Research Statement

Danny Dig

I enjoy doing research in Software Engineering (SE) in general, particularly in **interactive program analysis and transformation** and **software evolution**.

Research: Key statistics
- $7.4M as sole or lead PI, ($5.9M from NSF, $1.5M from industry). My share: $2.8M
- 90+ peer-reviewed papers
  - ★ 9 best paper awards at flagship/top conferences + 4 award runner-ups + 1 most influential award (N-10 years)
  - ★ 46 full conference papers (10+ pages) in flagship/top ACM/IEEE venues in SE (<25% acceptance ratio)
  - ★ 12 journal papers, including 6 in the flagship ACM/IEEE Transactions
- 13 tools released in industrial-strength products (e.g., NetBeans, Eclipse, Visual Studio), >40M downloads
- 7000+ citations, h-index of 44, >400 citations of top-cited paper

It is widely known that at least two-thirds of software costs are due to evolution, with some industrial surveys claiming 90%. For example, software evolves to add features, fix bugs, support new hardware (e.g., multi-cores), new versions of operating systems and libraries, and new user interfaces for new platforms (e.g., web or mobile devices). Although change is the heart of software development, programmers perform most software changes manually, through low-level text edits. This makes software development more expensive, time-consuming, and error-prone than it should be.

My group's research enables programmers to interactively and safely change large programs via **interactive refactoring tools** that preserve the existing behavior while improving other qualities (e.g., performance, readability, reliability, privacy and security, etc.) I successfully pioneered interactive program transformations by opening the field of refactoring in cutting-edge domains including **Generative AI and LLMs** [FSE24a&b, ICSE24Demo], **ML** [ICSE23, ICSE22a, TOSEM21], **mobile** [MobileSoft16, ASE15, SOFTW15, FSE14a, ICSE14a, MobileSoft14], **concurrency and parallelism** [ICSE19a, TOSEM15, STVR15, ECOOP14, FSE13, ISSTA13, ICST13, FSE12, ICSE11, SOFTW11, IWMSE11, ASE09a, OOPSLA09, ICSE09], **component-based** [ICSE22b, FSE20, ICSE19b, ICSE18, ICSE16, ICSE14b, ICSE14c, ECOOP13, ECOOP12, ICSE08, TSE08, ICSE07, ECOOP06, JSME06, ICSM05], **testing** [FSE17, ASE16, TSE10, ASE09b, ISSTA08, FSE07], and **end-user programming** [TOSEM19, FSE14b, ICSM12]. With my emphasis on interactive, domain-specific program transformations, my SE research serves as a bridge between the SE, PL, Security, HPC, and HCI groups. Also, my focus on moving my research to industry and my leadership of the IUCRC Center is very well aligned with President Randall's and Dean Brown's strategic vision for the College and the U.

Impact

I *(co-)authored 90+* journal and conference papers that appeared in top places in SE. According to Google Scholar my publications have been *cited 7000+* times. We[1] target the most selective venues in SE, thus 88% of our papers are in the flagship & top ACM/IEEE venues. We collaborate with academic researchers (MIT, UIUC, NCSU, ISU, PSU, TU Delft, Concordia, UBC) and industry partners (Boeing, Google, IBM, Intel, JetBrains, Microsoft, Oracle, Trimble, Uber).

While my group is pushing the frontiers of fundamental, long-term research, and we are winning prestigious awards in the top academic conferences in SE, we go the extra mile to move this research into practice. In SE, releasing software tools that professional developers use is one of the greatest tests of practical impact. Some of the techniques we developed are shipping with the official release of INTELLIJ, ECLIPSE, NETBEANS, ANDROIDSTUDIO, and VISUALSTUDIO popular programming environments, and are used daily by millions of Java, Kotlin, and C# programmers. Our most recent research on using LLMs as AI Assistant for refactoring was highlighted at the JetBrains AI Launch event on 12/06/23. Moreover, we discovered best practices for evolving ML code and educated developers from major industry players such as Amazon, Microsoft, Intel, Boeing, JetBrains, Trimble, and NEC through knowledge-sharing sessions. Additionally, our code automation tools applied hundreds of patches to best-in-class open-source code such as TensorFlow, Keras, NLTK, Cassandra, and projects from IBM, Microsoft, and Google and improved their performance and resource management.

As a lead or sole PI, my research ideas generated a significant amount of funding from NSF ($5.9M) and industry ($1.5M) in the last ten years. My share is $2.8M. Prior to 2012, I was Co-PI on industry grants of more than $10M.

As a service to the software engineering community, I have started two popular workshops: Workshop on Refactoring Tools, and Hot Topics On Software Upgrades, that already had eight and five instances, respectively. In 2014 I was the lead organizer of a Dagstuhl Seminar on the Future of Refactoring, which gathered the top 50 international experts on refactoring. I have chaired or co-chaired 16 workshops and 1 conference, and I have served as a member of 40+ program committees for all top conferences in SE. I am the Founder and Director of the NSF IUCRC Center on Pervasive Personalized Intelligence (`https://PPICenter.org`) that advances the science, practice, and education on IoT systems. We launched in 2020 with colleagues from two universities, industry members from several IoT verticals, and the NSF.

[1] when appropriate, I will use *we* instead of *I* to acknowledge the undergrads, PhD students, and collaborators whom I worked with

Research Approach

My research is driven by two important questions: (i) what software changes occur most often in practice and (ii) how can we automate them to improve programmer productivity and software quality? Answering these questions is relevant for practice, as well as intellectually challenging and rewarding.

