Towards a Quantum Software Modeling Language

Carlos A. Pérez-Delgado*
University of Kent
Canterbury, Kent, United Kingdom
c.perez@kent.ac.uk

Hector G. Perez-Gonzalez
Universidad Autónoma de San Luis Potosí
San Luis Potosí, SLP, México
hectorgerardo@uaslp.mx

ABSTRACT

We set down the principles behind a modeling language for quantum software. We present a minimal set of extensions to the well-known Unified Modeling Language (UML) that allows it to effectively model quantum software. These extensions are separate and independent of UML as a whole. As such they can be used to extend any other software modeling language, or as a basis for a completely new language. We argue that these extensions are both necessary and sufficient to model, abstractly, any piece of quantum software. Finally, we provide a small set of examples that showcase the effectiveness of the extension set.

CCS CONCEPTS

• **General and reference → General conference proceedings**; **Design**; • **Software and its engineering → System description languages**; **Unified Modeling Language (UML)**; **Software design engineering**; • **Theory of computation → Quantum computation theory**; **Quantum information theory**.

KEYWORDS

quantum computing, software engineering, UML

ACM Reference Format:
Carlos A. Pérez-Delgado and Hector G. Perez-Gonzalez. 2020. Towards a Quantum Software Modeling Language. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3387940.3392183

1 INTRODUCTION

Quantum computation rose to prominence after the discovery of quantum algorithms[5, 7] that can efficiently perform tasks that are intractable classically. These discoveries propelled research and interest in quantum computation. Today, there exists prototype quantum hardware with computational capabilities beyond that of any classical machine[1]. Further applications of quantum theory to computation have also been made in several areas of theory of computing, such as models of computation[6], data structures[8], and cryptography[2].

*Both authors contributed equally to this research.

Quantum computation has, until today, been studied almost exclusively 'in the small.' A general understanding of quantum computation, or, quantum programming 'in the large' is yet to be developed. Here we aim to set the foundations of a general framework for studying, developing, and conveying quantum programs. We aim to do so by developing a universal modeling language for quantum software. Rather than develop such a language from scratch, we have decided to start from the well-known Unified Modeling Language (UML)[3], and introduce a minimum set of extensions that allow it to effectively model quantum software.

Assuming UML to be a shared common-language upon which we can build, allows us to convey our original extensions much more succinctly. Our extension set can, however, be applied with little or no modification to any other modeling language.

2 Q-UML

Before discussing in depth the extensions we are introducing, we make a few fundamental observations on which we base the guiding principles for our extension set.

Our first observation is about the nature of quantum computation. The central difference between quantum and classical computation is in *how* it achieves its goals. Quantum computers have access to quantum algorithms[7], and quantum data-structures[8], that are unavailable to classical computers—hence their performance advantage. Algorithms and data-structures are, however, implementation details. Algorithms are an essential design choice while programming in the small. However, they are more often than not completely ignored in large-scale software architectural design. For instance, UML diagrams seldom portray algorithms and data-structures beyond a very high-level design perspective.

It would seem then that quantum computation introduces nothing to computation that needs to be captured in a software design diagram. This is not the case, and the reason for this is our second observation. Quantum computation changes the very nature of *information* itself. Quantum information is much richer than classical information. It is also much more challenging to store, transmit, and receive. If a module (class, object, *etc.*) needs to store, transmit or receive quantum information, then this is an important design consideration—which needs to be included in any effective software design.

A third observation here is that the classical vs. quantum nature of the information used by a module is an important consideration both when discussing its internal implementation *and* its interface. Furthermore, these two are separate and independent considerations.

A classical module, implementing some classical behavior, would have no need, or capability, to communicate quantum data. A quantum module may or may not have to; *i.e.* a module's quantum behavior may be completely part of its internal implementation and not appear as part of its interface. For instance, take a module implementing Shor's algorithm. Shor's algorithm uses quantum effects to efficiently factor a large integer into its prime factors. The implementation of this module must necessarily be quantum. Both the input (the large integer) and the output (the prime factors), consist of classical information. And hence, the *interface* of such a module can be strictly classical.

More generally, we can conceive of quantum software modules that have all classical inputs and outputs (like the above example), all quantum inputs and outputs, or a mix of both. A quantum software design must address, for each individual interface element, whether it is classical input/output, or if it is quantum. In short, whether a module communicates classically or via quantum information, and whether its internal implementation requires quantum hardware are important considerations that need to be captured in a design document.

The importance of such labelling should be clear. Quantum data can only be stored and transmitted with special hardware designed to do so. More importantly, from an abstract, device-independent, strictly software perspective: quantum and classical information are *not* interchangeable. Classical information is clone-able and admits fanout operations, while quantum information (in general) does not. On the other hand, quantum information has a much larger state-space.

Finally, it is true that quantum information is strictly a super-set of classical information—and hence a quantum module can communicate any classical information it desires using a quantum interface element. We argue, however, that using a quantum interface element and messaging when classical would suffice is bad quantum software design, for the reasons stated above.

