Static Analysis and Transformation for Quantum Programming Languages

Joshua A.C. Behler, Ali F. Al-Ramadan, Betis Baheri, Michael L. Collard,
Qiang Guan, Jonathan I. Maletic

Abstract

Quantum computing is advancing rapidly, with developers increasingly adopting quantum programming languages. However, despite progress in quantum research, there is a notable lack of development tools and technologies that assist developers in designing and analyzing quantum programs. This work aims to address this gap using a software engineering approach. Specifically, the srcML infrastructure is leveraged to automate the static analysis and refactoring of OpenQASM quantum programs by implementing a tool, QStatic. Then QStatic's potential is demonstrated by identifying and refactoring common code patterns. The approach enables the exploration and manipulation of quantum source code, effectively addressing numerous challenges in quantum programming, such as optimizing quantum gates to reduce runtime costs.

Introduction

Quantum computing is quickly evolving, capturing the attention of both researchers and industry practitioners. With companies investing in quantum computers, research, and talent, the quantum computing market is expected to grow from USD $1.3 billion in 2024 to an estimated USD $5.3 billion by 2029 with a compound annual growth rate of 32.7% [1]. Naturally, with the rise in quantum computing comes the rise of quantum programs. Quantum algorithms are now evolving beyond simple circuit representations to complex programs implemented in specialized quantum languages, leading to an explosion in the development of more sophisticated large quantum applications.

With this expansion of quantum programming, the demand for high-quality code and advanced development tools is becoming crucial. Recent studies highlight the importance of effective programming approaches in advancing quantum computing. For instance, Weidenfeller et al. [2] found that certain software strategies make quantum circuits more efficient, especially on systems with hardware limitations. Abbas et al. [3] also stress that algorithms are crucial for improving performance by optimizing the accuracy of quantum gates and reducing the time they take to run. Despite some progress, there remains a noticeable gap in the tools available to developers, ultimately hindering their ability to effectively design quantum programs [4, 5]. While various tools attempt to address this gap, they are still not sufficient. For example, QSmell [6] detects various quantum code smells using static and dynamic analysis. Similarly, LintQ [7] and QChecker [8] utilize static analysis to identify bugs in quantum programs. However, these techniques are restricted to specific frameworks, such as Qiskit, and lack generalizability to other quantum programming languages.

In this work, we present a more flexible and generalizable approach that utilizes the srcML infrastructure [9, 10] to automate the analysis and refactoring of OpenQASM quantum programs through the development and implementation of a static analysis tool, QStatic. This tool enables developers to effectively analyze quantum code and optimize performance through automated refactoring. We showcase QStatic's potential by identifying and refactoring various common quantum code patterns. Ultimately, our approach establishes a practical infrastructure that serves as a foundation for evolving quantum programming languages, empowering them to better meet the dynamic demands of quantum software development.

From Quantum Circuits to Code

As quantum computing grows in popularity, new tools are being developed to support the ever-growing infrastructure. A common way of expressing quantum algorithms is through quantum circuit diagrams, which serve as visual representations of qubits and the gates performed on them over time. These diagrams can effectively capture the physical structure and operational order of the hardware running quantum algorithms but can be difficult to understand, explain, and create.

Quantum programming languages provide an efficient means to represent and create quantum algorithms and programs [11]. Similarly to classical computing, which can be represented through circuit diagrams, low-level machine languages, and high-level programming languages, quantum computing has both low and high-level programming languages. Examples of low-level quantum programming languages include OpenQASM and Quil, and high-level languages include Q#, Scaffold, and Silq [12]. These languages and others combine quantum concepts such as qubit and quantum gates to merge these features with classical programming features such as looping, branching, and classical data. **Figure 1** showcases a quantum circuit and its equivalent representation in OpenQASM.

