Quantum Software Testing – Current trends & Emerging proposals

Antonio García de la Barrera[1], Ignacio García-Rodríguez de Guzmán[1], Macario Polo[2], José A. Cruz-Lemus[2]

{Antonio.GAmo, Ignacio.GRodriguez, Macario.Polo, JoseAntonio.Cruz}@uclm.es

[1]Department of Technologies and Information Systems, Esceula Superior de Informática, c/Paseo de la Universidad, 13071, University of Castilla-La Mancha, Ciudad Real, Spain.

[2]Institute of Technologies and Information Systems, c/Camino de Moledores s/n, 13071, University of Castilla-La Mancha, Ciudad Real, Spain

Abstract Quantum technology is rapidly improving the capacity of quantum computers, increasing the numbers of qubits, and fostering the development of quantum programs capable of solving time problems which, up to now, have been out of the reach of the most powerful classical computers. Unfortunately, quantum computational capacity is growing faster than the quantum software engineering knowledge required to avoid a new (quantum) software crisis. Since quantum software engineering is an open research issue, in this paper we present firstly a detailed view of the current state of the art identifying the following trends in testing techniques: i) general proposals, ii) statistic approaches based on repeated measurements, iii) the use of Hoare-like logics to reason about software correctness, and iv) a relevant line of research based on reversible circuit testing (partially applicable to quantum software unitary testing). On the other hand and considering the lack of techniques and tools based on quantum software engineering, we propose to use quantum software mutation. Based on several categories of error that have been identified as likely among quantum developers, we have developed a set of quantum mutant operators to improve quantum test suites and to reduce the number of quantum test cases together with a prototype to automate de generation of quantum mutants' circuits from an original one.

Introduction

In 1982, Nobel Laureate Richard Feynman asked: "What kind of computer are we going to use to simulate physics?", thereby inaugurating the "second quantum

revolution". In fact, from this point the very idea for a quantum computer was born, and quantum computer science began in earnest. Over the last three decades our understanding of "quantum computers" has expanded drastically, as the efforts to make real such an "exotic" computer have made steady yet remarkable progress [1]. Using various "counterintuitive" principles such as superposition and entanglement, quantum computers now yield faster computing speeds, providing high value in many different and important applications. In fact, there are thousands of very interesting applications for this new paradigm, covering several areas[2]: economics and financial services, chemistry, medicine and health, supply chain logistics, energy, agriculture, etc.

The prospects for quantum computing are indeed exciting, and extraordinary expectations are now fuelling a global effort to perfect quantum computing [3]. The most important companies (Google, IBM, Microsoft, Intel, Atos, Alibaba, etc.) are investigating how to take the most advantage of this new technology in their businesses. Also, many countries (China, U.S., Japan, Russia, U.K., etc.) are investing huge quantities of money in quantum technology. The involvement of governments is of great importance, as has been evidenced by the introduction of the National Quantum Initiative Act in the United States, the funding of the Institute for Quantum Computing by the Canadian government, or the European Union's "*Quantum Manifesto and Quantum Technologies Flagship*" initiative.

A variety of quantum computers is already available, such as IBM Q, IonQ, Rigetti, D-Wave, Microsoft Quantum, and Google Quantum. Tens of quantum programming languages exist (e.g., qGCL, Q Language, QML, Quipper, OpenQASM, Qiskit, Q#) [4], as well as software development kits (e.g.,Forest, Qiskit, Cirq, QDK, Orquestra) [4]. A comprehensive review of quantum computing literature, and a detailed overview of quantum software tools and technologies, plus quantum computer hardware development, can be found in [5].

As the Quantum Software Manifesto[1] states: "*Given the recent rapid advances in quantum hardware, it is urgent that we step up our efforts in quantum software*", stressing the importance of quantum software. It is necessary to go a step further and raise awareness of the need for Quantum Software Engineering (QSE) that can enable us to produce quantum software with the necessary quality and productivity [6].

The main current design for quantum computers is the so-called "gate-based quantum computing," very similar to today's classical approaches, which consist in dividing an algorithm into a sequence of a few very basic "primitive operations" or gates. In this kind of quantum computer, one of the tools most used for creating quantum programs is the quantum circuit. In fact, there exist several quantum circuit simulators (Quirk[2], QCEngine[3], etc.), and some quantum vendors such as IBM use

[1] https://www.qusoft.org/quantum-software-manifesto/
[2] https://algassert.com/quirk
[3] http://machinelevel.com/qc/

the circuit as the main element for Qiskit programming[4]. Quantum circuits are a very good artefact with which to design quantum programs, since the transformation from quantum circuits to quantum code is quite direct, and furthermore entails a more agnostic representation of a quantum algorithm (viewed from the point of view of programming language).

So, given the current state of quantum software development, the promising present and future of quantum computing, and the challenges posed by the Quantum Software Manifesto, we would like to focus in this chapter on the strengthening of Quantum Software Engineering. Therefore, and with the aim of contributing to the development of the quantum software testing process (focusing on quantum circuit level), (i) we firstly present a review of the state of the art of the current proposals about quantum software testing, followed with (ii) the proposal of the adaptation of the classical software mutation testing technique to the quantum software testing context.

The chapter is organized as follows: second section summarizes the review of the state of the art of the last years regarding quantum software testing; third section presents the adaptation of the mutation technique to quantum software; section forth presents a prototype to automate the apply the technique, and finally, section fifth outline some conclusions of our work.

Current trends on Quantum Software Testing

This section presents a general view of the current state of the art of quantum software testing, classified in (i) overviews, (ii) frameworks, (iii) probabilistic testing and verification, (iv) Hoare logic applications, and (v) reversible circuits testing.

Overview proposals

Some publications present overviews, challenges, and predictions about the current state of the art and the future of software engineering concerning quantum computing. They offer different perspectives about testing and verification, from state-of-the-art surveys to bug taxonomies.

In [7], Zhao presents a comprehensive literature review on quantum software engineering. The term "quantum software engineering" is defined in the paper as "the use of sound engineering principles for the development, operation, and maintenance of quantum software and the associated document to obtain economically quantum software that is reliable and works efficiently on quantum computers.". A quantum software cascade-based lifecycle is also presented. Then, a

[4] https://qiskit.org/

quantum software engineering state-of-the-art survey is conducted, summarizing the available technology with a focus on analysis, design, implementation, testing, and maintenance.

In [8], Miranskyy et al. discuss current classic software debugging tactics, showing which ones can be directly adopted for quantum software testing. They also list novel techniques suited for the quantum-computer-specific debugging issues, such as superposition or entanglement verification, or the possibility of test probability distributions employing approximate copies of the algorithm's output.

