QuTAF:

A Test Automation Framework for Quantum Applications

Fernando Betanzo Sánchez

Delft University of Technology

Electrical Engineering, Mathematics and Computer Science
Quantum & Computer Engineering
Quantum Information and Software

Master Thesis

QuTAF: a Test Automation Framework for Quantum Applications

Fernando Betanzo Sánchez

Committee Members Dr. David Elkouss
EEMCS
Delft University of Technology

Dr. Przemysław Pawełczak
EEMCS
Delft University of Technology

Supervisor Prof. Dr. Stephanie Wehner

November, 2020

Fernando Betanzo Sánchez
QuTAF: a Test Automation Framework for Quantum Applications
Master Thesis, November, 2020
Reviewers: Dr. David Elkouss and Dr. Przemysław Pawełczak
Supervisors: Prof. Dr. Stephanie Wehner

Delft University of Technology
Quantum Information and Software
Quantum & Computer Engineering
Electrical Engineering, Mathematics and Computer Science
Postbus 5
2600 AA and Delft

Abstract

The testing of quantum applications can be approached from three perspectives. The first one concerns the certification of the accuracy of the quantum device where the application is run. The second one has to do with the classical verification of the result output by the application. Yet a third one addresses the problem from the software engineering perspective. As new quantum applications that run in Noisy Intermediate-Scale Quantum devices are developed, there is an increasing need for tools that can help to find bugs and verify that these applications work as expected. In this thesis we design and develop such a tool. We introduce QuTAF, a test automation framework for quantum applications that is based in the `robot framework`. To the best of our knowledge, this is the first test automation framework software developed and used for testing quantum applications in a real quantum node. We prove that our QuTAF is capable of detecting minor deficiencies in current state-of-the-art quantum hardware, by running tests for small quantum applications executed in a networked quantum node. We also simulate and test two different failure scenarios to validate the capabilities of our QuTAF. We simulate quantum devices affected by depolarizing and dephasing noise, and find that our QuTAF is able to detect errors introduced by an increase in the depolarization probability, but is otherwise insensitive to the errors produced by the dephasing of quantum states. We also simulate bugs present in the quantum programs, and prove that our QuTAF is able to correctly identify these as failing test cases.

Acknowledgement

This Master thesis is the culmination of my research activities conducted at QuTech and the Technical University of Delft, during the year of 2020. It is a milestone that I set as my goal a little more than three years ago. Of course, this would not have been possible to achieve without the support of all the people, in Mexico and in The Netherlands, that stood beside me either physically or virtually.

Personally speaking, my first and deepest thank goes to my partner, Danny, the love of my life. Without your love, your kindness, your bravery, your understanding, your support, your incentive, and even your pressure, none of this would have been possible. I hope I can spend the rest of my life trying to pay you back. Thank you Sabina, my daughter, for reminding me that life is not always as hard as it seems to be. Thank you for pushing me to be a better person everyday. Thanks to you too, Joaquín, my baby boy. Thank you for bringing so much light and joy to our life. Thanks to Ravish Budhrani, Quint van Velthoven, Claudia Vázquez, and Gustavo López, for the good nights of entertainment and amusement. Last but not least, I want to thank my mother and father, Alicia and Felipe. There are just not enough words to thank you for all the support and motivation you have always provided. Without your teachings and your love, I would not be who I am.

Academically speaking, my first and deepest thank goes to my advisor, Stephanie Wehner. I would like to thank you for the opportunity of working on a project that combined two of my academic and professional passions, quantum computing and software testing. Thank you also for your understanding and flexibility, and for putting my well-being in front of everything else. Huge thanks to Wojciech Kozlowski for the countless discussions, the non-stopping guidance, the unfathomable patience, and for being there while it seemed that everything was going astray. I am truly grateful to have had you as my daily supervisor. Thanks Carlo Delle Donne and Ingmar te Raa for your work on the software of the networked quantum node, and for the willingness to debug and use my testing framework, even when the time for doing so was short. Thanks to Bas Dirkse for the lengthy discussions about testing quantum computers, and for always being quick in pointing me to the right direction. Thanks to Przemysław Pawełczak for the guidance, the willingness to collaborate, and for the nice encouragement words in moments of need. And thanks to Tina Nane for helping me put the hypothesis testing part of the thesis in solid mathematical ground.

Finally, I would like to thank the Foundation Justus & Louise van Effen for financially supporting international students through their Excellence Scholarship, without it, it would have been impossible for me to pursue the Master program of my dreams here in The Netherlands. Likewise, thanks to the *Fondo para el Desarrollo de Recursos Humanos* (Fund for the Development of Human Resources), from *Banco de México*, for their timely and outstanding financial support.

Contents

Acronyms

BPP	Bounded-error Probabilistic Polynomial time. 18
BQP	Bounded-error Quantum Polynomial time. 18
CDF	Cumulative Distribution Function. 27
IP	Interactive Proofs. 18
ISF	Inverse Survival Function. 28, 34, 52, 69
NetSquid	Network Simulator for Quantum Information with Discrete events. 37
NISQ	Noisy Intermediate-Scale Quantum. 1, 15, 70
NOS	Network Operating System. 32, 33
NV	Nitrogen-Vacancy centres in Diamond. 37, 43, 44, 54
PMF	Probability Mass Function. 24, 25, 69
QuTAF	Test Automation Framework for Quantum Applications. 31, 69
SF	Survival Function. 27
SUT	System Under Test. 13, 31, 35, 38, 70
TAF	Test Automation Framework. 14

1 Introduction

Experience is not what happens to a man; it is what a man does with what happens to him.

— **Aldous Huxley**
Writer and philosopher

The quantum applications that will be run in the near future, in the so-called Noisy Intermediate-Scale Quantum (NISQ) computers, give us already the possibility of tackling classically intractable problems of moderate size [25]. Examples of these are the simulation of medium sized quantum-many body systems [27], optimization problems like community detection in networks[28], or some quantum machine learning tasks [31].

As these applications are developed and run in different quantum back ends, there is a need for tools and mechanisms that can help us in verifying that they work as expected. These tools are required not only for attesting the quality of the quantum hardware, or the validity of the quantum computations performed, but also for detecting issues in the quantum programs themselves.

Moreover, almost all of the NISQ-era applications are based on hybrid quantum-classical algorithms, where classical and quantum computers work together in order to solve a problem. Thus, there exist the demand for validating also the complex interactions happening between the classical and the quantum software components.

During the early ages of the quantum computing field, most emphasis was given to the development of techniques for the certification of the accuracy of the quantum hardware, and the certification of the results of a quantum computation. For the former, methods such as quantum state tomography [14], randomized benchmarking [21], or self-testing of quantum systems [29], helped to corroborate the first quantum computing platforms. For the latter, the field of quantum interactive proof systems [12] was created to put in mathematical terms the question of how to use a classical computer to verify the result of a quantum computation.

Now, as the quantum devices improve and more quantum applications for small quantum computers are developed, it is time to look at the problem from a different perspective. In this work, we address the topic of testing quantum applications from the software engineering point of view. We implement a test automation framework that can be used for finding defects in quantum applications, and for verifying it works as per the requirements. The main contributions of listed below:

- We model the testing of quantum applications as a stochastic process. More specifically, it is modeled as a Bernoulli process that consists of a series of Bernoulli trials that succeed with probability p, and fail with probability $1 - p$.
- We use the Hypothesis Testing technique in order to compare the distribution of the observed testing outcomes with an expected probability distribution. Furthermore, the Binomial Inverse Survival Function and the Binomial Proportion Confidence Interval, are proposed as practical ways of performing the Hypothesis Testing.
- We design and develop a test automation framework that can perform functional tests on quantum applications. We base its development in an existing open source framework, the `robot framework`.
- We use our test automation framework for verifying the development of the Network Operating System of a networked quantum node. In particular, the framework is used for testing small quantum applications and verifying the integration between the classical software and the quantum hardware. We find that the framework is capable of detecting real deficiencies present in the quantum devices.
- We simulate different failure scenarios using the `NetSquid` tool, in order to showcase the capabilities and limits of our test automation framework. Scenarios like the lack of quality from the quantum devices, or programming errors found in the quantum programs, are simulated and tested.

1.1 Thesis Structure

Chapter 2

In this chapter we provide the background required for understanding this thesis. The first section is a recap of quantum information concepts such as qubits, quantum gates, quantum measurements and quantum errors. The second section runs through the subject of software testing, reviewing topics such as testing techniques, testing levels, and software test automation.

Chapter 3

Here, the question *what does it mean to test a quantum application?*, is discussed at length. Related works on the subjects of quantum hardware certification, the formal verification of quantum programs, and the testing of quantum applications, are considered. Then we debate about the differences between testing quantum and classical applications, and look at how much of the current testing knowledge can be applied, or adapted, to the testing of quantum software.

Chapter 4

This chapter deals with the topic of Hypothesis Testing. We introduce the stochastic process used for modeling the testing process, and then describe how to apply the Hypothesis Testing technique in order to test quantum applications. Here, we also depict how the Binomial Inverse Survival Function and the Binomial Proportion Confidence Interval can be used for achieving the same result.

Chapter 5

In this chapter we present QuTAF, the first (to the best of our knowledge) test automation framework for quantum applications. We first look at the requirements that any test automation framework that aims to test quantum applications, should meet. Then, we recount the design decisions made for developing QuTAF. We explain the selection of the `robot framework` as the basis for our test automation framework, and the use of `NetSquid` for performing the quantum simulations.

Chapter 6

Here we present the results of using QuTAF for testing quantum applications run on both a real and a simulated quantum node. We describe the tests conducted for the first use case, and discuss in detail an interesting example of a failing test case. Later, the simulations performed as part of the second use case and the results obtained, are also explained in detail.

Chapter 7

We conclude this work with a presentation of the key takeaways gathered from the development of QuTAF and a discussion of the main results obtained. An outlook of future research work is also given.

Background

2

We know very little, and yet it is astonishing that we know so much, and still more astonishing that so little knowledge can give us so much power.

— **Bertrand Russell**
Polymath

2.1 Quantum Computing

2.1.1 Qubit

We know classical information is treated in terms of bits. In the case of quantum information, we use qubits. A single qubit is a two-level quantum system expressed in terms of any orthonormal basis, for example the excited $|e\rangle$ and ground $|g\rangle$ states of an electron. In general, it is customary to use the computational basis $\{|0\rangle, |1\rangle\}$ as the *de-facto* basis, where:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Unlike classical bits, however, a qubit can be in state $|0\rangle$, state $|1\rangle$, or in a *superposition* of both. Mathematically, this means that an arbitrary qubit can be written as a linear combination of the states $|0\rangle$ and $|1\rangle$ as such:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.1}$$

Where α and β are the amplitudes of the basis states, $\alpha, \beta \in \mathbb{C}$, and we have a normalization restriction on these amplitudes such that $\|\alpha\|^2 + \|\beta\|^2 = 1$. A qubit is, then, a vector in a two-dimensional complex vector space. For reasons that will be clear in the next sections, the qubits that can be expressed like this, are also called pure quantum states.

2.1.2 Pure Multi-qubit Quantum States

We can also have multi-qubit pure quantum states, and we express the joint state of two or more qubits as a tensor product of the individual states. For example, for $|\psi\rangle$ defined above and $|\phi\rangle = \gamma|0\rangle + \delta|1\rangle$, we get:

$$|\psi\rangle \otimes |\phi\rangle = |\psi\phi\rangle = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle \tag{2.2}$$

Hence, with n qubits a space $\mathbb{C}^{2^n}$ is spanned. These types of systems are called separable quantum states. Any multi-qubit quantum state that is not separable, is said to be *entangled*.

