Non-Functional Requirements for Quantum Programs

Lorenzo Saraiva[1], Edward Hermann Haeusler[1], Vaston Costa[2] and Marcos Kalinowski[1]

[1]Pontificia Universidade Católica do Rio de Janeiro.
R. Marquês de São Vicente, 225 - Gávea, Rio de Janeiro - RJ, Brazil
[2]Universidade Federal de Catalão.
Av. Dr. Lamartine Pinto de Avelar 1120. Catalão - GO, Brazil

Abstract

Quantum computing is moving from a purely theoretical area to an area with practical applications, allowing considerable performance efficiency improvements. The goal of this paper is to discuss non-functional requirements for quantum programs. Based on experiences developing quantum software for real quantum hardware we analyze hardware-related constraints and derive a set of generic non-functional requirements for this type of program. We identified a set of five performance efficiency and reliability related non-functional requirements that should considered when implementing a quantum program for a quantum device. We also discuss available solution options to address the requirements. There are high level solutions to deal with the hardware-related constraints described in our identified requirements. While many of the them are specific to quantum programming languages and technologies, the scientific community is engaging to integrate these kind of solutions into the quantum software engineering life cycle in an agnostic way regarding quantum programming languages and technologies.

Keywords

Quantum software engineering, Non-functional requirements

1. Introduction

With the developments in the last decade, quantum computing is going from a purely theoretical area to an area with practical applications, achieving considerable performance efficiency improvements compared to classical computing. As it becomes more widespread and accessible, the demand for writing quantum software in a controlled, industrial manner will increase accordingly [1]. Quantum Software Engineering (QSE) arises as an area that is aiming to bring the principles and heuristics of software engineering to the quantum computing context.

Classical software development can be divided into phases, and together these phases compose

2nd Quantum Software Engineering and Technology Workshop, Oct 18–22, 2021, Virtual Conference
lorenzo.saraiva@hotmail.com (L. Saraiva); hermann@inf.puc-rio.br (E. H. Haeusler); vaston@ufcat.edu.br (V. Costa); kalinowski@inf.puc-rio.br (M. Kalinowski)
http://lattes.cnpq.br/1614733376309760 (L. Saraiva);
http://www.inf.puc-rio.br/blog/professor/@edward-hermann-haeusler (E. H. Haeusler);
http://lattes.cnpq.br/5192533875584788 (V. Costa); http://www.inf.puc-rio.br/~kalinowski (M. Kalinowski)
0000-0002-0103-9113 (L. Saraiva); 0000-0002-4999-7476 (E. H. Haeusler); 0000-0002-4488-7518 (V. Costa); 0000-0003-1445-3425 (M. Kalinowski)

what is called the software life cycle[2]. Over the years, many life cycle process models have been designed, such as waterfall, evolutionary, and spiral [3]. Also, aiming at improving production speed and stakeholders' collaboration, agile development approaches have gained popularity [4].

In the quantum context, there also have been different attempts to create models that accurately describe the development life cycle. Zhao[5] describes a QSE life cycle model that can be used as a reference. He divides the life cycle of a quantum program into requirement analysis, design, implementation, testing, and maintenance, and reviews the main articles published on each of the defined phases. The amount of research dedicated to each life cycle phase varies greatly: while the implementation and test phases have several academic articles written about them, there is not a single published article about quantum software requirement analysis yet [6][5], which was the initial motivation for this research.

In the requirements area, the term Non-Functional Requirement (NFR) has been used to refer to concerns not related to the software's functionality. Different authors characterize this concept in informal and unequal definitions. The definition we use in the context of this paper is the following: *non-functional requirements describe the non-behavioural aspects of a system, capturing the properties and constraints under which a system must operate* [2]. Non-functional requirements are typically documented textually either quantified or non-quantified [7]. The quantification depends on the type of non-functional requirement. For instance, performance is rather documented quantitatively while maintainability is rather documented non-quantitatively [7].

Quantum algorithms are often described in terms that facilitate proving correctness or deriving asymptotic complexity estimates, without considering requirements related to the specific computing device on which to execute them [8]. Still, the fact is that quantum computers have a series of properties that can be safely ignored if not specified and using simulators, but have to be accounted for if the plan is to run a quantum program on a real device. These limitations have their roots mainly in quantum physics interactions and properties beyond the scope of this article, but their effects and the limitations they create when writing quantum algorithms are observable and measurable.

2. Quantum Computation Fundamentals

In the realm of quantum computing, we have two ways of constructing (quantum) programs, which will influence requirements analysis and are therefore briefly described in this section.

