Dissertation

Supervisor: Carlos Perez Delgado
Programme: MSc Computer Science (Artificial Intelligence)
Word Count: 10,000

December 3, 2024

Acknowledgments

Abstract

List of Figures

List of Tables

List of Abbreviations

- AI - Artificial Intelligence
- COBYLA - Constrained Optimization by Linear Approximation
- MSc - Master of Science
- NISQ - Noisy Intermediate-Scale Quantum
- PUB - Primitive Unified Blocs
- SDK - Software Development Kit
- SU(2) - Special Unitary Group of Degree 2
- QUML - Quantum Unified Modelling Language
- UML - Unified Modelling Language
- VQE - Variational Quantum Eigensolver

Contents

1 Introduction

1.1 Problem Description and Incentive

Quantum Software Engineering (QSE) is an emerging research field seeking to develop and standardise software engineering principles in quantum technologies. Many of these standards have been adapted from classical software engineering (CSE), attempting to keep them as familiar as possible to CSE whilst being able to distinguish that quantum and classical systems are two fundamentally different hardware.

One area of interest in QSE is the adaptation of Unified Modelling Language (UML) to model quantum systems and illustrate their communication with classical systems. Two notable papers have introduced adaptations of UML for this purpose: Q-UML [**Prez-Delgado2022**] and a quantum UML profile [**Prez-Castillo2022**]. Both propose methodologies for incorporating quantum technologies into UML, aiming to broaden the scope of professionals who can contribute to the QSE field and promote early adoption of a standardised quantum modelling language whilst quantum technologies remain in relative infancy.

The critical question is which quantum UML adaptation offers the best solution. This involves evaluating their effectiveness for modelling quantum systems, their suitability in real-world applications and their potential for widespread adoption. Additionally, should the industry favour full-scale UML modelling or a simplified approach, such as flowcharts, for quantum system design, and which of the two adaptations can best accommodate these design preferences?

1.2 Goals and Objectives

This project aims to create and contrast UML diagrams using these two quantum UML approaches.

The first objective is to choose an appropriate real-world example as a starting point for the initial diagrams. The example chosen is the Variational Quantum Eigensolver (VQE), a hybrid quantum/classical algorithm. VQE is ideal for exploring how communication between quantum and classical machines can be represented in a modelling language.

The second objective is to choose the most suitable UML diagram to model the algorithm and compare the quantum UML methods. A sequence diagram was selected because it is one of the most commonly used UML diagrams and illustrates the communication between quantum and classical modules throughout the VQE algorithm.

The third objective is to determine a robust method for comparison and analysis. This will involve combining the author's observations, following established guidance on good diagram design, and feedback gathered from anonymous polling. Quantum computing and software engineering professionals would be ideal participants in this feedback process.

Finally, the last objective is to create multiple UML diagrams. A valuable exploration will involve modelling fundamental quantum properties such as entanglement and superposition, as proposed by the authors of the quantum UML literature. This approach will enable the use of other UML diagrams or the application of sequence diagrams in a new context.

2 Literature Review

2.1 Background

2.1.1 Q-SE 2020

Q-SE 2020 was the first international workshop held virtually in July 2020, co-located with ICSE 2020. Its purpose was to unite researchers and professionals to form a community that could advance quantum software development.

Paper submissions were requested, including "Towards a Quantum Software Modelling Langauge" [**Perez-Delgado2020**], which laid the groundwork for Q-UML, which was further expanded upon in the paper "A Quantum Software Modeling Language", presented in "Quantum Software Engineering" [**serrano2022quantum**].

The authors of "Quantum Software Engineering" [**serrano2022quantum**] developed their own quantum UML profile, first published in March 2021 [**Prez-Castillo2021**], which was later expanded on and presented in "Computing" [**Prez-Castillo2022**].

Both papers sought to adapt UML to integrate quantum computing principles.

2.1.2 UML 2

UML 2, simply referred to as UML, is a modelling language and the industry standard for designing and documenting computer systems. It includes 14 types of diagrams divided into structural and behavioural categories. UML 2 signifies that we are currently using the second version of this language.

Sequence diagrams, part of the behavioural category, belong to a sub-category of interaction diagrams, which includes communication, timing, and interaction overview diagrams.

Sequence diagrams model communication protocols between human and non-human entities. The horizontal axis represents the sequence of communication messages, while the vertical axis indicates the timing of interactions. Each component in a sequence diagram has a lifeline extending vertically, which may terminate if the element is no longer involved in the sequence.

As the industry standard for CSE, UML is the logical choice for adaptation for QSE. To evaluate its effectiveness, it should be tested and modelled using current quantum hardware.