Polo [9] introduces some challenges and ideas regarding the testing of quantum programs, providing a brief overview of the existing approaches to the subject. The work focuses on functional testing, discussing the place of the test suite in hybrid classic/quantum systems, white box testing (particularly mutation), and, finally, on the applicability of classic model-based testing in model representation of quantum circuits.

In [10], Huang and Martonosi survey a range of QC programs, performing debugging on small-scale simulations using different languages and technologies. Based on this experience, they conduct a comparative study assessing how the quantum environment can support testing and debugging. They also state that a quantum algorithm consists of three main conceptual parts: inputs, operations, and outputs, and they point out that bugs can result from a mistake in any of these stages. Finally, a bug taxonomy is presented, giving examples of each kind of bug together with how to prevent them.

In [11], Sodhi considers state-of-the-art Quantum Computing Platforms (QCPs) to identify all relevant characteristics from a software architecture perspective. By this means, the general architecture and typical programming model of a QCP are specified. Then, the significant characteristics of QCPs–from an architectural point of view–are listed and traced to the Quality Attributes, including testability, briefly evaluating the impact on these.

Finally, in [12], Miranskyy et al. discuss several use cases for quantum algorithms. Based on this, they address the use of quantum components as black-box artifacts in solution libraries and compare two approaches to the testing of quantum components: a unitary test perspective and a System of Systems (SoS) approach. As a result, they offer some tricks to analyze quantum programs during runtime.

Frameworks

Some of the selected publications present technological environments or methodologic-level approaches aiming to provide–or including–a frame or context for the testing activities.

In [13], Dey et al. addressed the need for systematic techniques for cost-effective quantum software development, remarking how the different behavior of quantum systems causes a barrier in the adoption of classic Software Development Life

Cycles (SDLCs). To this end, they propose a Quantum Development Life Cycle (QDLC) model based on classical waterfall models. For the testing stage, they propose a state reconstruction technique named quantum-state tomography. It is based on repeated preparation and measurement in which the preparation and measurement are repeated *22*n* times for an *n-qubit* system.

In [14], Campos and Souto establish the need for a benchmark to ease the reproducibility of quantum software engineering research. To this end, they propose the development of a framework named QBugs, which includes a catalog of quantum algorithms, a catalog of reproducible bugs, and supporting infrastructure to enable empirical and controlled experimenting. In addition, they plan to–with the use of GIT–automatically map each bug report labeled as "bug" or "issue" to the commits introducing and solving the bug, automatically adding the identified problem-solution tuple to the bug catalog.

In [15], Gomes et al. establish the need for pre-developed quantum software components to increase the community of developers and their effectiveness and efficiency. To this end, they propose the creation of quantum algorithm and data structure libraries, both for the development and testing of quantum programs.

In [16], Reutter and Vicary introduce a knot-based language to design and verify quantum algorithms. They offer a scheme for interpreting knot diagrams, called shaded tangles, as quantum programs, allowing to yield substantial new insight about how the program works along with a fully topological verification. Furthermore, it is observed that isotopic tangles yield equivalent programs.

Property-based testing is a structured method for automated testing using program specifications. In [17], Honarvar et al. introduce a property-based framework for quantum programs in Q#, concerning property specification, test case generation, and analysis of test results. The authors provide an overview of the framework's architecture and mode of use, a prototype, and some examples of its application results.

In [18], Steiger et al. introduce a framework, named ProjectQ, for quantum algorithm development. It transforms high-level domain-specific language (DSL) code to several low-level instruction sets, enabling developers to test quantum algorithms on efficient simulations or through the IBM Quantum Experience cloud service. For this purpose, it uses a Python-embedded DSL and a set of compilers. The framework includes tools for circuit drawing and resource estimation and allows extension mechanisms such as plug-ins.

Fuzz testing, colloquially known as “fuzzing,” is a set of software testing techniques implying the generation of a set of inputs aiming to finding errors and identifying security flaws. Grey-box fuzzing, the most deployed fuzzing strategy, combines light program instrumentation with the new input data generation. In [19], Wang et al. present QuanFuzz, a search-based test input generation tool for quantum software. It analyses the system under test by instrumenting the source code, identifying which parts of the source code are associated to the measurement results, and then mutates the initial input matrix, selecting those mutations which improve the probability weight for a value of the quantum register to trigger sensitive

branches. Benchmark results shows QuanFuzz achieves 20-60% higher coverage compared to traditional test input generation.

In [20], Betanzo introduces QuTAF, a test automation framework based on the robot framework[21], for quantum applications testing on real quantum machines. While focused on identifying hardware-related errors, it is proven that QuTAF can identify software bugs as failing test cases.

Quipper is a functional language that enables a high-level approach for the definition of quantum circuits. QPMC is a model checker developed for the verification of quantum protocols specified as Quantum Markov Chains. In [22], Anticoli et al. present Entangλe, a framework for translating Quipper-like programs into the QPMC model checker, allowing to perform automatic formal verification of quantum protocols.

In [23], Smelyanskiy et al. present a high-performance distributed quantum simulator for classic computers named qHiPSTER, which can simulate single-qubit gates and controlled two-qubit gates for testing purposes. It has been performance-checked for up to 40-qubit algorithms, achieving high performance and hardware efficiency limited by memory and bandwidth.

Probabilistic testing and verification

While classic computing shows a deterministic behavior, quantum physics properties as superposition mean that quantum computers deliver probabilistic measures when classical observations are made on qubits; that is, when a qubit in a superposition state is collapsed into a classical value, it takes a given value with a given probability. Some selected publications address quantum computing validation from a probabilistic perspective from circuit and software levels.

Krishnaswamy et al. [24] propose a general fault modeling method to capture both probabilistic and deterministic faults. The authors discuss how the behavior of quantum circuits is inherently probabilistic and they state that, while the goal of traditional testing has always been to detect the presence of faults, probabilistic testing aims to estimate fault probability—what the authors call "track uncertainty". This work presents a technology-agnostic, probabilistic equivalent called the "Probabilistic Transfer Matrix" (PTM) method. It is inspired by traditional fault models representing faults and deriving test vectors that propagate fault effects to outputs.

In [25], Huang and Martonosi address the problem of quantum software validation. They particularly highlight the need for new tools to write quantum algorithms as program code, citing the difficulty of probing the internal states of programs and interpreting such states even with existing observations. They also mention the lack of testing guidelines for quantum testing. Based on statistical tests over classical observations, they present quantum program assertions that allow programmers to determine whether a quantum program state matches the expected value in one of either classical, superposition, or entangled types of states. They use such assertions

to test three benchmark quantum programs and to lay out a strategy for using quantum programming patterns to place assertions and prevent bugs.

In [26], Li et al. propose Proq, a runtime assertion scheme based on projections (closed subspaces of the state space) and checked on projective measurement that reduces the number of assertions significantly compared to repeated executions. They prove that projection-based assertions can statistically assure that a quantum function is close to its expected behavior.