Although equivalent to each other, these two ways use slightly different computational models and manipulate different types of data. These data types are related to pure-state quantum algorithms and mixed-state quantum algorithms. Both can describe the current states of any quantum system, and the models can use circuits to represent the computation on the data.

A pure quantum state is represented by a unit vector in a Hilbert space $\mathcal{H} = \mathcal{H}_2^n$, where $\mathcal{H}_2$ describes the Hilbert space of a 2-state particle, also known as a **qubit** and, $\mathcal{H}_2^n$ is the n-times tensor product of this 2-state Hilbert space. Using Dirac's Bras and Kets notation, the observable states of $\mathcal{H}_2$ are $|0\rangle$ and $|1\rangle$ and a pure state one particle state is $|\alpha\rangle = c_0 |0\rangle + c_1 |1\rangle$, with $c_i \in \mathbb{C}$ and $||c_0||^2 + ||c_1||^2 = 1$, i.e., a linear combination of the states $|0\rangle$ and $|1\rangle$ with probability 1.

A general element of $\mathcal{H}_2^n$, i.e., a n-qubits pure state, is $\alpha = \sum_{i=0}^{2^n-1} c_i |i\rangle$. Quantum circuits can represent any physical quantum system that consist of n quantum two-state particles, for some n, named the canonical way of expressing quantum computing, where the gates are unitary linear operators. There are sets of universal unitary gates. The Tofolli and Hadamard gates form a universal set since one can describe any unitary operator by a circuit containing only occurrences of these gates.

A mixed quantum state is a mixture of pure states. It is represented by $\{\Psi\} = \{p_i |\Psi_i\rangle\}$, where $|\Psi_i\rangle$ are pure states and the p_i's form a probability distribution. That is, one can represent the corresponding quantum system in terms of the pure state $|\Psi_i\rangle$ with probability p_i. This representation is not unique. Different mixtures can describe the same physical system. The use of density matrices has some advantages and, it is equivalent.

Hence, we have Pure-state Quantum Computation (PsQC) and Mixed-state Quantum Computation (MsQC), and circuits can represent both. Additionally, the former can have a high-level representation in a Quantum Programming Language. While PsQC is gate based as unitary linear operators, MsQC is gate base by density matrices. A density matrix can represent any linear operator in a Hilbert space, which is a great advantage, as there is no need to orthonormalize density matrices at each step.

The representation of linear operators into density matrices is not injective, but it is surjective. All density matrix describes the mixture of its eigenvectors, with the probabilities being the corresponding eigenvalues. Note that diagonal density matrices correspond to probability distributions over classical states. Hence, density matrices are the linear operators that will describe quantum computing when dealing with data represented as mixed states.

3. Non-Functional Requirements in Quantum Software

As a first step to discuss NFRs for quantum computing, we analyzed the possibility of adapting an NFR model for embedded systems. Embedded systems have characteristics related to quantum computing in the sense that the software is designed specifically for the exact hardware it will be executed on. The Pecos Model [9] is a component based NFR model for embedded systems. It is possible to draw parallels between the Pecos Model and hybrid classical-quantum systems, where a classical environment interacts with a quantum device to perform specific computations in a black-box manner. This model is composed of components, data ports and connectors. In a hybrid classical-quantum system, the quantum processor unit and the classical processors represent the components and the layers in-between represent the connectors. However, after our analysis, we concluded that adapting the Pecos model to the quantum context would require significant effort, and that the parallels are not sufficient to justify doing so.

Indeed, in the quantum context, the two ways of constructing quantum programs, PsQC or MsQC, bring some relevant differences into the requirement analysis. The MsQC model allows measurements in/at intermediate parts of the quantum circuit. At the same time, the PsQC does not enable any measurements at any point of the circuit but the terminal nodes. While this difference affects functional requirements, it seems to be intrinsically linked to a characteristic of the program in terms of its algorithms design (if in circuit shape or Quantum Programming Language form), which makes a substantial difference also on non-functional requirements.

States superposition and entanglement are the most cited and used features that show advantages of quantum computation over classical computation. Grover quantum search, for example, shows up quadratic speed up over classical search. The superposition of states is mandatory to get Grover speed up over classical computing. The PsQC is the natural choice for implementing Grover unstructured quantum search. The use of MsQC has to take into account the entropy of the initial, of the mixed initial state. The literature has reported examples that depending on the entropy of initial (mixed) state, Grover quantum algorithm performs as bad as classical unstructured search, see [10] and [11]. They point out to the need of using entanglement to out-perform classical computation when implementing Grover in MsQC. We have also to mention the need of entanglement to have more accuracy in the generalization of Grover algorithm to more than one item existing in the quantum database [11]. Thus, we have to include hardware's ability to perform entanglement among the non-functional requirements.