2.1.2.1 Sequence Diagram

2.1.2.2 Class Diagram

2.1.3 NISQ

The Noisy Intermediate Scale Quantum (NISQ), known as the near-term quantum computer, employs qubits ranging from tens to hundreds. An NISQ device is sensitive to noise and requires error correction. We are currently in the NISQ era, which means quantum advantage has yet to be achieved.

Despite this, computations can still be performed on NISQ devices, with workloads typically split between classical and quantum computers. One such computation that can be accomplished through this hybrid approach is the Variational Quantum Eigensolver.

2.1.4 VQE

The Variational Quantum Eigensolver (VQE) is a hybrid algorithm that leverages both quantum and classical devices to find the ground state of a given physical system.

The algorithm starts with the physical system represented as a Hamiltonian and prepares a parametrised trial state called the ansatz. A quantum computer evaluates the ansatz by applying a quantum circuit to explore the parameter search space. A classical optimiser then adjusts the parameters in each iteration, guiding the quantum circuit's search and gradually refining the ansatz until the algorithm converges on the lowest energy estimate, the ground state of the physical system.

2.1.5 Qiskit

Qiskit is an open-source software development kit (SDK) created by IBM to access and utilise their cloud-based quantum computing services. Implemented in Python, Qiskit provides tools and libraries for quantum programming and experimentation. The VQE algorithm represented in the sequence diagram is based on the implementation of VQE in Qiskit [**IBM2024**], which interacts directly with the IBM Quantum Platform.

2.1.5.1 Qiskit and VQE

IBM Quantum Learning provides online tutorials[**Tutorial**] on implementing quantum algorithms utilising the Qiskit SDK. The UML diagrams created for this project were based on the tutorial to implement the VQE algorithm in Qiskit[**IBM2024**].

An instance of the class *SparsePauliOp* is used to create a Hamiltonian object by calling the *.from_list()* method for its construction. This Hamiltonian is a classical representation of Pauli operators, where each operator is given as a string (e.g., "X", "Y", "Z", or "I" for identity). Pauli operators are 2×2 matrices corresponding to spin measurements along the x, y, and z

axes[**DJORDJEVIC201229**]. Each operator string specifies actions on individual qubits. The VQE tutorial uses a 2-qubit system, acting on pairs of Pauli strings called "operator terms". The tensor product of these pairs is assigned a coefficient (represented as a complex number in Python) that defines the strength of each operator term. The linear combination of these terms represents the system's total energy, the Hamiltonian. Only non-zero operators and coefficients are stored, resulting in a sparse representation of operator terms and the Hamiltonian to reduce computational expense.

An instance of the class *EfficientSU2* is used to create the ansatz with the number of qubits the Hamiltonian holds passed to it in its construction. This class provides a hardware-efficient classical representation of a quantum circuit capable of creating parametrised quantum states. The circuit comprises layers of single-qubit operations along with C-NOT gates, which entangle the qubits. The Qiskit documentation defines SU(2) as *"the special unitary group of degree 2, its elements are 2×2 unitary matrices with determinant 1, such as the Pauli rotation gates"*[**EfficientSU2**], meaning that the circuit includes layers of operations that rotate the states of individual qubits, specifically using Pauli rotation gates. Each Pauli rotation gate holds a parameter which will be iteratively adjusted to find the lowest energy state of the ansatz.

The class *QiskitRuntimeService* interacts with the IBM Qiskit Runtime Service that provides cloud-based access to quantum hardware and quantum simulators. Creating an IBM account and giving a token when executing the code to access the service is necessary. The parameter *"ibm_quantum"* is given to the instance of *QiskitRuntimeService* to access the quantum computing platform available on the IBM cloud service. The method *least.busy()* is used to select the next available quantum hardware with the parameter *(simulator)* set to false to access the quantum hardware as opposed to a quantum simulator.

QiskitRuntimeService will instantiate a *IBMBackend* object which interacts with the selected quantum hardware. The attribute *target* of the *IBMBackend* object is accessed and passed as a parameter to an instance of the *StagedPassManager* class, created using the *generate_preset_pass_manager()* method. This process allows the pass manager instance to receive information regarding the constraints of the selected quantum hardware.

The *StagedPassManager* object will *"define a typical full compilation pipeline from an abstract virtual circuit to one that is optimized and capable of running on the specified backend"*[**StagedPassManager**]. The *.run()* method is executed on the pass manager instance to transform the ansatz circuit to be compatible with the selected quantum hardware. The *.apply_layout()* method is then called on the Hamiltonian, with the newly transformed ansatz passed as a parameter, to modify the Hamiltonian's layout to be compatible with the selected quantum hardware.

