Supporting Static Program Analysis and Transformation of Quantum-Based Languages

Joshua A. C. Behler
Computer Science Department
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

Ali F. Al-Ramadan
Computer Science Department
Kent State University
Kent, Ohio, USA
aalramad@kent.edu

Betis Baheri
Computer Science Department
Kent State University
Kent, Ohio, USA
bbaheri@kent.edu

Qiang Guan
Computer Science Department
Kent State University
Kent, Ohio, USA
qguan@kent.edu

Jonathan I. Maletic
Computer Science Department
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—The work aims to enable the use of common software engineering techniques and tools for quantum programming languages (e.g., OpenQASM). With the increased interest in quantum computing, researchers are adopting the use of higher-level quantum programming languages versus low-level circuit diagrams. While general purpose programming languages (e.g., C++, Python) are highly supported by a variety of software engineering tools, these novel programming languages for quantum computing have almost no support. Useable tools for debugging, static analysis, error detection, and transformation are currently non-existent. This work extends an existing software infrastructure (i.e., srcML) for the analysis, exploration, and manipulation of source code to OpenQASM. The srcML infrastructure, via parsing, generates abstract syntax information of programs to support high-level querying and analysis of the source code. With this, quantum developers can extract information and identify possible errors or inefficiencies in their programs. The paper presents the basic syntactic markup for OpenQASM. Also, a number of relevant quantum-based problems (e.g., iteration patterns, control recusion) are described and examples of how they are addressed using srcML is given.**

Keywords—quantum programming, static program analysis, openqasm

I. Introduction

Quantum computing has exploded in popularity in recent years. Quantum computing utilizes the principles of quantum mechanics to process information in ways that are fundamentally different from classical computing. The basic principle is the use of quantum bits, or qubits, which can exist in multiple states simultaneously. This allows us to solve certain complex problems more efficiently than classical computers. As more and more facets of industry and research adopt quantum computing to solve difficult problems, more technologies are being created to support the exploration of this field. One such technology is quantum programming languages. Languages such as OpenQASM [1], [2], Scaffold [3], and Q# [4] provide a means for developers to write quantum programs that can be executed on quantum hardware (or simulators). These languages, similar to traditional programming languages, allow developers to write algorithms and express quantum circuits. OpenQASM (for Quantum assembly language) is a low-level programming language that provides direct access to quantum hardware. It supports both classical and quantum instructions. However, unlike classical programming languages, quantum programming languages lack a wide range of software engineering tools to assist developers in the construction of programs. There are no tools for conducting basic static analysis of the code to assist in finding errors, tracking down inefficiencies, or identifying common patterns. The lack of such tools can slow the development and adoption of quantum programming languages. As quantum computing gains more attention, there will be a growing need for these tools that assist developers in understanding and working with quantum programs. By using such tools developers will be able to identify potential errors, optimize performance, and extract meaningful information. Moreover, the availability of such tools can encourage developers to adopt quantum programming languages rather than relying solely on quantum circuit analysis.

In this paper, we begin work on extending the srcML infrastructure [5], [6] to support OpenQASM. srcML is a robust set of tools that generates an XML representation of a program's source code and abstract syntax based on a predefined markup specification of the language's grammar. This XML format preserves all the syntactic information of the original code, including whitespace, and currently supports various traditional programming languages (e.g., C, C++, C#, Java). The srcML representation enables advanced code analysis and manipulation using standard XML technologies (e.g., XPath, XSLT). By providing a standardized, simple format for processing and analyzing source code, srcML enables a language to undergo static analysis. Adding quantum programming languages to srcML opens many possibilities for program analysis which were previously difficult or limited in scope.

We describe the process of integrating OpenQASM into srcML by creating a markup specification that defines how the language structures are labeled in the resulting XML

© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. This is the accepted version of the paper published in the Proceedings of the 2024 IEEE International Conference on Quantum Computing and Engineering (QCE), DOI 10.1109/QCE60285.2024.00103.

representation. We present the markup specification for OpenQASM, covering both its classical and quantum features. Furthermore, we introduce a static time analyzer that examines quantum operations performed on qubits within an OpenQASM program. Using the output of this analyzer, we identify four common quantum language patterns: *Horizontal Iteratio*n, *Vertical Iteration*, *Diagonal Iteration*, and *Control Recursion*. In addition to pattern identification, we discuss how the srcML representation of OpenQASM programs can facilitate quantum code refactoring and code smell detection.