2.1.3 Statistical Mixture of Multi-qubit Quantum States

When thinking about quantum systems whose state is not completely known, for example when we have an imperfect quantum source that generates a state $|0\rangle$ with probability p and state $|1\rangle$ with probability $1-p$, we see that the use of vector states, or their tensor products, is not sufficient. Hence, when we want to talk about an ensemble of quantum states, we use something called the *density operator* or *density matrix*, defined as:

$$\rho \equiv \sum_x p_x |\psi_x\rangle\langle\psi_x| \tag{2.3}$$

The density matrix has the following properties:

1. $\rho \geq 0$, it is positive semidefinite.
2. $\text{tr}(\rho) = 1$, the sum of the diagonal entries is 1.
3. $\text{rank}(\rho) = 1$, for pure states.
4. $\text{rank}(\rho) < 1$, for mixture of states.

As we can see, using density matrices we can correctly account for situations where there is a lack of knowledge on the actual state of a quantum system. It is important to remark the distinction of the statistical randomness of a mixture of quantum states, from the intrinsic randomness arising from quantum mechanical behavior, captured in the amplitudes used in the vector state representation. It is also important to notice that, given the above definition, different ensembles of quantum states can be described by the same ρ.

2.1.4 Quantum Gates

We transform and operate on the qubits by applying quantum gates. These are reversible logic gates that are used in the quantum circuits to perform a quantum computation. They are analogous to the classical logic gates used in classical circuits. However, they have a key distinction: all the quantum gates are reversible. This is to make sure that the information contained in the α and β coefficients in (2.1), i.e., the quantum information, is maintained throughout the computation. These quantum gates are usually represented as unitary matrices. For this work, the main single qubit quantum gates that we will use are the so-called Pauli and rotation gates, presented in Table 2.1.

2.1.5 Quantum Measurements

The simplest kind of measurement that can be performed on a pure quantum state is a projective measurement. For a single qubit, this can be interpreted as the probability of finding it in any of two orthogonal states. The post-measurement state would be one of these orthonormal states, i.e., a quantum state *collapses* after measurement. Recalling the normalization restriction on the α and β complex amplitudes of a single qubit, then the probability of finding $|\psi\rangle$ from Eq. (2.1) in $|0\rangle$ is simply the squared norm of the amplitude associated with it: $\|\alpha\|^2$, while it is $\|\beta\|^2$ for finding it in state $|1\rangle$.

On the other hand, there are cases where the post-measurement state is not of interest or when we want to perform a measurement on a fraction of qubits of a bigger subsystem. For these cases, the Positive Operator-Valued Measure formalism is used. We can construct a measurement operator M_k to find the probability of outcome k, so: $p(k) = \langle\psi| M_k^\dagger M_k |\psi\rangle$. If we define now a positive operator like:

$$E_k \equiv {M_k}^\dagger M_k \tag{2.4}$$

Where $\sum_k E_k = I$, then the set of E_m operators completely determines the probabilities of the k measurement outcomes.

We can apply the same measurement operator to a quantum state expressed as a density operator. In this case, the probability of obtaining outcome k is:

Tab. 2.1: Single-qubit quantum gates.

Description	Gate
Identity gate	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
NOT or bit-flip gate	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Phase-flip gate	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Bit-and-phase-flip gate	$Y = \begin{bmatrix} 1 & -i \\ i & 0 \end{bmatrix}$
Rotation around X by θ	$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
Rotation around Z by θ	$R_z(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & \exp(i\theta) \end{bmatrix}$
Rotation around Y by θ	$R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$

$$p(k) = Tr(M_k \rho {M_k}^\dagger) = Tr({M_k}^\dagger M_k \rho) = Tr(E_k \rho) \tag{2.5}$$

2.1.6 Quantum Errors

Similarly to classical information where we have different communication channels that model the loss of the information, like the binary symmetric channel or the erasure channel, there exist three main quantum channels that model the loss of quantum information:

1. **Amplitude damping channel**. This channel models the loss of quantum information that happens when a qubit loses energy do to its entanglement with the environment.
2. **Phase damping channel**. In this channel the loss of information happens when a qubit decoheres without losing energy, i.e., it dephases.
3. **Pauli channel**. It models the decoherence by approximating the combined amplitude and phase damping using the Pauli operators I, X, Y, and Z.

In this work, we will model the quantum errors using two special classes of Pauli channels, namely, the depolarizing and dephasing channels.

2.1.6.1 Quantum Depolarizing Channel

A regular Pauli channel $\mathcal{N}_P$ maps an input state $|\psi\rangle$ into a linear combination of the original state (I operator), the bit-flipped state (X operator), the phase-flipped state (Z operator), and the bit-and-phase-flipped state (Y operator):

$$\mathcal{N}_P(\rho) = (1 - p_x - p_z - p_y)I\rho I + p_x X\rho X + p_z Z\rho Z + p_y Y\rho Y \tag{2.6}$$

Where I, X, Z, and Y are the Pauli operators or Pauli gates; p_x, p_z, and p_y are the probabilities of encountering the respective Pauli errors; and ρ is the density matrix of the quantum state, which for pure quantum states is $\rho = |\psi\rangle\langle\psi|$.

The depolarizing channel, in turn, simply models the worst-case scenario by assuming that the three Pauli errors that can affect the qubit are equally likely, i.e., $p_x = p_z = p_y$. Then, the depolarizing channel $\mathcal{N}_{DPL}$ is:

$$\mathcal{N}_{DPL}(\rho) = (1 - p)\rho + \frac{p}{3}(X\rho X + Z\rho Z + Y\rho Y) \tag{2.7}$$

The depolarizing channel is hence characterized by a probability p, and it has a $\frac{p}{3}$ chance of causing a bit-flip, phase-flip, or bit-and-phase-flip error on the state ρ, and a $(1 - p)$ chance of leaving it unaffected.

2.1.6.2 Quantum Dephasing Channel

The quantum dephasing channel is a simpler Pauli channel model where the only error applied to a quantum state is a phase-flip, i.e., . It is also characterized by a probability p, and this value indicates the chance of applying a change in the base, i.e., a Z, to the state ρ. Likewise, $1 - p$ is the probability of applying the identity operator instead. The dephasing channel $\mathcal{N}_{DPH}$ is then:

$$\mathcal{N}_{DPH}(\rho) = (1 - p)\rho + p(Z\rho Z) \tag{2.8}$$

2.2 Software Testing

This section serves the purpose of being a brief introduction to the topic of Software Testing. It will address it from the Software Engineering perspective.

A discussion on what does it mean to test quantum applications and/or hardware, as well as the different approaches when testing classical vs quantum software, is deferred to the next section 3.4.

2.2.1 What Is Software Testing?

The concept of software testing has changed as much as the ideas and notions of software development have also evolved throughout the years. One of the first standards on software testing, the Standard for Software Test Documentation from the IEEE (IEEE Std 829-1983) [16] says that testing is:

> "The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items."

From the beginning, software testing was thought of as a process whose primary aim was to detect differences between the expected and observed functioning of a system, in order to make some evaluation about it. A few years later, the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990) [17] remarks that testing is:

> "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system component."

We can observe that, besides the previously identified activities, now the recording of the results was included as part of the testing process. Nowadays, testing is thought of as a an integral part of the systems development life cycle (SDLC), and the International Software Testing Qualifications Board (ISTQB) defines it as [13]:

> "Testing is the process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects."

As we can see, the modern definition of software testing involves a wider set of activities, and does not focus only on testing the system, but other related products around it. One thing that has not changed over the years is the main goal of software testing: finding defects or bugs in the system. But the analytical reader might have noticed already that the previously quoted definitions are rather vague when mentioning what are the tests actually validating. This is because there is a rather long list of characteristics or properties of a system that we might want to verify. In general, we can group these into two categories: functional and non-functional.

The tests that evaluate whether a system is behaving as per the requirements, in other words, that check *what* the system is doing is correct, are called functional tests. On the other hand, when we assess the security, reliability, performance, or maintainability of a software, i.e., *how* the system is doing it, we are conducting non-functional tests. In the rest of this work we will only focus on functional tests, so the testing of the non-functional qualities of a quantum application is left as an open problem for further research efforts.

2.2.2 Levels of Testing

Any sufficiently large and complex software system is composed of several parts, or modules, that interact and communicate between each other in order to solve a problem. Each of these modules is usually in charge of a single task and comprised of one or more subroutines that help to accomplish the task. In order to completely test a system we need to make sure that all of its components behave correctly, both in isolation and when put together. This translates into a need of identifying distinct testing levels that check the correct functioning of the subroutines, the modules, groups of modules, and finally the whole system. We will follow the categorization presented in the IEEE Software Engineering Body of Knowledge (SWEBOK) [3]: unit testing, integration testing, and system testing. It is important to mention that other standards and entities follow a similar line of reasoning when talking about testing levels.

The first level is the **unit testing**. It deals with the validation of individually testable components, or units. The answer to the question of what makes a unit of software varies from project to project, where it goes from a single subroutine or a whole module, to even a cohesive group of modules. In any case, unit tests verify the functionality of a single piece of software, in isolation from other pieces. Of course, making sure that all the components work independently does not ensure that the whole system will behave correctly, hence we need other higher levels of testing.

The next level of testing is called **integration testing**. As the name suggests, this level focuses on verifying the interaction among the previously tested units of software. It usually focuses on the interfaces between the components and the communication happening between them. One way this is accomplished is by following a systematic integration. In this approach, groups of related components are incrementally integrated, according to the specifics of the software architecture or a set of identified core functions. Then, more groups are gradually added until the whole system is tested. The other integration approach is called *big bang*, where all the components are integrated at once.

Finally, the last level of testing is the **system testing**. Here, we verify the behavior of the system as a whole and check that it meets the specified requirements. If the previous two levels of tests were executed correctly, then most of the defects and bugs of the system would have been caught by now. So, this last level of testing is better suited for assessing the non-functional requirements of the system, like security, reliability, or performance. Again, given that in this work we are focusing on functional tests, the two levels that will have more prevalence are unit and integration tests.

2.2.3 Testing Techniques

We have identified the distinct levels of testing needed in so we can properly verify the functionality of a software system. Furthermore, there is also a set of testing techniques that can be employed in order to perform the required tests. These techniques are based on how the test cases are generated, and in general are grouped in three main categories that cover a spectrum of the amount of information about the internal workings of a system used: white-box, black-box and gray-box testing.

On one side we have **white-box testing**, also called glass-box or structural testing. This is a testing approach that verifies the internal workings of a component, based on information about its design and implementation. It usually takes a look at the source code in order to design test cases that exercise the different logical paths of the system. This approach is more suited for performing tests at the unit level, as the communication and integration of components is often only available in specification documents, and not in the form of code. One limitation from this technique is that missing requirements, or even unimplemented functions, are difficult to catch, as they are not present in the source code of the software.

On the other side of the spectrum we find the **black-box testing**. This technique relies only on the specification of the requirements of a system, and does not look at how a component is implemented. It works by giving the component an input, observing the output, and comparing it against the expected one. It is suited for performing checks across all the testing levels, but it is particularly useful for integration and system tests. For example, trying to cover all paths that a specific piece of data follows, by using a white-box testing approach, is usually infeasible due to the sheer volume of options and combinations. However, the final value of the data has usually a known or expected value, which can then be tested using the black-box approach. Then, by changing the input provided, we can test other relevant or important paths without having to enumerate all of them. The disadvantage of this approach is, of course, that some parts of the system might never be tested or that they might be unnecessarily over tested.

Standing in the middle we have the **gray-box testing**. This is, as expected, a combination of both black-box and white-box testing techniques. Here, the test cases are designed with partial information about the internal workings of a system, for instance the data structures or algorithms used, in order to derive better suited tests. Then, they are run in a black-box fashion, where the input is also better chosen given the knowledge about the underlying structure of the system. So, gray-box testing can alleviate some of the shortcomings of the black-box testing technique while requiring less information than white-box testing.

2.2.4 Test Automation

The topic of test automation is so broad that it would deserve its own section. Since we are, however, limited in space, in this subsection we will focus on answering those questions relevant the present work: what is test automation, why is it needed, what are the benefits and drawbacks of automating tests, and finally what is a test automation framework. For a more complete treatise on the subject, we refer the reader to [8].