Another difference is that, in classical computation, the programming language is chosen for an implementation mainly for its adequacy to whatever is being programmed. The most used languages all have their known strengths and weaknesses, and there is a relative consensus on which language should be used for what. Abstraction power suitability, efficiency, and many other pragmatic features of the programming languages are mixed with market directions, technologies preferences and availability considerations should be taken into account. This is even a bit fuzzier on quantum computing. The few existing companies still compete for the spotlight, advertising their own quantum programming language or development technology as the best choice for the problem you have in hand. This choice is hard, indeed.

Aware of these differences, in this paper, as a first step, we propose a set of NFRs that reflect quantum computing specific hardware-related constraints. These NFRs were identified based on experiences implementing quantum algorithms in a set of academic projects. We classify each NFR according to the ISO 25010 quality model.

The first NFR we cover is relatively simple, but it is worthy of mention since there is usually not a need for it to be considered in the classical context. The biggest quantum computer known at the moment of the writing of this article has 72 qubits - reflecting the state-of-the-art. We can imagine that when quantum processors become integrated into classical computers, the small number of available qubits will be a real issue when considering the price factor - more qubits will mean a more expensive device. If someone wants to reach the highest number of devices in the early stages of quantum spreading, they will have to cater to this limitation by writing programs that use a number of qubits within that limit. Thus, an essential generic non-functional requirement that should be considered for the quantum computing context is: **NFR1 - the program should use a maximum of n qubits, where n is the number of qubits available in the target quantum device.** In terms of its classification in the ISO 25010 quality model, it classifies within the *performance efficiency* product quality characteristic. More specifically, being related to its *resource utilization* subcharacteristic. It is noteworthy that this requirement is a hard cutoff — if the computer doesn't have at least the necessary number of qubits, you will not be able to run the quantum program.

Another aspect of a quantum circuit is its depth. The depth is the maximum length of any (directed) path from any input wire to any output wire[12]. Quantum devices need to maintain quantum states stable throughout the whole process of the program execution. If a quantum circuit is too deep, the device might not be able to maintain the necessary quantum state for

a sufficient time frame, thus introducing errors caused by decoherence. Those errors become an even more significant problem because it's not trivial to pinpoint how deep a circuit can be in a specific quantum device. The results might vary even in devices with the same amount of qubits. Another issue is that even the mere act of detecting errors in quantum programs is much more complex than on classical ones[13], and so is correcting them. Thus, another important generic non-functional requirement is: **NFR2 - the program should be designed considering the maximum circuit depth so that the target device can maintain a stable quantum state for the necessary period to execute the algorithm**. If your circuit is too deep, it will gradually decohere and eventually collapse into a classical state, losing quantum behaviour. Regarding its classification in the ISO 25010 quality model, it affects the *reliability* product quality characteristic, due to potential errors produced by decoherence. Unfortunately there is currently no subcharacteristic this requirement could mapped against in the ISO 25010. Nevertheless, it should be related to other non-functional requirements concerning the reliability *fault tolerance* subcharacteristic, e.g., specifying the need for related error correction protocols.

When talking about the complexity of a classical algorithm, we use the asymptotic complexity and big O notation[14]. In quantum algorithms, on the other hand, the complexity measure usually used is the number of T gates present in the algorithm[15]. This measure has a similar origin as the circuit depth issues, since T gates are the ones with greater implementation cost. Therefore, increasing the number of T gates used also increases the overall energy necessary to keep the quantum state stable, potentially inducing decoherence. Hence, another important generic requirement is: **NFR3 - the program should be designed considering the number of T gates so that it does not exceed the limit of the target device**. This requirements could also be classified within the *reliability* product quality characteristic. As for NFR2, ignoring it will increase the chance of decoherence and, consequently, errors. Thus, *fault tolerance* mechanisms should be employed considering such chances.

