Locus: A Proposal for Quantum Software Composition

Javier Zayas Gallardo
ITIS Software, Universidad de Málaga, Málaga, Spain

Francisco Chicano
ITIS Software, Universidad de Málaga, Málaga, Spain

Carlos Canal
ITIS Software, Universidad de Málaga, Málaga, Spain

Juan Manuel Murillo
Quercus Software Engineering Group, Universidad de Extremadura, Cáceres, Spain

Abstract

The way quantum programs based on circuits are built today has many analogies with how assembler routines were developed in the past. This way of writing programs is not only tedious but also makes it difficult to achieve good quality attributes such as reusability or maintainability. One of the main advances in improving the quality of classical software, as well as increasing programmers' productivity, was raising the level of abstraction in programming languages. This allowed developers to move away from direct bit-level operations – such as rotation, shifting, addition, or carry handling – toward working with a set of basic data types like Integer, Float, or Character, along with well-defined operations on them. We believe that introducing similar mechanisms in the field of Quantum Programming will help to achieve comparable benefits in quantum software. This paper presents the authors' preliminary efforts in this direction, introducing the concept of *locus*.

2012 ACM Subject Classification Software and its engineering → Software creation and management

Keywords and phrases Locus, Quantum programming, Quantum circuits

Digital Object Identifier 10.4230/OASIcs.Programming.2025.17

Funding This work is partially supported by the AI4Software network (RED2022-134647-T).
Francisco Chicano: Francisco Chicano is supported by the University of Malaga under grant PAR 4/2024.
Juan Manuel Murillo: Juan M. Murillo is supported by the projects Q-SERV (PID2021-124054OB-C31), RuralServ (TED2021-130913B-I00), PDC2022-133465-I00, and RCIS (RED2022-134148-T) funded by the Ministry of Science and Innovation. Also by grant Ref.2024/00640/001 by the Regional Goberment of Extremadura.

1 Introduction

The development of quantum computers in recent years has fostered the emergence of the discipline of Quantum Software Engineering (QSE) [12]. This discipline addresses the special requirements of quantum software in contrast to classical software, providing appropriate engineering processes for building quantum programs [8].

Many features of quantum systems make the constructs of classical programming languages inadequate for programming them. Some of these include dealing with a machine state that incorporates superposition, the fact that information units (qubits) can be manipulated in dimensions of phase and amplitude, the collapse of the qubits' superposition state when measured, and the impossibility of replicating all or part of the quantum state once it has been measured. The mechanisms provided by quantum programming languages to handle these characteristics build on quantum gates that can be applied to one or multiple qubits.

These mechanisms are similar to those found in the assembly language of a classical processor, where operations are performed on a bit or register [8]. As a result, building quantum programs nowadays is not only a tedious task but also, much like assembly routines, leads to very poor results in terms of reusability, maintainability, and compositionality.

The aim of this work is to contribute a new abstraction for quantum programming languages, designed to facilitate the creation of quantum programs as a composition of operations implemented as circuits. The proposed abstraction is called Locus (from the Latin for "place"), and it defines a place in the quantum program specified by a set of qubits and the operations that have been applied to those qubits up to that point. In this way, a Locus represents the quantum sub-state determined by the state of its associated qubits at the moment it is created. The purpose of defining a Locus is to apply an operation to it. This operation is simply a quantum circuit applied to the qubits of the Locus, that is aimed to produce a specific transformation.

The concept of a Locus may resemble that of a variable in a classical program, which defines part of the machine's state on which operations (appropriate to the variable's type) are to be applied. However, it differs in that two Locus instances over the same qubits, but at different points in the program, are considered two distinct Locuses. The specific operation to be applied to a Locus can be determined at a later stage or delegated to another part of the program. In this sense, applying an operation to a Locus can be regarded as a form of dependency injection into the Locus. The benefits it provides are analogous to those of this mechanism in classical programming: decoupling of code, which facilitates reusability and maintainability, as well as support for patterns such as inversion of control. To illustrate these aspects, the abstraction has been implemented in Qiskit, and several usage examples are provided in the paper.

