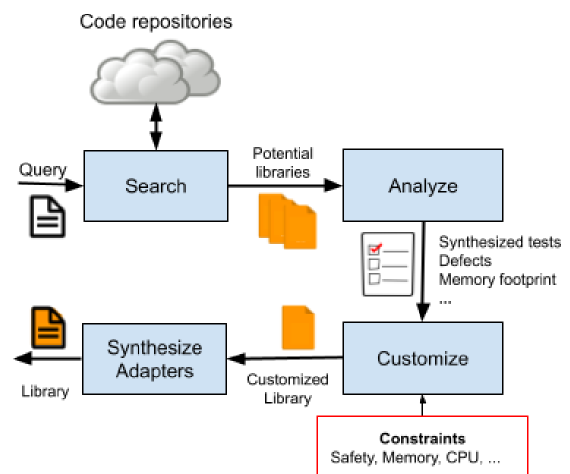
**My research aims to develop techniques, tools, and workflows that improve developer productivity and software quality**. Software systems now use a vast suite of software components and tools, including software libraries, build systems, interactive development environments, testing frameworks, and compilers. As a result, the developer experience and the resulting reliability, security, and performance of the software systems that developers produce are directly related to the quality of the software components and tools that developers can discover and utilize.

My recent research focuses on designing **program analysis techniques** for **software libraries**. Software libraries play a critical role in the software development process. They expose APIs that provide useful functionality and create abstractions that enable developers to focus on the core application logic, leading to modular software development. Furthermore, several factors influence optimal library utilization – (a) **awareness** of the most appropriate libraries, (b) the ability to **reason** about a library across various dimensions that include correctness, security, performance, and memory usage, and (c) the **ease of incorporating** a library to serve the functional requirements of the application.

Changing trends in software engineering have exacerbated the complexities of modular software development, including (a) an order of magnitude increase in available libraries,[1] which imposes a cognitive burden on the developer to make the right choices, (b) fast-paced software development resulting in the deployment of under-tested software products in a race to capture the market, negatively impacting software quality, (c) heightened churn of software developers where code ownership is in constant flux, compromising software support, (d) a broad spectrum of programming languages with custom syntax, semantics and recommended best practices, affecting the ability to reason about code, and (e) growth in the number of smart devices and appliances from numerous vendors, with varying configurations that include memory capacity, processing power, operating system, networking support, etc., thus creating a need for software customization. These changes necessitate a **fundamental rethink** of the **software development process** surrounding the usage of software libraries.

## Research Vision

**I envision a future where developers are armed to navigate the constantly evolving software ecosystem effectively**. My goal is to design approaches to **discover, reason, customize, and adapt code** to efficiently build defect-free software systems to realize this vision. I am particularly interested in exploring the following research questions: a) Is it possible to search for code with constraints on program properties that include functionality, performance, memory consumption, binary size, and concurrency? b) Can we automatically construct targeted tests that expose the violation of a specified property? c) Is it possible to automatically construct code hybrids that combine the desired properties of input code samples? d) Can we automate the integration of software libraries into user applications? Successful research in this area will reduce the complexities of modern software development.




I have investigated a subset of these research questions in the context of object-oriented languages, specifically for Java. The vibrant community of Java developers and a large corpus of available code make it an appropriate candidate to explore the above questions. Also, the complexity associated with class inheritance, polymorphism, data encapsulation, concurrency, data/interface accessibility, and aliases makes Java well-suited to evaluate the applicability and rigor of any proposed solution.

To ease the discovery of Java classes based on input functionality from extensive collections of codebases, I designed a **search technique** [2] and evaluated it on open source

[1] GitHub (`http://github.com/about`), a popular source code hosting platform, hosts around 200 million repositories.

Java classes. The technique leverages the complementary strengths of distributed embedding-based search and program analysis to yield meaningful results. I also built the **first set of algorithms** that reason about the safety of accessing a class under concurrency by automatically **synthesizing targeted tests and executions** to reveal thread-safety violations in classes [4, 7, 5, 8]. These techniques employ a carefully crafted combination of static and dynamic analysis, constraint solving, and automated code generation to construct defect-revealing multithreaded tests. Further, to ease library integration, I developed an approach to synthesize **verified adapters** [3] by integrating symbolic execution, constraint solving, and program synthesis, enabling a drop-in replacement of classes.

## Searching for Java classes

Developers maintaining software systems often need to replace one of the classes in the source codebase with a functionally equivalent class. Potential use cases include deprecation of deployed classes, the need to change vendors to satisfy organizational needs or intellectual property constraints, improved performance or memory usage, or the desire to identify and use better versions that may contain fewer defects. While software repositories often have suitable class replacements, finding such replacements can be challenging owing to the extant code bases (billions of lines of code) and the intricacies of object-oriented languages. The challenges include (a) functionality spread across multiple source files and classes, (b) complex type hierarchies created by a mixture of primitive types, user-defined types, generics, and wild cards, (c) varying accessibility levels for the APIs and the data, (d) distinct algorithms that offer the same functionality, (e) unique implementation styles, and (f) finally, the vast search space of classes rendering a pure program analysis based approach intractable.