The standard weakest precondition calculus, introduced by Dijkstra [27] and extended to probabilistic programs by Morgan et al. [28], has been successfully employed to reason about the correctness of classic software. In [29], Feng et al. extend the proof rules presented by Morgan et al. for classic probabilistic loops so that they can be used to prove any correct assertion about quantum loops.

In [30], Baltag et al. present a decidable logic for reasoning about the correctness of quantum programs. It captures system properties through probabilistic predication formulas, stating that a given quantum state will collapse to a state which satisfies a given condition with a given probability. They propose first-order quantifiers ranging over quantum states and two second-order quantifiers, one ranging over quantum-testable properties, the other over quantum "actions". This technique is used to describe the correctness of Quantum Teleportation, Quantum Search Algorithm, and the Deutsch-Jozsa algorithm.

Hoare logic applications

Formal verification involves proving the correctness of an algorithm against a formal specification. Hoare provided in [31] a set of logical rules allowing to reason about the correctness of software, which has been the basis for a wide variety of testing research, including some approximations for Hoare-like logic for verifying quantum programs.

In [32], Barthe et al. propose a relational program logic based on a quantum analogue of probabilistic couplings in order to perform a verification of the properties of quantum programs, such as reliability of quantum teleportation against noise and uniformity for samples generated by the quantum Bernoulli factory.

In [33], Liu et al. formalize the theory of Quantum Hoare Logic (QHL), particularly the syntax and semantics of quantum programs; they establish rules for QHL and verify the soundness and completeness of the deduction system for partial correctness of quantum programs.

In [34], Zhou et al. derive a variant of QHL, named *applied Quantum Hoare Logic* (aQHL) which significantly simplifies verification of quantum programs. It is developed by restricting QHL to projections, a class of preconditions and postconditions, and adding several rules for reasoning about the robustness of quantum programs.

In [35], Ying et al. study the definition of invariants in quantum programs (an invariant of a software at a given location is an assertion that is always true when the location is reached, allowing to check partial correctness of a program). They also address the problem of generating additive invariants for quantum software by reducing it to a Semidefinite Programming (SDP) problem and applying an SDP solver.

In [36], Kakutani presents a Hoare-style logic for the verification of quantum, probabilistic programs. Hartog's probabilistic Hoare logic [37] is extended in this work, and the QPL language [38] is taken as the target.

Finally, Sun and He present the basic idea of *categorical logic for quantum programs* (CLQP) [39]. CLPQ combines the logic of quantum programming (LQP)—an extension of quantum Hoare logic—with categorical quantum mechanics (CQM). They present its syntax, semantics, and proof system along with a proof-of-concept over Deutsch's algorithm's correctness.

Reversible circuits testing

From the point of view of circuit verification, reversibility provides some interesting properties, such as the conservation of energy and, thus, of information [40]. In the current state-of-the-art, the most used "high-level" models and languages in quantum algorithm definition are still representations of circuits, thus keeping some of the circuits' properties that are interesting for unitary testing of quantum algorithms. This is the reason why this topic has been considered relevant for this work. Some of the selected publications focus on reversible quantum circuit verification and the applicability of the classical reversible circuit existing techniques and approaches to their quantum equivalents.

In Patel et al. [41], the test-set generation problem is considered, focusing on how it is affected by the reversibility of circuits. It is demonstrated that reversibility simplifies the problem in a significant way. An algorithm for finding complete test sets is presented and compared to conventional automatic test pattern generation (ATPG, hereinafter), obtaining test sets approximately half the size of the ATPG ones. The authors also discuss how this work may be extended to reversible quantum circuits, considering the inherent differences between the deterministic fault-free classical circuits and the probabilistic fault-free quantum circuits.

In [42], Mondal et al. propose a fault detection scheme for any type of reversible circuits consisting of entirely positive, negative, or mixed controlled Toffoli gates. It is suitable for large circuits and has been tested on several benchmark circuits, detecting faults, and identifying the faulty zone. After presenting the results, a comparative analysis with other works is performed.

Finally, Zamani et al. [43] present a test generation method for reversible circuits, adoptable by Built-in self-test (BIST) implementations, which achieves a high fault coverage. In the proposed approach, each test pattern is the output of the circuit

to the previous test pattern. A test generation algorithm to minimize test time is also presented, achieving 100% fault coverage. Encouraging results were found in the application of the benchmark simulation experiments of the proposed method.

Analysis of the current state of the art

According to the state of the art presented in the previous sections, a brief analysis of the different findings will be presented, in order to outline in what extent current proposals support the emerging quantum software testing process of Quantum Software Engineering.

One of the main identified strengths is that experience with classic software engineering has enabled the community to become aware of the need for a Quantum Software Engineering–and more specifically a Quantum Software Testing Engineering–in a very incipient state of quantum computing. This has enabled the development of elements such as lifecycles [13], bug taxonomies [10, 14], or testing frameworks [20] significantly earlier than their classic counterparts. On the other hand, the low abstraction level of the models and languages used for quantum algorithm definition implies that some techniques developed for quantum circuit testing may be possibly applied to the unitary testing of quantum software. Another point that derives from the low abstraction level is that the graphical Platform-Independent Models (PIM) are–with some exceptions [16]–based on a broadly accepted metamodel, favoring standardization, abstraction, reuse, and independence from technological providers.

On the other hand, the main weakness found is the lack of a settled and well-proven body of knowledge on Quantum Software Engineering, as the state-of-the-art testing of quantum algorithms and protocols is still rudimentary [17]. Even though the tendency in SE is to raise the level of abstraction, the artifacts used for quantum algorithm specification are still mainly circuit representations, lacking the advantages of higher-level versions. Moreover, the lifecycles proposed so far [7, 13] are cascade-based approaches. While state-of-the-art classic SE lifecycles have transformed testing processes into a set of transversal and iterative tasks, waterfall models offer a more isolated and sequential vision of testing and verification. Besides, there is still a lack of off-the-shelf components [15], such as testing libraries, and technological environments that support verification and testing activities [17, 22] are still reduced in number, matureness, and integration.

However, there is a vast body of knowledge on classic testing engineering. Although some techniques, processes, and activities have been assessed and adapted there are still others that are yet to be addressed, for example, the integration of quantum processes and artifacts on classical platforms like KDM [44] or UML [45] is an emerging field of study. Furthermore, the Quantum as a Service approach that is expected to prevail in the near future [46] will make it possible to approach integration testing and verification management through already established "as a

Service" practices, such as component integration techniques, provider Service Level Agreements or continuous integration processes [47]. Finally, from the perspective of Model-Driven Architectures (MDA) [48], the low abstraction level of current platform-independent and specific models (PIMs/PSMs) eases the mapping effort between different models and languages significantly. This favors MDA's practices such as reuse and automatic transformation and testing of high-level executable models.