A cost function method, defined as *cost_func*, is constructed to facilitate access to quantum hardware. It accepts an estimator (written later in the code) and the components of a primitive unified bloc (PUB) as its parameters. The PUB object comprises an array of initial guesses for the parameters of the ansatz in the form of a list (these parameters are also written later in the code), the

Hamiltonian represented as a list of observables, and a quantum circuit, this being the ansatz. These components form a tuple and are assigned to the variable *"pub"*, which is given as a parameter for the estimator object's *.run()* method. The *.result()* method returns a container of PUB results[**PrimitiveResult**] with slicing utilised to access the estimated energy of the ansatz, which is stored in the variable *"energy"*. A dictionary stores the parameters, iteration, and energy estimate each time the estimator's *.run()* method is executed. Its initial values consist of a placeholder for the parameters, *"0"* for the iteration and an empty list for the energy estimate, with each energy estimate being appended to the list per execution. Finally, the cost function method returns the value of the *"energy"* variable as its result after being called.

A random array of initial guess parameters, assigned to the variable *x0*, is constructed using NumPy's constant π and its *random.random()* function. This function generates an array of random floating-point numbers, scaled to the range of [0,π2]. The ansatz attribute *num_parameters* sets the size of the array to match the number of parameters to assign to each of the Pauli rotation gates in the ansatz circuit.

An instance of the *Session* class is created with the *IBMBackend* instance as a parameter. This *Session* instance is then passed to the *Estimator* object, where it is assigned to the estimator's *mode* attribute. This *Estimator* object, which operates within the cost function, uses the *Session* object to execute computations on the specified quantum backend. The *Session* object enables the grouping of iterative calls to the quantum computer[**Session**] when the *Estimator*'s *.run()* method executes, efficiently managing the allocation of jobs to quantum resources.

An instance of the classical *Minimize* function from the SciPy package is initialised with the Constrained Optimization by Linear Approximation (COBYLA) method selected to minimize a scalar function, in this case, the energy estimate returned by the cost function. The *Minimize* function's parameters include the *cost_func*, the initial guess array *x0* for the ansatz parameters, along with the *ansatz*, *Hamiltonian*, and *estimator*, all passed to the cost function for execution.

During the optimisation loop, the cost function calculates and returns the energy estimate of the ansatz, and the *Minimize* function iteratively updates the parameters in *x0* to reduce this energy estimate. Each call to the cost function runs 10,000 shots on the quantum circuit via the estimator. The loop continues until the energy estimate converges to the lowest achievable value, indicating the ground state energy.

Upon completion of the optimisation loop, successful termination of the process is verified by comparing the solution parameter and evaluation count against the stored solution parameter and iteration count within a dictionary maintained by the cost function. These results are then visualised using the Matplotlib package, plotting a graph with the number of iterations on the x-axis and the energy estimates on the y-axis.

2.2 Related Work

2.2.1 Q-UML

Q-UML is a methodology for integrating quantum computing into the UML standard. The paper establishes a fundamental axiom and five core design principles to consider when designing a UML quantum extension. As the axiom asserts, while the design should adhere to classical standards as closely as possible, it must acknowledge the fundamental differences between classical and quantum hardware. The design principles describe when and how to label components in a diagram as classical or quantum.

In Q-UML, quantum elements are visually distinguished using bold font for textual labels and double lines for diagrammatic components.

2.2.2 Quantum UML Profile

A quantum UML profile is an extension of UML designed explicitly for hybrid information systems incorporating classical and quantum software.

In UML, a profile is a structural diagram that extends the language to support domain-specific custom models. The quantum UML profile introduces a set of stereotypes, tagged values and constraints to represent quantum concepts.

3 Design and Implementation

3.1 Plant UML

The initial version of the VQE sequence diagram was drafted using the Plant UML plugin for PyCharm, an open-source tool that enables users to generate UML diagrams from plain text.

To create the Q-UML version of the VQE sequence diagram, a combination of Plant UML and Lucidchart was used to include the double lines and bold text necessary to distinguish quantum components. However, for creating the quantum UML profile, Plant UML alone may suffice, as it should have the tools required to extend UML to represent quantum-specific elements.

3.2 Lucidchart

Lucidchart is a web-based application that creates general-purpose diagrams, including UML diagrams, through a graphical user interface (GUI). It is used when creating Q-UML adaptations as it allows the customisation of elements to contain bold text and double lines necessary to distinguish quantum components.

The first completed diagram is a PNG file created in Lucidchart. An attempt was made to keep it to a format similar to the Plant UML output. Consistency must be considered to ensure a fair comparison between both diagrams; therefore, even if the VQE quantum UML profile diagram can be reproduced in

Plant UML alone, it may need to be translated into Lucidchart to ensure this consistency is upheld.