This paper is structured as follows. Section 2 describes the motivation for this project. The following Section 3 explains the concept we propose in the article: Locus. This section will be divided into two subsections: 3.1 will describe the notion and use of Locus and subsection 3.2 will present some examples. Section 4 includes a discussion and related works. Finally Section 5 concludes the article with some conclusions.

2 Motivation

The principle of separation of concerns (SoC) introduced by Dijkstra [2] is ubiquitous in classical software development.

The proven effectiveness of this principle in classical software development suggests that it should be adapted for quantum software development. We can already identify opportunities to apply this design principle in many examples of quantum algorithms. For instance, Grover's search algorithm [5] consists of an alternation between two operations: the oracle and the diffusion operator. These two operations can be identified as separate modules.

Two further examples are Shor's algorithm [10], which employs the Quantum Fourier Transform (QFT) [1] internally as a standalone module – and the Quantum Approximate Optimization Algorithm (QAOA) [3], which is built from successive layers, each containing a problem-dependent module, U_P, and another, U_M, that depends on the so-called mixing Hamiltonian (see Fig. 1).

The standardized mechanisms currently provided by quantum programming languages are those related to the sets of supported quantum gates. These are operations that manipulate single qubits (by applying a phase along any axis) or qubit registers (by creating entanglement among the qubits in the register). These types of instructions are very low-

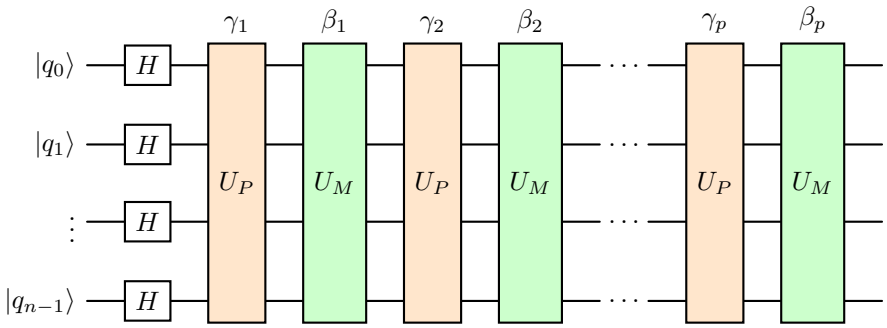

Figure 1 Quantum Approximation Optimization Algorithm: separation of concerns.

level and comparable to the instructions found in classical assembly languages for each processor. To provide programmers with higher abstraction mechanisms, some languages such as Qiskit[1] [7] offer builder operations such as *Custom Gates*. A custom gate allows us to encapsulate functionality in modules that can be applied and reused in different programs. It is, in fact, a powerful abstraction mechanism that should not be confused with a simple gate. A custom gate can be parameterized in terms of the qubits it involves, and even in the structure of the internal circuit it implements.

For a custom gate to be reusable, it must be well-designed [9]. For example, a custom gate that implements a generic QFT can be created. The QFT can then be reused in various programs requiring this functionality, such as an implementation of Shor's algorithm. To enable that reuse, the QFT must be parameterized with the number of qubits in the circuit implementing the QFT. This prevents the need to build QFT custom gates for different circuit sizes. Such parameters allow to adjust how the circuit implementing the custom gate is constructed. The custom gate mechanism is complemented by the ability to specify, at instantiation time, how the qubits of the custom gate are connected to those of the receiving circuit.

In this way, custom gates are a mechanism that gives the programmer the freedom to create their own levels of abstraction in the form of complex operations. At the same time, they allow for the construction of reusable quantum code libraries. However, they still encourage building programs in a sequential manner. Gates and the different parts of the circuit are composed (using the `append` instruction) one after another. Once a custom gate is added to a circuit, its function and the way it connects to the circuit become fixed. As a result, common practices in classical programming – such as expressing conditional or deferred behavior – become difficult to implement. To address this gap, the next section introduces the *Locus* mechanism, a placeholder in the circuit where functionality will be added later. Locus can be used as a complement of custom gates.