Automating a test means to translate it into a form, usually as code, so that the test can be automatically executed as if it was performed by a human tester. It involves the automation of all the actions taken by the tester and all the elements needed by the test. So, it includes: the set up of the System Under Test (SUT), the set up of the preconditions, exercising the SUT, the observation and collection of the output, and the verification against the expected output. While the term test automation might be thought of as a type of testing, it is better described as a separate activity or a separate skill, needed in order to improve the testing of a system. In essence, performing a test manually or in an automated fashion should have no impact on its effectiveness, i.e., the capacity of the test to detect defects in the software.

The main reason behind test automation is an economic one. It is generally agreed that automating a test costs more, in terms of time and effort, than running it manually. However, if the test ought to be run repeatedly, it is actually more cost-effective to automate it than to run it manually. Another advantage of the automation is the ability to test scenarios or cases that are infeasible with a manual approach, for example: subtle changes in a graphical user interface, or the simulation of the input from hundreds of users. Other benefits obtained when automating are: consistency and repeatability of the tests, the reduction of human errors, and the ease of performing cross-platform testing.

On the other hand, there are some risks and caveats that we need to consider before immediately jumping into the automation bandwagon. The most important one is that we should only automate good tests. With good tests we mean those that have been proven to exercise the software to cover all the requirements. Remember that automation makes us be able to perform controlled and fast repeated checks, but does not change the qualities of the original test executed. Now, since automating a test usually means to create software that test other software, we need ways to make sure that the testing code is also working correctly. Furthermore, the maintainability of the testing software is crucial for being able to reuse and expand the tests, since otherwise the pay-off of the automation is diminished. Lastly, it is important to mention that having tests that run automatically and do not fail, does not necessarily mean that our system has no defects. If there is a change in the requirements, or the automated tests do not have the sufficient coverage, then we would get a false sense of confidence.

Finally, in order to answer the last question, i.e., what is a Test Automation Framework (TAF), we need to know first what a testing framework is. According to Ghanakota [11], a testing framework is a set of assumptions, concepts, practices, and rules that form a platform for developing tests. Operationally, the main responsibilities of a testing framework are: providing a common way of expressing the expectations, creating a mechanism to run or drive the system under test, executing the actual tests, and reporting the results. When possible, the framework should be application-agnostic, easily expandable, and easily maintainable. So, a test automation framework should cover all of the mentioned responsibilities and qualities of a testing framework, but with a focus on providing a suitable mechanism for automating the execution and reporting of the tests.

3 Testing Quantum Applications

At the heart of quantum mechanics is a rule that sometimes governs politicians or CEOs — as long as no one is watching, anything goes.

— **Lawrence M. Krauss**
Theoretical physicist

If you were to give a talk with the title "Quantum Computing: Testing", in a quantum computing related forum or conference, there would be at least three different themes that people would think about when reading the title of the presentation. If the people have a background in the design of quantum computing platforms, they might think about the verification of the quantum hardware, asking how to certify if a quantum device is functioning accurately. If, on the other hand, the people are more related to the computer science area, they might think instead of the validation of a quantum computation. They would probably relate the topic to how to test if a computation was performed in an actual quantum computer, or how to verify a quantum computation if the problem solved is not classically tractable. Lastly, another small percentage of the people might think about it from the software engineering point of view, thinking on how to test quantum applications and find bugs on them.

This work tries to provide a better understanding on how to test quantum applications, from the last mentioned point of view, i.e., from the software engineering perspective. So, in section 3.4 we discuss what does it mean to test quantum software, and how much of the classical software testing knowledge presented in the previous chapter, can we apply to the testing of quantum applications. Still, we know that we need to account for the deficiencies and particularities of the quantum hardware, as these quantum applications are currently run in the so-called Noisy Intermediate-Scale Quantum (NISQ) machines. Hence, in section 3.1, we also explore the techniques available for verifying the accurate functioning of a quantum device, and how to incorporate them into the testing process. For completeness, in section 3.2, we also include a brief review on the verification of quantum computations from the computer science perspective.

3.1 Verification of Quantum Hardware

Generally speaking, when we verify a quantum device by testing the accuracy of its output, we say that we certify the device. When certifying a quantum device, we usually want to compare it against another one, or to a different version or configuration of the same device. For this, we have developed the notion of benchmarks, which are certification methods that assign a common performance measure that is reproducible and allows us to do the comparison. These benchmarking methods, also called certification protocols, are limited by the very same feature that makes a quantum computer more powerful than its classical counterpart: its ability to perform computations on exponentially larger state spaces.

The benchmarking tools available range from the full blown quantum state tomography [14] that completely reconstructs a quantum state by performing measurements on several copies of the same state, to the fidelity estimation technique [9] that gives an estimate of the fidelity between an ideal quantum state and the actual state produced. In [7], Eisert et al. provide a comprehensive account of the certification protocols currently available. Figure 3.1 shows the classification they do of the tools, based on the amount of information gained, and how weak or strong are the assumptions made about the quantum device.

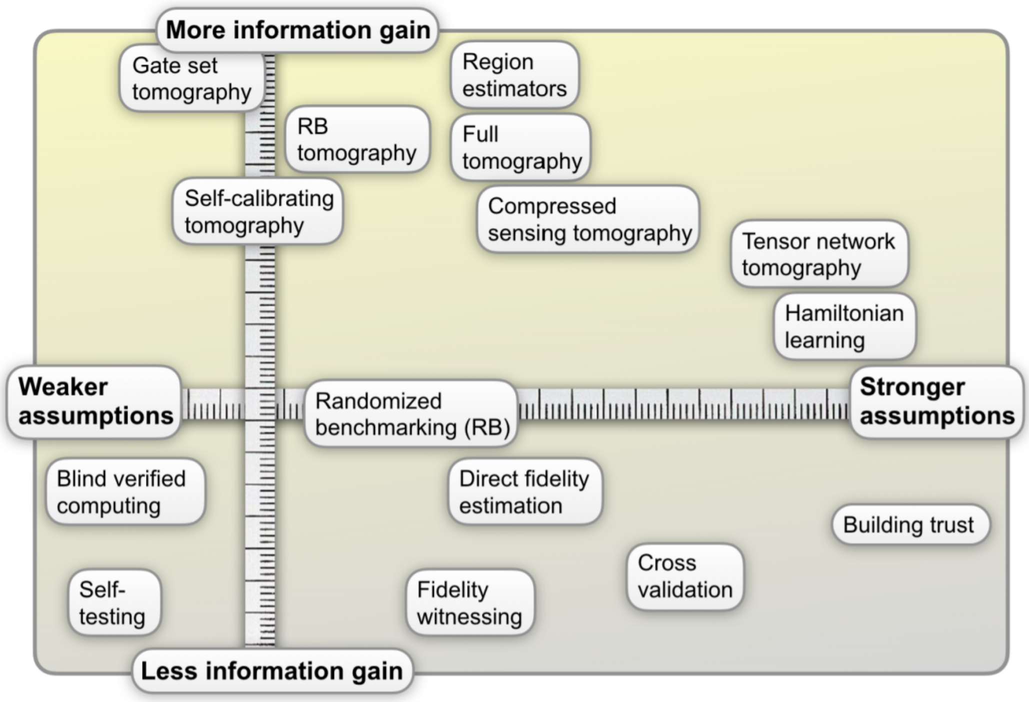


Fig. 3.1: Classification of some of the quantum certification protocols currently available. Taken from [7].

Another important characteristic of a benchmark is how complex it is, in terms of classical and quantum computational effort. Usually, when a certification protocol offers more information gain, then it also requires more time and power to compute. If we want to test quantum applications in practice, then we would need to choose those tools that provide a good trade off between the information gain and the computation complexity.

Note also that almost all of the benchmarks shown in Figure 3.1 make use of the same measures of quality for comparing a target ideal density matrix (ρ_1) with the experimentally or physically created one (ρ_2): the Trace Distance $D(\rho_1, \rho_2)$, and the Fidelity $F(\rho_1, \rho_2)$. A complete explanation of these metrics is found in [24].

The benchmarks so far discussed are all used for gaining confidence about the quality of the quantum hardware, as they give us a comparable and reproducible measure of its performance. However, when testing quantum applications, we often take this for granted. In other words, before the tests are run, we already trust that the quantum device is performing as accurately as possible, and we go on to verifying that the quantum application functions correctly. Hence, due to the computational complexity of the protocols and the fact that they focus on the verification of the accuracy of the device, we will not make direct use of any of the benchmarks.

3.2 Verification of Quantum Computations

If the benchmarking tools from above are used for certifying the accuracy of trusted devices, the methods discussed in this section then focus on the verification of the computation performed on an untrusted quantum computer. The techniques outlined here focus on answering the following question: how do we check that a quantum computer produces the right results, when the problem, and even sometimes the verification of the solution, is considered to be intractable in a classical computer. In order to answer the question, first we need to formulate it in more complexity theoretical terms. Scott Aaronson, in his renowned quantum computing blog, puts it like this [1]:

> “If a quantum computer can efficiently solve a problem, can it also efficiently convince an observer that the solution is correct? More formally, does every language in the class of quantumly tractable problems (BQP) admit an interactive proof where the prover is in BQP and the verifier is in the class of classically tractable problems (BPP)?”

The key idea is to bring the question into the realm of the interactive proof (IP) systems. An interactive proof system is simply a model that consists of two entities: a verifier and a prover. In this case, the verifier is a machine in the Bounded-error Probabilistic Polynomial time (BPP) class, the class of decision problems that can be solved efficiently by a probabilistic classical computer. The prover is in the Bounded-error Quantum Polynomial time (BQP) class, the quantum equivalent of BPP, and is considered more powerful than the verifier. The verifier wants to check a reported solution, so he interacts with the prover by exchanging messages, until the verifier is convinced of the solution is correct. For quantum interactive proof systems, the verifier should accept or reject a valid solution with high probability.

This remained an open question since its conception, until in 2018 Mahadev proposed the first protocol allowing the verification of quantum computations through purely classical means [22]. Before that, some weaker formulations had already been solved, like the case where the verifier has access to a quantum computer limited in power, or where the verifier interacts with a pair of entangled quantum provers. Another recent key result is from Ji et al., where it is shown that $\mathtt{MIP}^* = \mathtt{RE}$, i.e., the class of languages that can be decided by a classical verifier interacting with multiple all-powerful quantum provers ($\mathtt{MIP}^*$) that share entanglement, is equal to the class of recursively enumerable languages ($\mathtt{RE}$) [18].

3.3 Verification of Quantum Applications

The topic of testing quantum programs and quantum applications from a software engineering perspective has also been researched already. The most explored theme is that of the verification of quantum programming languages, which aims to verify a quantum application by performing an analysis of the semantics of the source code. Here, a collection of formal methods have been developed for validating that a quantum program is able to produce an expected result, without actually running the application. An excellent account of this topic is found in this PhD thesis from Robert Rand [26].

A common understanding derived from the works on quantum programming formal verification techniques, is that a quantum program is necessarily a probabilistic program. Then, the knowledge so far acquired in regards to the verification of classical probabilistic systems, can be used as a basis for the development of new methods for verifying quantum applications. An example of this is in [32], where Ying et al. model quantum programs as quantum Markov chains, in order to be able to prove some properties about them.

On the other hand, we also have works that have researched the requirements for testing quantum applications looking at some more typical classical software testing topics. For instance, in [23], Miranskyy and Zhang discuss how to define a set of software engineering practices that can be applied to the testing of quantum software. They explore the applicability of the software testing approaches like black-box and white-box testing, illustrate the difficulties associated with them, and outline some potential solutions. Furthermore in [15], Huang and Martonosi classify the type of errors that a programmer can incur in, when creating a quantum program.