Finally, it is crucial to point out the different threats that must be considered concerning quantum software testing. Firstly, quantum programming is less intuitive and more complex, thus more error-prone due to quantum mechanics [49, 50]. The fact that a significant proportion of practitioners have a background in physics or mathematics rather than computer science can also be a source for reluctance about the introduction of SE practices such as higher abstraction level artifacts or the adoption of novel development/testing processes [51]. Secondly, the QaaS predictable future combined with the low abstraction level of artifacts can derive in low portability of models and code and solid dependency of the hardware providers.

From classic to quantum software testing: redefining the mutation technique

Introduction

Considering the previous analysis of the state of the art, it is obvious the lack of techniques and tools to implement a Quantum Software Engineering-based quantum software testing process. In order to start tackling with such situation, the current section presents a redefinition of one of the most powerful techniques to perform software testing in classic software development: software mutation.

Mutation testing has been widely used to improve the quality of test suites ever since the inception of structure programming [52] to optimised applications in object-oriented development [53, 54]. Also, mutation testing has been applied to different software domains [55]. Mutants are usually generated by automated tools that, through mutation operators, introduce syntactic changes in the programs. Most of the time, these changes can be interpreted as small mistakes that a *competent programmer* [56] might commit and that, under given circumstances, could lead to the program showing an unexpected behaviour. This situation is more common for programmers who come from the field of classical computing [57].

Thus, mutation tools can imitate simple human errors and are premised on the *coupling effect* (i.e., if a test suite is sensitive enough to detect simple faults, it will

be also able to detect more complex faults [58]). Each mutation operator is specialised in introducing one particular type of change that may cause an error.

To the best of our knowledge, mutation is a technique that to date has not been exploited in quantum circuit development, except for two proposals which do mention the concept of mutation. The first of these is [59], where the authors apply metamorphic testing to quantum software (written in Q#). In this approach, the concept of "mutant" is only mentioned as a part of the validation of the approach, and as a method to modify the original source code. Although this approach is focused on Q# programs, mutants proposed in this paper are just possible examples of modifications that could be carried out on quantum software. The second such proposal is [60], in which the authors present a novel quantum mutant-based fault injection technique, based on the replacement of certain gates by other ones.

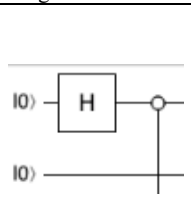
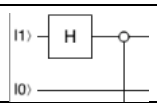
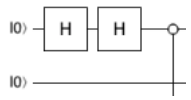
Quantum specific errors and operators

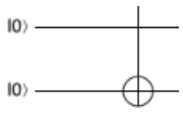
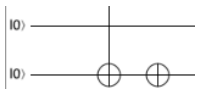
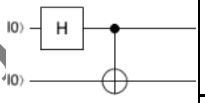
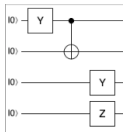
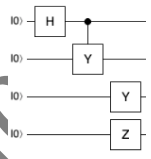
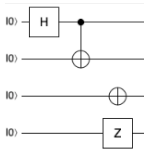
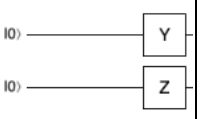
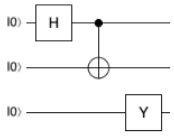
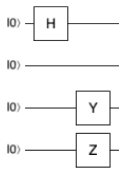
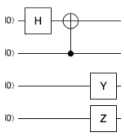
Quantum Computing is a young discipline and as yet there are few fault models. Two of the most significant are those produced by Lukac et al. [61] and by Biamonte et al. [61], who, respectively, collect: (1) faults due to *quantum noise* or gate-building construction and (2) *programmer faults*.

We are concerned here with the second type, which we have grouped into the categories: *missing gates*, *wrong gates*, *bridging faults* (a multi-qubit gate which connects wrong qubits), and *initialisation errors*. We have included an additional category for representing *entanglement faults.*

The proposed, designed and implemented mutation operators appear in **Table 1**. Each operator includes its description and one example of its application. Note that each operator can be applied several times to the same circuit, producing one mutant circuit per application.

The *wrong gate* operators swap X, Y, Z and H gates with the others, thus showing a similar behaviour. As a result, we give only three examples of its application.

Family	Operator	Description	Examples	
			Original circuit	Mutant circuit
Initialization	Change initial value	Changes the initial value of a qubit		
	First gate duplication	Duplicates a one-qubit gate placed in the first column of the circuit		

	Further gate duplication	Duplicates a one-qubit gate placed in the 2nd and next columns		
Wrong gate	Swap X-Y	These operators swap one Pauli or Hadamard gate with a different Pauli or Hadamard gate		 *Swap H-Y*
	Swap X-Z			
	Swap X-H			
	Swap Y-X			 *Swap X-Y*
	Swap Y-Z			
	Swap Y-H			
	Swap Z-X			 *Swap Y-X*
	Swap Z-Y			
	Swap Z-H			
	Swap H-X			
	Swap H-Y			
	Swap H-Z			
Missing gate	One-qubit gate removal	Removes a one-qubit gate at any location of the circuit		 *Removed the 4th qubit*
	Multi-qubit gate removal	Removes a multi-qubit gate at any location of the circuit		 *Removed the 2nd column*
	Control gate removal	Removes one control gate in a multi-qubit gate		
Bridging faults	Swap controls and controlled qubits	Swaps the control gate with one of the controlled gates		

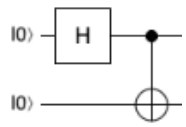
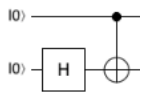
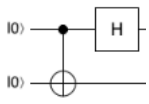
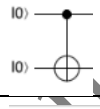
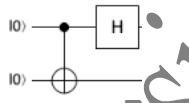
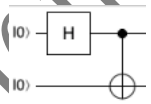
Entanglement faults	Wrong entanglement initialisation	Moves the H gate to the X row		
	Entanglement corruption	Shifts the H gate to the right of the control column		
	Force unentanglement	Removes the H gate in the first column		
	Force entanglement	Detects inverse situations to an entanglement and converts them into entanglements		

Table 1. Quantum mutation operators

Quantum mutation support tool

Description of the prototype

QuMu is a tool for the mutation testing of quantum circuits. It is based on the circuit representation given by the Quirk quantum circuit simulator[5]. In Quirk, a circuit is represented by an ordered set of columns, and on each column, there is an ordered set of gates. Quirk exports the circuits as a JSON object and, from this, QuMu creates a circuit representation. In Fig. 1., the diagram shows the most meaningful operations in each class. Note that *Gate* is an abstract class that has as many concrete specialisations (Figure 6) as types of gates that we want to mutate.