A problem often overlooked in algorithm design and implementation will arise when dealing with real quantum devices: not all qubits are connected to each other. Figure 1 depicts the connectivity map of a series of IBM quantum computers[1]. One can note that the number of connections between qubits can be quite far from the maximum of $n(n-1)/2$ possible connections. Since it is only possible to create two qubit gates between qubits that are connected, this can be quite limiting, especially when working with quantum algorithms that use highly entangled states with several qubits[16]. Thus, an additional generic non-functional requirement that arises is: **NFR4 - the program should be implemented minimizing the number of gates between qubits that are not physically connected on the target device.**

In the Figure 1, using the Yorktown device as a reference, if the programmer wants to apply a two qubit gate between qubits 0 and 3, such as a CNOT gate or SWAP gate, it would be necessary to apply first a SWAP gate between qubit 2 and either 0 or 3, creating a connection between 0 and 3. While this works in simple cases, this will still be a limiting factor when writing more complex algorithms. *Transpilers* [17] can be used to circumvent this problem, which will transform a generically programmed quantum circuit into an adapted quantum circuit for the specific device. The drawback is that this is an automated process, and it tends to create quantum circuits that are bigger than the optimal, which may imply in time behaviour

[1] https://www.flickr.com/photos/ibm_research_zurich/albums/72157663611181258/

efficiency loss. Hence, this requirement can be related to the ISO 25010 *performance efficiency* and *reliability* characteristics. Not minimizing the number of gates between qubits that are not physically connected and using transpilers will increase circuit size, increasing resource utilization and chances of errors due to decoherence.

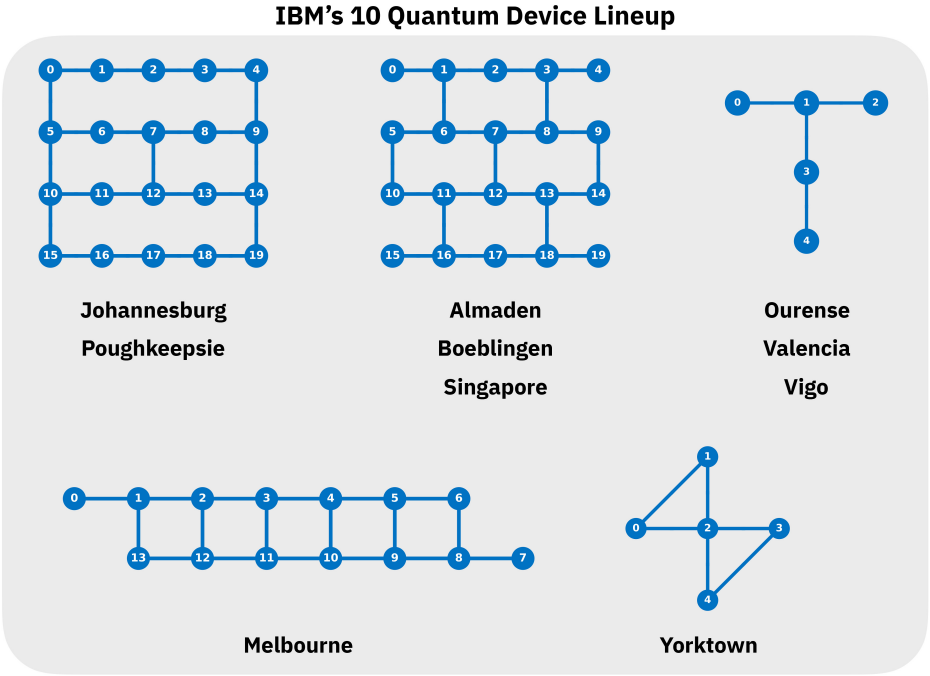

Figure 1: Qubit connectivity map of IBM quantum computers.

Physically constructing quantum gates can be a daunting task. Because of that, real quantum devices usually only implement a few quantum gates that compose a universal set. Thus, we consider the following generic non-functional requirement: **NFR5 - the program should be implemented minimizing the use of gates that are not available in the target quantum device.** In a similar manner as in the qubit connectivity mentioned in NFR4, a transpiler can be used to transform the original quantum circuit with high-level multi-qubit gates to a circuit composed exclusively of the available quantum gates, increasing the circuit depth and possibly creating decoherence. Therefore, this requirement can also be related to the ISO 25010 *performance efficiency* and *reliability* characteristics.