I enjoy connecting seemingly unrelated areas of computer science and making novel contributions. For example, using LLMs + IDE to assist in software development tasks, bringing recent advances from SE to ML engineers and data scientists and modernizing their development tools, connecting parallel and mobile computing with interactive techniques from software design, adapting proven software engineering principles into the world of spreadsheet developers, designing scalable program analyses using data mining techniques, etc. I devise techniques and theories that generalize to solve larger classes of problems, as well as build and deploy tools for automating program changes.

Automating changes is challenging as it requires complex code transformations that span multiple, non-adjacent program statements and requires deep inter-procedural analyses that globally reason about objects shared through the heap. A key problem is designing program analyses that are *accurate* yet fast enough to be used in an *interactive* tool.

I validate rigorously my research by employing empirical methods (e.g., case studies, controlled experiments, interviews) in the evaluation stage (did we built the *tool right*?) and also in the formative stage (are we building the *right tool*?). I place high value into starting a new research direction with large-scale, empirical formative studies that allow data-driven decisions for designing novel software engineering tools.

I happily go the extra mile necessary to move my research into practice. I maintain strong ties with the industry groups building the major integrated development environments (e.g., PyCharm, NetBeans, Eclipse, Visual Studio, IntelliJ IDEA, AndroidStudio), and I contribute to open-source software. My current industry sabbatical at JetBrains (the leading IDE provider) further develops these relationships.

While I enjoy building novel techniques and tools, my greatest joy comes from investing myself into people. I have already graduated 5 PhDs (of which one is faculty at CMU), and I am currently supervising 1 PhD and 2 MS students. In the past I graduated 11 MS students (all published a paper with me). I inspired half of them to continue for PhD. Among the 22 undergrads that have done research with me, I published with 16 of them, and 9 enrolled in grad school. My goal is to help my students maximize their potential and inspire them to become tomorrow's leaders in technology.

In the future I will continue to lead research on interactive program transformations that make software evolution easier, faster, and safer. No matter what is today's winning technology, today's brand new programs are tomorrow's legacy programs. Program transformations are becoming more important as existing software is aging and software evolution becomes the primary paradigm of software development. In the new AI Assistant era there would be even more opportunities for improving AI-generated code.

Previous Research Results

I will describe seven kinds of interactive program transformations that we have successfully automated over the last few years. (i) Generative AI Programming Assistant – this ushers us into a new era of software development when LLMs become effective AI assistants for developers that change code. (ii) Automate code evolution for AI/ML engineers and data scientists – this is important given the huge impact that AI has on our society and that most data engineers and scientists are not trained as professional software developers. (iii) Increase the scalability and precision of program analysis used in refactoring and other analysis tools – this is important for handling today's ultra-large codebases of hundreds of millions of LOC and for increasing the reliability of static analysis tools. (iv) Improve responsiveness in mobile apps via asynchrony – this is essential to sustain the exponential growth in the development and use of mobile apps in the recent years. (v) Retrofit parallelism into existing sequential programs – this is crucial to allow existing programs to run efficiently on the now-pervasive multi-core systems. (vi) Upgrade component-based applications to use the latest API of their components – this is important as all major software systems nowadays are built from software components that continuously evolve. (vii) Generate test suites for complex transformations and evolve obsolete tests – this is important to ensure safety.

Generative AI Programming Assistant for code changes

Many are asking will Generative AI and Large Language Models (LLMs) put software developers out of business? Will it displace my research and make it irrelevant? In 2023, during my industry sabbatical at JetBrains, I've been part of a team that brings a fresh perspective on how Generative AI is helping with the nine hardest things programmers have to do. In this new GenAI era, our role as computing researchers is even more crucial. With my colleagues at JetBrains we have been developing solutions that enable developers to use Generative AI to (i) explain code, bug fixes, summarize recent changes, (ii) generate documentation, code, commit messages, suggest names, etc. By surveying more than 18,000 developers, we learned that a top concern remains the trustworthiness of the solutions provided by Generative AI. While many solutions resemble the ones produced by expert developers, LLMs are known to produce hallucinations, i.e., solutions that seem plausible at first, but are deeply flawed. I present below two examples on how my PhD group in collaboration with JetBrains researchers developed novel research that synergistically combines the creative potential of

LLMs with the safety of static and dynamic analysis from program transformation systems.

First [FSE24a, ICSE24], we are leveraging LLMs for code refactoring, a common practice in software development where developers make behavior-preserving changes to their code to prevent the accumulation of technical debt. Despite extensive research and the development of automated recommendation and application of refactorings, studies reveal that these tools often fail to recommend refactorings that align with developers' preferences and acceptance criteria. Given that LLMs have been trained on large code corpora, they are more likely to imitate human behavior. If we harness the familiarity of LLMs with the way developers program, we could suggest refactorings that developers are likely to accept.

Our formative study on thousands of refactorings from open-source projects revealed that LLMs are very effective in giving expert suggestions, yet they are unreliable: up to 63% of the suggestions are hallucinations. We developed a novel approach that synergistically combines the creative potential of LLMs with static analysis to enhance the refactoring suggestions generated by LLMs. Additionally, we utilize the safety measures of static analysis within IDEs to execute refactorings safely. Starting from candidates suggested by LLMs, we filter, further enhance, and then rank suggestions based on static analysis techniques from program slicing. Our thorough empirical evaluation on thousands of refactorings from open-source projects shows that our approach outperforms the previous state of the art by a factor of 6x. We conducted surveys with dozens of professional developers using our approach and 80% of the respondents agreed with the recommendations provided by our approach. Some commented "These suggestions made me look at this code with new eyes once more, and I will try to refactor it in a different way."