The contribution of this paper can be summarized as:

- The creation of a markup specification which begins the process of adding OpenQASM to the srcML Toolkit.
- The manual markup of OpenQASM code examples into their srcML format and the creation of a tool which performs analysis of the execution order of the source code.
- A cursory exploration of coding patterns, refactorings, and code smells in quantum code that srcML enables the analysis of

The rest of the paper is organized as follows. Section II discusses related works in quantum software engineering and program analysis. Section III describes the process of extending srcML to support OpenQASM, including the markup specification and the creation of srcML files. Section IV introduces the static time analyzer and its application in identifying quantum patterns. Section V explores quantum code refactoring identification. Finally, Section VI concludes the paper and outlines future research directions.

II. Related Work

There is little literature on static program analysis tools for quantum programs. LintQ [7] and QChecker [8] are two recent works that focus on bug detection in Qiskit-based quantum programs. LintQ identifies potential bugs by mapping a set of quantum-specific abstractions to program information. This information is extracted using CodeQL, a static analysis engine. QChecker detects bugs by directly analyzing the abstract syntax tree (AST) representation of Qiskit programs. Kaul et al. [9] implement a tool, QCPG, that extends Code Property Graphs to analyze quantum programs written Qiskit or OpenQASM. The QCPG includes information from both the classical and quantum parts to enable static analysis of both domains. Other problem-specific techniques to analyze quantum code include entanglement analysis [10] to detect possible entanglement errors within and between modules in Q#. Zhao et al. [11] identify and categorize common bug patterns in quantum programs written in the Qiskit programming language. Nayak et al. [12] propose a new framework, Q-PAC, that is used to detect bug-fix patterns automatically in Qiskit quantum code using ASTs, regular expressions, and semantic checks However, these approaches have a few limitations. First, they are tightly coupled to specific quantum programming languages or frameworks. LintQ relies on CodeQL and Qiskit-specific abstractions, QChecker is tightly coupled with Qiskit's AST structure, and the QCPG prototype is currently specifically implemented for Qiskit and OpenQASM. Extending these tools to analyze programs written in other quantum languages can be problematic without major modifications to the design. Work dealing with pattern detection is tied to the specific syntax and semantics of Qiskit. Other work dealing with problem-specific techniques, such as entanglement analysis for Q#, are not easily generalizable to other problem domains.

Our work here recognizes the need for a more general approach to quantum program analysis. We leverage the srcML infrastructure to represent quantum programs in a standardized XML format to allow for a flexible and language-agnostic program information extraction process that can be applied to various quantum programming languages. This approach can easily capture the structure and semantics of quantum programs to facilitate a wide range of static analysis tasks such as bug detection, optimization, and pattern extraction. A key advantage of our approach is that supporting a new quantum programming language only requires the addition of a markup specification, which defines the language's elements, sub-elements, and their relationships. Our work complements existing tools and techniques by providing a versatile foundation for general-purpose static analysis tools.

III. OpenQASM in srcML

srcML is a powerful, robust, and highly scalable infrastructure used for transforming source code into a structured XML representation for multiple programming languages. It includes a powerful parsing technology that allows for efficient transformation of source code to XML and vice versa without any loss of information (of the original source code). The XML representation adds abstract syntax information directly into the source code. Because of this, a user has access to the source code and abstract syntax tree simultaneously. An abstract syntax tree (AST) is a hierarchical representation of the syntactic structure of a program. Each element in the srcML AST corresponds to a specific syntactic construct in the source code, such as a function, a loop, or an expression. The elements are nested according to the structure of the code, with parent elements containing child elements that represent their constituent parts. For example, a function declaration element may contain child elements for the function name, parameters, and function body. Figure 1 details a small example of OpenQASM source code and its corresponding srcML markup. srcML differs from other ASTs in that alongside the (exact) original source code, the whitespace and comments are preserved in the XML. Compiler generated ASTs drop quite a lot of original syntactic information. Making it impossible to do such things as refactoring. srcML allows for the in-depth exploration, analysis, and manipulation of source code by leveraging various common XML technologies (e.g., XPath).