4. Discussion on how to address the non-functional requirements

The restrictions listed are admittedly low-level, as are the NFRs they create. Dealing directly with them would be the same as accessing individual registers on a classic computer, which is not something usually done by a classical developer[18]. Even though this could be desirable in an eventual scenario of extreme need for optimization for a specific and probably limited quantum

device, this approach is not practical when considering an average quantum programmer who, for the most part, is not necessarily developing for any specific device. Naturally, when designing a quantum algorithm, lowering qubit count, circuit depth, and gate count should always be one of the objectives. Still, developers are typically used to do that as the general intention, not considering the specific limits of quantum hardware. And later, when the algorithm is finished, knowing which quantum device is sufficient or ideal to run the program is not trivial. For this reason, there is considerable research about high-level approaches to choosing the correct device, optimizing depth and adapting the circuit.

Quantum computers with qubits numbers in the 50-100 range are already a reality. So, instead of making quantum algorithms with theoretical speedups but an unrealistically high number of qubits and circuit depth, some researchers are focusing on these near-term quantum machines, called Noisy Intermediate-Scale Quantum devices (NISQs). As the name goes, these devices have a non-negligible amount of noise, relatively short decoherence times, and qubits in the two digits range. Still, they have a pretty significant advantage: *they already exist.* For that reason, close term quantum development will be focused in hybrid quantum-classical software, where an application sends a specific task to an integrated quantum processor and receives an output. The life cycle model as described by Zhao[5] is quite abstract and is more fitting to be used as a guide for research in the area of QSE than as a manual for writing software for NISQs. Weder *et al.*[19] proposed a life cycle model with a more practical approach, tackling the difficulties a programmer face when trying to program quantum software, which includes dealing with constraints related to our identified NFRs. The model has ten phases, but the ones that are closely related to the requirements of this article are:

- Quantum Hardware Selection
- Readout-Error Mitigation Preparation
- Compilation and Hardware-dependent Optimization

The authors also introduce the concept of *quantum provenance* [20] as the relevant data that should be collected from a quantum computer and further analyzed to effectively select a suitable quantum computer for the execution of a circuit or to assist in the process of quantum compilation and optimization.

The **Quantum Hardware Selection** phase comprises an analysis of the quantum circuit and further selection of suitable hardware. This expresses well the intention behind this phase, but the exact way you select the quantum hardware is not defined. Salm *et al.*[21] have made an initial draft on how to automate the selection of quantum devices, specifically for current day NISQs. Given a chosen algorithm and a chosen input range, they describe the steps to generate a first-order logic statement that will accurately define the requirements to run that algorithm with that specific input range. The circuit representation of this algorithm is then transpiled for possible hardware, and from the transpiled circuit, you can gather depth and qubit numbers and check if the device is a good choice.

Suchara *et al.* [22] have developed related work but without the emphasis on current quantum computing, instead of having a more general approach. On the other hand, their framework, the QuRE toolbox, is a bit more robust. It takes extra factors such as different potential error correction algorithms to estimate qubit number, gate number, and execution time accurately.

These data could theoretically guide the selection of quantum devices and error correction protocol. Since the focus of QuRE is to quickly compare the properties of large quantum algorithms [22] that are still beyond our reality, it remains a primarily theoretical framework.

Quantum Volume (QV) is a metric designed to represent the largest random circuit of equal width and depth that a quantum computer successfully implements [17]. QV is at a higher abstraction level when compared to the NFRs defined in this paper, since it abstracts some of those aspects in a single number metric. Another metric, devised by Sete *et al.* [23], for classifying quantum computers with a single number is the Total Quantum Factor (TQF). Both these metrics mainly consider constraints related to the described requirements - number of qubits, decoherence times and qubit connectivity, combined with other particular aspects (e.g., possible parallel operations for QV and longest gate execution time for TQF).

Regarding **Readout-Error Mitigation Preparation**, a noiseless quantum computer is still far from our reality. NISQs devices have a non-negligible amount of noise and relatively short decoherence times [24]. A quantum computer can have different types of errors. One stems from the fact that gates aren't always perfectly executed down to the quantum level, resulting in slight deviations. Several error-correction codes are attacking this issue [25][16]. Still, these techniques are based on redundancy and use extra qubits, called *ancilla*, so this increase in qubit number has to be taken into account when applying one. Another type of error are *readout-errors* that occur when the measurement is disturbed by some noise. There are readout-errors protocols to reduce the influence of these errors [19]. During the readout-error mitigation preparation phase, one has to choose an error mitigation approach, which are based on the *unfolding problem*, where the goal is to estimate the algorithm behaviour assuming a variable could always be measured exactly and then comparing actual execution results with the estimated ones [26]. Since the chosen error-correction model will have an impact on fidelity and possibly on the number of qubits used, one may store it as part of quantum provenance data [19].