In our second work [FSE24b] we employ LLMs to dramatically improve the effectiveness of Transformation by Example (TBE) systems that automate frequently occurring code changes. TBE systems are limited by the quality and quantity of the provided input examples. Thus, they miss transforming code variations that do not have the exact syntax, data-, or control-flow of the provided input examples, despite being semantically similar. We harness LLMs' creativity to produce semantically equivalent, yet previously unseen variants of code changes that meet three criteria: correctness (semantic equivalence to the original change), usefulness (absence of hallucinations), and applicability (aligning with the primary intent of the original change). We instantiate these into our tool PyCraft, which synergistically combines static code analysis, dynamic analysis, and LLM capabilities. Using these richly generated examples, we inferred transformation rules and then automated these changes, resulting in an increase of up to 39x, with an average increase of 14x in target codes compared to the previous state-of-the-art TBE tool that relies solely on static analysis. We submitted patches generated by PyCraft to a range of projects, notably esteemed ones like microsoft/DeepSpeed and IBM/inFairness. Their developers accepted and merged 83% the 86 code changes.

This shows the usefulness of our novel approach and ushers us into a new era when LLMs become effective AI assistants for developers and instead of displacing developers it makes them more productive. We **demoed our research to Tim Bates, former CTO of Lenovo**, his reaction was: "OEMs are still playing catch-up in the software game, lagging behind tech juggernauts ... Enter the AI code assistant, the great equalizer. It's the ace up the sleeve for OEMs lacking that end-user mojo, leveling the playing field against giants like Apple and Amazon. Ever chatted with OEM bigwigs about letting their engineers buddy up with AI assistants? Trust me, it's a game-changer for cooking up smarter, more user-focused solutions."

Automate code evolution for AI/ML engineers and data scientists

A thrust of my research group [ICSE23, ICSE22a, TOSEM21] is to automate code evolution as we entered into a new era of Software Development, dubbed Software 2.0. In this era, programmers augment the traditional software development with programming with learned models. Engineers amass training data and feed it into an ML algorithm that will synthesize an approximation of the function whose partial definition is that training data. They use libraries such as TensorFlow, Scikit, and Keras. Python has become the lingua franca for developing ML systems, both in commercial and open-source. As we show in our TOSEM21 paper, the ratio of new Python projects that use ML libraries increased from 2% in 2013 to 50% in 2018. So software 2.0 is here to stay.

My group has a track record of starting a new research direction with formative studies to determine what are the software development pain points. This ensures that we are solving the right problem. It also helps us distill best practices from the community. In our TOSEM21 paper we presented the first and largest study on the challenges of evolving ML systems. To conduct this study we performed static analysis and mined software repositories for the 3000 top-rated ML systems in GitHub, and surveyed 100 ML developers. We found that ML systems face unique challenges and pressure to change as they evolve: when the Data changes (e.g., new data that needs cleaning), when the ML model changes (e.g., when we need to retrain the model), in addition to regular code evolution. Moreover, despite the fact that many ML developers are not professionally-trained software engineers, the tools for developing and evolving ML code are in their infancy. Unlike in traditional software development where developers use IDEs and tools to help them change code, when it comes to changing ML code, developers make all these changes by hand. These changes are tedious, error-prone and time-consuming, This is what makes ML code development expensive.

To advance the science and tooling in ML software evolution, in our ICSE22a paper we conducted the first and most fine- grained study on repetitive code change patterns in a diverse corpus of 1000 top-rated ML systems comprising 58 million SLOC. To conduct this study we reuse, adapt, and improve upon the state-of-the-art change mining techniques. Our novel tool, R-CPatMiner, mines over 4M commits and constructs 350K fine-grained change graphs and detects 28K frequent change patterns. Using thematic analysis, we identified 22 pattern groups and we reveal 4 major trends of how ML developers change their code. We surveyed 650 ML developers to further shed light on these patterns' applications.

Inspired by these results, in our ICSE23 paper we introduced a novel, data-driven approach to automate many new kinds of program transformations that ML engineers use. We showed that it is feasible to employ a novel automated workflow that mines frequent code changes from thousands of open-source repositories, infers the transformation rules, and then transplants them automatically to new target sites. We designed, implemented, evaluated and released this in a tool, *PyEvolve*. At its core is a novel data-flow, control-flow aware transformation rule inference engine. Our technique allows us to advance the state-of-the-art for program-by-example tools; without it, 70% of the code changes that PyEvolve transforms would not be possible to automate with the current state of the art techniques. Our thorough empirical evaluation of over 40,000 transformations shows 97% precision and 94% recall. By accepting 90% of CPATs generated by PyEvolve in best-in-class open-source projects such as TensorFlow, PyTorch, and AWS, developers confirmed that our tool is useful. Their feedback included: "your changes are cleaner and faster", "changes are good, I'm not sure why I missed that". This shows that even in the most finely-tuned, best-in-class projects produced by experts, they miss opportunities to use best practices, thus our techniques can help.