In both of the works mentioned above, the debugging of quantum applications is found to be considerably more challenging than the debugging of classical programs. This is due to two main factors: the no-cloning theorem that stop us from creating copies of a qubit, and the fact that measuring a qubit ultimately destroys the quantum information it contains. Hence, it is not possible to debug a quantum program interactively as is usually done for classical applications.

3.4 Testing Classical vs. Testing Quantum

Here, we summarize the particularities and difficulties about testing quantum applications identified in the previous sections, and discuss how they can be approached and solved bye the software testing topics outlined in section 2.2. Then, we arrive at a first set of general requirements that our test automation framework for quantum applications should meet.

3.4.1 Probabilistic Nature of Quantum Tests

The most important difference between a quantum test and a classical test, is that the first one is a probabilistic process. This probability comes from the intrinsic probabilistic nature of quantum mechanics. We observe it in the decoherence of a qubit, in the outcomes of a quantum measurement, or in the errors introduced by the application of noisy quantum gates. The fact that the testing process is a probabilistic process, means that we need to perform several repetitions of a test, in order to arrive at a conclusion. Thus, the outcome of a quantum test is a probability distribution obtained from the observed outcomes of all the repetitions executed.

This also implies that, in order to know whether a test passed or failed, the expected outcome of a test needs to be expressed as a probability distribution too. Comparing a single value or realization with a probability distribution is almost meaningless. Hence, the verdict about the final result of a quantum test involves the comparison of two probability distributions. In the next chapter, we will see how to model the testing of a quantum application as a probabilistic process, and how to compare two distributions using the statistical inference technique known as Hypothesis Testing.

3.4.2 Debugging of Quantum Applications

When testing a system, we do not only want to identify if there is an issue or not, but also find out the root cause of the problem, if it exists. This is referred to as *debugging* or *triaging*, and as discussed in section 3.3, doing it for quantum applications is harder than for classical ones. This is due to the no-cloning theorem and the destructive nature of quantum measurements.

The debugging process starts with making sure that the reported failure is not the result of an issue in the test definition or the testing tools used, but a real problem in the quantum application being tested. For classical applications we have a set of procedures we can follow in order to discard this source of errors. We may, or may not, be able to apply or adapt these for the debugging of quantum applications.

The first thing we usually do is simply repeat the failing test case. By doing this, we can get rid of probable temporary or intermittent issues that were present in the system when the test was run. Unfortunately, repeating a test of a quantum application translates into making hundreds, if not thousands, of executions. Depending on the type of test, or the availability of the quantum device, this might not be possible to do.

If we are able to repeat the test, the next step is to compare the failed execution with the new one. If the test fails again, we can try to compare the test execution logs to find the reason for it. When repeating the test is not possible, we can also compare the failed test case against a similar one, or to a previous run of the same test. All of these approaches assume that the test tools used are capable of generating the given execution logs. More importantly, logs from the quantum operations actually executed by the quantum

A different approach for the debugging would be to execute the same test in a simulated environment, that closely resembles the real physical one. The effectiveness of this is determined by how well characterized the quantum device is, and how well we can simulate it. From section 3.1 we know that fully determining the quality of the quantum hardware is computationally hard. Additionally, the simulation is not always a viable option, as we know that some quantum applications are classically intractable and so impossible to be simulated efficiently. Then, we can resort to performing a formal validation of the program with the methods described in section 3.2. However, it is not clear how to compare the results obtained using these techniques, with the ones from the failed test.

Hypothesis Testing

4

In practical life we are compelled to follow what is most probable; in speculative thought we are compelled to follow truth.

— **Baruch Spinoza**
Philosopher

4.1 Modeling the Testing Process

Suppose we are given a quantum black-box that receives as input the description of a quantum state, for example "$|0\rangle$" or "$|+\rangle$", prepares the specified qubit, and then measures it in the Z basis, giving as output a description of the measurement outcome: "0" or "1", depending on whether a $|0\rangle$ or a $|1\rangle$ was found. If we want to test this device, we just need to provide a known input and compare the resulting output with the expected one. Following our example, giving a "$|1\rangle$" as input, we would expect a "1" as output. However, as we have seen in section 3, this is not so simple. Because of factors like imperfect measurements and the inherent decoherence of the qubits, even if the quantum state is correctly prepared, we would sometimes see a wrong output. Likewise, the same factors could make a wrongly prepared quantum state be measured and give the right answer. This means that, in order to derive a conclusion, we need to test the quantum black-box more than once.

Lets say we apply 10 times the previous test of giving a "$|1\rangle$" as input, and we get as output "1" eight times, and "$+0$" two times. If we mark a successful test with a `1` and a failed test with a `0`, and assume that the test realizations are completely independent, then we have a Bernoulli process composed of 10 Bernoulli trials. This means that each test realization can be thought of as a random variable that takes on the values `1` and `0` with a probability of success p.

$$X = \begin{cases} \texttt{1} & \text{with Probability} = p \\ \texttt{0} & \text{with Probability} = 1 - p \end{cases} \tag{4.1}$$

We can make an estimation of the p parameter of a Bernoulli process by taking the ratio between the number of successes and the total number of trials. In the case of our example, the proportion of successes would be $8/10$ or 80%. Of course, if we repeat the same test another 10 times, we might get a different outcome. So, the question is now how likely are we to actually obtain a different result, or in other words, what is the *real* probability of success p of the analyzed Bernoulli process. Assuming that each of the test executions is independent of each other and has the same probability of success, a series of Bernoulli trials can be characterized by a Binomial distribution, usually denoted by $B(n, p)$. The Probability Mass Function (PMF) of the Binomial distribution is:

$$f(k, n, p) = \Pr[k; n, p] = \Pr[X = k] = \binom{n}{k} p^k (1-p)^{n-k} \tag{4.2}$$

The PMF of the Binomial distribution gives us the discrete probability of getting exactly k successes in n independent Bernoulli experiments. Then, the problem of comparing an expected output with an obtained one, translates into the problem of comparing two probability distributions. In order to know if the quantum box that we are testing is functioning correctly, we need a way of comparing the distribution of the series of test realizations with the expected distribution. For this, we will use a method of statistical inference called hypothesis testing.

4.2 Hypothesis Testing

Hypothesis testing is a statistical inference technique for testing a claim or hypothesis about a parameter in a population using data measured in a sample, i.e., it is a systematic way of knowing whether a group or population matches a certain criteria. The method of hypothesis testing consists of four steps. The steps are now explained by applying them to our example.

4.2.1 Step 1. State the Hypothesis

The first step is to identify the claim or hypothesis that we would like to test. In our case, we want to know whether the proportion of successful tests is the one we expect, according to the specifics of the quantum box being tested. Say the claim is that the quantum black box can produce a $|1\rangle$ state and measure it in the Z basis, with a fidelity of 0.98. Then, we would expect the proportion of correct tests to be around 98%. More specifically, we would like to know if the ratio is actually at least 0.98, since it is fine if the fidelity is better, but not if it is worst. Following the hypothesis testing technique we start by stating a ***null hypothesis***, denoted by H_0, which is something presumed to be true. In this case our null hypothesis would be that the proportion of correct tests is at least 98% or, in other words, that the probability of success p of the testing procedure is ≥ 0.98. On the other hand, in the ***alternative hypothesis***, or H_1, we directly contradict the null hypothesis and state what we think it is wrong about it. So, our null and alternative hypothesis are:

$$\begin{aligned} H_0 &: p \geq 0.98 \\ H_1 &: p < 0.98 \end{aligned} \tag{4.3}$$

4.2.2 Step 2. Collect Data and Calculate a *Test Statistic*

The next step is to perform measurements from a sample to collect data and define a test statistic to use. A test statistic is, in essence, a mathematical formula that allows us to summarize the data collected. This will be used as the evidence for the test. A natural test statistic to use is the sum of the outcomes, simply expressed as $\sum_{i=1}^{n} x_i$ for a series of Bernoulli random variables. Another option of test statistic is the mean number of successes, expressed like $\frac{1}{n}\sum_{i=1}^{n} x_i$, when a success is marked with `1` and a fail with `0`, as we are doing. For our example we will use the first one. We will then imagine that we performed 100 tests on the quantum black box, of which 93 were successes, so $\sum_{i=1}^{n} x_i = 93$.

4.2.3 Step 3. Compute the p-value

Once we have collected the data, we can proceed to answer the question: is the evidence (the test statistic) good enough to reject the H_0? This is accomplished by calculating the p-value. The p-value is defined as the likelihood of obtaining an outcome, given that the null hypothesis is true. Then, in order to make a decision, we compare the p-value against something that quantifies the notion of a *good enough* evidence. This is called the **level of significance** of the test, denoted by α, and it tells us when we are going to accept or reject the null hypothesis. For example, a typical $\alpha = 0.05$ level of significance means that when the probability of obtaining the test statistic measured in a sample is less than 5%, i.e. p-value $< \alpha$, then we can conclude that the outcome is too unlikely, and consequently we reject H_0.

In the previous subsection 4.1 we have already established that for a series of Bernoulli random variables, the probability of seeing exactly k successes is given by the Probability Mass Function (PMF) of the Binomial distribution. Then, using equation (4.2) we can calculate the probability of obtaining the observed outcome given that the null hypothesis H_0 is true, by setting $p = 0.98$. Following our example, knowing that $k = 93$, we calculate the p-value as:

$$\begin{aligned} p\text{-value} := f(93, 100, 0.98) &= \Pr(X = 93) \\ &= \binom{100}{93} 0.98^{93} (1 - 0.98)^{100-93} \\ &= \binom{100}{93} 0.98^{93} (0.02^{7}) \\ &\approx 0.00313 \end{aligned} \tag{4.4}$$

4.2.4 Step 4. Make a Verdict

We can now proceed to make a verdict about the null hypothesis. If the p-value is not sufficiently small, in other words, if p-value $\geq \alpha$, then the data is consistent with the null hypothesis and we *fail to reject* H_0. The expression *fail to reject* H_0 can be a bit misleading, yet it is used to make clear the notion that when not enough evidence has been gathered against H_0, we can safely say that H_0 is not rejected, but not that H_0 is *accepted*. On the other hand, when p-value $< \alpha$, we do have strong evidence against the null hypothesis, so we can safely say that we reject H_0 in favor of H_1. Hence, given that the p-value of our example is $\approx 0.0.003$, if the significance level is set at $\alpha = 0.05$, we conclude that the data is not consistent with the null hypothesis and we reject H_0.

It is important to mention that, whenever we perform a hypothesis test, we can incur in errors when deriving the conclusion of rejecting or not the null hypothesis. The first error happens if we reject the null hypothesis when it was actually true, this is called a *false negative* or `TYPE I` error. The maximum probability of committing this type of error is α, while $1-\alpha$ is called the confidence level of the test. Conversely, we can also accept the null hypothesis when it was actually false, this is a *false positive* or `TYPE II` error. The maximum probability of committing this type of error is β, and $1 - \beta$ is the power of the hypothesis test. When using the hypothesis testing technique, it is important to set suitable values for both the α and β, depending on the objective of the test and the context or domain of the problem. Table 4.1 shows a summary of these errors.

The α and β parameters are closely related. In general, if we want to increase the power of the test, i.e., reduce β, we need to increase the parameter α. In our case, we want the power of the testing procedure to be as high as possible, or equivalently, the parameter β to be as close to 0 as possible, since we do not want to wrongly conclude that the proportion of successes is above the expected one. It is worth noting that another way of increasing the power of a hypothesis test is by increasing the number of samples used for the test.

Tab. 4.1: Types of errors for the Hypothesis Testing.

		Decision	
		Retain the null hypothesis	Reject the null hypothesis
Truth in the population	True	Correct $1-\alpha$	`TYPE I` error α
	False	`TYPE II` error β	Correct $1-\beta$

4.3 Hypothesis Testing in Practice

In the previous section we outlined the theoretical background and formal procedure that needs to be followed for using the hypothesis testing technique. The first step of the process is to select a null and an alternative hypothesis. For our purposes, the null hypothesis contains the information of the expected or desired probability distribution. And the evidence for its rejection comes from the observed data, i.e., the test outcomes. However, the same objective of comparing two probability distributions via the hypothesis testing technique can be accomplished by using other functions of the Binomial distribution.