3.3 VQE Sequence Diagram

3.3.1 SD QUML

This diagram has been completed and is attached at the end of this report. It illustrates the sequence of messages between modules necessary to execute the VQE algorithm.

Multiple Pauli operators define the Hamiltonian "sparsely" instead of creating a complete matrix. It acts upon two qubits at a time, and the qubits count is passed to the ansatz object. The ansatz is a circuit that creates a trial state for the experiment.

These objects are transformed through a pass manager to be compatible with a quantum computer, the "backend". This process does not require communication with the quantum hardware, as the information is held classically; therefore, the diagram does not depict these modules as quantum.

A session is created and configured to the same backend, passing multiple circuits to an estimator object while the session handles resource management. This estimator is a quantum object, depicted as such in the diagram, as it utilises quantum hardware to estimate the values of the quantum circuits.

The methods `minimize` and `cost_func` contain the estimator object and are also depicted as quantum. The `cost_func` takes the Hamiltonian, ansatz, and parameters (initially a NumPy array of random values) and combines them into a new variable, `pub`, which is then given to the estimator object. The estimator executes 10,000 shots, defined during its creation, meaning the quantum circuit runs 10,000 times. This message is marked as a quantum message with **bold text**, indicating that the communication uses quantum hardware. The estimator then returns an energy estimate, converted to classical information before being received by the `cost_func`, therefore depicted as a classical message.

The `cost_func` method updates the parameters and records critical information in a `Dictionary` object per iteration. The `minimize` method, acting as an outer loop, receives the energy estimate and uses a classical optimiser (COBYLA) to guide the process into finding the lowest energy estimate.

The loop is completed once the energy estimate returned by `cost_func` converges to the lowest energy estimate, and the parameters and iterations are verified with the `Dictionary` object. The final step involves plotting each iteration and energy estimate using the Matplotlib Python library.

3.3.2 SD Quantum UML Profile

This has yet to be completed and will be the next step in the process.

3.4 VQE Class Diagram

3.4.1 CD QUML

3.4.2 CD Quantum UML Profile

4 Results and Analysis

4.1 Author Observations

My initial observations of the first diagram suggest that while the bold text and double lines used to distinguish quantum components are visible, they are more easily identifiable in examples provided in the Q-UML paper. This is likely due to the larger scope of the VQE sequence diagram and the fact that it is primarily composed of classical elements. Additional design choices could be considered to better differentiate between classical and quantum components, although these would fall outside UML standards. For example, different colours could represent classical and quantum modules. This feature would only be readily available in applications like Lucidchart but could be incorporated with Plant UML by developing an extension. Once the quantum UML profile version of the VQE sequence diagram is completed, more detailed observations and comparisons between the two diagrams will be possible.

4.2 Diagram Literature

Once both diagrams are completed, it will be crucial to consult relevant literature on effective diagram design to make an informed assessment. Research papers exploring good diagram design, particularly concerning UML, should be considered. An example of such work is the paper "Improving Information System Design: Using UML and Axiomatic Design" [**CAVIQUE2022103569**], which provides valuable insights into improving system information design.

4.3 Polling

An anonymous questionnaire should be developed to present both VQE sequence diagrams to professionals and researchers in quantum computing and software engineering. Ideally, multiple UML diagrams would be included. The questionnaire should ask participants about the clarity of the diagrams, their understanding of the content being depicted, and their overall observations on the application of UML to quantum technologies. Additionally, gathering information on the industry sectors respondents are associated with would be valuable, providing insight into the diversity of perspectives within the results.

5 Conclusions

The paper will conclude with an overall assessment of the advantages and disadvantages of each method. It will address whether the primary goal and all four main objectives were achieved. Additionally, there will be a discussion on whether UML is the most suitable approach for representing quantum systems or if a more general flowchart format might be more appropriate. This will include considering which method could be used in such a case. I assume that Q-UML would remain the most applicable, as the quantum UML profile is inherently based on UML formatting.

6 Future Work

Depending on whether the fourth objective is achieved, further exploration into modelling fundamental quantum concepts would be beneficial for evaluating the effectiveness of applying UML to quantum systems. Plant UML is currently the most convenient tool for creating UML diagrams, as Lucidchart is significantly more labour-intensive. Developing an extension for Plant UML or other open-source tools like Mermaid to incorporate Q-UML formatting would be highly advantageous for future diagram creation. Such an extension would also increase the utility of this method, making it more accessible for continued exploration and broader adoption in the QSE field.

References

7 Bibliography

8 Appendix