Increase the scalability and precision of interactive program analysis

Another major thrust of my group's research [ICSE22b, TSE22, FSE20, ICSE19a, ICSE19b, ICSE18, FSE17, FSE16a, FSE16b, ASE16, ICSE16] is to (i) scale up program analyses so that our community can analyze large codebases of hundreds of millions of LOC, and (ii) increase the precision of static analysis so that others can trust the results.

Today's code bases used in industry at companies such as Google, Microsoft, Amazon are so large that they cannot be loaded in traditional IDEs, so companies keep code in the cloud. The current generation of refactoring tools (i.e., interactive source-to-source transformations) require to load in memory the whole codebase to analyze, which is not feasible to do at this scale. For example, we collaborated with software engineers at Google, where the Java codebase is 300M LOC. Together we developed the next generation of refactoring tools [ICSE19a] that work in a distributed manner, using MapReduce on the cloud, thus scaling up an inter-procedural whole program analysis. Our empirical evaluation shows that we efficiently performed global refactorings across the entire Google's code base in a matter of 30 minutes. Moreover, we generated 139 performance-related refactoring patches in 7 best-in-class, highly optimized open-source applications. For example, the developers of CASANDRA, a big-data processing engine used in each of the Fortune-500 companies, accepted 15 of our refactoring patches. Jeff Jirsa, a CASANDRA lead developer wrote about one of our refactorings: *This patch was deep inside the database and actually is very important for performance*. This shows that our new approach is safe, applicable, and useful.

As an example of improving the reliability of static analysis, consider our research [TSE22] on inferring where in the code refactoring already took place. This is important for understanding the evolution of code, program comprehension, and enables many more applications such as better code merging, code reviews. Refactoring inference has been a very active line of research that stifled in the past due to results which were not accurate enough to be used in other tool chains. Our new inference tool, RefactoringMiner, works at finer granularity: it applies differential static analysis on consecutive code commits, without requiring that users experiment with thresholds. With a 98% precision and 87% recall, RefactoringMiner is significantly more accurate than previous tools. Also, it is ultra-fast (7x faster than previous state of the art) so it can be used with continuous integration tools every time when a user commits code. Moreover, our release of the dataset and the tools last year enabled other researchers to publish 65 papers.

The secret to obtaining ground-breaking results in our field is the collaboration with other researchers that bring a plethora of novel approaches and skills (e.g., statistical learning [FSE16a], genetic algorithms [FSE16b], MapReduce [ICSE19a]). In return, we have given back to the community by providing research infrastructures and datasets that other researchers can build upon. We extend this community refactoring infrastructure funded by a recent NSF CCRI grant.

Improving responsiveness in mobile apps via asynchrony

At OSU I opened the field of refactorings for the domain of mobile apps [MobileSoft16, ASE15a, ASE15b, IEEE Software15, FSE14a, ICSE14a, MobileSoft14]. According to Gartner, mobile and wearable apps get more than 200 billion downloads/year. The number one performance problem that plagues mobile apps is executing blocking I/O operations (e.g., accessing the web, cloud, database) synchronously, which freezes the UI and frustrates users. The key solution is to use asynchronous programming. Asynchrony helps apps stay responsive because they can continue with other work.

With my students, we conducted large-scale formative studies, one of them [ICSE14a] won the **ACM SIGSOFT Distinguished Paper award at ICSE** (the flagship SE conference), to understand how programmers underuse or misuse asynchronous programming. In this formative study we analyzed 1387 apps, comprising 12M SLOC produced by more than 3000 developers. Besides serving as inspiration for our research, this study has several practical implications. First, it is a tremendous resource for educating developers who learn about asynchronous programming from seeing both positive and negative examples. Our companion webpage shows hundreds of relevant examples of how to use async APIs. We received more than 40,000 unique visitors within 1 year. Second, it provides value for researchers, tool vendors, and the software testing/verification community. For example, Microsoft library designers confirmed that our findings are useful and will influence the future development of async constructs in C#.

Grounded on such studies, our refactoring research addresses key challenges such as reasoning about a programming model that inverts the flow of control, designing novel inter-procedural analyses to determine non-interference of asynchronous operations with the main thread of execution, preserving the behavior of exceptions, etc. Such static analyses in the context of event-driven systems pose unique challenges.

Our growing list includes refactorings for adding asynchrony into synchronous code [FSE14a] or replacing callback-based execution of legacy async code with modern async constructs that have pause-and-replay semantics [ICSE14a]. Another refactoring [ASE15a] converts between two fundamentally different async constructs in Android: it changes shared-memory communication into message passing that resembles distributed programming. This refactoring eliminates memory leaks, lost results, and wasted energy.

Our refactoring tools generated several hundreds of patches that correctly introduced asynchrony in mobile apps and were accepted by their developers, thus showing the practical value of our research. For example, in our ICSE14a paper, we applied and reported refactorings in 10 apps. 9 replied and accepted each one of our 28 refactorings. Using our static analysis, we found and reported misuses in 19 apps. All replied and accepted each of our 286 tool-generated patches. **Microsoft has incorporated one of our techniques** into the official release of Visual Studio. We are currently working with Google to incorporate some of our analyses into its static checkers that any app would need to pass before it is accepted into the Google Play store.

One of our refactoring tools enables Android developers to upgrade their apps from a static to the new dynamic permission model in Android 6. This improves the security and privacy of apps as it reduces the risk of permission abuse from malware. Inspired by a large-scale formative study of how developers use permissions, we designed and implemented a toolset that recommends and inserts permission-related code. Google engineers **demo-ed our toolset at Google I/O 2016**. We are integrating it into the **official release of Android Studio**, used by 90% of Android developers.