I designed a new technique and implemented a system, ClassFinder [2], for automatically finding replacement Java classes. Given a query class, ClassFinder automatically searches large codebases to identify and rank potential replacement classes. ClassFinder combines two complementary techniques: embedding-based class ranking and method compatibility matching. Embedding-based class ranking maps the method names in a class to a high-dimensional vector, with the cosine similarity metric over the resulting vector space serving as a proxy for the intended functional similarity between the classes. This ranking eliminates the irrelevant classes and narrows the search space. Subsequently, method compatibility matching computes a type-similarity metric between matched methods and fields to effectively derank candidate replacement classes that 1) operate with incompatible or overly specific types, 2) contain empty or placeholder methods, or 3) contain semantic differences. ClassFinder was evaluated on several open-source Java query classes with a search corpus of ≈**600 thousand open-source classes. The results indicate that ClassFinder can effectively find suitable replacement classes.**

## Synthesizing Verified Adapters

Manually updating an application to use a different class can be cumbersome and error-prone. For instance, even when class versions are updated, backward compatibility is not always maintained. Furthermore, ensuring that the application's behavior is unchanged can become non-trivial because an existing class in the library can differ from the chosen replacement class in the new library across multiple dimensions. These include signatures of the provided interfaces, the underlying functionality offered by these interfaces, and the internal data representation which impacts their implementation. This motivates the need for a technique that can synthesize an **adapter** for a given replacement class so that the synthesized adapter is **equivalent** to the existing class.

I designed and implemented a new algorithm [3], which, given a pair of original and replacement classes, automatically synthesizes an adapter class that implements the same interface as the original class by leveraging the APIs offered by the replacement class. To perform this replacement, the system constructs an inter-class equivalence predicate that defines state equivalence between the instances of the two classes. These predicates are synthesized by symbolically executing methods defined by the classes to identify a set of relevant symbolic expressions and equating them appropriately. The algorithm generates sketches that encode a search space of candidate method invocation sequences for each adapter method. These sketches are solved using the inter-class equivalence predicate to synthesize the required adapter class. Thus, the technique can synthesize **intricate adapter** classes that are **guaranteed to be equivalent** to the corresponding input classes.

## Analyze: Synthesizing multithreaded tests

Analyzing third-party libraries for software quality issues that include software bugs, memory bloat, execution bottlenecks, and security issues is essential to ensure the reliability of a client application. Further, detecting **concurrency bugs in libraries** can be challenging due to many plausible library usage scenarios and the intricacies associated with concurrency bugs. Successfully detecting these bugs in libraries requires identifying: i) the methods that need to be invoked concurrently, ii) the inputs passed to these methods, iii) the execution context leading up to the method invocations, and iv) the interleaving of the threads that cause the erroneous behavior. Unfortunately, neither fuzzing-based testing techniques nor over-approximate static analyses are well-positioned to detect subtle concurrency defects while retaining high accuracy alongside sufficient coverage. While dynamic analysis techniques [6] can be helpful, I observed their success is critically dependent on the availability of defect-revealing multithreaded tests. Without a priori knowledge of the defects, manually constructing such tests is non-trivial.

As part of my Ph.D. thesis, I designed the **first set of algorithms** to automatically **generate targeted multithreaded tests** to detect concurrency bugs in libraries [4, 7, 5, 8]. The key insight underlying the design is that a subset of the properties observed when the defects manifest in a concurrent execution can also be observed in a sequential execution or via static analysis. I explored two design variants: (a) path-agnostic test synthesis and (b) path-aware test synthesis. The path-agnostic analysis analyzes the execution traces obtained from executing the input sequential tests and produces a concurrent test that creates the necessary execution state conducive for triggering deadlocks, data races, or atomicity violations. The path-aware test synthesis technique explores newer paths that are not covered by the input sequential tests. It is a directed, iterative and scalable algorithm where each step of the iterative process includes statically identifying sub-goals towards the goal of failing an input specification, generating a plan toward meeting these goals, and merging the paths traversed dynamically with the plan computed statically via constraint solving to generate a new test. The approach reports **complete reproduction scenarios**, guaranteed to be true, for the bugs it finds. The tests automatically synthesized by these techniques helped expose more than **300 concurrency bugs** in popular libraries (Oracle Java Development Kit, Google Guava Collections, HyperSQL DataBase, Apache OpenNLP, etc.), including many previously unknown bugs that were subsequently fixed.