[5] https://algassert.com/quirk

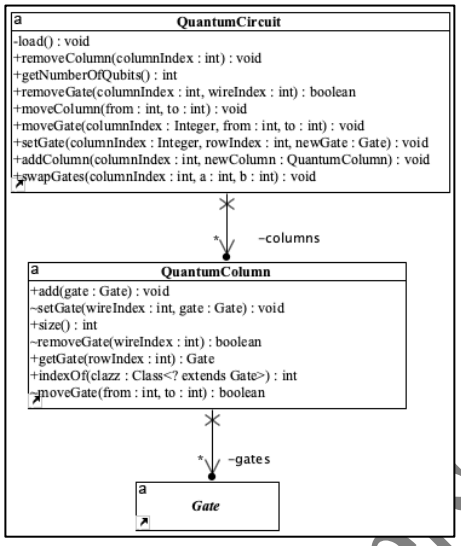


Fig. 1. QuMu representation of a quantum circuit

Mutation operators are specialisations of the abstract Operator class (Fig. 5), which declare two abstract operations:

- *isApplicableTo* returns *true* whether this operator is applicable to a quantum circuit passed as a parameter. If it is, then the operation saves in the *mutablePositions* map whichever rows are mutable in each column.
- If it was determined that the operator was applicable to the circuit, *apply* goes through all the columns and rows saved in *mutablePositions* and generates a mutant for each mutable position. Every mutant is then saved in a database.

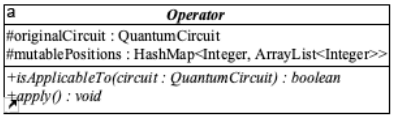


Fig. 2. A generic QuMu mutation operator

The operators are organised in a hierarchical structure (Fig. 3) like that shown in Table I, which allows for operations which are common to several operators to be reused: for example, the isApplicableTo method in the abstract EntanglementOperator class is valid for all three of its specialisations.

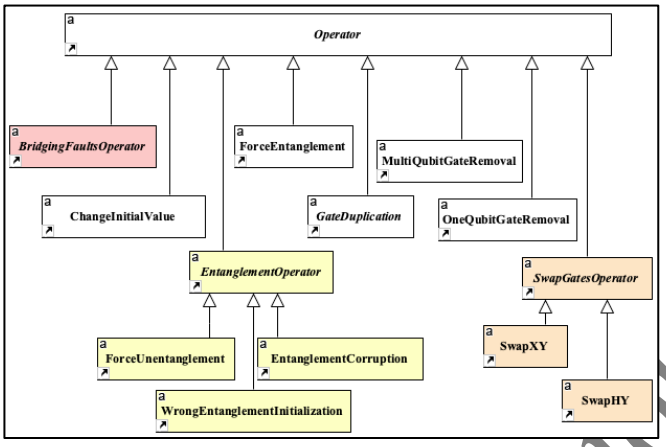


Fig. 3. Partial view of the architecture of the operators

The determination of the applicability of an operator to a given circuit relies on reflective programming: for example, the isApplicableTo method in OneQubitGateRemoval looks for gates implementing the IOneQubitGate interface (Fig. 4). Currently, the only gates implementing this interface are X, Y, Z and H - and so the operator would not be applicable to the Square of Z gate. If we want to apply this operator to this gate, we only need to ensure that SquareOfZ implements the IOneQubitGate.

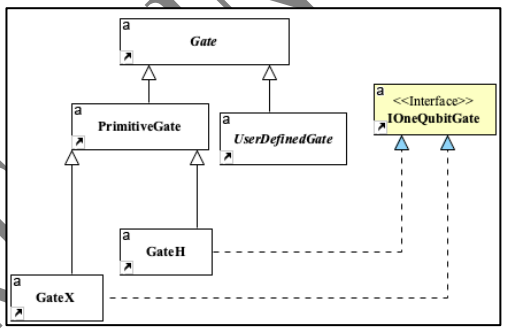


Fig. 4. isApplicableTo uses reflection to decide the applicability of an operator

Once the two subsystems (i.e., the circuit with its columns and gates, and the operators) have been defined, the generation and execution of mutants is performed by two engines:

- The first engine (mutant generation) iterates over the selected mutation operators and over the columns and gates of the circuit.
- The second engine (mutant execution) uploads every mutated circuit onto the Quirk website, performs the execution simulation of the mutant, downloads the results, and saves them in the database.

Quantum software mutation example

Fig. 5 shows the top side of QuMu, which is implemented as a web application. The tester may either load one of circuits saved in the database (shown in the dropdown list of the section "Original circuit") or may paste its Quirk's JSON specification into the "Your circuit" section.

The circuit is drawn in the third section ("Loaded circuit"), which is an iframe element containing the Quirk web page.

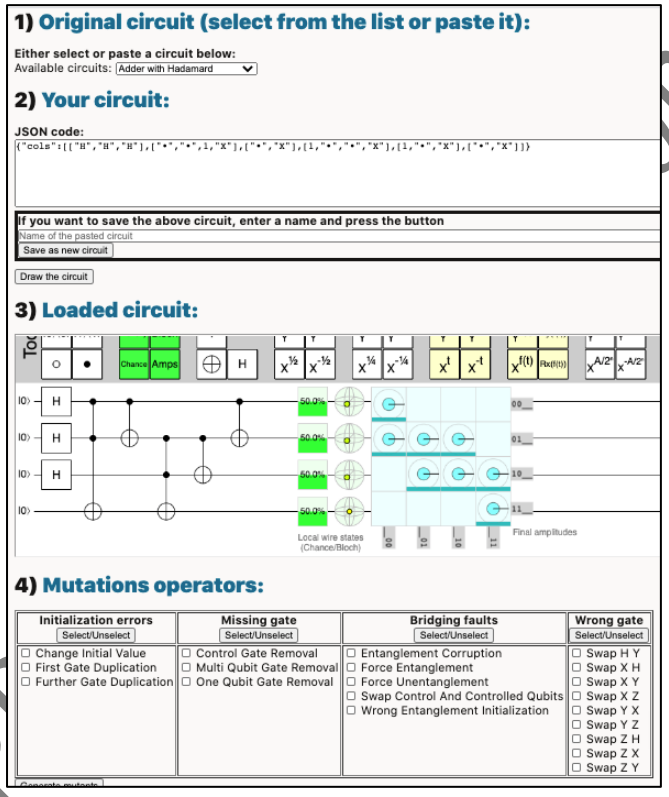


Fig. 5. Top side of QuMu

In section 4 (Fig. 5), the tester selects the operators they want to apply to the circuit and presses the Generate mutants button. When the server receives the request, it generates the mutants, saves them in the database, and returns the results to the user-agent.