Retrofitting Parallelism into Existing Sequential Programs

As a visiting research professor at Illinois, I opened the area of interactive transformations for retrofitting parallelism into sequential programs [TOSEM15, STVR15, ICST13, FSE12, TR12, SOFTW11, IWMSE11, ICSE11, ICSE09, TR09, ASE09a, OOPSLA09]. In the multi-core era, programmers turn to parallelism when they need to optimize their programs for performance. They also use parallelism to enable new applications and services, not previously possible, e.g., better quality of service, improved security, and richer user experiences.

One approach to parallelizing an existing sequential program is to rewrite it from scratch, but this requires a lot of programmer effort. Another approach is to use an *automatic parallelizing compiler*. Despite continuous improvements on compilers, many times programmers still need to change their programs to make them more parallel. Knowing where to introduce parallelism requires domain knowledge and understanding of the program's algorithms and data structures. In practice, the most widely used approach is to parallelize a program incrementally by changing the existing code. Each small step can be seen as a behavior-preserving transformation, i.e., a *refactoring*. Programmers prefer this approach because it is safer: they maintain a working, deployable version of the program. Also, the incremental approach is more economical than rewriting a program from scratch.

Refactoring tools. My group's research automated more than a dozen refactorings to change sequential programs into parallel programs that are thread-safe, scalable, and efficient. Our refactorings fall into three categories. First, refactorings for thread-safety make a program thread-safe, e.g., by synchronizing accesses to shared state via library classes. Second, refactorings for throughput add multi-threading via task and loop parallelism. Third, refactorings for scalability replace lock-based synchronization with accesses to lock-free, highly scalable data structures. These interactive refactorings combine the strengths of the programmer and the computerized tool.

I will illustrate the challenges of implementing one refactoring that converts a mutable into an immutable class. Since its state cannot be mutated once an object is properly constructed, the immutable class is thread-safe. Immutability plays an important role in other fields, e.g., computer security, memory optimizations, and distributed computing. Our refactoring tool, IMMUTATOR, first makes the class and its fields final. Second, it finds all methods in the class that mutate the transitive state and converts them into factory methods that return a new object whose state is the old state plus the mutation. This is similar with factory methods in immutable classes like `String` where `toUpperCase()` returns a new `String` with some characters replaced. Third, it finds the objects that enter into or escape from the object's transitive state and clones these objects, so that the state can not be mutated by a client who holds a reference to state objects. IMMUTATOR uses a demand-driven pointer-analysis.

We evaluated IMMUTATOR by (i) running it on 346 classes from popular open-source projects, (ii) a case-study of how open-source developers refactor manually, and (iii) a controlled experiment. The results show that (i) the refactoring is widely applicable, (ii) several of the manually-performed refactorings are not correct (the code contains subtle mutations and non-cloned entering/escaping objects) whereas refactoring with our tool is safer, and (iii) refactoring with our tool is fast (2.3 seconds/class) and saves the programmer from analyzing 84 methods and changing on average 42 lines per refactored class. The participants in the controlled experiment took an average of 27 minutes per refactoring. Thus IMMUTATOR, dramatically improves programmer productivity.

Our empirical evaluation of the whole toolset shows that it is useful: it reduces the burden of analyzing and modifying code, it is fast enough to be used interactively, it correctly applies transformations that open-source developers applied incompletely, and the refactored code exhibits good speedup. Referring to our refactoring tool RELOOPER, Doug Lea, the architect of the Java concurrency library, wrote on the Java concurrency mailing list: *I was very impressed with* RELOOPER *... I expect it will be useful to just about anyone interested in exploring these forms of parallelization.*

Another one of our refactorings, LAMBDAFICATOR [FSE13], is **shipping with the official release of the NETBEANS IDE**, which is used daily by millions of Java developers. LAMBDAFICATOR converts imperative Java code into functional style iterators that provide unobtrusive parallelism. For example, it converts `for` loops that iterate over collections to functional operations (e.g., `map` or `filter`) that use lambda expressions.

When appropriate, I am not afraid to try ideas that are not mainstream. For example, in data race detection, much of the research in the recent years has shifted to dynamic race detection, because the community lost hope in approaches purely based on static analysis. Our ISSTA13 results restored hope that static race detection can be very effective in finding races in real programs with very few false positives, thus it won the ACM SIGSOFT Distinguished Paper Award.

Porting to new Parallel Programming Languages. We have also developed tools to port existing code to new parallel programming languages. As a collaborator to the compiler group at Illinois, we developed Deterministic Parallel Java (DPJ) [OOPSLA09], a language that aims to make parallel programming deterministic by default. At the heart of DPJ is a type and effect system that ensures that the parallel tasks are non-interfering. Our tool, DPJIZER [ASE09a] reduces the burden of writing annotations, by automatically inferring the method effects using a constraint-based algorithm.

Software analytics with actionable insight to inform better decisions. Our empirical study [IWMSE11] of more than 200 refactorings used in practice inspired us and several other researchers to build refactorings for parallelism. In our FSE12 paper, we analyzed 655 open-source applications that adopted Microsoft's new parallel libraries, comprising 17.6M lines of application code written in C#. Our companion webpage shows thousands of relevant examples of how to use parallel APIs. Our webpage received **more than 120,000 unique visitors** within 3 years. Microsoft library designers confirmed that our findings are useful and have already influenced the development of their libraries.

