Jiasi Shen Research Statement

Software plays a central role in numerous aspects of human society. Current software development practices involve significant developer effort in all phases of the software life cycle, including the development of new software, detection and elimination of defects and security vulnerabilities in existing software, maintenance of legacy software, and integration of existing software into more contexts. The goal of my research is to **improve software development by automating tasks that currently require substantial manual engineering effort.**

My research focuses on developing automatic techniques that **analyze, manipulate, and transform software.** During my Ph.D., I have developed techniques that automatically extract, transform, and augment functionality present in existing software. This research pioneered a novel approach, **automatic software rejuvenation**, which takes an existing program, learns its core functionality as a **black box**, formulates it as a precise model, and uses the model to generate a new program. The new program delivers the same core functionality but is potentially augmented or transformed to operate successfully in different environments. This research enables the rejuvenation and retargeting of existing software and provides a powerful way for developers to express program functionality that adapts flexibly to a variety of contexts.

Motivation. Software functionality can be expressed in many forms, including high-level descriptions, diagrams, pseudo-code, and concrete implementations. When it is expressed in concrete implementations, however, much flexibility is lost. The core functionality of the software becomes difficult to separate from other implementation details, such as platform dependency and security properties, and the associated human knowledge becomes difficult to manipulate or transform. This complexity is especially problematic given the following challenges. First, as new computing platforms emerge, much simple functionality (e.g., retrieving data or counting) has to be implemented repeatedly in different places. Moreover, as these platforms become increasingly complex and difficult to use, their usage barriers grow higher even though much of the implemented core functionality is relatively simple.

Approach. My thesis research focuses on analyzing software to extract its core functionality and formulate it as a model. I aim to make this model **flexible, comprehensible, and exact.** This model can then enable the automatic insertion of new features such as different platforms and systematic security checks, enable the generation of new code that is clean and maintainable, and enable rigorous mathematical reasoning. These capabilities can help developers automatically manipulate and transform software, thereby reducing manual engineering effort.

To achieve this goal, I identify recurring patterns in software, capture these patterns formally, and then exploit their properties by developing targeted algorithms with strong theoretical guarantees. I have instantiated this strategy on two domains: the domain of database-backed programs and the domain of stream-processing programs. I have developed targeted techniques that automatically extract, transform, and augment software functionality [6, 7, 4, 5]. Each technique has two main steps: (1) **inferring** a black-box program's core functionality and (2) **regenerating** the functionality into a new implementation. My program inference techniques involve three key concepts: (1) a domain of supported programs defined formally as a domain-specific language (DSL), (2) a targeted algorithm for the domain that exploits the domain knowledge, and (3) a rigorous mathematical proof, where possible, based on the DSL. Here, an inference algorithm learns the program functionality and formulates it as a **precise model represented in the DSL**. This model enables the automatic regeneration of a new program that preserves the original core functionality but, by its construction, adapts to various new contexts.

Benefits. This paradigm can deliver benefits in many aspects of the software life cycle. The first benefit is **automatically generating high-quality software** from simple prototypes that implement only the core functionality, such as by **generating correct-by-construction code augmented with checks that eliminate security vulnerabilities**. For example, my techniques regenerate data-retrieval programs to add systematic checks that prevent SQL injection vulnerabilities [6, 7, 4]. My group has also used the DSL-based approach to regenerate string-processing libraries, where the new versions no longer invoke original potentially dangerous dependencies, eliminating software supply-chain vulnerabilities [8]. The second benefit is **automatically improving program comprehension and producing cleaner code**, making the code more transparent and the developers more productive. For example, the techniques developed by me and my collaborators produce models of a program that characterizes the program's externally visible behavior, rather than its internal implementation details [6, 7, 4, 8, 3, 5]. These models can help developers understand programs whose source code is obscure, obfuscated, or unavailable. Preliminary results indicate that these models can help developers understand the business logic flowing across multiple source files built with sophisticated frameworks [2]. The third benefit is **automatically extracting the human knowledge in software and retargeting it to different languages and platforms** that provide more powerful features. For example, our techniques regenerate data storage and retrieval programs, where the new versions are augmented with back-end databases and front-end web interfaces [3]. As another example, I supervised a Master's thesis that regenerates Python programs to use an external database instead of the original in-memory data structures [1, 9]. Our technique can also regenerate stream-processing programs with parallelism, so that the new versions perform the same computations but run faster [5]. In short, my research promises to enable developers to express software functionality in flexible new ways that adapt to a variety of contexts.