For the circuit shown in Fig. 5, QuMu generates 58 mutants. Fig. 6 shows the 15th mutant, generated by the Control Gate Removal operator to the control gate at the CNOT gate, in the fifth column.

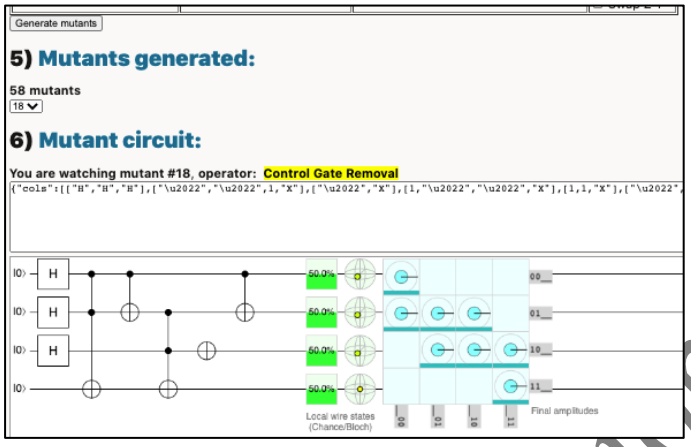


Fig. 6. Detail of the 18th mutant for the circuit in Fig. 5

According to the distribution matrixes obtained from the execution of mutants they are classified into three different states: alive mutants, killed mutants and injuried mutants.

Killed mutants

In Quirk all the circuits include a matrix of *n* x *n* (with *n* being the number of qubits) on the right side, which shows the distribution of probabilities of each pair of qubits. Each cell is represented by a complex number *(r, i)* that represents the amplitude matrixes.

Fig. 7 details the matrixes which correspond to the original and to the 18th mutant. Since the distribution of the results is different, this mutant can be clearly marked as killed.

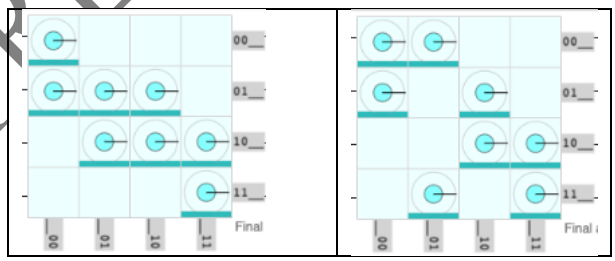


Fig. 7. Probabilities in the original (left) and in the 18th mutant, which is killed

Alive mutants

The circuits in Fig. 8 are the original adder circuit (top row) and its 15th mutant, whose change has also been generated by the *Control Gate Removal* operator (i.e., the control gate at the second column has been removed). Both distribution matrixes are exactly equal and, hence, this mutant is *alive.*

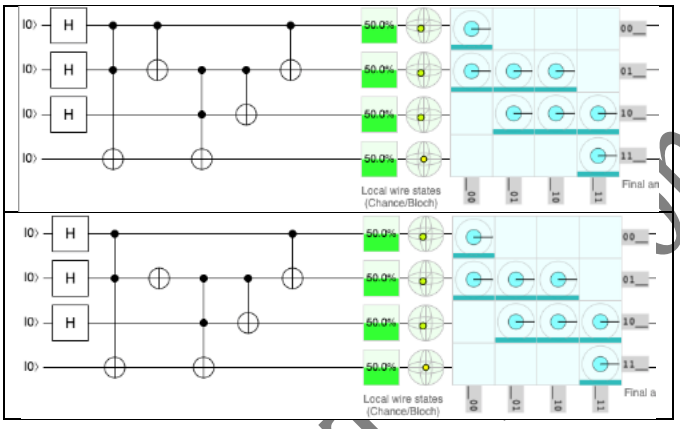


Fig. 8. Original circuit (top) and the 15th mutant, which remains alive

Injured mutants

There are other mutants that offer the same result as the original but leave the qubits with a different phase. The *Change Initial Value* operator has produced the Mutant 1, the first qubit input value of which has been changed from 0 to 1.

As is seen in Fig. 9, the original and the mutant distribution matrixes are almost identical, but there is a difference in the final phase of the first qubit: the quantum particle points outside the paper in the original circuit, and inside the paper in the mutant. However, the absolute values of the complex numbers representing each cell in the matrix are the same and, thus, both outputs are indistinguishable when the qubits are measured.

We refer to this type of mutant as "injured".

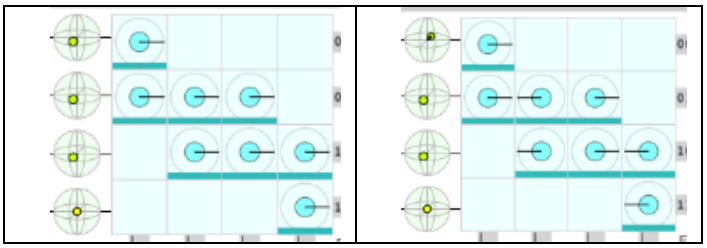


Fig. 9. The output probabilities are the same, but the phases are different (original versus Mutant 1)

Showing the analysis results in QuMu

QuMu includes a mutant execution engine and a mutant results analyser. Once the mutants have been generated, the first one iterates over each successive mutant, reads the output produced by Quirk, extracts the probability distribution matrix, and saves it onto a database. Then, the results analyser compares the distribution matrixes of the original circuit with every mutant, showing, respectively, either a killing or an injured matrix (Fig. 10).

7) Result analysis:

Analyze results

Killing matrix			
58 mutants	53 killed mutants Score (in [0, 1]): 0.9137	56 injured mutants Score (in [0, 1]): 0.9656	Operator
Mutant	Killed	Injured	
m1	Alive	Injured	Change Initial Value
m2	Alive	Injured	Change Initial Value
m3	Alive	Injured	Change Initial Value
m4	Killed	Injured	Change Initial Value
m5	Killed	Injured	First Gate Duplication
m6	Killed	Injured	First Gate Duplication
m7	Killed	Injured	First Gate Duplication

Fig. 10. First rows of the killing & injured matrixes for the Hadamard adder

Conclusions and future work

Quantum computing is no longer a promise, but a reality and its impact in current and future society trends has been proven and recognized by academy and governments (which play crucial responsibilities on the success of Quantum Software Engineering [62]) since “quantum computers have the potential to solve tasks that we don’t even dare dream of today and that classical computers can never solve” [63]. However, quantum software (as a new and substantially different paradigm) lacks

a methodological background to ensure a quality development process and quality products, as it has been claimed in different studies. A Quantum Software Engineering study field is emerging to avoid a potential quantum software crisis, and many areas must be developed. That means that we must commence quantum software engineering right now, thus seeking to be prepared, and endeavouring to avoid low quality quantum software plagued with errors and productivity problems.