As powerful and interesting as quantum programming languages are, there are still many issues that quantum programs face. To start with, running quantum programs is very expensive. Researchers and companies that use or operate quantum machines must make sure that the code they are running is correct and will generate the desired results. Running an expensive task on a quantum machine, only to find that the program written has a bug can be devastating (and costly) [5]. Similarly, running an inefficient piece of software can unnecessarily waste time and money, when a more efficient equivalent version can be used. Classical computing faced similar issues in the past, and as tools improved, writing high quality code became both cheaper and faster.

Beyond execution, quantum programs also struggle with readability. Many quantum programs are translated directly from circuits and, as a result, are mainly a string of quantum gate calls. This can be difficult to read and understand. Using software engineering concepts such as encapsulation and procedural abstraction can drastically improve the readability of the code. To solve these issues, we take a software engineering approach to examining quantum source code and aim to create development tools which can support quantum programming languages.

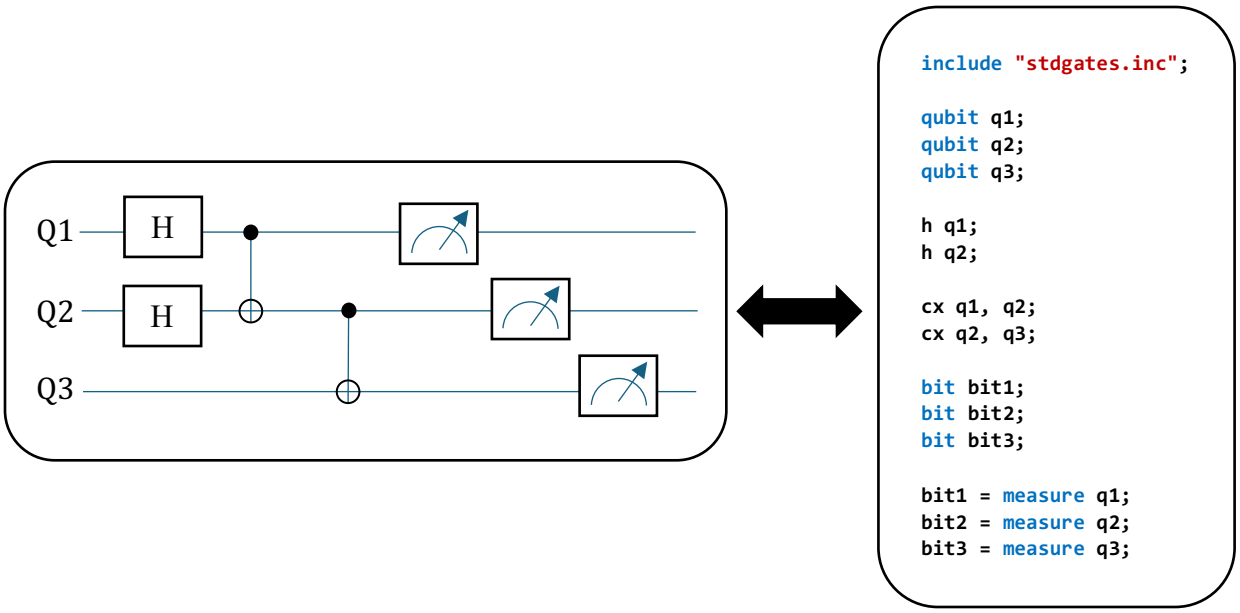


Figure 1: An example quantum circuit (left) and an equivalent representation in OpenQASM (right).