In summary, the guiding principles behind any quantum software modeling language must include the following:

(1) **(Quantum Classes):** Whenever a software module makes use of quantum information, either as part of its internal state/implementation, or as part of its interface, this must be clearly established in a design document.
(2) **(Quantum Elements):** Each module interface element (*e.g.* public functions/methods, public variables) and internal state variables can be either classical or quantum, and must be labelled accordingly.
  (a) **(Quantum Variables):** Each variable should be labelled as classical or quantum. If the model represents data types, the variables should also specify the classical (*e.g.* integer, string) or quantum (*e.g.* qubit, qubit array, quantum graph state) data type,
  (b) **(Quantum Operations):** For each operation, both the input and output should be clearly labelled as either classical or quantum. Whether the operation internally operates quantumly should also be labelled.
(3) **(Quantum Supremacy):** A module that has *at least* one quantum element is to be considered a quantum software module, otherwise it is a classical module. Quantum and classical modules should be clearly labelled as such.
(4) **(Quantum Aggregation):** Any module that is composed of one or more quantum modules will itself be considered a quantum module, and must be labelled as such.
(5) **(Quantum Communication):** Quantum and classical modules can communicate with each other as long as their interfaces are compatible, *i.e.* the quantum module has classical inputs and/or outputs that can interface with the classical module.

We will argue in Sec. 2.3 how these extensions are not only necessary, but also sufficient in order to design and represent quantum software. First, in the following two sections we put these principles into practice as a set of concrete extensions to UML.

2.1 Class Diagram Extensions

UML is a very graphical language, meant to convey a lot of meaning in a very small amount of space. As such, it makes sense to use a graphical way to represent quantum software elements. We chose to do this by use of **bold** text to denote quantum elements, and double lines to denote a quantum relationship or quantum communication.

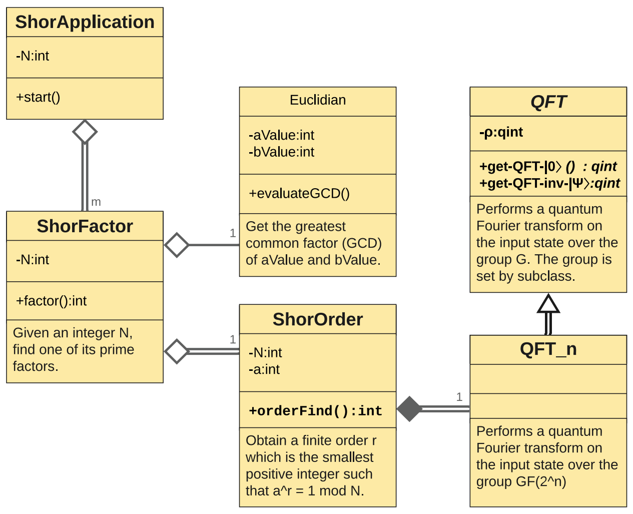
ShorApplication
-N:int
+start()
m
ShorFactor
-N:int
+factor():int
Given an integer N, find one of its prime factors.
Euclidian
-aValue:int
-bValue:int
+evaluateGCD()
Get the greatest common factor (GCD) of aValue and bValue.
1
ShorOrder
-N:int
-a:int
+orderFind():int
Obtain a finite order r which is the smallest positive integer such that a^r = 1 mod N.
1
QFT
-ρ:qint
+get-QFT-|0〉 () : qint
+get-QFT-inv-|Ψ〉:qint
Performs a quantum Fourier transform on the input state over the group G. The group is set by subclass.
QFT_n
1
Performs a quantum Fourier transform on the input state over the group GF(2^n)

Figure 1: Q-UML class diagram of Shor's Algorithm. Quantum classes and interface elements are presented in bold text, and quantum relationships use double-lines.

For attributes, the name will be bold if it is represented using quantum information. For methods, we use the following convention. If any of the inputs are quantum, these are bold. If the output or datatype of the method is quantum, then the datatype should also be bold. For backwards compatibility with regular UML, whenever the input or output datatypes of a method are omitted, these will be assumed to be classical in nature. If a class/object has any quantum attributes or methods then it itself is considered quantum, and its name shall also be bold.

Relationships between classes will use double-lines whenever the relationship is quantum in nature. For inheritance, if the superclass is quantum then the subclass, and the inheritance relationship, will also be quantum. (the converse is not necessarily true however). In the case of aggregation and composition, if a class/object being aggregated/composed is quantum, then the class/object to which it is aggregated/composed into, as well as that relationship will also be quantum. Association relationships do not have any special rules, beyond the need of a quantum class/object to have a classical interface if it is to associate with classical classes/objects.

Fig. 1 showcases a Q-UML diagram that exemplifies the above rules.

2.2 Sequence Diagram Extensions

Sequence diagrams in UML allow us to portray the *dynamic* relationship between modules in a software program. As we did before for static relationships, we extend the existing language in order to allow us to differentiate between classical and quantum messages. As previously discussed, this is essential information. Quantum information behaves differently from classical information; it can store/portray different data; it admits different operations; and, it requires different hardware to store, send, and receive.