3 Locus

In this section we will discuss the Locus concept we present as well as an early phase of the implementation of the concept by showing some examples.

[1] We take Qiskit as a reference since it supports advanced abstractions and it is the most widely used quantum programming language.

3.1 Notion and use of *locus*

A Locus in a quantum program determines a specific location within that program. A Locus is created to inject a quantum circuit at that specific point. It designates the set of qubits to which it applies and the position in the quantum program where it is applied. The circuit injected at the Locus will operate on the same qubits designated by the Locus. The input of this injected circuit will be connected to the circuit before the Locus, and its output will be connected to the circuit after the Locus.

The code[2] shown in Listing 1 introduces the basic syntax to declare and use a Locus in Qiskit programs. It is an informal description intended to provide an initial idea of how to operate with a Locus in its various forms. First, Line 6 creates and instantiates a locus on qubits 1, 2, and 3 of a circuit (built in Lines 1–3).

Later, once a Locus has been created and instantiated, a circuit can be injected into it. Lines 10–13 illustrate the steps for that. First, a circuit (`myCustomGate`) is created and then it is injected into the locus. `inputConnection` and `outputConnection` are optional parameters specifying how the logical qubits of `myCustomGate` must be connected to the circuit before and after the locus. For example, for `myCustomGate` a value [2,1,0] for `inputConnection` will specify that the qubits 0, 1 and 2 of `myCustomGate` will be connected to qubits 2, 1 and 0 of the Locus, respectively.

Lists of locus can also be created and instantiated (Lines 17–24). Later, circuits can be injected into each locus in the list (Lines 26 and 27), and the same circuit can even be injected into all the locus in the list (Line 29).

Alternatively, a Locus can be created and later instantiated with different instructions (Lines 35 and 42). With this approach, the same Locus can be instantiated multiple times in different locations (Line 46). This is useful when the same circuit is to be injected in various points of the program, always using the same input-output connection pattern.

3.2 Some examples using Locus

This section presents some practical examples of Locus usage. The purpose of these examples is to show how the concept of Locus encourages the programmer to conceive a quantum program as a series of modules, each performing a specific functionality, and composed using Locus.

3.2.1 Locus as architectural connectors

The simplest use of Locus is as an architectural connector. In this scenario, an application can be conceived as a series of routines in the form of circuits, each performing a well-defined functionality, with Locus instances between them establishing the connections among the different circuits. For example, Grover's algorithm can be envisioned as three connected routines: an initialization to create the state of perfect superposition, an oracle that applies a phase mark to the searched values, and a routine that performs amplitude amplification on the marked values. Between these three routines, a Locus can be instantiated whose sole responsibility is to implement the connection between the different circuits. Listing 2

[2] All the instructions presented here correspond to the implementation of the Locus developed by the authors in Qiskit, which is available in the accompanying Zenodo repository `https://doi.org/10.5281/zenodo.15359664`.

```
        nqbits = 4
        myCircuit = CircuitBuilder (nqbits)
        # intructions to build the circuit before the locus
        ...
        # the locus is created and instantiated at qbits 1 to 3
        myLocus = myCircuit.newLocus ([1,2,3])

        ...

        n = 3
        myCustomGate = createCustomGate(n)

        myLocus.inject (myCustomGate, inputConnection, outputConnection)

        ...

        steps = range (2)
        myLocusList = LocusList (steps)

        for step in steps:
            # Here something can be added to myCircuit
            locusqubits = qbitsinvolved(step)   # Calculation of the qbits involved in
                                                # the locus at each step
            myLocusList[step] = myCircuit.newLocus (locusqubits)
            # Also here something can be added to mycircuit

        for step in steps:
            myLocusList[step].inject(myCustomGate(step), computeInputConnection(step),
                                     computeOutputConnection(step))
        # Or
        myLocusList.inject (myCustomGate, inputConnection, outputConnection)

        ...

        numqbits = 3    #number of qbits of the Locus
        # A locus of numqbits is created
        myLocus = Locus.create(numqbits)

        steps = range (2)

        for step in steps:
            # Here something can be added to myCircuit
            locusqubits = qbitsinvolved(step)   # Calculation of the qbits involved
                                                # in the locus at each step
            myCircuit.putLocus (locusqubits)
            # Also here something can be added to mycircuit

        # Then the same circuit can be injected en all the instances of the locus using
        # the same connection schema
        myLocus.inject (myCustomGate, inputConnection, outputConnection)