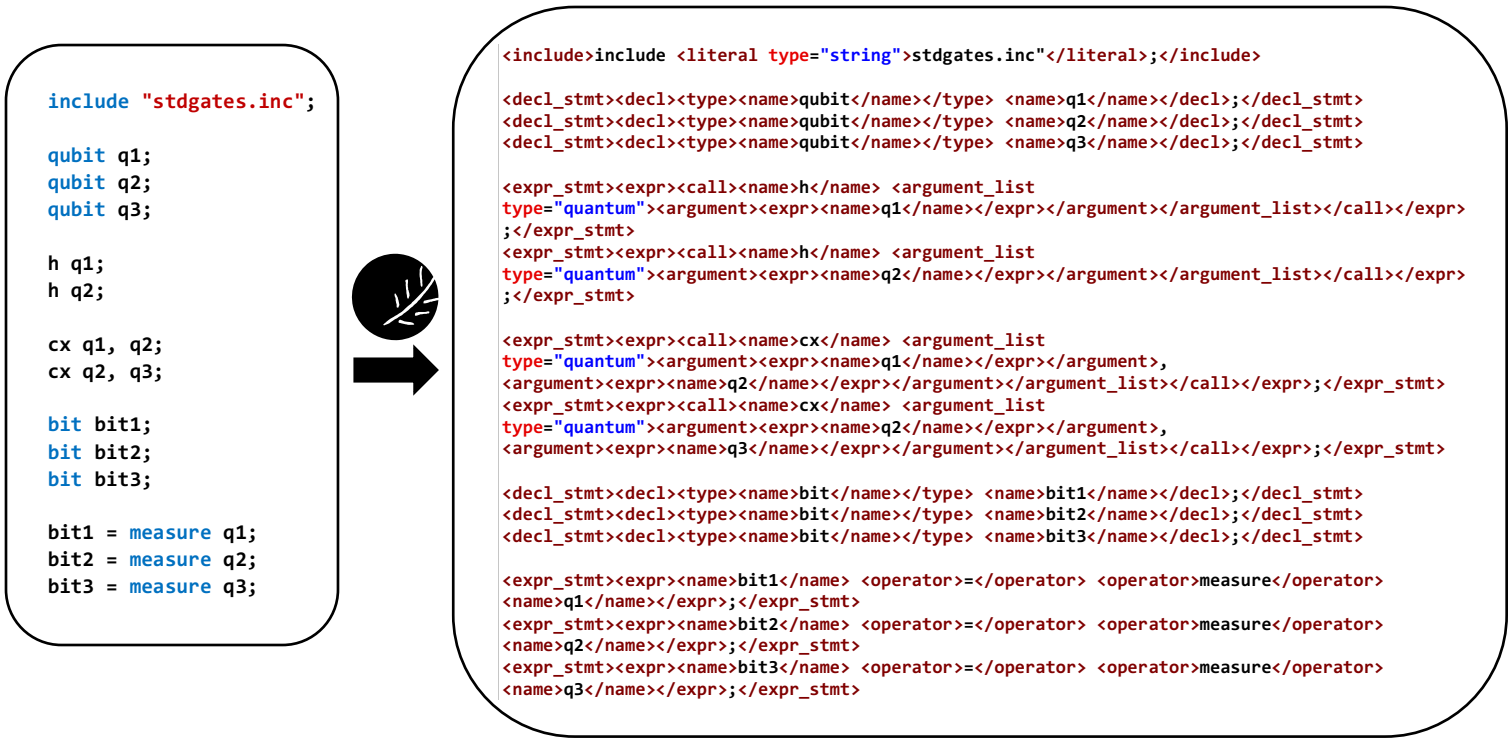
Static Analysis for Quantum Source Code

Software engineering broadly aims to support the development, debugging, testing, maintenance, and understanding of software systems. Specifically, we are interested in static program analysis, which is the analysis of source code (without execution).

Static program analysis is performed by examining the written source code and identifying patterns, idioms, and other aspects. This information allows researchers to construct tools to better support software development, including the detection of code smells, refactoring the source code, and developing tools to aid developers – such as linters.

Static program analysis is normally done on an abstract syntax tree (AST) representation of the source code as defined by the grammar for the programming language. However, little work has been done on producing usable ASTs for quantum programming languages. Highly specialized static analysis tools that work solely on a single language do exist. For example, Xia and Zhao [13] created a method for statically analyzing the entanglement information of Q# programs by generating control flow graphs. Such tools work well but are highly specialized and are usually not generalizable to other quantum languages or useful for developing other kinds of static analysis.