To achieve all the benefits that quantum computing offers, though, this new paradigm will need to be developed in an appropriate way. Testing processes and tools are of particular importance - as is evident from classic software engineering. As a first step in our walkthrough to the statement of a standardized quantum software testing process, we present in this chapter (i) an analysis of the current state of the art about quantum software testing, and aligned to our aim of contributing to create a body of knowledge for quantum software testing (and in turn, contribute to Quantum Software Engineering), (ii) we propose the redefinition of the quantum software mutation-based testing technique, together with a prototype to make feasible the application of the proposal.

The analysis of the state-of-the-art reveal that quantum physics characteristics have an important impact on the verification and validation of quantum software, as well as superposition, which turn the deterministic nature of (classic) software into an stochastic one for quantum software. Other quantum characteristics, such as the inability to clone/copy the exact state of a qubit, make impossible to check the intermediate state of an particular qubit. Despite these physical barriers to perform testing on quantum programs, several trends have been identified: (i) proposals dealing with the stochastic nature of quantum software, (ii) adaptations of the Horale logic, and (iii) the use of quantum circuits' reversibility property.

Finally, a redefinition of the software testing based on mutation technique has been proposed for quantum software testing. An initial set of errors has been identified, and the corresponding mutant operators has been designed to simulate the errors in "quantum circuits under tests". In order to it feasible to apply the technique, a prototype has been developed, and initial experiments shows not only the possibility to identify "*killed* and *alive*" quantum mutants, but also a new type of quantum mutant, the "*injured*" one. More research must be carried out in order to find new errors, determine the most common ones, refinements in the redefinition on the technique.

Acknowledgements

The research work presented in this paper is framed within the following projects: TESTIMO (*Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla La Mancha y Fondo Europeo de Desarrollo Regional FEDER, SBPLY/17/180501/000503*), and "QHealth: Quantum Pharmacogenomics Applied to Aging", 2020 CDTI Missions Program (Center for the Development of Industrial

Technology of the Ministry of Science and Innovation of Spain). We would like to thank all the aQuantum members, especially Guido Peterssen and Pepe Hevia, for their help and support.

References

1. Maslov, D., Nam, Y., and Kim, J., *An Outlook for Quantum Computing [Point of View]'.* Proceedings of the IEEE, 2019. **107**(1): p. 5–10,.
2. López, M.A. and Silva, M.M.D., *Quantum Technologies: Digital Transformation, Social Impact, and Cross-sector Dis-ruption'*. 2019.
3. Humble, T.S. and DeBenedictis, E.P., *Quantum Realism'.* Computer, 2019. **52**(06): p. 13–17,.
4. LaRose, R., *Overview and Comparison of Gate Level Quan-tum Software Platforms'.* Quantum, 2019. **3**: p. 130,.
5. Gill, S.S., *Quantum Computing: A Taxonomy, Systematic Review and Future Directions'.* ArXiv, 2020. **2010**(15559).
6. Piattini, M., Peterssen, G., Pérez-Castillo, R., Hevia, J.L., Serrano, M.A., Hernández, G., García-Rodríguez de Guzmán, I., Paradela, C.A., Polo, M., Murina, E., Jiménez, L., Marqueño, J.C., Gallego, R., Tura, J., Phillipson, F., Murillo, J.M., Niño, A., and Rodríguez, M. *The Talavera Manifesto for Quantum Software Engineering and Programming*. in *QANSWER*. 2020.
7. Zhao, J., *Quantum Software Engineering: Landscapes and Horizons.* arXiv preprint arXiv:2007.07047, 2020.
8. Miranskyy, A., Zhang, L., and Doliskani, J. *Is Your quantum Program Bug-Free?* in *Proceedings - 2020 ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2020*. 2020.
9. Usaola, M.P. *Quantum Software Testing*. in *QANSWER*. 2020.
10. Huang, Y. and Martonosi, M. *QDB: From quantum algorithms towards correct quantum programs*. in *OpenAccess Series in Informatics*. 2019.
11. Sodhi, B., *Quality Attributes on Quantum Computing Platforms.* arXiv preprint arXiv:1803.07407, 2018.
12. Miranskyy, A., Zhang, L., and Doliskani, J., *On Testing and Debugging Quantum Software.* arXiv preprint arXiv:2103.09172, 2021.
13. Dey, N., Ghosh, M., and Chakrabarti, A., *QDLC--The Quantum Development Life Cycle.* arXiv preprint arXiv:2010.08053, 2020.
14. Campos, J. and Souto, A., *QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments.* arXiv preprint arXiv:2103.16968, 2021.

15. Gomes, C., Fortunato, D., Fernandes, J.P., and Abreu, R. *Off-the-shelf components for quantum programming and testing*. in *CEUR Workshop Proceedings*. 2020.
16. Reutter, D. and Vicary, J. *Shaded tangles for the design andverification of quantum programs (extended abstract)*. in *Electronic Proceedings in Theoretical Computer Science, EPTCS*. 2018.
17. Honarvar, S., Mousavi, M.R., and Nagarajan, R. *Property-based Testing of Quantum Programs in Q#.* in *Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020*. 2020.
18. Steiger, D.S., Häner, T., and Troyer, M., *ProjectQ: an open source software framework for quantum computing.* Quantum, 2018. **2**: p. 49.
19. Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., and Sun, J., *QuanFuzz: Fuzz testing of quantum program*. arXiv preprint arXiv:1810.10310, 2018.
20. Betanzo Sanchez, F., *QuTAF: A Test Automation Framework for Quantum Applications.* 2020.
21. Bisht, S., *Robot framework test automation*. 2013: Packt Publishing Ltd.
22. Anticoli, L., Piazza, C., Taglialegne, L., and Zuliani, P., *Entanglе: A translation framework from quipper programs to quantum markov chains*, in *Communications in Computer and Information Science*. 2018. p. 113-126.
23. Smelyanskiy, M., Sawaya, N., and Aspuru-Guzik, A., *qHiPSTER: the quantum high performance software testing environment (2016).* arXiv preprint arXiv:1601.07195.
24. Krishnaswamy, S., Markov, I.L., and Hayes, J.P., *Tracking uncertainty with probabilistic logic circuit testing.* IEEE Design & Test of Computers, 2007. **24**(4): p. 312-321.
25. Huang, Y. and Martonosi, M. *Statistical assertions for validating patterns and finding bugs in quantum programs*. in *Proceedings - International Symposium on Computer Architecture*. 2019.
26. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., and Xie, Y., *Projection-based runtime assertions for testing and debugging Quantum programs.* Proceedings of the ACM on Programming Languages, 2020. **4**(OOPSLA).
27. Dijkstra, E.W., Dijkstra, E.W., Dijkstra, E.W., and Dijkstra, E.W., *A discipline of programming*. Vol. 613924118. 1976: prentice-hall Englewood Cliffs.
28. Morgan, C., McIver, A., and Seidel, K., *Probabilistic predicate transformers.* ACM Transactions on Programming Languages and Systems (TOPLAS), 1996. **18**(3): p. 325-353.
29. Feng, Y., Duan, R., Ji, Z., and Ying, M., *Proof rules for the correctness of quantum programs.* Theoretical Computer Science, 2007. **386**(1-2): p. 151-166.