```

Listing 1 Code showing the different alternative syntax to create and manipulate Locus.

shows the implementation. Even though the example uses trivial connectors for the sake of simplicity, the reader can easily understand that this is not a limitation. The decision on how the connection is established can be delegated, for example, to a function that evaluates certain conditions in order to determine the connection scheme.

This approach allows, on the one hand, the construction of cleaner routines that are free from any influence of the environment in which they will be used, and on the other, connectors with a well-encapsulated and maintainable implementation.

3.2.2 Locus for deferred instantiation

Another use of the locus is as a deferred instantiation. This can be applied to various scenarios such as selecting certain circuits depending on the depth of the circuit or, as we will see in the following example, depending on which simulator to use. An example is to use a certain oracle or another to build an implementation of Grover's algorithm depending on which simulator is used. In this scenario we can build the circuit with Grover's algorithm leaving as an unknown which oracle is going to be used with the help of the Locus. Once we have the circuit created we check which simulator will be used and depending on the answer we will inject in the locus an implementation of oracle or another one. Listing 3 shows the implementation.

4 Discussion and related works

The Locus concept is intended to provide programmers with the ability to conceive their quantum applications as a composition of modules. The goal is to enable the construction of quantum applications using a top-down approach, shifting across different levels of abstraction beyond that of quantum gates. In addition to its use in quantum programming languages, Locus can also be adopted in higher-level representations of quantum applications, such as Architecture Description Languages (ADLs) or modeling languages.

Some current quantum programming languages also offer abstractions beyond quantum gates. Perhaps the most well-known are those found in Qiskit, such as Registers and Custom Gates. These abstractions, when used together, are powerful and enable basic composition mechanisms. Although custom gates are primarily designed to encapsulate the implementation details of a circuit segment, they can also be used to abstract the connections between different parts of a quantum program. If such parts are also provided as quantum gates, it is possible to compose complex quantum programs as combinations of custom gates, with the composition itself supported by the custom gate mechanism.

The idea of creating reusable quantum code can be found in other proposals, such as the one by Guo et al. [6]. In this work, ansatzes are mentioned as methods of designing and creating quantum algorithms. Its main contribution is to assist quantum algorithm designers by offering a structured repository of design patterns that enable the selection and reuse of ansatzes for variational quantum algorithms. Our Locus proposal can extend this work by introducing a new abstraction level that complements ansatz selection. While Guo et al. focus on the structure of the circuit, Locus focuses on how to organize, compose, and inject these structures in a flexible and maintainable manner. Locus enables deferred circuit injection, separation of concerns, and modularity promoting reuse and adaptability, which align closely with the reuse and abstraction goals emphasized by Guo et al.. Conversely, the ansatz catalog can directly enhance the Locus concept by serving as a structured repository of candidate modules to inject.

```
1
2         # Grover's algorithm as a threee routines composition supported by Locus
3         nqbits = 4
4         myCircuit = CircuitBuilder (nqbits)
5
6         def superpostion(nqubits)
7             superpostion = CircuitBuilder(nqubits)
8             for i in range nqubits
9                 superpostion.h(i)
10            return superpostion
11
12        def oracle(nqubits, target)
13            oracle = CircuitBuilder(nqubits)
14            bin_target = f"{target:0{nqubits}b}"
15            for i, bit in enumerate(reversed(bin_target)):
16                if bit == '0':
17                    oracle.x(i)
18            oracle.h(nqubits - 1)
19            oracle.mcx(list(range(nqubits - 1)), nqubits - 1)
20            oracle.h(nqubits - 1)
21            for i, bit in enumerate(reversed(bin_target)):
22                if bit == '0':
23                    oracle.x(i)
24            return oracle
25
26        def difusser(nqubits)
27            difusser = CircuitBuilder(nqubits)
28            for q in range(nqubits):
29                difusser.h(q)
30            for q in range(nqubits):
31                difusser.x(q)
32            difusser.h(nqubits-1)
33            difusser.mcx(range(nqubits-1), nqubits)
34            difusser.h(nqubits-1)
35            for q in range(nqubits):
36                difusser.x(q)
37            for q in range(nqubits):
38                difusser.h(q)
39            return difusser
40
41        # The architecture of the application is created next. The oracle is created for value 3.
42        myCircuit.append(superposition(nqubits))
43        conector1 = myCircuit.newLocus([0,1,2,3])
44        myCircuit.append(oracle(nqubits, 3))
45        conector2 = myCircuit.newLocus([0,1,2,3])
46        myCircuit.append(difusser(nqubits))
47
48        # Finally connections are made. In this example, trivial connections are made.
49        conector1.inject(CircuitBuilder(nqubits), [0,1,2,3], [0,1,2,3])
50        conector2.inject(CircuitBuilder(nqubits), [0,1,2,3], [0,1,2,3])
51
52
```