Compared to the other languages that srcML supports, OpenQASM is very simple in terms of syntax and number of features. However, because of OpenQASM's status as a quantum programming language, many new tags need to be defined in srcML to support these features. OpenQASM also supports classical programming constructs, which srcML already supports in other languages and thus also supports them in OpenQASM.

```
for uint i in [0: 3] {
    x q[i];
}
```

```
<for>for <control><init><decl><type><name>uint</name></type><name>i</name></decl></init> <range>in
<expr><index>[<expr><literal>0</literal><operator>:</operator>
<literal>3</literal></expr>]</index></range></control> <block>{<block_content>
    <expr_stmt><expr><call><name>x</name> <argument_list type="quantum">
    <argument><expr><name><name>q</name><index>[<expr><name>i</name></expr>]</index></name></expr></argument>
    </argument_list></call></expr>;</expr_stmt>
</block_content>}</block></for>
```

Figure 1: An example fragment of OpenQASM code (top) and its corresponding srcML (bottom). Note that all original source code text is preserved (indicated by the bold text), including whitespace and comments. Each XML tag represents the syntactic context of the source code element.

A. OpenQASM Markup Specification

Internally, srcML maintains parsers that handle the conversion from source code to srcML. These parsers are manually crafted using custom-made language grammars. When a new language is introduced to srcML, the parsers must be updated to support the syntax of the new language and handle edge cases that invariantly appear.

This process is very manual and can cause great difficulty when trying to add a language that is syntactically very different from previous languages. To avoid this, we are currently developing a set of parsing generation tools that can take a *markup specification* file and produce a parser for the language it represents. The markup specification is an XML document that defines what elements the language uses, what sub-elements and keywords each element has, and what sub-elements are optional or required. Currently, the parser generator is still under development and not fully functioning – however, the format for the markup specifications is well-defined, and allows for the markup of source code files.

To add OpenQASM to srcML, we create the markup specification for the language. Table 1 details the contents of the markup specification, including the new tags that are introduced to account for quantum programming's unique features.

Category	srcML Elements
Statement	**barrier**, **box**, break, **calibration**, decl_stmt, **defcalgrammar**, expr_stmt, extern, for, if_stmt, **include**, **measure**, **pragma**, **reset**, **version**, while
Function/Gate	argument, argument_list, call, function, **gate**, **gate_decl**, parameter, parameter_list, return
Extra Markup	literal, operator, name, type

Table 1: A table listing the statement tags srcML used to mark up OpenQASM. Bold elements are new elements introduced for OpenQASM.

The markup specification for OpenQASM supports the current version of OpenQASM 3.0. Because 3.0 is backwards compatible with OpenQASM 2.0, the markup specification will markup either version as encountered. An example being the change from `measure` being an operation in 2.0 to being a statement in 3.0. If `measure` is encountered within an expression, the markup specification treats it as an `<operator>` element. If it is used with its statement syntax, it is given the new `<measure>` element.

Beyond the addition of new elements in the markup, other additions are introduced in the OpenQASM markup specification – the most substantial of which involve qubits. OpenQASM supports passing both classical values and qubits to functions and gates as parameters. Passing classical values to these syntactically resembles how classical languages pass them – with a comma-separated list of values within parentheses. For qubits, OpenQASM uses a slightly different syntax, where parentheses are not used, and the qubits are just listed between commas. Calls, gates, and functions support having both kinds of parameters, meaning that some kind of disambiguation is needed within srcML. We introduce the special attribute `type="quantum"` onto the qubit parameter and argument lists, which allows a researcher to collect classical, qubit, or both lists as needed. Alongside this, a new type of literal in srcML has been added - `<literal type="qubit">`. This literal is for the physical qubit syntax in OpenQASM, where values like `$0` can be used to represent hardware qubits when ran on physical machines.