Our best-paper award research [ICST13] analyzed quantitatively and qualitatively common misuses of concurrent collections in Java. We focused on CHECK-THEN-ACT idioms, where a check on the collection (such as non-emptiness) precedes an action (such as removing an entry). In 28 widely-used open source Java projects, we discovered 60 new bugs. The developers confirmed and accepted our patches. In addition, our results already influenced the design of the Java 8.0 concurrent library: Doug Lea introduced new API methods in `ConcurrentHashMap` as a result of our findings.

Automated Upgrading of Component-Based Applications

For my PhD dissertation I developed a set of techniques to automatically and safely upgrade applications to use the latest version of the components (libraries, frameworks) upon which they are built [ICSE08, TSE08, ICSE07, ECOOP06, JSME06, ICSM05]. Ideally, the interface of a component never changes. In practice, new versions of software components often change their interfaces and so require applications that use the components to be changed. My goal was to learn what these component changes are and to automatically integrate them into applications.

To be practical, my solution addresses the needs of both component and application developers: application developers want an automated and safe (behavior-preserving) way to upgrade component-based applications to use the newer versions of components; component developers are reluctant to learn any new language or write any specifications extraneous to the regular component development. The current state-of-the-practice for component evolution uses text-based tools, but syntax alone is too poor to express the complexities of API evolution. My main insight is to **treat API changes as first-class citizens**: the evolution is expressed in terms of program transformations with well defined semantics.

After finding that over 80% of the API changes that break existing applications are refactorings [ICSM06, JSME06], I used refactorings to formally express component evolution. To reduce the burden of component developers writing annotations that describe the component evolution, we need to automatically detect refactorings applied between two versions of a component. One approach to detect refactorings is to extend the refactoring engine to record refactorings. I collaborated on the integration of record-and-replay in the official release of Eclipse 3.2. However, refactoring logs are not available for existing versions of components. Therefore, it is important to be able to infer them.

I developed a novel algorithm [ECOOP06] that detects applied refactorings using a combination of fast syntactic analysis (that I adapted from the experts in Data Mining at Illinois) and precise semantic analysis. Empirical evaluation of REFACTORINGCRAWLER, our implementation, shows that it scales to real-world components, and its accuracy in detecting refactorings is over 85%, a dramatic improvement over the 20% accuracy of the previous solutions. REFACTORINGCRAWLER has been used at six research institutions to detect refactorings for the purpose of mining software repositories and program comprehension. REFACTORINGCRAWLER is used in industry (at CuramSoftware and Google) to validate that the version release documentation contains all API changes in a given release.

When solving the upgrading problem, I solved a larger class of problems: refactoring-aware software merging [ICSE07, TSE08]. I developed the world's first smart versioning tool that semantically merges API changes.

Testing and evolution

I always enjoy opportunities for collaborative research. I would like to highlight the successful collaboration with the software testing group led by Darko Marinov at Illinois. Combining my expertise on refactoring and transformation systems with their expertise on generating complex test inputs, our group developed techniques to identify flaky tests which are extremely detrimental for continuous integration practices [FSE17, ASE16]. In another collaboration, we developed a technique [FSE07] for automated testing of refactoring engines. We have applied our technique to testing Eclipse and NetBeans, two popular open-source IDEs for Java, and we have exposed 21 new bugs in Eclipse and 24 new bugs in NetBeans. Our technique was later incorporated in the testing infrastructure at Sun Microsystems (now Oracle).

Future Research

In the medium term, my research focus is on developing a fresh perspective on a programming environment that places transformations at the center of software development. NSF previously funded us to work on this research with $2.2M, and our strong results will provide a great basis for the future work. We are at the cusp of a major breakthrough in increasing programmer productivity by leveraging Generative AI and LLMs for software engineering tasks. My group pioneered the use of Generative AI as an AI assistant for code transformations - I believe there is more room to leverage GenAI both for professional developers and end-user programmers. I am particularly excited to lead research that brings the latests advancements from the software refactoring community to ML developers and data scientists in STEM fields. We are partnering with PyCharm, the leading IDE for Python development, to conduct discovery sessions to learn about software evolution tasks that are good candidates for automation in ML and STEM codebases. Below I describe exemplars of research that my group and the PPI Center that I lead are excited to work on. Long term, I will continue to contribute to SE (and indirectly to ML/AI, Security, HPC) by expanding these areas with practical, scalable, and safe transformations.

Extending code automation through the use of Generative AI. Our preliminary studies reveal the potential of GenAI not only to automate tasks previously considered non-automatable but also to broaden the scope of already automated techniques in software engineering beyond our prior understanding. My goal is to further extend these advancements. While on sabbatical at JetBrains I learned about the *discoverability challenge*: state of the art IDEs like IntelliJ support more than 2000 features, yet developers are unaware of most of these features and many times perform the same tasks manually. While LLMs can suggest refactorings and program enhancements, they do not teach developers and students about the safe ways to apply these transformations within the IDE. I plan to study existing IDE automations that suffer from a lack of discoverability and late awareness, and also those whose suggestions do not align with what experts would actually perform. While many LLM suggestions have WOW effect, the inherent limitations of LLMs (e.g., hallucinations and unconventional suggestions – up to 70% of suggestions as indicated by our recent studies) underscore the necessity of supplementary techniques like fine-tuning and validation tools grounded in static/dynamic code analysis. I strongly believe for a synergistic integration of LLM-based suggestions with IDEs and static/dynamic analysis-based safety components. This will reliably harness the full power of LLMs in advancing automation within software engineering, and educate developers and students thereby alleviating the discoverability and delayed awareness problems.