Listing 2 Use of Locus as architectural connectors.

```
1
2         from qiskit import QuantumCircuit, QuantumRegister, Aer
3         from qiskit.circuit import Gate
4
5         def createOracleA(nqubits, target):
6             oracle = CircuitBuilder(nqubits)
7             bin_target = f"{target:0{nqubits}b}"
8             for i, bit in enumerate(reversed(bin_target)):
9                 if bit == '0':
10                    oracle.x(i)
11            oracle.h(nqubits - 1)
12            oracle.mcx(list(range(nqubits - 1)), nqubits - 1)
13            oracle.h(nqubits - 1)
14            for i, bit in enumerate(reversed(bin_target)):
15                if bit == '0':
16                    oracle.x(i)
17            return oracle
18
19        def createOracleB(nqubits, target):
20            oracle = CircuitBuilder(nqubits)
21            bin_target = f"{target:0{nqubits}b}"
22            for i, bit in enumerate(reversed(bin_target)):
23                if bit == '0':
24                    oracle.x(i)
25            oracle.mcz(list(range(nqubits - 1)), nqubits - 1)
26            for i, bit in enumerate(reversed(bin_target)):
27                if bit == '0':
28                    oracle.x(i)
29            return oracle
30
31        nqubits = 3
32        myCircuit = CircuitBuilder(nqbutis)
33
34        # Creation of the circuit architecture including Grover's algorithm
35        # and including locus to define the oracles later on.
36        for i in range(nqubits):
37            myCircuit.h(i)
38
39        steps = 2
40        oracles = LocusList(steps)
41        for i in range(steps):
42            myCircuit.append(difusser)
43            oracles[i] = myCircuit.newLocus([0,1,2], 'Oracle')
44
45        # Choice of which oracle to inject into the locus depending on which simulator is being used.
46        backend = Aer.get_backend('qasm_simulator')
47        backend_name = backend.name()
48        print("Backend:", backend_name)
49        if 'aer' in backend_name:
50            oracles.inject(createOracleA(nqbutis, 3), [0,1,2], [0,1,2])
51        else:
52            oracles.inject(createOracleB(nqbutis, 3), [0,1,2], [0,1,2])
```

Listing 3 Deferred circuit instantiation Using Locus.

Another related proposal is by Fernández-Osuna et al. [4]. They provide a large-scale analysis of how foundational quantum design patterns, such as initialization, superposition, entanglement, and oracle, are used in practice across Qiskit-based quantum programs. Our Locus proposal aligns naturally with these goals by introducing a higher-order abstraction for composing quantum operations. Locus provide support for the implementation of the architectural patterns so the instantitation of the different components can be deferred by injection. This compositional mechanism thus complements existing patterns by offering a flexible framework to structure and integrate them in scalable quantum software.