The markup specification uses the `<call>` element for both calls to functions and gates, as syntactically they are identical due to both allowing for classical and qubit parameters.

B. Creating srcML Files

With the markup specification, OpenQASM source code files can now easily be parsed and marked up into srcML, allowing for static analysis to be performed. The parser for this is not yet released but will be in the near future. For this work, the suite of OpenQASM 3.0 example files is marked up. The 16 files within the examples provide a fairly comprehensive use of the OpenQASM markup specification, containing at least one use of every element defined.

Because of the nature of the srcML Toolkit we can convert code marked up OpenQASM srcML back into the original source code. This allows for any srcML file created through other means (such as manual markup) to be processed and analyzed as normal. Thus, one can use XPath or our srcQL query language on the marked up code. Also, XSLT or XPath can be used to do transformations or refactorings on the code.

IV. Static Analysis

Currently, most developers that work with quantum algorithms typically think of quantum programming on the

circuit and gate level, with many works such as the Quantivine tool by Wen et al. focusing on how to better visualize very large quantum circuits [13]. Because of this focus from researchers, there is a hesitation to adopt quantum programming languages into workflows, and as a result most implementations of OpenQASM and other quantum programming languages do not support all of the languages' features.

srcML makes performing static program analysis simple due to the plethora of XML processing tools. By leveraging these tool, we create a prototype that enables the static analysis of the execution order of OpenQASM code to identify patterns and shortcuts. Providing researchers and developers with this information allows for quicker and easier understanding of quantum code and enables the adoption of the language.

A. Execution Order Analysis

An advantage of working with quantum circuits instead of programming languages is the ability to directly visualize the occurrence of operations with respect to time. While OpenQASM does allow a user to specify timing and control the minutia of execution, the code is organized like any other programming language, with execution jumping around to different blocks during execution. OpenQASM also supports classical control flow structures like if/else statements and loops, which can further obscure execution order at first glance. Unless a developer is intimately familiar with the custom gates and functions within the code, it can be difficult to understand what is happening to each qubit during execution.

To alleviate this problem, our tool traverses the XML structure of an OpenQASM srcML file and catalog each operation performed on a qubit with respect to time in execution. To accomplish this, we analyze every noteworthy statement within the code for changes made to a qubit. If a change is found, the change and the qubit is saved to a log and the search moves to the next statement.

B. Implementation

Each noteworthy element within OpenQASM requires a specific approach to successfully analyze and gather any notable changes. We list the process for analyzing each element:

<decl_stmt>: For every decl_stmt found in the srcML, we check to see if the variable being declared is a qubit. If it is, we save the name of the qubit into a list and give it an empty list of operations for use in storing any changes later on.

<reset>: If the tag is a reset statement, the qubit(s) that the reset operates on are given a reset operation in their list.

<gate>s and <function>s: If the tag is either a gate or a function definition, the name of the gate/function is grabbed and saved to a dictionary alongside the block of code it contains. Saving the block of code is important, as when a call is made to the function/gate, the statements within the block need to be analyzed further. Because each function could call other functions, this process can become quite nested, and we ensure that all statements are evaluated in order.

<measure>: If a tag is a measure statement, the measure operation is given to the qubit(s) that are being measured.

<for>: If the tag is a for loop, multiple steps must be taken to ensure proper evaluation. First, the range of the for loop is evaluated to determine which values the looping variable will hold, and thus determine how many times it will execute. OpenQASM only supports range-based for loops, so we reasonably assume that the for loop will have a limited execution. For each value that the looping variable will hold, a copy of the for loop's block is made, and any reference to the looping variable within each copy is replaced with the corresponding value. All the copies are then added to the list of statements, and our execution continues. This loop unrolling allows us to track precise changes to qubits which are stored in arrays.

<expr_stmt>: Expression statements require the most processing of all the other tags, due to calls being present inside of them. If a `<call>` sub-element is found within the expression statement, the quantum arguments that the call passes are saved. We check to see if the name of the call matches with any of the standard library gate calls or any function/gate defined within the code. If a match is found a copy of the corresponding function/gate's block is made. Each parameter is replaced with the corresponding argument from the call, and the block's statements are added to the execution stack.