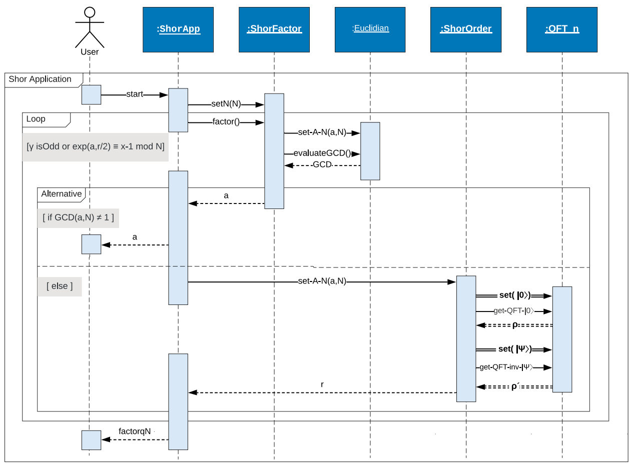

Figure 2: Q-UML sequence diagram of Shor's Algorithm. Quantum classes are presented in bold text, and quantum messages use double-lines.

Like before, we make use of **bold** text to markup quantum modules, and double lines to portray quantum messages. Fig. 2 shows a Q-UML sequence diagram. Note how even though the *relationship* between **Shorfactor** and **ShorOrder** is quantum, the messaging between them is *not*. This illustrates an important point. A module is marked as quantum if it uses quantum resources in any form, either directly as part of its internal implementation or as part of an aggregated module. If a sub-module (in UML a *composed* class or object) is quantum, then the encompassing module must also be marked as quantum. In a static (*e.g.* class) diagram, the quantum composition relationships inform us—especially in the case of a seemingly classical module that does not *in itself* use quantum resources—which composed modules are using quantum resources.

Also, note the communication between the objects **ShorOrder** and **QFT_n**. The module **QFT_n** operates on a quantum state. Hence, both 'set' messages are quantum. Likewise, the return messages ρ and ρ' are quantum states. However, the request to perform a quantum Fourier transform (QFT) or a QFT inverse operation can (and therefore should) be communicated classically. This diagram showcases the level of granularity available to us using these diagrams with the proposed extensions.

2.3 Discussion

We have proposed a minimal series of extensions to existing software modeling languages. We exemplify our additions in UML, but these extensions are easily applicable to any other modeling language, or be used as the basis for a new modeling language.

We've argued the necessity of each of the extensions in previous sections. We can argue as well, that these extensions are not only necessary, but also sufficient to fully model quantum software. To make this argument, we appeal to the fact that all quantum computation is simulable using classical computation albeit with an efficiency loss. Other than their use of quantum information and algorithms, quantum computers are indistinct from classical ones. Hence, from a high-level design perspective, the only information element that needs to be considered when developing quantum software is when quantum (rather than classical) information is being used.

The one remaining information element we have not discussed is algorithm efficiency. If quantum computation is to be used, it will most likely be due to the efficient algorithms at its disposal. That said, algorithm efficiency is not a solely quantum consideration. UML itself does not inherently have language elements for algorithm efficiency (beyond user-defined notes). It does, however, have several extensions used and proposed for this purpose(see *e.g.*[4]). Other modeling languages may also have definite algorithm efficiency elements. We argue that it is best to use existing language elements when they are available.

ACKNOWLEDGMENTS

CP-D would like to acknowledge funding through the EPSRC Quantum Communications Hub (EP/T001011/1). The authors would also like to thank Joanna I. Ziembicka for useful comments during the preparation on this manuscript.

REFERENCES

[1] Frank Arute *et. al.* 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5
[2] Charles H Bennett and Gilles Brassard. 2014. Quantum cryptography: public key distribution and coin tossing. *Theor. Comput. Sci.* 560, 12 (2014), 7–11.
[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. 2005. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series).* Addison-Wesley Professional.
[4] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. 2003. Performance modelling with the Unified Modelling Language and stochastic process algebras. *IEE Proceedings - Computers and Digital Techniques* 150, 2 (March 2003), 107–120. https://doi.org/10.1049/ip-cdt:20030084
[5] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing (STOC '96).* ACM, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866
[6] Carlos A. Pérez-Delgado and Donny Cheung. 2007. Local unitary quantum cellular automata. *Phys. Rev. A* 76 (Sep 2007), 032320. Issue 3. https://doi.org/10.1103/PhysRevA.76.032320
[7] Peter W Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science.* Ieee, 124–134.
[8] Liming Zhao, Carlos A. Pérez-Delgado, and Joseph F. Fitzsimons. 2016. Fast graph operations in quantum computation. *Phys. Rev. A* 93 (Mar 2016), 032314. Issue 3. https://doi.org/10.1103/PhysRevA.93.032314