Also, [11] introduces high-level quantum data types into the Rhyme language to simplify quantum programming. These types extend classical primitives such as integers, floats and strings to the quantum domain, allowing developers to express superposition and entanglement using the familiar classical function syntax. While Rhyme abstracts typed quantum variables, Locus instantiated in the qbits correponding to a variable provide support for injecting the implementation of the different operations over the variable. structures how and where these operations are composed and injected into the circuit.

Finally, as shown in the examples, Locus can be used in conjunction with custom gates. In fact, Locus does not add any new functionality. Everything that can currently be done with Locus can also be achieved using registers and quantum gates. The difference lies in the approach: while registers and gates encourage building the application as a sequence of circuits that are appended to the existing one, Locus promotes thinking in terms of distinct functionalities that will be composed within a defined structure. Thus, custom gates are primarily conceived to encapsulate implementations within a circuit, whereas Locus is intended for broader use cases, including higher-level structural composition. Moreover, Locus allows deferring the implementation of some of these functionalities to a moment after their instantiation.

5 Conclusions

In this work, we have presented Locus, a mechanism that helps quantum application programmers approach development following a top-down methodology. Using this approach, they can generate multiple levels of abstraction between which they can shift. The concept of Locus is currently implemented as a prototype through additional syntax in Qiskit.

Locus facilitates deferring and delegating the implementation of application functionalities. However, all contextual conditions that may be evaluated to decide when and how a functionality is implemented belong to the (classical) Qiskit program that builds the quantum circuit. Ideally, a more powerful mechanism would be desirable, for instance, defining the circuit to be executed based on the result of measuring part of the quantum state. This can be done in Qiskit using *IfElseOp*, a mechanism that is mostly used for error mitigation. Even when the use of *IfElseOp* and *WhileLoopOp* operations imposes some limitations, exploring the possibilites of using them in conjuntion with Locus opens many possibilities for building complex dynamic and reactive quantum programs. As future work, we are already working on putting together all this concepts.

References

1 D. Coppersmith. An approximate fourier transform useful in quantum factoring, 2002. `arXiv:quant-ph/0201067`.

2 Edsger Wybe Dijkstra. On the role of scientific thought (EWD447). In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

3 Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014. arXiv:1411.4028.

4 Miriam Fernández-Osuna, Ricardo Pérez-Castillo, José A Cruz-Lemus, Michal Baczyk, and Mario Piattini. Exploring design patterns in quantum software: a case study. *Computing*, 107(5):1–31, 2025.

5 Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237866.

6 Xiaoyu Guo, Takahiro Muta, and Jianjun Zhao. Quantum circuit ansatz: patterns of abstraction and reuse of quantum algorithm design. In *2024 IEEE International Conference on Quantum Software (QSW)*, pages 69–80. IEEE, 2024. doi:10.1109/QSW62656.2024.00021.

7 IBM Quantum. Ibm quantum guides, 2024. Accessed: 2025-05-29. URL: https://docs.quantum.ibm.com/guides.

8 Juan Manuel Murillo, Jose Garcia-Alonso, et al. Quantum software engineering: Roadmap and challenges ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025. doi:10.1145/3712002.

9 Javier Sanchez-Rivero, Daniel Talaván, et al. Some initial guidelines for building reusable quantum oracles. In Flavia Monti, Pierluigi Plebani, Naouel Moha, Hye-young Paik, Johanna Barzen, Gowri Ramachandran, Devis Bianchini, Damian A. Tamburri, and Massimo Mecella, editors, *Service-Oriented Computing – ICSOC 2023 Workshops*, pages 197–208, Singapore, 2024. Springer Nature Singapore.

10 Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. IEEE, 1994. doi:10.1109/SFCS.1994.365700.

11 Tamás Varga, Yaiza Aragonés-Soria, and Manuel Oriol. Quantum types: going beyond qubits and quantum gates. In *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering*, pages 49–52, 2024. doi:10.1145/3643667.3648225.

12 Jianjun Zhao. Quantum software engineering: Landscapes and horizons, 2021. doi:10.48550/arXiv.2007.07047.