## Future Directions

I will continue to execute my research vision and use my experience to address other open challenges in building high-quality software systems by exploring the following research directions:

**Code search and recommendation systems**: Effective code search and recommendation systems can be a source of positive disruption to the software development process. For example, building **search strategies** that can be deployed earlier in the software development cycle can **influence the choice of the programming language and the set of libraries**. Comprehensively addressing this problem requires building search techniques beyond functional equivalence, motivating the need for **parameterized code search** techniques with constraints on program properties that include performance, memory consumption, binary size, concurrency, IP compatibility, and application context. Also, expanding the expressiveness of code search queries with richer formats (e.g., natural language, DSL) can cater to a diverse set of users and reduce the entry barrier for new programmers. Eventually, building recommendation systems that analyze available code and an application to suggest relevant software artifacts can pave the way for **automated code evolution**.

**Synthesizing library hybrids**: Developers are interested in ensuring that their application conforms to specific system properties, such as thread safety, memory footprint, and energy efficiency. Ensuring these properties often requires combining functionalities and implementations from different libraries, as no one library can meet all requirements. I am interested in exploring the problem of automatically synthesizing library hybrids from a diverse set of libraries so that the synthesized library provides the necessary functionality while adhering to a user-defined set of system properties. I think this can be achieved by **building algorithms** that can **carve** out semantically meaningful **components from a library** relevant to an application and construct adapters to **integrate carved components**. This can be challenging as these components may be harvested from diverse sources with custom coding styles, underlying algorithms, and data structure choices. My experience with synthesizing executions, dynamic and static analyses, and building verified adapters can help design approaches to address this problem.

**Specification-driven benchmark generation**: The growth and adoption of newer programming languages with custom syntax, semantics, defects, and best practices have necessitated the continuous need for new program analyses. To meet this growing need, oftentimes, researchers propose analyses with varying assumptions and tradeoffs concerning precision and scalability. However, the availability of numerous analyses for a given language and the absence of a standard benchmark to measure their relative strengths makes it harder for a user to select the right program analysis technique. Comparing different analyzers requires a comprehensive and realistic code corpus that can serve as an unbiased reference to **analyze the analyzers**. My vision is to automatically generate such corpora by leveraging large open-source code repositories and input specifications to build techniques that combine the complementary strengths of program synthesis and learning. Also, I expect such a technique to serve as a teaching tool that generates realistic examples for programmers to learn the language idioms. My experience with a wide variety of property-driven generation of programs along with search strategies to identify similar code can be useful in addressing this problem.

**Applications to other domains and programming models**: Program analysis has been successfully applied in other fields, including databases, distributed systems, operating systems, security, and high-performance computing. I am interested in building novel techniques that can cater to the custom requirements of these domains. While at Microsoft Research, I jointly developed an approach for optimizing big data queries using program synthesis and static analysis [9]. I also explored the application of path profiling to improve branch prediction in C compilers during my internship at Google. As part of a collaboration between MIT and Aarno Labs, I developed formal approaches for provenance tracking in the context of detecting security vulnerabilities such as Adups FOTA malware in Android devices [1]. I will continue to explore this direction and am especially keen on exploring the application of my prior work on generating multithreaded tests in other domains such as operating systems, distributed systems, software-defined networks, and event-driven programs.

## Closing Remarks

Novel program analysis techniques that can search, reason, customize and adapt software will be instrumental in managing the rapid growth of code and emerging programming languages. I look forward to continuing to contribute to this space.

## References


[1] Michael Gordon, Jordan Eikenberry, Anthony Eden, Jeffrey Perkins, Malavika Samak, Henny Sipma, and Martin Rinard. *Clearscope: Full Stack Provenance Graph Generation for Transparent Computing on Mobile Devices*. MIT Technical Report 2020.

[2] Malavika Samak, Jose Pablo Cambronero, and Martin C. Rinard. *Searching for Replacement Classes*. (under submission). arXiv: `2110.05638`.

[3] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. *Synthesizing Replacement Classes*. ACM POPL 2020.

[4] Malavika Samak and Murali Krishna Ramanathan. *Multithreaded Test Synthesis for Deadlock Detection*. ACM OOPSLA 2014.

[5] Malavika Samak and Murali Krishna Ramanathan. *Synthesizing Tests for Detecting Atomicity Violations*. ACM ESEC/FSE 2015.

[6] Malavika Samak and Murali Krishna Ramanathan. *Trace Driven Dynamic Deadlock Detection and Reproduction*. ACM PPoPP 2014.

[7] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. *Synthesizing Racy Tests*. ACM PLDI 2015.

[8] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. *Directed Synthesis of Failing Concurrent Executions*. ACM OOPSLA 2016.

[9] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. *Optimizing Big-Data Queries Using Program Synthesis*. ACM SOSP 2017.