Finally, the **Compilation and Hardware-dependent Optimization** phase is composed of optimizations based on hardware characteristics and compilation to machine instructions. This process is built into the different hardware-specific compilers available, but there are hardware-independent compilers and transpilers. A compiler for a classical language consists of a sequence of steps used to transform the source code into a suitable representation. Similarly, quantum compilers are the ones responsible for transforming a program written in a high-level quantum language, such as Q# or Cirq, into machine instructions that can be executed by a quantum computer, considering the listed characteristics of the selected hardware.

The first phase of the compilation goes from the high-level representation to a Quantum Intermediate Representation (QIR) based on the quantum circuit model, which may represent other forms of quantum computing besides circuit based, such as adiabatic computation[27]. Developing a widely adopted QIR is still being attempted by several researchers[28][29], with even Microsoft recently releasing their own QIR. Still, the most relevant quantum computing companies use a specific one for their language, such as OpenQASM[30] for IBM's Qiskit. The second phase of the compilation consists of going from the QIR to the low-level language representing the machine instructions of the chosen hardware, such as QASM. In this step, we can use any of the hardware-specific compilers like IBM or Rigetti[19] and be limited to their hardware, or try to use a more generic compiler. Here is where many hardware-dependent

optimisations occur, considering the constraints listed in the requirements. Of the non-hardware-dependent compilers, one that stands out is the |ket⟩[8]. This hardware-independent compiler focuses on NISQs devices and aims to maximise the overall fidelity of the computation when executing algorithms that will be affected by noise. The t|ket⟩ achieved two remarkable feats. The first is the fact that it is both languages agnostic and retargetable. It accepts several different languages like Qiskit, QSAM, Quil, Quipper, ProjectQ and Cirq, covering a good part of the widely used quantum programming languages. It also can be deployed on different hardware, like Rigetti, Honeywell and Google. In a field where there still exists a feeling of "every man for himself", and competing companies are constantly trying to outdo each other, having a compiler with these characteristics is an advance towards a unification that will likely be beneficial for the average quantum programmer. The second feat are the results: t|ket⟩ offers significant improvement in terms of gate count and circuit depth over other compilers when evaluated on realistic quantum circuits and real quantum devices [8].

Transpilers and compilers are relatively similar: they take source code and transform it into a version in a different representation. The main difference is that the compiler converts the code into a lower-level abstraction while the transpiler converts it into another representation with the same level of abstraction, be it in the same or a different language. In the quantum context, transpilation is the process of rewriting a given input circuit to match the topology of a specific quantum device and/or to optimize the circuit for execution on present-day noisy quantum systems [31]. Recently, several transpilers have been developed to increase accuracy on NISQs, and since they're less work-intensive than developing a compiler, smaller teams can do valuable research. There is a promising result in a just-in-time transpiler [32], tested on IBM quantum computers. The qubits on each quantum computer are periodically calibrated and tested against reference values to check their fidelity levels, and this feedback is open to the public. The just-in-time transpiler uses this data in real time to remap the circuit and avoid using low fidelity qubits and had good results, showing that there is room for improvement on calibrations and that they should be performed as much as possible. This transpiler focused on optimization. On the other hand, we have works such as the SABRE [33]. The SABRE is an algorithm designed to tackle the qubit mapping problem in NISQs. This problem is known to be NP-complete [34], and mathematical solutions can get prohibitively time consuming as the number of qubits increases. The SABRE combines a series of heuristics to avoid falling into the drawbacks of previous works and ensure flexibility, scalability, controllability, and high-quality initial mapping, with results up to exponential speedup on various benchmarks [33].

A promising work that is going in the direction of unifying quantum computing - taking off the developer the hassle of navigating all these different quantum programming languages, frameworks and SDKs - is Q|Path⟩. The Q|Path⟩ is an all encompassing platform for quantum applications life cycle management and development for quality quantum software. From the creation of the quantum algorithm through its development, testing and implementation, to its deployment and reuse[35]. It provides a wide array of tools to develop quantum software and it also supports the execution in actual quantum devices in a transparent way regardless of the platform where they are executed. The platform aims to abstract the process of dealing with hardware-related constraints, specific framework issues, quantum data collection and even connection between different quantum platforms in order to democratize the access to quantum computing and allow the programmer to fully focus at solving the problem in hand.

It does so by providing tools with similar objectives as the ones we presented for the phases of the quantum life cycle by Weder *et al.* [19], such as automating the process of choosing the adequate device do execute a quantum program, collecting relevant data from the execution, connecting classical and quantum parts to create hybrid programs and managing the hybrid quantum-classical software production.