Refactoring for ML and STEM Codebases. Given the rise of data science, researchers in the broader STEM disciplines are relying on computer programs to model the phenomena they are studying. However, most of these scientists are not trained as programmers. Their first priority is to create computer models that help them answer the research questions important in their communities, whereas the internal code quality is of secondary concern. However, these systems are prone to technical debt issues, especially when such systems are long-lived. In the long run, research progress is stifled as the acquired technical debt prevents them from further extending, reusing, and adopting existing STEM codes.

We compared the code quality of 3,000 ML code bases primarily used by STEM researchers with 3,000 traditional code bases used in enterprise systems. To make the comparison fair, we selected projects implemented in the same language (Python - which is the widely-used programming language in the Data Science community), and we selected projects that have the same level of maturity and popularity (as measured by stars in GitHub). Our results show a stark contrast between the code quality of the two sets of projects. The majority of STEM projects in our representative corpus predominantly use scripts that do not encapsulate units of computations in neither object-oriented classes nor functions. For example, only 12% of the STEM projects use classes and only 31% use functions. In contrast, in enterprise systems these rates are more than double: 40% for classes and 67% for functions. We also found that only 50% of the STEM projects use the try-catch statements to sanitize their code and catch runtime exceptions, as opposed to 90% of the enterprise projects using try-catch statements. Independently of us, other researchers observed that Data Science codebases do not follow established software engineering best practices, e.g., Data Science projects suffer from a significantly higher rate of functions that use an excessive number of parameters and local variables.

To bring the benefits of refactoring to the STEM community, we will generalize the refactoring support beyond the main programming language used in refactoring research (i.e., Java) via a *data-driven approach*: first, we will mine millions of refactorings automatically by implementing multiple language adapters to expand the Java refactoring mining tools (e.g., our previously developed RefactoringMiner [TSE20]) to support Python and other data science languages used by the STEM community. Second, using this rich data-set, we will recommend and synthesize refactorings using program-by-example for new code sites to modernize code.

Launching a new Industry-University Cooperative Research Center (IUCRC) or Industry-funded Center.

In 2020 NSF awarded $1.5M to University of Colorado and Oregon State University to form a new IUCRC (see `https://PPICenter.org`) under my leadership. Companies from several IoT verticals (high-tech, automotive, precision ag, etc.) are supporting our Center. Despite the challenges caused by the pandemic, in 2022 we have grown the industry memberships and added a new university site at Oakland University. As the travel restrictions are lifting up, we are poised for exponential growth: at our first in-person IAB meeting on April 7-8, 2022 in Detroit, MI, we had 60+ industry executives and leaders eager to learn about our PPI Center and how to join us. While I am the Founding Director of this new Center, the current results are the fruition of a 5-year long process involving a dedicated team of faculty, administrators, and industry partners at both universities. I plan to leverage the lessons that I learned along this journey to launch another IUCRC or an Industry-funded Center at the U in an area that leverages the strengths of the SoC.

The **Industry Center is perfectly aligned with President Randall's and Dean Brown's strategic vision**. First, it **accelerates the research impact**. In his inaugural address on 03/23/22, President Randall talked about "picking up our clock speed by increasing the velocity of our engagement to speed up knowledge transfer." An industry-funded center affords our faculty such an accelerated impact. For example, one of our PPI faculty worked with our industry-member Intel and they were able to speed up end-to-end industrial AR applications that are important for Intel clients by more than 2X over the state of the art, which is a significant improvement in user experience.

Second, an **industry-funded center diversifies the U's research portfolio** by growing industry support on par with that provided by NSF, and helps the U to make progress toward the $1B/year research funding. I have led discovery sessions with 100+ C-level executives from industry. I plan to share this experience and join forces with others at the U who are conducting discovery sessions for the establishment of a FinTech epicenter at the U.

Third, an industry-funded center **builds on collaborative, interdisciplinary research theme** at the U. I am excited about launching together with faculty from SoC a new IUCRC or an industry-funded center. Aligning with the strong research & faculty in SoC, there are tremendous synergies with the Systems group (e.g., POWDER with their testbed for smart connected IoT services and 5G), Data Science, PL (verification and validation), Security, HCI (usability, ethics, privacy), and the SCI Institute (with visualizations for various smart IoT verticals). Together, we can make the SoC an R&D partner for the vibrant high-tech industry in the SLC and Silicon Slopes, so that we bring additional value, beyond the workforce development.

Leveraging the industry-funded center so that the whole Utah team can benefit. I am committed to using the Center for other initiatives and the bigger picture at the U. (i) Mentoring junior faculty: the Center leadership team coaches junior faculty on how to successfully access industry funding opportunities, serve the society, and tackle the big challenges that have a significant impact. (ii) Enhance the Master of Software Development program in the SoC, using the feedback from the key industry partners. For example, Bob Wold – a VP from Trimble, is dedicated to use the relationship with our PPI Center faculty for continuing education of their existing workforce. (iii) Internships for students: the industry Center will provide experiential learning for students, and better placement in key roles in industry. (iv) Hiring new faculty: The connections with industry will serve as a strong attraction for future faculty hires. (v) Going after bigger projects like ERCs and NSF Expeditions: having a strong team with proven track record of working together, and leveraging the access to industry and data will position the team to win bigger programs in the future.