When the tool finishes analyzing the file, it has a list of every qubit and every operation performed on a qubit with respect to time. This data is representable in two main formats sorted by time– as a list of operations per each qubit, or a list of operations performed on all qubits. Figure 2 showcases some example output from a simple piece of OpenQASM code.

```
include "stdgates.inc";

qubit[3] q;

reset q;

for uint i in [0: 2] {
  cx q[i], q[(i+1)%3];
}

bit[3] result;

measure q -> result;
```

Time	q[0]	q[1]	q[2]
1	reset	reset	reset
2	ctrl	cx	
3		ctrl	cx
4	cx		ctrl
5	measure	measure	measure

Figure 2: An example of the execution order analysis performed on a simple OpenQASM file. Left is the input code, and right is the order analysis output

C. Limitations

Currently, there are some limitations to what the analyzer can evaluate. While loops are currently not supported, as evaluating the looping condition is non-trivial and out-of-scope currently. Additionally, any qubits that are passed classically to a function are not processed. This does mean that certain OpenQASM files are not completely evaluated. Alongside this, if an expression statement makes multiple calls, or a call is found inside a different statement (such as the condition of an if), the call is not processed. This is typically rare, but occasionally happens when using the `durationof` built-in function.

V. Quantum Pattern Refactoring

Using the output from the execution order analysis, we can identify many different patterns, optimizations, and refactorings which are of interest to quantum programmers.

Refactoring code is the process of changing or reorganizing code while maintaining the same external functionality [14]. Typically, refactoring is done to eliminate code smells – a segment of code that may be problematic. Work by Chen et al. [15] has explored eight different quantum computing specific code smells and how to prevalent them. Other work by Zhao [16] explores types of refactoring on Q# programs, and discusses both generic programming refactoring like the changing of a variable name to more quantum focused issues, such as consolidating measurement operations. Using srcML and the available transformation languages, quantum code can be analyzed for code smells, and then refactored automatically.

Identifying code smells and refactoring them can be extremely helpful for quantum programmers. Typically, quantum programs are run on publicly available quantum machines provided by third-party companies – IBM Quantum is one such example. Running these programs can take a large amount of time, particularly as large queues of developer submitted jobs will form. Ensuring that your code is as efficient as possible and has no issues is vital to efficiently work on these platforms, not to mention saving costs.

A. *Iteration Patterns*

In quantum computing, iteration patterns are small patterns of code where similar operations are performed in order on qubits. The patterns are defined to help with the visualization of large quantum circuits, such as for the Quantivine tool [13]. For quantum code, these patterns can be used to identify locations where the code can be either condensed into various loops to promote readability, or loops can be unrolled to improve performance on hardware. There are three iteration patterns: horizontal iteration, vertical iteration, and diagonal iteration.

Horizontal iteration is defined as the same operation occurring on the same qubit multiple times in a row. To detect this, we look at the qubit-oriented data and identify places where the same operation appears multiple times in a row on a qubit. By searching this way, we also avoid missing instances where the operations are distant from each other, and other operations are performed on unrelated qubits in-between. Figure 3 showcases some examples of horizontal iteration within OpenQASM.

```
x q;
x q;
x q;
```

```
for uint i in [0:2] {
  x q;
}
```

```
x a;
x b;
y b;
x a;
z b;
x a;
```

Figure 3: Three examples of horizontal iteration occurring. Highlighted calls are part of the iteration.

Vertical iteration is the same operation being performed on multiple different qubits in a row. On a quantum circuit diagram, these are easy to identify as the operations will take up the same vertical time slice on the circuit. Adapting this for OpenQASM, we define vertical iteration as being the same operation performed on unique qubits one after the other, with no other operations occurring in between. This differs from horizontal iteration, in that other operations cannot occur between parts of the vertical iteration. Figure 4 details some examples of vertical iteration in OpenQASM code.

```
x a;
x b;
x c;
x d;
```

```
for uint i in [0:10] {
  x q[i];
}
```

```
x q[1];
x q[0];
x q[2];
```

Figure 4: Three examples of vertical iteration occurring. Highlighted calls are part of the iteration. Note that if no other operations occur, the order in which the qubits are used does not matter.