5. Conclusion and further considerations

We have reviewed a few hardware-related constraints that exist when executing quantum programs in real quantum devices, from which we derived a set of NFRs for such programs. These NFRs are exceedingly low level to be treated directly by a regular programmer.

Indeed, there is ongoing work on high level solutions to deal with hardware-related constraints in a more automated manner, aiming near term devices instead of theoretical quantum computers. These solutions are an integral part of quantum programs development - thinking about individual connections of qubits shouldn't be the norm. Therefore, these solutions should be included in the QSE life cycle, and not only as a footnote in the classical implementation phase. The model proposed by Weder *et al.*[19] is a great step in the direction of describing the process of programming quantum software for NISQs, incorporating the described constraints in the life cycle, releasing the burden of addressing the low level NFRs from the developer.

Another relevant point to be made is that the current state of quantum computing development, with competing companies developing overlapping products with the objective of becoming the main player is a problem to be addressed, and tends to make things harder for the average programmer. For that reason, projects such as t|ket⟩ and Q|Path⟩ are extremely important, with the potential to take a good amount of hassle off the hands of the programmer, favouring the overall development of quantum algorithms.

References

[1] M. Piattini, G. Peterssen, R. Pérez-Castillo, J. L. Hevia, M. A. Serrano, G. Hernández, I. G. R. de Guzmán, C. A. Paradela, M. Polo, E. Murina, et al., The talavera manifesto for quantum software engineering and programming., in: QANSWER, 2020, pp. 1–5.

[2] L. Chung, J. C. S. do Prado Leite, On non-functional requirements in software engineering, in: Conceptual modeling: Foundations and applications, Springer, 2009, pp. 363–379.

[3] N. M. A. Munassar, A. Govardhan, A comparison between five models of software engineering, IJCSI 7 (2010) 94.

[4] M. Kuhrmann, P. Tell, R. Hebig, J. A.-C. Klunder, J. Munch, O. Linssen, D. Pfahl, M. Felderer, C. Prause, S. Macdonell, J. Nakatumba-Nabende, D. Raffo, S. Beecham, E. Tuzun, G. Lopez, N. Paez, D. Fontdevila, S. Licorish, S. Kupper, G. Ruhe, E. Knauss, O. Ozcan-Top, P. Clarke, F. H. Mc Caffery, M. Genero, A. Vizcaino, M. Piattini, M. Kalinowski, T. Conte, R. Prikladnicki, S. Krusche, A. Coskuncay, E. Scott, F. Calefato, S. Pimonova, R.-H. Pfeiffer, U. Pagh Schultz, R. Heldal, M. Fazal-Baqaie, C. Anslow, M. Nayebi, K. Schneider, S. Sauer, D. Winkler, S. Biffl, C. Bastarrica, I. Richardson, What makes