To enable static analysis of quantum programming languages, an AST representation is necessary. To achieve that, we utilize srcML (www.srcML.org), a highly scalable infrastructure designed to transform source code across multiple programming languages into a structured XML representation [9, 10]. The srcML format allows us to access the source code's syntax, which in turn supports static program analysis. **Figure 2** showcases an example of OpenQASM srcML using the example from **Figure 1**.



```
include "stdgates.inc";

qubit q1;
qubit q2;
qubit q3;

h q1;
h q2;

cx q1, q2;
cx q2, q3;

bit bit1;
bit bit2;
bit bit3;

bit1 = measure q1;
bit2 = measure q2;
bit3 = measure q3;
```

```
<include>include <literal type="string">stdgates.inc"</literal>;</include>

<decl_stmt><decl><type><name>qubit</name></type> <name>q1</name></decl>;</decl_stmt>
<decl_stmt><decl><type><name>qubit</name></type> <name>q2</name></decl>;</decl_stmt>
<decl_stmt><decl><type><name>qubit</name></type> <name>q3</name></decl>;</decl_stmt>

<expr_stmt><expr><call><name>h</name> <argument_list
type="quantum"><argument><expr><name>q1</name></expr></argument></argument_list></call></expr>
;</expr_stmt>
<expr_stmt><expr><call><name>h</name> <argument_list
type="quantum"><argument><expr><name>q2</name></expr></argument></argument_list></call></expr>
;</expr_stmt>

<expr_stmt><expr><call><name>cx</name> <argument_list
type="quantum"><argument><expr><name>q1</name></expr></argument>,
<argument><expr><name>q2</name></expr></argument></argument_list></call></expr>;</expr_stmt>
<expr_stmt><expr><call><name>cx</name> <argument_list
type="quantum"><argument><expr><name>q2</name></expr></argument>,
<argument><expr><name>q3</name></expr></argument></argument_list></call></expr>;</expr_stmt>

<decl_stmt><decl><type><name>bit</name></type> <name>bit1</name></decl>;</decl_stmt>
<decl_stmt><decl><type><name>bit</name></type> <name>bit2</name></decl>;</decl_stmt>
<decl_stmt><decl><type><name>bit</name></type> <name>bit3</name></decl>;</decl_stmt>

<expr_stmt><expr><name>bit1</name> <operator>=</operator> <operator>measure</operator>
<name>q1</name></expr>;</expr_stmt>
<expr_stmt><expr><name>bit2</name> <operator>=</operator> <operator>measure</operator>
<name>q2</name></expr>;</expr_stmt>
<expr_stmt><expr><name>bit3</name> <operator>=</operator> <operator>measure</operator>
<name>q3</name></expr>;</expr_stmt>
```

Figure 2: OpenQASM source code from **Figure 1** (left) and its srcML representation (right)

In previous work [14], we established a foundational methodology for integrating quantum programming languages into the srcML infrastructure, focusing on OpenQASM 3.0. This involved designing srcML-based ASTs that leverage classical language concepts (e.g., variables, control flow) while introducing generalizable syntactic categories for quantum-specific constructs (e.g., qubit registers, quantum gates). To ensure our srcML extensions could support future quantum language adoption, we aligned OpenQASM's quantum features with patterns from other quantum languages. Additionally, we developed a prototype tool to demonstrate execution-order analysis.

Building on this foundation, the current work introduces QStatic[1], a tool designed to automate both static analysis and refactoring for OpenQASM. QStatic directly implements automated refactoring rules for common quantum patterns including:

- Iteration patterns: Restructuring repetitive code into loops for improved readability and maintainability
- Hadamard gate reduction: Replacing redundant operations with optimized equivalents.
- Code encapsulation: Modularizing frequently used operations into custom gate definitions to shorten code and improve readability.

QStatic aims to improve quantum programming by assisting to reduce manual effort, minimize errors, and enhance the efficiency of quantum code. This advancement bridges the gap between

[1] The tool is available at github.com/srcML/QStatic

theoretical analysis and practical tooling, accelerating the development and adoption of good-quality scalable quantum software.