Ongoing and Past Research

My broad research agenda is unique. **I am the first to approach the black-box inference problem by using DSLs to structure the problem space and solution.** I have applied this DSL-based approach to the database domain, where I developed techniques that infer and regenerate black-box data-retrieval programs [6, 7, 4], and to the stream-processing domain, where I developed a technique that automatically parallelizes black-box Unix shell commands [5]. The results indicate that this approach can address three challenges in previous research: inference of full functionality, scalability, and unambiguously correct answers.

Inferring and Regenerating Black-Box Programs

Inferring models from the observed program behavior is a long-standing research goal for reverse engineering, program synthesis, and program comprehension. I have developed techniques to infer models of data-retrieval programs fully, automatically, and precisely. These models enable the automatic regeneration of new implementations that suit many additional purposes such as correct-by-construction security, clean code, and automatic migration.

Techniques. My research developed several ways to infer models of black-box data-retrieval programs; they do so without analyzing the source code or the binary and without instrumentation, so they work with programs written in any language or coding style. To make the inference tractable, I identify target classes of supported black-box programs and characterize these classes with DSLs that capture the externally visible behavior of the programs. My techniques work with programs whose behavior is expressible using the DSLs and **guarantee exact and unambiguously correct answers**. My techniques use the DSLs internally to structure the inference algorithms. These algorithms **automatically choose inputs to run a program and observe its behavior**, using novel **active learning** techniques that eliminate uncertainty efficiently even if the input space is infinite. The inferred functionality is formulated as a precise model in the corresponding DSLs. These inferred models enable the automatic generation of code that implements the same core functionality but with additional valuable capabilities.

I pioneered the line of DSL-based black-box inference research with a technique [3] that infers black-box programs with data storage and retrieval commands by executing pairs of commands to infer how they interact. I then developed two more systems, Konure [6, 7] and Shear [4], that infer and regenerate black-box programs that retrieve data from relational databases, such as the data-retrieval components in task managers and blogging platforms. Both systems generate inputs and database values using a satisfiability modulo theories (SMT) solver, run the program, observe its database traffic, and progressively refine a DSL-based hypothesis of the program. Shear additionally resolves loop-related ambiguities by experimenting with the program. That is, it repeatedly runs the program with the same input, altering the database traffic at chosen execution points to elicit different behaviors that depend on the program structure. All three systems formulate the inferred functionality as a model in their DSLs and use this model to regenerate a new version of the program.

Contributions. Our results indicate that my DSL-based approach can achieve goals that are infeasible for general computations. The first goal achieved is **inferring the full black-box functionality.** In contrast to previous approaches that infer only partial models such as state transitions, my three techniques infer complete programs that execute and can potentially replace the original program. The second goal achieved is **scalability for large search spaces.** Konure explores an unbounded search space of candidate programs within a polynomial number of solver invocations, in contrast to most prior program synthesis techniques that enumerate a finite search space. Similarly, Shear scales well along several dimensions of program complexity. The third goal achieved is the **synthesis of unambiguously correct program structures.** Konure and Shear infer the unambiguously correct program structure for any black-box program expressible in their DSLs. This property distinguishes Konure and Shear from most prior techniques that rely on heuristics to choose among multiple nonequivalent candidates.

Results and Implications. I have applied Konure and Shear to over 40 data-retrieval commands in 8 open source applications built with Ruby on Rails (these applications use 3 or more languages) and Java. Each command is inferred within 2.5 hours on a laptop computer and then regenerated in Python. The new versions (1) have a more explicit code structure than the original Ruby on Rails applications and (2) eliminate SQL injection vulnerabilities in the original Java application. These results highlight the effectiveness and efficiency of my DSL-based approach. Because these techniques require no analysis of the source code or the binaries, they work with not only today's software using any language but also future software in the years to come.

Automatic Parallelization of Black-Box Programs

Programs that process large datasets are prevalent. One common way to make them run fast is through parallelism. With my collaborators, I have identified a new way to parallelize black-box stream-processing programs.

Technique. I have developed KumQuat [5], which automatically parallelizes black-box stream-processing programs such as Unix shell commands. This parallelism is achieved by discovering a way to split and merge workloads based on observations of the program behavior. The appropriate merge operator is synthesized automatically by KumQuat, in contrast to prior techniques that require manually hard-coded rules. KumQuat generates inputs to run the black-box program, observes its properties related to parallelism, and formulates these properties as a model in a DSL. KumQuat then uses this model to synthesize the appropriate merge operator and regenerate a new version of the original computation that is parallel and (typically) executes faster than the original. I developed theorems that characterize the conditions under which KumQuat will produce an unambiguously correct result.