4.3.1 The Binomial Inverse Survival Function

First, we need to introduce the Binomial Cumulative Distribution Function (CDF), $F(k,n,p)$, which indicates the probability that the Bernoulli random variable X will take a value less than or equal to k:

$$F(k,n,p) = \Pr[X \leq k] = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i} \tag{4.5}$$

Where $\lfloor k \rfloor$ means the greatest integer less than or equal to k. Furthermore, we can also define the Survival Function (SF) of the Binomial distribution, which is in turn the probability that the random variable will take a value strictly larger than k:

$$S(k,n,p) = 1 - F(k,n,p) = \Pr[X > k] \tag{4.6}$$

Finally, we arrive at the Inverse Survival Function (ISF) of a Binomial distribution:

$$Z(\alpha, n, p) = S^{-1}(\alpha, n, p) \tag{4.7}$$

This Inverse Survival Function returns the smallest non-negative integer k for which $F(k, n, p) \geq 1 - \alpha$, or conversely the smallest non-negative integer k for which $S(k, n, p) \leq \alpha$. Is in this last interpretation where we can notice the relation between the hypothesis testing process and the Binomial ISF. Note that the α of the $Z(\alpha, n, p)$ has a similar meaning as the significance level chosen for the hypothesis testing: it is the probability of observing an outcome. Then, using the Binomial ISF we can get a number k_{min} that represents the minimum expected number of successful outcomes, that guarantees that the probability of observing more than k_{min} successes is $> \alpha$, which is what the step 3 of the hypothesis testing technique does.

We will exemplify the use of the ISF with the same previous test of the preparation of the $|1\rangle$ quantum state. In this case, suppose we performed 1000 repetitions of the test, and that the expected fidelity is still 0.98. Also, the same significance level of $\alpha = 0.05$ is chosen for the hypothesis test. Using the Binomial ISF, we can get a minimum number of expected successful outcomes k_{min} that is consistent with the null hypothesis:

$$\begin{aligned} k_{min} &= Z(\alpha, n, p) \\ &= Z(0.05, 1000, 0.98) \\ &= 987 \end{aligned} \tag{4.8}$$

The value $k_{min} = 987$ is the minimum expected number of successful outcomes, where a significance level of 0.05 and a population of 1000 outcomes is chosen. Then, for deriving a conclusion about the null hypothesis, i.e., whether the proportion of actual observed outcomes is ≥ 0.98 or not, we simply count the number of observed successful outcomes, k_{obs}, and compare it against the minimum expected. If $k_{obs} \geq k_{min}$, we *fail to reject* the null hypothesis, otherwise, if $k_{obs} < k_{min}$ we have sufficient evidence to reject the null hypothesis.

4.3.2 The Binomial Proportion Confidence Interval

Another approach for performing the hypothesis testing is to calculate the binomial proportion confidence interval from the observed outcomes. This is an interval estimate of the probability of success of a Binomial distribution. Remember that in chapter 3 we recognized that when testing quantum states in superposition, we would need to specify the expected proportion of successes as a range. Hence, this alternative approach can help us to compare the specified expected interval with the estimated one obtained from the test outcomes.

Since the Binomial distribution is a discrete probability distribution, it is computationally hard to calculate when the number of trials is large. In consequence, when calculating the confidence interval, we actually only approximate the real Binomial distribution using different techniques. Each of these vary in accuracy and computational intensity. For a detailed comparison we refer the reader to [30], where Vollset compares 13 methods for computing binomial confidence intervals. We list here the most used approximation techniques: the asymptotic normal approximation, the Wilson score interval, the Jeffreys interval, the Clopper-Pearson interval, and the Agresti-Coull interval.

Irrespective of the approximation method chosen, the calculation of a binomial proportion confidence interval requires only the following information: the number of trials run, the number of successes observed, and a confidence level. The confidence level of the interval is how certain we want to be that the probability of success of the Binomial distribution actually lies within the calculated interval. We illustrate the usage of the binomial proportion confidence interval with an example.

Imagine we have a test that verifies the preparation and measurement of a qubit in superposition. Specifically, the $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ quantum state is prepared, and it is measured in the computational basis. We know that in a perfect setting, if we measure $|+\rangle$ in the Z basis we have an equal probability of having a `0` or a `1` as output, as half of the times the measurement would find the state in $|0\rangle$, and the other half in $|1\rangle$. But since there is no such thing as a perfect quantum device, instead of a single value like $p = 0.5$, we are given a range for the probability of observing a $|0\rangle$: $p = [0.48, 0.55]$, with a confidence of 95%.

Say that after running the test 1000 times, the outcome is `0:511` and `1:489`, meaning that the output was `0` in 511 occasions, and `1` the rest of the time. Since we were given the range for the probability of the `0` outcomes, the number of observed successes is 511. Note that we could also have been given an interval for the `1` outputs, in which case we would say that the number of successes is 489. Given the provided confidence level of 95%, we know that the significance level of the test needs to be $\alpha = 0.05$, as confidence = $1 - \alpha$. Then, using the normal approximation, the binomial proportion confidence interval is calculated to be $[0.480017, 0.541982]$. Since this estimated interval calculated from the test outcomes, lies within the expected one and has the same confidence, we can say that the test passes.

5 A Test Automation Framework for Quantum Applications

Without ambition one starts nothing. Without work one finishes nothing. The prize will not be sent to you. You have to win it.

— **Ralph Waldo Emerson**
Essayist, philosopher and poet

In chapter 2 we discussed the general responsibilities that any test automation framework should fulfill. There, also, we identified that the main advantage behind the automation of a test was being able to run it repeatedly, in a controlled fashion. Later, in chapter 3, we compared the requirements for testing classical and quantum programs, where the probabilistic nature of quantum phenomena was the key difference between the too. Putting this together, it is not difficult to conclude that the only feasible solution for testing quantum applications is the use of a test automation framework. This framework should allow us to perform several repetitions of a single test, gather the results, and perform the necessary statistical checks on them, to conclude whether the test passed or failed. This chapter presents QuTAF, a test automation framework for verifying quantum applications. In section 5.1 we describe the requirements for the QuTAF, while in sections 5.2 and 5.3 we detail its design and implementation, respectively.

5.1 Requirements

We have seen that the main responsibilities of a classical test automation framework are: providing a common language for defining the tests, driving the System Under Test (SUT), executing the tests, and reporting the results. We derive the requirements for our automation framework from this set of operational responsibilities, dealing with the particularities of testing quantum applications outlined in section 3.4. It is also important to mention that the proposed test automation framework is designed with two main use cases in mind:

A) **Testing a networked quantum node.** Our framework will be used for the verification of the Network Operating System (NOS) developed for a networked quantum node. The NOS software stack includes all the components that provide the usual functionalities required from any networked system, like scheduling, routing, memory management, and others. These are, of course, all classical software components that can be tested with any classical test automation tool. Hence, the QuTAF will be used, instead, for testing the integration between the classical software and the quantum hardware, by verifying that small quantum applications are run correctly.

B) **Testing simulated fail scenarios.** Here we will validate the capabilities of the QuTAF itself, by testing quantum applications that are purposefully simulated to fail. These fail scenarios simulate problems coming from the quantum hardware, as well as issues introduced in the quantum programs by the programmers.

From this, we can conclude that one of the main requirements for the QuTAF is that it should work for the verification of quantum programs running on both simulated and real quantum nodes.

The first requirement actually coming from the mentioned responsibilities, is that the automation framework must provide a standard way of defining the tests. This means that the tester using the TAF, should be able to write test cases using a common set of rules and terms, so that the tests are easily understandable and maintainable. For QuTAF, however, the two use cases mentioned above add more demands for this test specification language.

In use case A), for example, we run quantum applications to verify the integration and communication between the classical software stack and the quantum device. In this case, the quantum program to run is actually defined as part of test case itself. Furthermore, the same test definition language will be used for specifying the failing test scenarios in B). Thus, the requirement for the QuTAF is that it should let us easily specify small quantum circuits, along with the pertinent checks and verifications associated to them. Crucially, it should provide a way for specifying the expected result of a test as a probability distribution. At the end, as we know, we will compare this distribution with the one calculated from the test execution.

This takes us to the next responsibility of a classical TAF: driving the SUT and executing the tests. For the case of testing quantum applications, there are actually no extra demands to fulfil this requirement. This is not surprising if we take into account that any quantum node is already controlled by a classical system that executes the quantum gates and measurements on the physical qubits. Therefore, there is no difference between driving a classical system directly, or a quantum system through a classical interface. For the QuTAF in particular, in A) we will drive the SUT via an API exposed by the NOS. Additionally, we can safely assume that a similar method will be available for B), as the majority of quantum simulation tools provide an API for simulating full quantum circuits, or individual quantum operations.

Now, executing the tests also implies comparing the obtained output with an expected one, and in section 3.4 we saw that due to the probabilistic nature of the quantum tests, this comparison is not trivial. In fact, the result of a quantum test run is a probability distribution, instead of a simple integer or string value. So, we selected the statistical hypothesis testing technique as the solution for comparing the obtained probability distribution with an expected one. The use of the hypothesis testing technique introduces the following requirements for the QuTAF: being able to execute the test cases for a configurable number of times, gather the outcomes, and calculate their probability distribution.

Finally, for a better understanding of the test results, the report generated by our QuTAF should include not only the final decision of whether a test passed or failed, but also: the probability distribution calculated from the outcomes, the actual number of times that the test was executed, and the significance level (α) chosen for the hypothesis testing.

5.2 Design

From the previous section we can conclude that, although there are indeed some extra requirements when testing quantum applications, the main functionalities of our QuTAF will actually be those of a regular or classical test automation framework. This means that we can look into the list of existing test automation softwares, and choose one to serve as the basis for our QuTAF. These solutions are flexible and extensible enough to cover diverse testing needs, however, there does not exist one that can cover the testing requirements of all possible software systems. So, during the design phase, we put the requirements from section 5.1 into more technical terms, so that we can find the test automation solution that best fit the requirements.

As usual, the first question to answer is how are we going to define our test cases. For classical software, this refers to the programming languages and testing approaches supported by the test automation software: data-driven, keyword-driven, or behavior-driven. In our case, the requirement of being able to specify a quantum circuit as part of the test case, can be fulfilled by a TAF that employs either the keyword-driven or the behavior-driven approach. Using the keyword-driven approach, we can define keywords for each quantum operation, and then reuse them in all the test cases, as shown in Figure 5.1. On the other hand, using the behavior-driven approach, we can provide a description of the quantum circuit in plain text, as depicted in Figure 5.2.

```
# Keywords
Initialize Qubit  qubit=<int>
Apply Gate        qubit=<int>  gate=['X','Y','Z','H']
Measure Qubit     qubit=<int>
Verify Outcome    outcome=[0, 1]

# Test Case
Initialize Qubit  qubit=0
Apply Gate        qubit=0  gate='X'
Measure Qubit     qubit=0
Verify Outcome    outcome=1
```

Fig. 5.1: Example of a quantum program defined and tested using a keyword-driven approach.

```
Given that I initialize qubit 0
  And I apply the X gate to qubit 0
  And I measure qubit 0
 When I run this quantum program
 Then I expect to observe the outcome: 1
```

Fig. 5.2: Example of a quantum program defined and tested using a behavior-driven approach.

The next question is about the actual execution of the tests. From the requirements, we see that the test runner of the underlying TAF should be easily configurable. This is because the test cases will specify the number of repetitions to perform, the parameters of the expected probability distribution, and the hypothesis testing settings selected. The test runner must also calculate the probability distribution of the observed outcomes and use the hypothesis testing technique for comparing it against the expected probability distribution. Explicitly, it ought to be able to calculate both the Binomial Inverse Survival Function (ISF) described in section 4.3.1, and the proportion confidence interval outlined in section 4.3.2.

