominently, how can we provide personalized feedback to thousands of students? Tutors can hardly be physically present with students in online education. There is a pressing need for automated tutoring systems. Given my expertise in automated program repair, automated feedback generation for programming assignments seems to be a good starting point. Instead of generating a patch for a bug, it is possible to automatically generate hints and feedback. This area of research on technologies for online education is wide open, and I think I can make a valuable contribution to this area, given my research background and passion for education.