Diagonal iteration is similar to vertical iteration in that the same operation must be performed on different qubits each time. However, diagonal iteration requires the operation to need two qubits as arguments, and the qubits must form a "chain", where each operation links to the previous one through a single qubit. On a circuit diagram, this forms a step-like pattern which travels diagonally across the board. The for loop in Figure 2 is an example of diagonal iteration, and Figure 5 details further examples of diagonal iteration.

```
cx $0, $1;
cx $1, $2;
cx $2, $3;
```

```
for uint i in [0:5] {
  swap q[i], q[i+1];
}
```

Figure 5: Two examples of diagonal iteration occurring. The highlighted calls are a part of the iteration

B. *Control Recursion and Mirroring*

Control recursion is defined simply as an operation being under two or more control qubits [17]. Identification of this pattern is easy, and requires only looking at each operation row in the data for more than one ctrl entry. Alongside this pattern, Huang and Martonosi introduce the Mirroring pattern, which involves the deallocation of qubits by reversing any previously performed operations. OpenQASM provides the reset statement to accomplish the same thing, so we do not provide any additional checks for this pattern. To find all places the qubits are reset back to `|0>`, searching for all `reset` operations is sufficient.

C. *CNOTs in Hadamard Bases*

Along with identifying patterns, srcML can be used to identify optimizations to the source code, and implement them as transformations using srcML. One such example of this is the use of a CNOT between calls to the Hadamard gate. According to the book by Rieffel and Polak, performing a CNOT on two qubits between calling the Hadamard gate on both qubits before and after the CNOT is equivalent to calling a CNOT with the arguments reversed [18 p.80]. Figure 6 gives an example of this transformation.

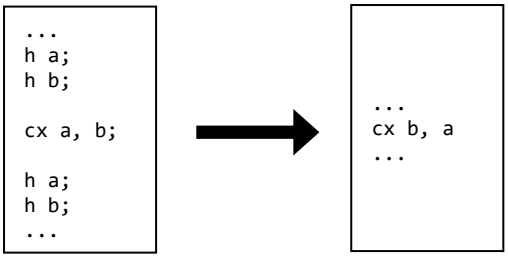


Figure 6: An example of a code optimization, where a CNOT between two sets of calls to the Hadamard gate in a code fragment can be replaced with the reversed CNOT and removal of the Hadamard gates.

D. Combining Gate Calls

In OpenQASM, all standard library and user defined gates are fundamentally defined by two primitive gates, `U` and `gphase`. `U` is the unitary matrix gate and `gphase` is the global phase gate. By analyzing all operations linearly on each individual qubit, sums of the underlying `U` calls made to each qubit can be constructed and replace consecutive gate calls. Table 2 details an example of this on two qubit calls.

Gate Call	Equivalent U call	Sum
`x $0;`	`U(π, π/2, π) $0;`	
`y $0;`	`U(π, π/2, π/2) $0;`	`U(2 * π, π/2, 3* π/2) $0;`

Table 2: An example of how two different gate calls can be summed into one singular unitary matrix operation

Reducing code in this manner provides both benefits and deficits. Positively, condensing code in this manner can reduce the amount of overhead, execution time, and amount of noise on certain hardware as the amount of `U` calls made in total is reduced. Negatively, this will obscure the code and make it much harder to sight-read. Custom-named gates are easier for a reader to understand at a glance.

E. Code Reusability

Quantum programs are highly susceptible to repeating code. Due to direct translation from quantum circuits to quantum code, a lot of quantum programming code can end up being very linear, with little to no encapsulation within functions and gates.

By analyzing the series of operations, the source code can be refactored by identifying common subsets of operations which are used multiple times within the file. These subsets can then be encapsulated into a function or gate, and all instances of the original calls can be replaced with a single call to the new function/gate. This can serve to make quantum code more readable, as it will reduce file size and encapsulate commonly occurring series of operations into a single location.