Now, given that none of the available TAFs is actually prepared to simulate quantum applications out of the box, we need to pick a separate tool that takes care of this task. This is complicated by the fact that the number of quantum simulation platforms is probably as high as the number of existing test automation solutions. Luckily, the requirements from the failing test scenarios of B), described in detail in section 6.2, help us prune the list.

The most important requirement is that the chosen software should be able to simulate noisy qubits and imperfect quantum operations. Talking about the quantum gates, it should be able to apply the two most common quantum error models: the quantum depolarizing channel and the quantum dephasing channel. For the case of the qubits, it should allow to apply error models that incorporate relevant quantum physical parameters like the T_1 and T_2^* times. Finally, we would like the simulation software to easily interface with the selected test automation solution.

The last question is how are the test results going to be presented. Almost all the test automation softwares already incorporate one or more ways of reporting the test results. But, for the reports from our QuTAF, we want to include relevant information from the test cases, like the parameters chosen, or the calculated probability distribution. Hence, the test automation software should provide an API or a way of personalizing and extending the final test reports.

5.3 Implementation

Given all the requirements and design considerations from above, we opted for using the `robot framework` [10] as the test automation solution that would be the basis for the implementation of our QuTAF. Also, we chose `NetSquid` [4] as the tool to perform the classical quantum simulations for use case B). In this section, we first give a brief description of both the `robot framework` and the `NetSquid` tool, along with the reasons for choosing them. Later, the overall architecture of the solution is set forth. We end with an example of how a qubit initialization test case is defined.

5.3.1 Robot Framework

The `robot framework` is an open source test automation framework, initially developed at Nokia Networks, that has been successfully used for the testing of a variety of software projects. It is an extensible keyword-driven automation framework based in Python, with a rich ecosystem consisting of libraries and tools developed as separate projects by the community [10]. The framework works by processing the test data, executing the corresponding test cases, and generating logs and reports. In general, the `robot framework` has no information about the System Under Test (SUT), and the interaction is handled entirely by the test libraries, which drive the SUT either directly or by means of test tools. The modular architecture of the `robot framework` is shown in Figure 5.3.

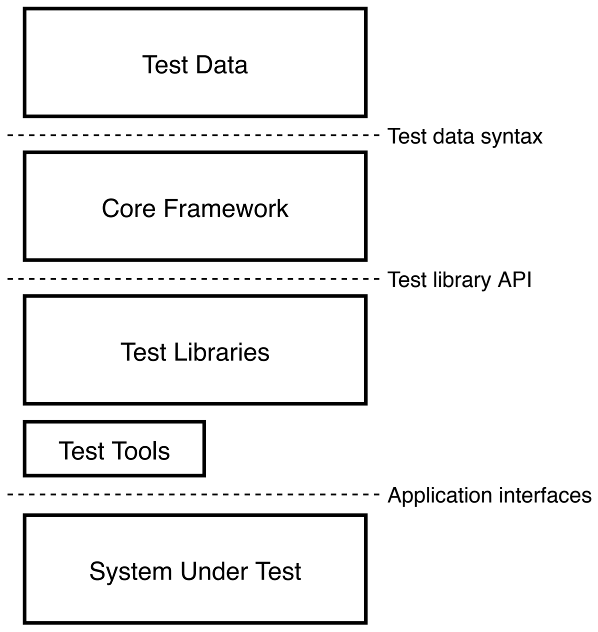


Fig. 5.3: Architecture of the `robot framework`.

By providing an extensible syntax with human-readable keywords, the framework allows us to define test steps and combine them to create more complex test scenarios. This can be used to create quantum circuits that are agnostic of the back end they are run in. We start by defining keywords for quantum operations like qubit initialization, quantum gates, and quantum measurements, like we saw in Figure 5.1. Then, we create other keywords that represent quantum programs, by simply using the keywords of the quantum operations associated to them. We can also define other keywords that take care of the necessary validations. Finally, we generate the test cases by mixing the quantum program with the validation keywords. Furthermore, we can run the same test cases on different quantum systems, by selecting the appropriate testing libraries.

This last characteristic, the possibility of running the same test cases on different systems, in one of the key factors behind the selection of the `robot framework`. As we can see from Figure 5.3, this is possible due to the modular architecture of the framework, where the test data is independent from the test libraries, which are the ones that dictate how to interact with the SUTs. In section 5.3.3 we provide the details of how this is actually implemented in our QuTAF.

Another reason for choosing the `robot framework` is that it provides an API that exposes its internal reporting mechanism. The framework creates, by default, logs and reports that include the essential information. However, there exist several tools and plug-ins that generate better or more detailed test reports, by taking advantage of this exposed API.

The final property that closed the case for choosing the `robot framework` over other solutions, is that it is Python-based. Remember that the quantum node from use case A) provides a Python API interface for specifying the quantum instructions to run on the quantum node. Further, the chosen tool for performing the quantum simulations, `NetSquid`, detailed in the next section, is also available as a Python package. Hence, a Python-based automation framework was key for easily implementing our QuTAF.

5.3.2 NetSquid

`NetSquid` is the Network Simulator for Quantum Information with Discrete events, a platform for simulating all aspects of quantum networks and quantum computing systems. It is based on a generic discrete-event simulation engine that allows to accurately simulate quantum systems that are subject to physical non-idealities. It does this by keeping track of the quantum state decoherence, based on the elapsed time between events [4].

Although specifically designed for the simulation of complex quantum networks, the `NetSquid` platform has been successfully used for the modelling of quantum devices based on Nitrogen-Vacancy centres in Diamond (NV), like in [5]. Hence, we can be sure that the simulations run for the use case B), will be comparable to the test results obtained from A), as the quantum device attached to the networked quantum node uses NV-based qubits.

Another advantage of using `NetSquid`, is that it offers four quantum state representation options: ket vectors, density matrices, stabiliser tableaus, and graph states with local Cliffords. The one that we are interested in is the density matrix representation. This representation allows for the accurate simulation of the evolution of noisy quantum states, and the errors introduced by applying imperfect quantum gates and quantum measurements.

Additionally, the `NetSquid` Python package supports the following quantum error models (in `netsquid.components.models.qerrormodels`): a phenomenological noise model based on the T_1 and T_2^* times of the qubit (`T1T2NoiseModel`), the depolarizing noise model (`DepolarNoiseModel`), and the dephasing noise model (`DephaseNoiseModel`). The last two models receive a depolarizing or dephasing rate that can be applied in two ways: as the probability that the qubit will see the corresponding error with the passing of time, or as the exponential error rate per unit time.

Lastly, it also allows to define custom noisy quantum measurement operations, using the `netsquid.components.instructions.IMeasureFaulty` component. The noise model in this case is a classical probabilistic error model, where we specify the probabilities of committing an error while measuring and observing the outcome 0 (`p0`) or outcome 1 (`p1`).

5.3.3 QuTAF Architecture

The final architecture of our QuTAF is depicted in Figure 5.4. As it is expected, the architecture is largely determined by the test automation software used as basis. There, we can see that the `robot framework` provides the mechanisms for carrying off the main operational responsibilities of the QuTAF: processing the test scripts, executing the tests by driving the System Under Test (SUT), and gathering the results. The `robotmetrics` tool is used to generate a report that includes the expected and calculated proportion of successful outcomes. The `NetSquid` package is used in the `SimulationLibrary.py` test library file. We now describe each of the components of our QuTAF in detail.

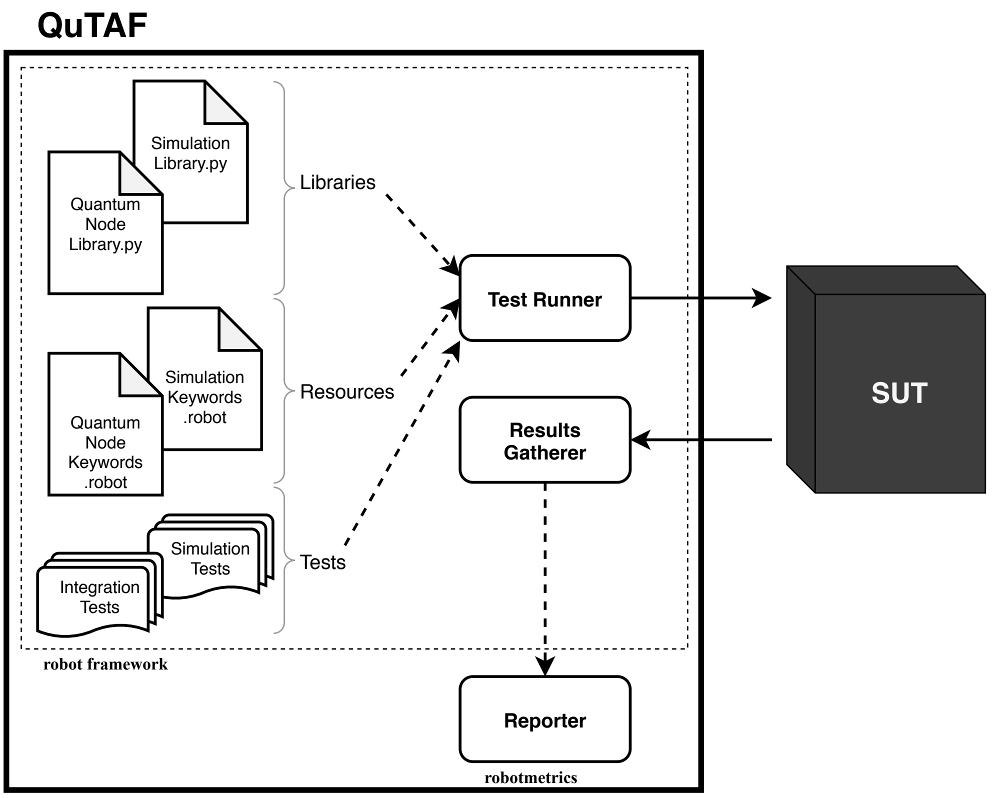


Fig. 5.4: General architecture of QuTAF, the proposed test automation framework for quantum applications.

Libraries. The testing libraries provide the means for driving the SUT. In practical terms, they are Python classes instantiated by the `robot framework`, either at the start of each test suite, or for the whole test run. In our case, we have two classes corresponding to the two use cases or two SUTs to be used. The

QuantumNodeLibrary.py allows us to test the networked quantum node. And the SimulationLibrary.py lets us test a simulated quantum device by using the NetSquid package. In these libraries we do the translation from the quantum operations specified as keywords, to the actual operations available in the systems tested. Here we also specify how the validations are performed. So, it is also in the testing libraries where the the statistical packages that help us calculate the probability distributions, namely scipy.stats and statsmodels.stats, are loaded.

Resources. In the robot framework, resource files provide a mechanism for sharing keywords and variables among tests. In our case, it is in the *Keywords.robot files where we define those keywords that are relevant only for testing a specific system. These files include keywords for common tasks such as test initialization or test tear down, as well as keywords related to the validations performed for each SUT.

Tests. As the name suggests, these are the files that contain the actual steps executed as part of a test. In the robot framework, each test.robot file is a test suite that is composed of one or more test cases. The test suites consist of a test set up and at least one test case. It can also contain definitions of local variables and keywords. An example of a test suite with a single test case that verifies the initialization of a qubit is shown in Figure 5.5.

Test runner and **results gatherer**. While they are depicted as separate entities in Figure 5.4, these components are actually parts of the the same core of the robot framework. The core uses the libraries and resources to run the tests on a given SUT. It is also in charge of keeping track of the test outcomes, and of creating an output.xml log file that contains the information about a test run. The core is, hence, the one that makes everything work.