Results and Implications. I have applied KumQuat to parallelize 70 open-source Unix shell scripts. The optimized 16-way parallel executions of long-running scripts achieve 11.3$\times$ median speedup. These results highlight the effectiveness of KumQuat over a wide range of computations. More broadly, KumQuat demonstrates a new way to use black-box observations of a program to improve its implementation, enabling a powerful form for developers to express program functionality productively, precisely, and flexibly.

Future Directions

My research philosophy is to **identify recurring patterns and exploit them as opportunities.** Many such patterns exist in today's software, which contains enormous amounts of human effort. I aim to replace such human effort with automatic techniques. In the future, software will serve even more critical tasks in human society. It is important to **reduce the fundamental inefficiencies in how people work with software.** My research agenda can help enhance our abilities to work with software through three directions.

One direction is **developing more black-box techniques for more sophisticated domains of computations and for new purposes.** In more sophisticated domains (e.g., microservices, distributed systems, operating systems, and software-defined networks), different software may share common goals such as security, performance, and reliability. These common goals may require manual engineering effort that recurs across many places. More broadly, I anticipate that the software for emerging hardware platforms (e.g., heterogeneous architectures, quantum processors, modern energy-efficient analog systems, embedded systems, and other specialized devices) may also contain recurring patterns. I aim to reduce such inefficiency so that many capabilities are provided automatically by tools that encapsulate knowledge. I propose to work with domain experts to identify patterns and opportunities, develop targeted techniques that improve software quality, and help realize these systems' enormous potential.

Another direction is **developing techniques that dive deeper into software in principled ways to expand the scope of supported computations.** I have started exploring this direction with Shear, which infers programs by intervening in program executions. I anticipate that exploiting other aspects of the program execution can enable many other interesting opportunities. For example, tracking internal arithmetic computations may enhance the range of computations that my techniques can support, an important step towards solving a broader class of long-standing legacy issues in many organizations and increasing the real-world impact. A key research question is how to balance the trade-off between supporting more sophisticated computations and supporting more varied languages, platforms, and implementation styles. My future research group will develop techniques that analyze software based on the perspective of users, rather than on the implementation details, so that the techniques produce comprehensible results that users can understand, maintain, and trust.

The third direction is **developing automatic techniques that eliminate inefficiencies during software development, making developers more productive and software more powerful.** I envision three angles from which automatic techniques can reduce inefficient human effort: across usage scenarios, across computation platforms, and across program implementations. I plan to start by dissecting software into conceptual components at appropriate levels of granularity. I plan to develop techniques that identify, extract, and transform reusable components across various software. For example, two programs may share similar business logic but provide different user interfaces to fit their usage scenarios. Their data structures may differ but still manipulate similar information. I am interested in developing techniques that automatically identify the reusable business logic and graft parts of one program into the other while keeping developer intervention minimal. These techniques can help developers express human knowledge, effort, and creativity in more productive ways. More generally, I aim to automate the understanding, extraction, and adaptation of human knowledge embedded in all kinds of software.

Human society will continue to develop and rely on software. I aim to capture the maximal benefits from the human effort and knowledge embedded in software. I am also excited to conduct and lead research on software techniques that enable new ways to express human creativity.

References

[1] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 62–78, 2019.

[2] Jürgen Cito, Jiasi Shen, and Martin Rinard. An empirical study on the impact of deimplicitization on comprehension in programs using application frameworks. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 598–601, 2020. Registered report.

[3] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. Active learning for inference and regeneration of computer programs that store and retrieve data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, page 12–28, 2018.

[4] Jiasi Shen and Martin Rinard. Active loop detection for applications that access databases. 2021. https://hdl.handle.net/1721.1/138144.

[5] Jiasi Shen, Martin Rinard, and Nikos Vasilakis. Automatic synthesis of parallel unix commands and pipelines with kumquat. *CoRR*, abs/2012.15443, 2021.

[6] Jiasi Shen and Martin C. Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 269–285, 2019.

[7] Jiasi Shen and Martin C. Rinard. Active learning for inference and regeneration of applications that access databases. *ACM Transactions on Programming Languages and Systems*, 42(4), January 2021.

[8] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. Supply-chain vulnerability elimination via active learning and regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1755–1770, 2021.

[9] Jerry Wu. Using dynamic analysis to infer python programs and convert them into database programs. Master's thesis, Massachusetts Institute of Technology, 2018.