This also makes future development easier, as the function/gate can now be called instead of copying the exact operations again.

VI. Conclusions and Future Work

By supporting for OpenQASM in the srcML infrastructure, a large realm of program analysis is made available to the quantum community. We demonstrate a tool that uses srcML to perform some simple analysis, and then provide examples of what can be achieved through this analysis.

In the future, our qubit analyzer can be improved to support more of OpenQASM's classical and quantum features. Features like branching, delays, boxes, and others can support finer-grain analysis, giving researchers more insight into their code.

Additionally, once the Parser Generator for srcML is working, we plan to expand support for other quantum programming languages. The elements added in the OpenQASM markup specification are designed to be as generically named and formatted as possible, so future quantum languages can be added and reuse the elements. Doing this will ensure that analysis can be as language agnostic as possible.

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] A. W. Cross *et al.*, "OpenQASM 3: A broader and deeper quantum assembly language," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–50, Sep. 2022, doi: 10.1145/3505636.

[2] A. W. Cross, L. S. Bishop, J. Smolin, and J. M. Gambetta, "OpenQasm Live Specification — OpenQASM Live Specification documentation." Accessed: Nov. 30, 2023. [Online]. Available: https://openqasm.com/index.html

[3] A. J. Abhari *et al.*, "Scaffold: Quantum Programming Language," Jul. 2012. Accessed: Apr. 02, 2024. [Online]. https://www.semanticscholar.org/paper/Scaffold%3A-Quantum-Programming-Language-Abhari-Faruque/427e9698df9c2080547ca97411fb8d13fadee381

[4] K. M. Svore *et al.*, "Q#: Enabling scalable quantum computing and development with a high-level domain-specific language," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, Feb. 2018, pp. 1–10. doi: 10.1145/3183895.3183901.

[5] 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*, Williamsburg, Virginia, USA: IEEE, Sep. 2011, pp. 173–184. doi: 10.1109/SCAM.2011.19.

[6] 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 *29th IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 516–519. doi: 10.1109/ICSM.2013.85.

[7] M. Paltenghi and M. Pradel, "LintQ: A Static Analysis Framework for Qiskit Quantum Programs." arXiv, Oct. 01, 2023. doi: 10.48550/arXiv.2310.00718.

[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)*, May 2023, pp. 50–57. doi: 10.1109/Q-SE59154.2023.00014.

[9] M. Kaul, A. Küchler, and C. Banse, "A Uniform Representation of Classical and Quantum Source Code for

Static Code Analysis," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Sep. 2023, pp. 1013–1019. doi: 10.1109/QCE57702.2023.00115.

[10] S. Xia and J. Zhao, "Static Entanglement Analysis of Quantum Programs," in *2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*, May 2023, pp. 42–49. doi: 10.1109/Q-SE59154.2023.00013.

[11] P. Zhao, J. Zhao, and L. Ma, "Identifying Bug Patterns in Quantum Programs," in *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, Jun. 2021, pp. 16–21. doi: 10.1109/Q-SE52541.2021.00011.

[12] P. K. Nayak, K. V. Kher, M. B. Chandra, M. V. P. Rao, and L. Zhang, "Q-PAC: Automated Detection of Quantum Bug-Fix Patterns." arXiv, Nov. 29, 2023. doi: 10.48550/arXiv.2311.17705.

[13] 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, doi: 10.1109/TVCG.2023.3327148.

[14] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. in Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, 2018.

[15] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 358–370. doi: 10.1109/ICSE48619.2023.00041.

[16] J. Zhao, "On Refactoring Quantum Programs in Q#," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Sep. 2023, pp. 169–172. doi: 10.1109/QCE57702.2023.10203.

[17] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proceedings of the 46th International Symposium on Computer Architecture*, in ISCA '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 541–553. doi: 10.1145/3307650.3322213.

[18] E. Rieffel and W. Polak, *Quantum computing: a gentle introduction*. in Scientific and engineering computation. Cambridge, Mass.: MIT Press, 2011. Accessed: Nov. 28, 2023. [Online]. Available: http://www.books24x7.com/marc.asp?bookid=52592