Reporter. The robot framework has a built-in test reporter that creates HTML reports based on the XML log file generated by the core. However, in order to extend the final report, we use the robotmetrics tool [2]. This software parses the output.xml log file using the API provided by the robot framework, and generates a dynamic HTML dashboard report. By modifying the source code of the robotmetrics, we are able to present the obtained and expected proportions for each of the test cases, in the Test Metrics tab of the dynamic dashboard report.

5.3.4 Test Definition Example

Figure 5.5 shows an example of how a qubit initialization test case is defined in our QuTAF. The first thing we notice are the distinct sections that compose the test file. At the top we have the *** Settings *** section, where the global characteristics

```
1  *** Settings ***
2  Documentation    Qubit initialization and measurement test
3  Library          SimulationLibrary.py
4  Resource         SimulationKeywords.robot
5  Test Setup       Start Simulation
6  Suite Teardown   End Simulation
7
8  *** Test Cases ***
9  Initialize qubit and measure in Z
10     [Documentation]  Initialize qubit 0 and measure in the Z basis.
11     [Tags]       initialization  simulation
12     Append Op To Quantum Program  op=INI  qubit=0
13     Append Op To Quantum Program  op=MSR  qubit=0  basis=Z
14     Simulate Program And Verify    runs=1000  proportion=0.98
15
16 *** Keywords ***
17 Simulate Program And Verify
18     [Documentation]  Simulate a quantum program using NetSquid and
19     ...              verify that the calculated proportion of successes
20     ...              is greater than or equal to the expected one
21     [Arguments]      ${runs}  ${proportion}
22     Simulate Quantum Program    repetitions=${runs}
23     Verify Outcomes Proportion  expected=${proportion}  alpha=0.05
```

Fig. 5.5: Example of a `.robot` test suite file that contains a single qubit initialization test case.

of the test are designated. After a documentation string we find the selection of the `SimulationLibrary.py`, which tells us that we will run the tests using the simulated quantum node testing library. We also see that the `SimulationKeywords.robot` is a resource file. This resource file contains definitions of the keywords used throughout the test suite. Remember that in the `robot framework`, each `.robot` test file is actually a test suite.

The next section, labeled as `*** Test Cases ***`, is where all the test cases that are part of this suite are specified. Each test case starts with a title, it has an optional documentation string and optional tags, and then it contains a list of keywords that are the actual test steps to be executed. These keywords are defined either in the resource files, or in the current test file, in the corresponding `*** Keywords ***` section. The local keywords, however, have a scope that is limited to the current test suite.

Using the `Simulate Program And Verify` keyword as example, we see that the keywords have a mandatory name and an optional documentation string. They can also receive positional or key-valued arguments, as observed in line 21. And, more importantly, they are created out of other existing keywords that are local, built-in, or found in the resource files.

6 Validation

Failure is the condiment that gives success its flavor.

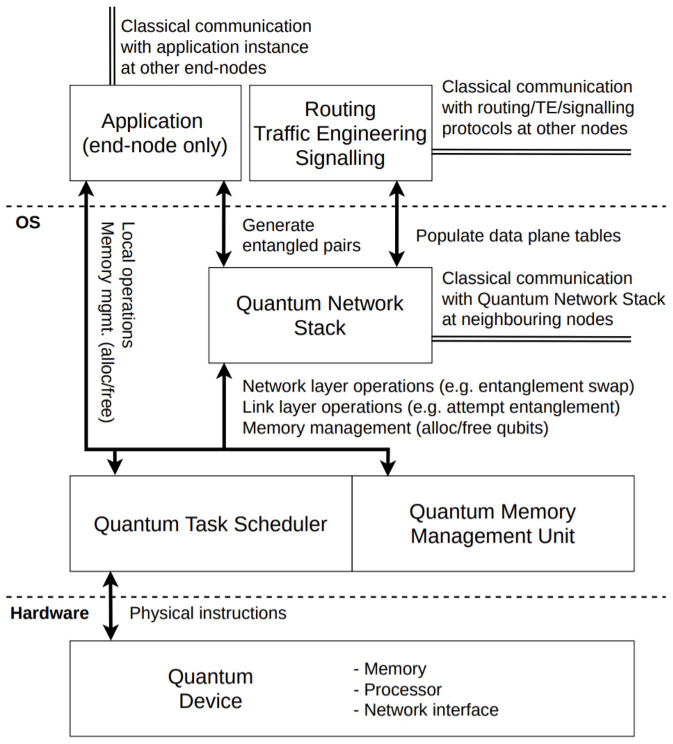
— **Truman Capote**
Writer and actor

The test automation framework for quantum applications presented in the previous chapter was developed with two main use cases in mind: A) verifying the software developments for a networked quantum node, and B) validating the workings of the QuTAF itself. In section 6.1 we present the results obtained for A), where our test automation framework was used for testing the integration of the classical software stack with the control software of the quantum device. The results from B) are in section 6.2, where three failure scenarios were simulated: a base case, tests that fail due to the quality of the quantum hardware is worse than expected, and tests that fail because of errors introduced by the programmers of the quantum programs.

6.1 Testing a Real Quantum Node

6.1.1 A Networked Quantum Node

The networked quantum node tested is an integrated system that serves as the basis for a first quantum internet prototype that will connect a few cities in The Netherlands. It provides the capabilities of running both local and distributed quantum applications. The system is composed of a classical compute node, a quantum compute node based on Nitrogen-Vacancy centres in Diamond (NV), as well as classical and quantum networking interfaces. The classical board is in charge of performing the required computations for the application, transport, network, and link layers of the system. While the quantum node is responsible for the physical layer, and it is controlled by another classical board that can receive instruction messages from the compute board, and as send measurement results and status messages back. More details about the general architecture of the software stack for the networked quantum node can be found in [20] and [19].



Classical communication with application instance at other end-nodes
Application (end-node only)
Routing Traffic Engineering Signalling
Classical communication with routing/TE/signalling protocols at other nodes
OS
Local operations Memory mgmt. (alloc/free)
Generate entangled pairs
Populate data plane tables
Quantum Network Stack
Classical communication with Quantum Network Stack at neighbouring nodes
Network layer operations (e.g. entanglement swap)
Link layer operations (e.g. attempt entanglement)
Memory management (alloc/free qubits)
Quantum Task Scheduler
Quantum Memory Management Unit
Hardware
Physical instructions
Quantum Device
- Memory
- Processor
- Network interface

Fig. 6.1: Proposed architecture for the components of a networked quantum node. From [19].

In Figure 6.1 taken from [19], we see an example of the architecture for a general networked quantum node. In our case, the quantum device at the bottom of the figure is the quantum compute node, and everything that is above the hardware pointed line, is part of the classical compute board. The physical instructions are sent as messages between the compute board and the quantum device, or more precisely, between the compute board and the classical controller of the quantum device. It is worth mentioning that, by the time the integration tests were performed, the networked quantum node had only a single NV memory qubit attached, and no actual quantum networking interface. So, only single-qubit quantum operations were tested. It is also worth recalling that, ultimately, the QuTAF sees the whole networked quantum node as a black-box, and uses its python API to execute the tests.

6.1.2 Integration Tests

The tests run for the networked quantum node have two goals in mind. The first one is certifying a proper communication between the compute node and the quantum device. This is easily accomplished by observing the messages transmitted, and counting the number of errors happening. Nevertheless, knowing that a quantum instruction message has been successfully transmitted, does not certify that the operation is correctly performed. Hence, the second goal of the tests is to verify that the quantum instructions, i.e. the qubit initialization, single-qubit gates, and quantum measurements; are also properly executed.

For the first goal, the networked quantum node is capable of reporting back the number of sent and failed messages. So, calculating the proportion of successfully transmitted messages is straightforward. For all the test cases, the threshold is set to 0.9999. Consequently, if at least 99% of the messages are correctly transmitted between the compute node and the quantum device, then the communication test is considered a pass.

On the other hand, for verifying the correct application of a quantum instruction, we have to run a small quantum circuit, and then validate the quantum state produced. For example, if we want to test that a X gate is correctly applied, we initialize a qubit in $|0\rangle$, apply the X gate, and measure the qubit in the Z basis. Then, according to the particularities of the quantum hardware, like the T_1 and T_2 decoherence times of the qubits, and the errors induced by the quantum gates and measurement operations, we define the expected probability distribution of successful measurements.

The QuTAF can then calculate the probability distribution of the test outcomes, and compare them using the hypothesis testing technique. Since the testing process can modeled as a Bernoulli process whose probability distribution is determined by the probability of success, then the tests for the quantum instructions are also characterized by a proportion: the expected probability of success of the testing process.

Notice that depending on the operations tested, and more properly on the produced quantum state, we need to specify the expected proportion either as a single number or as a range. For the previous example, given that we are measuring in the Z basis, we can say that we expect 0.90 or 90% of successful measurements. Then, if the proportion calculated from the test outcomes is lower than that, we say that the test fails. However, if we prepare the state $|+\rangle$ and we measure it in the Z basis, then ideally we would expect a 0.50 or 50% of successful measurements. Then, we could say that getting any distribution different than this one from the test outcomes, means that the test failed. In reality, though, this number will change due to the inherent quantum errors previously described. So, we resort to say that the test passes if the probability of success calculated from the observed outcomes, falls within an expected range.

Another option for testing the $|+\rangle$ state, or any superposition quantum state, is to perform the measurement in a different basis. Nonetheless, depending on the type of quantum hardware tested, this is not always possible. For example, the underlying

NV-based quantum device used by the networked quantum node can only perform measurements in the computational basis. There, measurements in different basis are accomplished by rotating the qubit towards the axis of the selected basis. These pre-measurement rotation gates are run as part of the quantum circuit tested. This, in turn, makes it difficult to know if a test failed because of a problem in the quantum instruction that is being verified, or because the pre-measurement rotation gates were incorrectly applied.

Table 6.1 lists all the integration tests run in the networked quantum node. Each test case has an associated set of quantum operations performed, as well as the number of times the test was executed. The number of 30000 repetitions is chosen in order to have a greater confidence on the calculated probability of success, while keeping the significance level at 0.05. It is also selected to get an outstanding number of messages for verifying the communication. The only exceptions to this are the negative test cases, with ids T.2.a to T.2.c, that are only repeated 10 times. As mentioned before, the expected proportion of successful messages is set to 0.99 for all the test cases. On the other hand, the expected proportion of successful measurement outcomes is calculated individually for each of the test cases, based on the calibration of the quantum hardware. These values are listed in table 6.2, along with the calculated proportion, and the test verdict.

Tab. 6.1: Description of the integration tests run on the networked quantum node.

Id	Description	Operations	Repetitions
T.1	State $\|0\rangle$ initialization and measurement in Z	1. Initialize qubit 0 2. Measure qubit 0	30,000
T.2.a	Initialization of invalid qubit	1. Initialize qubit 1	10
T.2.b	Unsupported 2-qubit gates	1. Initialize qubits 0, 1 2. Apply CNOT with qubit 0 as control and qubit 1 as target	10
T.2.c	Unsupported move qubit operation	1. Initialize qubits 0, 1 2. Move qubit 0 to qubit 1	10
T.3.a	State $\|1\rangle$ preparation rotating around X and measurement in Z	1. Initialize qubit 0 2. Apply π rotation around X on qubit 0 3. Measure qubit 0	30,000

Continues on next page...