30. Baltag, A., Bergfeld, J.M., Kishida, K., Sack, J., Smets, S.J.L., and Zhong, S., *Quantum probabilistic dyadic second-order logic*, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2013. p. 64-80.
31. Hoare, C.A.R., *An axiomatic basis for computer programming.* Communications of the ACM, 1969. **12**(10): p. 576-580.
32. Barthe, G., Hsu, J., Ying, M., Yu, N., and Zhou, L., *Relational proofs for quantum programs.* Proceedings of the ACM on Programming Languages, 2020. **4**(POPL).
33. Liu, J., Zhan, B., Wang, S., Ying, S., Liu, T., Li, Y., Ying, M., and Zhan, N., *Formal verification of quantum algorithms using quantum hoare logic*, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2019. p. 187-207.
34. Zhou, L., Yu, N., and Ying, M. *An applied quantum hoare logic*. in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019.
35. Ying, M., Ying, S., and Wu, X. *Invariants of quantum programs: Characterisations and generation*. in *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. 2017.
36. Kakutani, Y., *A logic for formal verification of quantum programs*, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2009. p. 79-93.
37. den Hartog, J. *Verifying probabilistic programs using a hoare like logic*. in *Annual Asian Computing Science Conference*. 1999. Springer.
38. Selinger, P., *Towards a quantum programming language.* Mathematical Structures in Computer Science, 2004. **14**(4): p. 527-586.
39. Sun, X. and He, F., *A first step to the categorical logic of quantum programs*. Entropy, 2020. **22**(2).
40. Fredkin, E. and Toffoli, T., *Conservative logic.* International Journal of theoretical physics, 1982. **21**(3): p. 219-253.
41. Patel, K.N., Hayes, J.P., and Markov, I.L., *Fault testing for reversible circuits.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2004. **23**(8): p. 1220-1230.
42. Mondal, B., Bandyopadhyay, C., and Rahaman, H. *A testing scheme for mixed-control based reversible circuits*. in *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*. 2016. IEEE.
43. Zamani, M., Tahoori, M.B., and Chakrabarty, K. *Ping-pong test: Compact test vector generation for reversible circuits*. in *2012 IEEE 30th VLSI Test Symposium (VTS)*. 2012. IEEE.
44. Jiménez-Navajas, L., Pérez-Castillo, R., and Piattini, M. *Reverse Engineering of Quantum Programs Toward KDM Models*. in *International*

Conference on the Quality of Information and Communications Technology. 2020. Springer.

45. Pérez-Castillo, R., Jiménez-Navajas, L., and Piattini, M., *Modelling Quantum Circuits with UML.* arXiv preprint arXiv:2103.16169, 2021.
46. Leymann, F., Barzen, J., Falkenthal, M., Vietz, D., Weder, B., and Wild, K., *Quantum in the cloud: application potentials and research opportunities.* arXiv preprint arXiv:2003.06256, 2020.
47. Bratman, H. and Court, T., *The software factory.* Computer, 1975. **8**(5): p. 28-37.
48. OMG, *MDA Guide v1. 0.* Business Process Integration chapter, 2003.
49. Ying, M., *Floyd-hoare logic for quantum programs.* ACM Transactions on Programming Languages and Systems, 2011. **33**(6).
50. Piattini, M., Serrano, M., Perez-Castillo, R., Petersen, G., and Hevia, J.L., *Toward a quantum software engineering.* IT Professional, 2021. **23**(1): p. 62-66.
51. Theocharis, G., Kuhrmann, M., Münch, J., and Diebold, P. *Is water-scrum-fall reality? on the use of agile and traditional development practices*. in *International Conference on Product-Focused Software Process Improvement*. 2015. Springer.
52. Agrawal, H., *Design of mutant operators for the C pro-gramming language'*. 1989, Software Engineering Research Center, Purdue University, West Lafayette.
53. Polo, M., Piattini, M., and García-Rodríguez, I., *Decreasing the Cost of Mutation Testing with Second-Order Mutants'.* Softw. Test. Verif. Reliab, 2009. **19**(2): p. 111–131,.
54. Deng, L. and Offutt, A.J., *Reducing the Cost of Android Mutation Testing'*. 2018.
55. Deng, L., Offutt, J., Ammann, P., and Mirzaei, N., *Mutation Operators for Testing Android Apps'.* Inf. Softw. Technol, 2017. **81**(C): p. 154–168,.
56. DeMillo, R.A., Lipton, R.J., and Sayward, F.G., *Hints on Test Data Selection: Help for the Practicing Programmer'.* Comput-er, 1978. **11**(4): p. 34–41,.
57. Li, G., Zhou, L., Yu, N., Ding, Y., Ying, M., and Xie, Y., *Projection-Based Runtime Assertions for Testing and Debugging Quan-tum Programs'.* Proc. ACM Program. Lang, 2020. **4**.
58. Offutt, A.J., *Investigations of the Software Testing Coupling Effect'.* ACM Trans. Softw. Eng. Methodol, 1992. **1**(1): p. 5–20,.
59. Honarvar, S., Mousavi, M.R., and Nagarajan, R. *Property-Based Testing of Quantum Programs in Q#'*. 2020.
60. Boncalo, O., Udrescu, M., Prodan, L., Vladutiu, M., and Amaricai, A., *Assessing quantum circuits reliability with mutant-based simulated fault injection'*, in *2007 18th European Confer-ence on Circuit Theory and Design*. 2007. p. 942–945,.

61. Lukac, M., Kameyama, M., Perkowski, M., Kerntopf, P., and Moraga, C., *Fault Models in Reversible and Quantum Circuits'*, in *Advances in Unconventional Computing: Volume 1: Theory*, A. Adamatzky, Editor. 2017, Springer International Publishing: Cham. p. 475–493.
62. Piattini, M., Peterssen, G., and Pérez-Castillo, R., *Quantum Computing: A New Software Engineering Golden Age.* SIGSOFT Softw. Eng. Notes, 2020. **45**(3): p. 12–14.
63. EQF, *Strategic Research Agenda. European Quantum Flagship'*. 2020, European Commission.