Quantum Code Exploration

srcML enables us to inspect quantum source code and identify patterns that might signal inefficiencies or other issues. To demonstrate the usefulness of this approach, we introduce an infrastructure, QStatic, that leverages srcML to perform execution order analysis and refactoring on the OpenQASM source code. Execution order analysis involves gathering and analyzing the order of operations performed within the source code, specifically focusing on the order of quantum gate calls and the qubits affected.

We perform execution order analysis through a multi-step process designed to clarify the sequence of quantum operations. First, when quantum calls are encapsulated within a for loop, we use loop unrolling, a process of expanding a for loop into its equivalent individual calls, to identify the qubits that are affected and their order. Next, we expand calls to user-defined gates and functions to capture the fundamental quantum operations performed. Quantum operations that occur inside of if-statements are labeled with the if-statement's condition, allowing future support of control flow analysis. Gathering this data allows us to tackle some of the many problems facing quantum programming. Static analysis is vital for improving code readability and refactoring to increase performance.

Identifying Iteration Patterns for Refactoring

Iteration patterns are small patterns of code where similar operations are performed in a certain order on qubits. Wen et al. defined three types of iteration patterns for use in their Quantivine visualization tool [15]: vertical (performing the same operation on different qubits), horizontal (performing the same operation on the same qubit), and diagonal (performing the same binary operation on a "step-like" pattern of qubits). Their tool uses the patterns to create more concise visualizations of quantum circuits to support scalability.

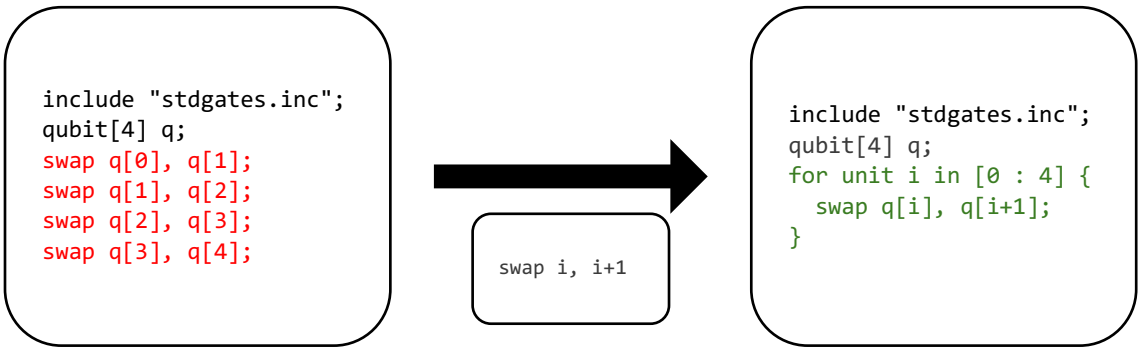


Figure 3: OpenQASM quantum code which contains a series of swaps (left) and a refactored form which encapsulates the swaps in a for loop (right). The code being refactored showcases an example of diagonal iteration.

For quantum source code, these patterns are useful for identifying sections of source code where multiple statements can be condensed into a loop. Utilizing QStatic's ability to support transformations, we can refactor source code that contains any of the three iteration patterns to condense the various statements and operations into a for loop and improve the readability of the source code. An example, demonstrating a series of swaps on an array of qubits is shown in **Figure 3**.

Encapsulating Common Code Snippets

In a similar vein to iteration patterns, certain operations may be performed numerous times consecutively, but without an easy way to wrap them into a loop. In these cases, the operations can be contained into a custom gate definition. This new gate takes the place of all the various calls, shortening the code and improving its readability.

Figure 4 demonstrates an example of this process. We identify two sections of code which contain the same quantum gate calls in a row. We record these calls and check that their argument patterns are identical. Using this pattern, we create a new gate and replace all instances of the concurrent gate calls with a single call to the defined gate. This makes the code easier to understand and develop as the program evolves.