agile software development agile, IEEE Transactions on Software Engineering (2021) 1–1. doi:10.1109/TSE.2021.3099532.
[5] J. Zhao, Quantum software engineering: Landscapes and horizons, arXiv:2007.07047 (2020).
[6] P. E. Z. Junior, V. V. de Camargo, A systematic mapping on quantum software development in the context of software engineering, arXiv:2106.00926 (2021).
[7] S. Wagner, D. M. Fernández, M. Felderer, A. Vetrò, M. Kalinowski, R. Wieringa, D. Pfahl, T. Conte, M.-T. Christiansson, D. Greer, C. Lassenius, T. Männistö, M. Nayebi, M. Oivo, B. Penzenstadler, R. Prikladnicki, G. Ruhe, A. Schekelmann, S. Sen, R. Spínola, A. Tuzcu, J. L. D. L. Vara, D. Winkler, Status quo in requirements engineering: A theory and a global family of surveys, ACM Trans. Softw. Eng. Methodol. 28 (2019). URL: https://doi.org/10.1145/3306607. doi:10.1145/3306607.
[8] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, R. Duncan, t| ket>: a retargetable compiler for nisq devices, Quantum Science and Technology 6 (2020) 014003.
[9] R. Wuyts, S. Ducasse, Non-functional requirements in a component model for embedded systems, in: International workshop on specification and verification of component-based systems, OOPSLA, 2001.
[10] S. Bose, L. Rallan, V. Vedral, Communication capacity of quantum computation, Phys. Rev. Lett. 85 (2000) 5448–5451.
[11] E. Biham, D. Kenigsberg, Grover's quantum search algorithm for an arbitrary initial mixed state, Phys. Rev. A 66 (2002) 062301.
[12] A. C.-C. Yao, Quantum circuit complexity, in: Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science, IEEE, 1993, pp. 352–361.
[13] N. M. Linke, M. Gutierrez, K. A. Landsman, C. Figgatt, S. Debnath, K. R. Brown, C. Monroe, Fault-tolerant quantum error detection, Science advances 3 (2017) e1701074.
[14] I. Chivers, J. Sleightholme, An introduction to algorithms and the big o notation, in: Introduction to programming with Fortran, Springer, 2015, pp. 359–364.
[15] A. Paler, I. Polian, K. Nemoto, S. J. Devitt, Fault-tolerant, high-level quantum circuits: form, compilation and description, Quantum Science and Technology 2 (2017) 025003.
[16] D. A. Lidar, T. A. Brun, Quantum error correction, Cambridge university press, 2013.
[17] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, J. M. Gambetta, Validating quantum computers using randomized model circuits, Physical Review A 100 (2019) 032328.
[18] R. Pérez-Castillo, M. A. Serrano, M. Piattini, Software modernization to embrace quantum technology, Advances in Engineering Software 151 (2021) 102933.
[19] B. Weder, J. Barzen, F. Leymann, M. Salm, D. Vietz, The quantum software lifecycle, in: Proc. ACM Int. Work. on Arch. and Paradigms for Eng. Quantum Soft., 2020, pp. 2–9.
[20] B. Weder, J. Barzen, F. Leymann, M. Salm, K. Wild, Qprov: A provenance system for quantum computing, IET Quantum Communication (2021).
[21] M. Salm, J. Barzen, U. Breitenbücher, F. Leymann, B. Weder, K. Wild, The nisq analyzer: Automating the selection of quantum computers for quantum algorithms, in: Symposium and Summer School on Service-Oriented Computing, Springer, 2020, pp. 66–85.
[22] M. Suchara, J. Kubiatowicz, A. Faruque, F. T. Chong, C.-Y. Lai, G. Paz, Qure: The quantum resource estimator toolbox, in: IEEE 31st Int. Conf. on Comp. Design, 2013, pp. 419–426.
[23] E. A. Sete, W. J. Zeng, C. T. Rigetti, A functional architecture for scalable quantum

computing, in: 2016 IEEE Int. Conf. on Rebooting Computing (ICRC), 2016, pp. 1–6.
[24] J. Preskill, Quantum computing in the nisq era and beyond, Quantum 2 (2018) 79.
[25] M. D. Reed, L. DiCarlo, S. E. Nigg, L. Sun, L. Frunzio, S. M. Girvin, R. J. Schoelkopf, Realization of three-qubit quantum error correction with superconducting circuits, Nature 482 (2012) 382–385.
[26] L. Brenner, R. Balasubramanian, C. Burgard, W. Verkerke, G. Cowan, P. Verschuuren, V. Croft, Comparison of unfolding methods using roofitunfold, International Journal of Modern Physics A 35 (2020) 2050145.
[27] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, I. L. Markov, A layered software architecture for quantum computing design tools, Computer 39 (2006) 74–83.
[28] D. Ittah, T. Häner, V. Kliuchnikov, T. Hoefler, Enabling dataflow optimization for quantum programs, arXiv:2101.11030 (2021).
[29] K. Hietala, R. Rand, S.-H. Hung, X. Wu, M. Hicks, Verified optimization in a quantum intermediate representation, arXiv:1904.06319 (2019).
[30] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta, Open quantum assembly language, arXiv:1707.03429 (2017).
[31] A. Cross, The IBM q experience and QISKit open-source quantum computing software, in: APS March Meeting Abstracts, volume 2018, 2018, pp. L58–003.
[32] E. Wilson, S. Singh, F. Mueller, Just-in-time quantum circuit transpilation reduces noise, in: IEEE ICQCE), 2020, pp. 345–355.
[33] G. Li, Y. Ding, Y. Xie, Tackling the qubit mapping problem for nisq-era quantum devices, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 1001–1014.
[34] M. Y. Siraichi, V. F. d. Santos, S. Collange, F. M. Q. Pereira, Qubit allocation, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, 2018, pp. 113–125.
[35] G. Peterssen, Advantages of agnostic development of quantum algorithms and apps for the real world with qpath, 2021. URL: https://www.quantumpath.es/2021/02/25/advantages-of-agnostic-development-of-quantum-algorithms-and-apps-for-the-real-world-with-qpath/.