Leveraging my relationship with JetBrains for becoming a preferred partner university. JetBrains provides scholarships and other funding for students to conduct research thesis projects with collaborators from JetBrains. This enables students to have access to state-of-the-art program analysis tools and ML models, datasets on various programming tasks usages, etc. Leveraging my relationships and trust with JetBrains, I plan to establish a master agreement between JetBrains and the U so that many students and faculty can benefit. I want to connect other faculty colleagues with JetBrains researchers. This will position the SoC as leader in LLM-based applications, which will help us recruit strong faculty for years to come.

JetBrains provides practical hands-on tutorials, video courses and podcasts on learning programming languages (e.g., Kotlin – released by JetBrains and now the official language for developing Android apps) and professional SW development (e.g., testing, program understanding, debugging, version control management, continuous integration, etc). I will leverage my relationships with JetBrains to bring such resources to programming classes in the SoC, making them more modern and highly practical. Having access to these highly-polished resources will also lighten up the load for the TAs for programming-intensive classes.

In summary, I am looking forward to long-term collaboration. I am committed to helping you create a non-dependent, high-performance culture so that the whole team at the U can benefit.

Select References

For a complete list, see my CV.

[ICSE23] M. Dilhara, A. Ketkar, and D. Dig. PyEvolve: Automating Frequent Code Changes in Python ML Systems. In *International Conference on Software Engineering*, pp 995-1007, May 2023. Acceptance ratio: 26% (209/796).

[ICSE22a] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig. Discovering Repetitive Code Changes in Python ML Systems. In *International Conference on Software Engineering*, pp 736-748, May 2022. Acceptance ratio: 26% (197/757).

[ICSE22b] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, T. Bryksin. Inferring and Applying Type Changes. In *International Conference on Software Engineering*, pp 1206-1218, May 2022. Acceptance ratio: 26% (197/757).

[TSE22] N. Tsantallis, A. Ketkar, D. Dig. RefactoringMiner 2.0. In *IEEE Transactions on Software Engineering (TSE)* 48(3), July 2022, pp 930-950.

[TOSEM21] M. Dilhara, A. Ketkar, D. Dig. Understanding Software-2.0: a Study of Machine Learning library usage and evolution. In *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30 (4), July 2021, pp 1-42.

[ICSE19a] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, E. Aftandilian. Type Migration in Ultra-Large-Scale Codebases. In *International Conference on Software Engineering*, pp 1142-1153, May 2019. Acceptance ratio: 21% (109/529).

[ICSE18] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, D. Dig. Accurate and efficient refactoring detection in commit history. In *ICSE*, pp 483-494, May 2018. Acceptance: 20% (105/502).

[FSE17] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility In *International Symposium on the Foundations of Software Engineering*, pp 197-207, Sept 2017. Acceptance ratio: 24% (72/295). **ACM SIGSOFT Distinguished Paper Award**

[FSE16] A. Nguyen, M. Hilton, M. Codoban, H. Nguyen, L. Mast, E. Rademacher, T. Nguyen, D. Dig. API Code Recommendation Using Statistical Learning from Fine-grained Changes In *International Symposium on the Foundations of Software Engineering*, pp 511-522, Nov 2016. Acceptance ratio: 27% (74/273). **ACM SIGSOFT Distinguished Paper Award**

[ICSE16] J. Kim, D. Batory, and D. Dig. Improving Refactoring Speed by 10X. In *International Conference on Software Engineering*, pp 1145-1156, May 2016. Acceptance: 19% (101/530). **Runner-up ACM SIGSOFT Distinguished Paper Award**.

[ASE15] Y. Lin, S. Okur, and D. Dig. Study and Refactoring of Android Asynchronous Programming. In *International Conference on Automated Software Engineering*, pp 1–11, Nov 2015. Acceptance ratio: 20.7% (60/289).

[ICSE14a] S. Okur, D. Hartveld, D. Dig, and A. van Deursen. A Study and Toolkit for Asynchronous Programming in C#. In *International Conference on Software Engineering*, pp 1117-1127, May 2014. Acceptance ratio: 20% (99/499). **ACM SIGSOFT Distinguished Paper Award**.

[FSE13] A. Gyori, L. Franklin, J. Lahoda, and D. Dig. Crossing the Gap between Imperative and Functional Programming through Refactoring. In *International Symposium on the Foundations of Software Engineering*, pp 543-553, Aug 2013. Acceptance ratio: 20% (51/251).

[ISSTA13] C. Radoi and D. Dig. Practical Static Data Race Detection for Java Parallel Loops. In *International Symposium in Software Testing and Analysis*, pp 178-190, July 2013. Acceptance ratio: 26% (32/124). **ACM SIGSOFT Distinguished Paper Award**.

[ICST13] Y. Lin and D. Dig. Check-then-Act Misuse of Java Concurrent Collections. In *International Conference on Software Testing, Verification and Validation*, pp 164-173, March 2013. Acceptance: 25% (38/152). **Best Paper Award**

[ICSE11] F. Kjolstad, D. Dig, G. Acevedo, M. Snir. Transformation for Class Immutability. In *International Conference on Software Engineering*, pp 61–70, May 2011. Acceptance ratio: 14% (62/441). **Runner-up ACM SIGSOFT Distinguished Paper Award**