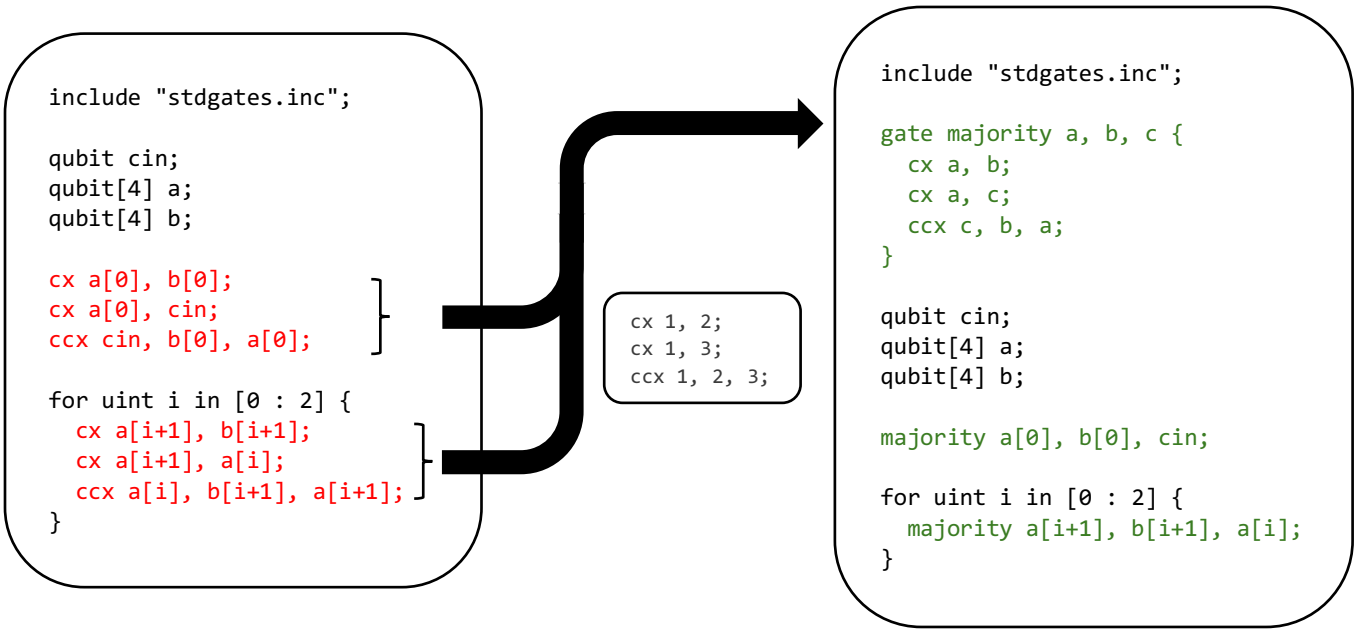


Figure 4: OpenQASM code which repeats a series of cx, cx, and ccx calls in two places (left) and a refactored form where the calls are encapsulated into a custom gate call (right)

Improving Efficiency

Companies that offer access to their quantum computers typically charge end users to run code, sometimes with a long wait list. These wait times can be exacerbated by small errors in the code – if something is slightly wrong it becomes costly and time consuming to fix and rerun. Improving the efficiency of quantum programs can save researchers, end users, and quantum companies both time and money.

srcML enables the identification of various inefficiencies and provides the means to replace them with more efficient code. For instance, consider the presence of a CNOT gate in a Hadamard basis. When a CNOT is performed on two qubits between two Hadamard gate calls on both qubits, the result is equivalent to solely performing a CNOT with the target and control qubits flipped. This is illustrated in **Figure 5**.

Using srcML, we can refactor this code by first identifying areas where the five operations are performed on two qubits in the correct order, deleting all Hadamard gate calls, and replacing the CNOT call with the reversed call. This not only reduces the number of statements within the source code but also provides a marginal increase in execution. Running the two different programs in Figure 5 through the qubit simulator in Amazon's *braket* Python module, we find that the single CNOT call executes slightly faster regardless of how many qubits we ask the simulator to test. While improvements are minor with smaller amounts of qubits, every single run is faster on the reversed CNOT than with the Hadamard basis. To test the speed of the two programs, we ran *braket's* simulator 100 times with an increasing number of shots. That is, 100 times with 100 shots, 100 times with 1000 shots, all the way to 10^7 shots. Each of the 100 runs are then averaged out together. At 10^7 shots, we see a speed increase of three seconds from the reversed CNOT compared to the Hadamard basis. While this is a relatively simple example for a developer to identify and fix themselves, and the speed improvements are minor, it demonstrates how quantum programs can contain inefficiencies that can be automatically identified and improved by using srcML.

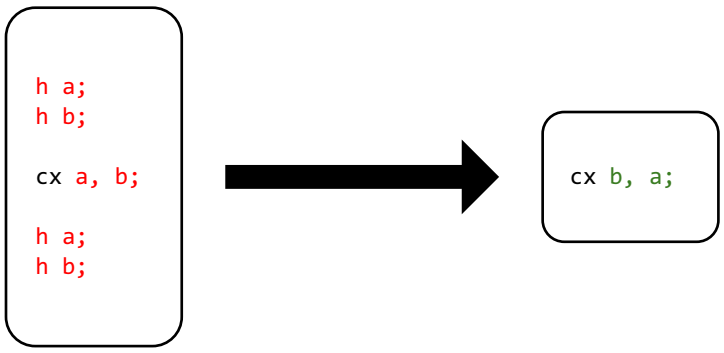


Figure 5: Two OpenQASM code snippets, one showing a CNOT in a Hadamard basis (left) and the other showing the reversed CNOT (right). The operations on the qubits produce the same result, with the single CNOT being more concise.

Conclusion

General tools for static analysis have the potential to significantly enhance the current quantum software ecosystem, bringing development tools up to the standards of those available for traditional programming languages such as C++. As quantum software becomes more mainstream, developers will demand better tools to aid in their work. Developing and maintaining these tools requires a robust infrastructure, and srcML is well-equipped to address many of the challenges involved. This is due to its ability to handle systems written in multiple programming languages (i.e., polyglot systems).

Our research represents the initial phase of what can be achieved with static analysis for quantum programs by implementing and showcasing one such tool, QStatic. We plan to extend this work to create a suite of tools that assist quantum developers with everyday tasks and integrate it into the srcML infrastructure. This integration will allow us to support quantum languages across other tools in our infrastructure. For example, srcML has a slicing tool that can be adapted to slice certain qubits and identify all lines in a program that affects any given qubit. Additionally, the infrastructure supports other advanced tasks, including source code querying and manipulation, enabling quantum programmers to search for specific patterns within source code and making the identification of certain patterns or features very simple.

Acknowledgements

This work was supported in part by grants from the US National Science Foundation: CNS 20-16465, OAC 22-38734, OAC 22-30111, OAC 22-17021, OAC 23-11950.

References

[1] “Quantum Computing Market Global Forecast to 2029,” *MarketsandMarkets*, 2024. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/quantum-computing-market-144888301.html.

[2] J. Weidenfeller *et al.*, “Scaling of the quantum approximate optimization algorithm on superconducting qubit based hardware,” *Quantum*, vol. 6, p. 870, Dec. 2022.

[3] A. Abbas *et al.*, “Challenges and opportunities in quantum optimization,” *Nature Reviews Physics*, vol. 6, no. 12, pp. 718–735, Dec. 2024.

[4] P. Liimatta, P. Taipale, K. Halunen, T. Heinosaari, T. Mikkonen, and V. Stirbu, “Research Versus Practice in Quantum Software Engineering: Experiences From Credit Scoring Use Case,” *IEEE Software*, vol. 41, no. 6, pp. 9–16, Nov. 2024.

[5] M. Salam and M. Ilyas, “Quantum computing challenges and solutions in software industry—A multivocal literature review,” *IET Quantum Communication*, vol. 5, no. 4, pp. 462–485, 2024.

[6] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed, “The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing,” in *Proceedings of the 45th International Conference on Software Engineering*, Melbourne, Victoria, Australia, 2023, pp. 358–370.

[7] M. Paltenghi and M. Pradel, “Analyzing Quantum Programs with LintQ: A Static Analysis Framework for Qiskit,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, p. 95:2144-95:2166, Jul. 2024.

[8] P. Zhao, X. Wu, Z. Li, and J. Zhao, “Qchecker: Detecting bugs in quantum programs via static analysis,” in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, 2023, pp. 50–57.

[9] M. L. Collard, M. J. Decker, and J. I. Maletic, “srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration,” in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, 2013, pp. 516–519.

[10] M. L. Collard, M. J. Decker, and J. I. Maletic, “Lightweight Transformation and Fact Extraction with the srcML Toolkit,” in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 173–184.

[11] B. Heim *et al.*, “Quantum programming languages,” *Nat Rev Phys*, vol. 2, no. 12, pp. 709–722, Dec. 2020.

[12] F. Ferreira and J. Campos, “An exploratory study on the usage of quantum programming languages,” *Science of Computer Programming*, vol. 240, p. 103217, Feb. 2025.

[13] S. Xia and J. Zhao, “Static entanglement analysis of quantum programs,” in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, 2023, pp. 42–49.

[14] J. A. C. Behler, A. F. Al-Ramadan, B. Baheri, Q. Guan, and J. I. Maletic, “Supporting Static Program Analysis and Transformation of Quantum-Based Languages,” in *Proceedings of the IEEE International Conference on Quantum Computing and Engineering (QCE)*, Montréal, Québec, Canada, 2024.

[15] Z. Wen *et al.*, “Quantivine: A Visualization Approach for Large-scale Quantum Circuit Representation and Analysis,” *IEEE Trans. Visual. Comput. Graphics*, pp. 1–11, 2023.

Joshua A. C. Behler is a PhD student in the Department of Computer Science at Kent State University, where he contributes to the srcML and iTrace projects. His research interests are centered around program comprehension, software visualization, and software maintenance. Contact him at jbehler1@kent.edu.

Ali F. Al-Ramadan is a PhD student in the Department of Computer Science at Kent State University and a contributor to the srcML project. His research focuses on program analysis, code manipulation, and transformation. He is a member of IEEE-CS. Contact him at aalramad@kent.edu.

Betis Baheri is an Assistant Professor in the Department of Cybersecurity at SUNY Canton and the founder of the Quantum Shield Laboratory (QSL), specializing in quantum computing, high-performance computing, machine learning, and cybersecurity. He honed his experience in large-scale computational systems at the Texas Advanced Computing Center and has collaborated with Los Alamos and Pacific Northwest National Laboratories. Baheri received his Ph.D. in Computer Science from Kent State University. Contact him at bbaheri@kent.edu.

Michael L. Collard is an Associate Professor in the Department of Computer Science at The University of Akron, where he co-directs the srcML project. His research interests include software evolution, program comprehension, reverse engineering, source-code transformation, and source-

code differencing. Collard has a Ph.D. in Computer Science from Kent State University. He is a member of IEEE-CS. Contact him at collard@uakron.edu.

Qiang Guan is an Associate Professor in the Department of Computer Science at Kent State University, where he directs the Green Ubiquitous Autonomous Networking System lab (GUANS) and serves as a member of the Brain Health Research Institute (BHRI). Before joining Kent State, he worked as a computer scientist on the Data Science at Scale team at Los Alamos National Laboratory. His current research interests include high-performance computing applications, quantum computing systems, HPC-Cloud hybrid systems, and virtual reality. For more details, please refer to http://www.guans.cs.kent.edu. Contact him at qguan@kent.edu.

Jonathan I. Maletic is a Professor in the Department of Computer Science at Kent State University where he co-directs the srcML project. His research interests are centered on software evolution, with a focus on comprehension, static analysis, exploration, and manipulation of large-scale software systems. Maletic has a Ph.D. in computer science from Wayne State University. He is a member of IEEE-CS and ACM. Contact him at jmaletic@kent.edu.