Id	Description	Operations	Repetitions
T.3.b	State $\|1\rangle$ preparation rotating around Y and measurement in Z	1. Initialize qubit 0 2. Apply π rotation around Y on qubit 0 3. Measure qubit 0	30,000
T.4.a *to* T.4.o	Rotation by $k \times \frac{\pi}{16}$ around X and measurement in Z ($k = 1, 2, ..., 15$)	1. Initialize qubit 0 2. Apply $k \times \frac{\pi}{16}$ rotations around X on qubit 0 3. Measure qubit 0	30,000
T.5.a *to* T.5.o	Rotation by $k \times \frac{\pi}{16}$ around X and measurement in Z ($k = -1, -2, ..., -15$)	1. Initialize qubit 0 2. Apply $k \times \frac{\pi}{16}$ rotations around X on qubit 0 3. Measure qubit 0	30,000
T.6.a *to* T.6.o	Rotation by $k \times \frac{\pi}{16}$ around Y and measurement in Z ($k = 1, 2, ..., 15$)	1. Initialize qubit 0 2. Apply $k \times \frac{\pi}{16}$ rotations around Y on qubit 0 3. Measure qubit 0	30,000
T.7.a *to* T.7.o	Rotation by $k \times \frac{\pi}{16}$ around X and measurement in Z ($k = -1, -2, ..., -15$)	1. Initialize qubit 0 2. Apply $k \times \frac{\pi}{16}$ rotations around Y on qubit 0 3. Measure qubit 0	30,000
T.8	State $\|+\rangle$ preparation and measurement in Y	1. Initialize qubit 0 2. Apply $\frac{\pi}{2}$ rotation around Y on qubit 0 3. Apply $\frac{\pi}{2}$ rotation around X on qubit 0 4. Measure qubit 0	30,000
T.9	State $\|1\rangle$ preparation in small rotations around X and measurement in Z	1. Initialize qubit 0 2. Apply $2 \times \frac{\pi}{16}$ rotation around X on qubit 0 3. Repeat step 2 for 7 times 4. Measure qubit 0	30,000

6.1.3 Results

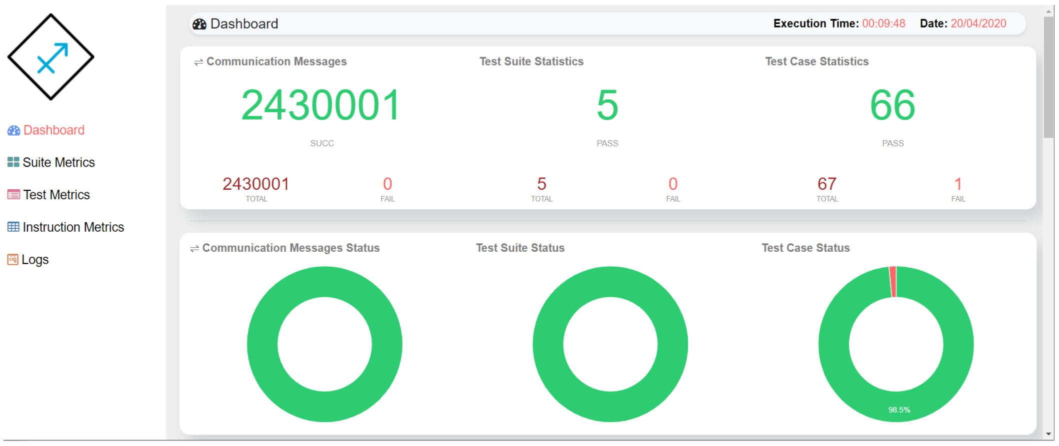
Before delving into the test results, it is worth mentioning that due to a tight schedule and the limited resources available, it was not possible to use the QuTAF to execute the test cases listed in 6.1. This means that the QuTAF was not used to drive the networked quantum node, and the tests were performed by direct execution on the classical compute node. For each test case, a test log was generated. Nevertheless, the QuTAF was later used for performing the analysis of the outcomes, so the different techniques discussed in section 4.3.1 and section 4.3.2 were indeed used for deriving the results presented in Table 6.2.

Looking at the Table 6.2, we immediately see that the only test case that is deemed as failed is the one with id T.1. This was caused by a miscalibration of the quantum device. A detailed analysis about this test case is deferred to section 6.1.1. Then, it is also easy to notice that the tests T.2.a, T.2.b, and T.2.c, have no expected or calculated probability of success, but they are marked as PASS or SKIP. This is because these are negative tests whose goal was to verify that the system was capable of returning an error when receiving an invalid request. The T.2.b test is marked as SKIP since the functionality was not available by the time the tests were performed.

If we focus on the column that shows the expected probability of success, we can observe that the test cases that prepare $|1\rangle$ states, or states close to $|1\rangle$, have an expected proportion of success that is closer to 1.0, compared to those that test states prepared as $|0\rangle$. The reason for this is that the measurement operation in the quantum device has an asymmetric probability of failure. By this we mean that, when doing a quantum measurement, the quantum node is more likely to commit an error when measuring $|0\rangle$ states, than when measuring $|1\rangle$ states.

The test cases that prepare a $|1\rangle$ can also give us some information about the quality of the rotation gates applied by the quantum device. For instance, comparing the calculated probability of success from test T.3.a and test T.3.b, we can conclude that the X and Y rotation gates have a similar performance, since both prepared a $|1\rangle$ with almost the same probability of success. On the other hand, if we compare the calculated probabilities between tests T.3.a and T.8, we see then that the X rotation is not error free, since when we apply the gate 8 times instead of only 1, the success probability drops from ≈ 0.943 to ≈ 0.921.

In Figure 6.2 we show the dashboard of the dynamic report generated by the QuTAF, as a different way of presenting the same results listed in Table 6.2. We see that all the communication messages were successfully transmitted, and there is only one test case marked as failed.



Dashboard Execution Time: 00:09:48 Date: 20/04/2020

Communication Messages	Test Suite Statistics	Test Case Statistics
2430001 SUCC	5 PASS	66 PASS
2430001 TOTAL, 0 FAIL	5 TOTAL, 0 FAIL	67 TOTAL, 1 FAIL

Communication Messages Status — Test Suite Status — Test Case Status

Fig. 6.2: Dashboard of the test report generated by the QuTAF after analyzing the outcomes of the integration tests run on the networked quantum node.

Tab. 6.2: Results of the integration tests run on the networked quantum node. Each test case has an expected probability of success calculated based on the parameters of the quantum hardware, and an actual probability of success calculated from the test outcomes.

Id	Details	Probability of success		Test Result
		Expected	Actual	
T.1	State $\|0\rangle$ initialization	0.91	0.8919	FAIL
T.2.a	Unsupported initialization	-	-	PASS
T.2.b	Unsupported 2-qubit gate	-	-	SKIP
T.2.c	Unsupported move operation	-	-	PASS
T.3.a	Rotation around X to $\|1\rangle$	0.93	0.9433	PASS
T.3.b	Rotation around Y to $\|1\rangle$	0.93	0.9464	PASS
T.4.a	$\pi/16$ rotation around X	[0.85,1.00]	[0.8625,0.8757]	PASS
T.4.b	$2\pi/16$ rotation around X	[0.80,0.95]	[0.8403,0.8543]	PASS
T.4.c	$3\pi/16$ rotation around X	[0.75,0.85]	[0.8061,0.8213]	PASS
T.4.d	$4\pi/16$ rotation around X	[0.70,0.80]	[0.7367,0.7537]	PASS
T.4.e	$5\pi/16$ rotation around X	[0.65,0.75]	[0.6815,0.6997]	PASS
T.4.f	$6\pi/16$ rotation around X	[0.55,0.65]	[0.6171,0.6361]	PASS
T.4.g	$7\pi/16$ rotation around X	[0.50,0.60]	[0.5192,0.5388]	PASS
T.4.h	$8\pi/16$ rotation around X	[0.40,0.50]	[0.4399,0.4595]	PASS
T.4.i	$9\pi/16$ rotation around X	[0.35,0.45]	[0.3765,0.3955]	PASS
T.4.j	$10\pi/16$ rotation around X	[0.25,0.35]	[0.2856,0.3034]	PASS
T.4.k	$11\pi/16$ rotation around X	[0.17,0.27]	[0.2082,0.2244]	PASS
T.4.l	$12\pi/16$ rotation around X	[0.09,0.19]	[0.1530,0.1674]	PASS
T.4.m	$13\pi/16$ rotation around X	[0.03,0.13]	[0.1036,0.1158]	PASS

Continues on next page...

Id	Details	Probability of success		Test Result
		Expected	Actual	
T.4.n	$14\pi/16$ rotation around X	[0.00,0.10]	[0.0626,0.0724]	PASS
T.4.o	$15\pi/16$ rotation around X	[0.00,0.10]	[0.0471,0.0557]	PASS
T.5.a	$-\pi/16$ rotation around X	[0.00,0.10]	[0.0716,0.0820]	PASS
T.5.b	$-2\pi/16$ rotation around X	[0.05,0.15]	[0.0968,0.1086]	PASS
T.5.c	$-3\pi/16$ rotation around X	[0.10,0.20]	[0.1489,0.1631]	PASS
T.5.d	$-4\pi/16$ rotation around X	[0.15,0.25]	[0.2080,0.2242]	PASS
T.5.e	$-5\pi/16$ rotation around X	[0.20,0.30]	[0.2691,0.2867]	PASS
T.5.f	$-6\pi/16$ rotation around X	[0.30,0.40]	[0.3515,0.3703]	PASS
T.5.g	$-7\pi/16$ rotation around X	[0.35,0.50]	[0.4253,0.4447]	PASS
T.5.h	$-8\pi/16$ rotation around X	[0.45,0.55]	[0.5056,0.5252]	PASS
T.5.i	$-9\pi/16$ rotation around X	[0.55,0.65]	[0.5841,0.6033]	PASS
T.5.j	$-10\pi/16$ rotation around X	[0.65,0.75]	[0.6775,0.6957]	PASS
T.5.k	$-11\pi/16$ rotation around X	[0.70,0.80]	[0.7389,0.7559]	PASS
T.5.l	$-12\pi/16$ rotation around X	[0.75,0.85]	[0.7974,0.8130]	PASS
T.5.m	$-13\pi/16$ rotation around X	[0.80,0.90]	[0.8359,0.8501]	PASS
T.5.n	$-14\pi/16$ rotation around X	[0.80,0.95]	[0.8684,0.8814]	PASS
T.5.o	$-15\pi/16$ rotation around X	[0.85,1.00]	[0.8734,0.8862]	PASS
T.6.a	$\pi/16$ rotation around Y	[0.85,1.00]	[0.8623,0.8755]	PASS
T.6.b	$2\pi/16$ rotation around Y	[0.80,0.95]	[0.8429,0.8569]	PASS
T.6.c	$3\pi/16$ rotation around Y	[0.75,0.85]	[0.8049,0.8201]	PASS
T.6.d	$4\pi/16$ rotation around Y	[0.70,0.80]	[0.7456,0.7624]	PASS
T.6.e	$5\pi/16$ rotation around Y	[0.65,0.75]	[0.6882,0.7062]	PASS
T.6.f	$6\pi/16$ rotation around Y	[0.55,0.65]	[0.6085,0.6275]	PASS
T.6.g	$7\pi/16$ rotation around Y	[0.50,0.60]	[0.5219,0.5415]	PASS
T.6.h	$8\pi/16$ rotation around Y	[0.40,0.50]	[0.4512,0.4708]	PASS
T.6.i	$9\pi/16$ rotation around Y	[0.35,0.45]	[0.3744,0.3934]	PASS
T.6.j	$10\pi/16$ rotation around Y	[0.25,0.35]	[0.2992,0.3174]	PASS
T.6.k	$11\pi/16$ rotation around Y	[0.17,0.27]	[0.2040,0.2200]	PASS
T.6.l	$12\pi/16$ rotation around Y	[0.09,0.19]	[0.1564,0.1710]	PASS
T.6.m	$13\pi/16$ rotation around Y	[0.03,0.13]	[0.1026,0.1148]	PASS
T.6.n	$14\pi/16$ rotation around Y	[0.00,0.10]	[0.0638,0.0738]	PASS
T.6.o	$15\pi/16$ rotation around Y	[0.00,0.10]	[0.0447,0.0531]	PASS
T.7.a	$-\pi/16$ rotation around Y	[0.00,0.10]	[0.0609,0.0707]	PASS
T.7.b	$-2\pi/16$ rotation around Y	[0.05,0.15]	[0.0945,0.1063]	PASS
T.7.c	$-3\pi/16$ rotation around Y	[0.10,0.20]	[0.1375,0.1513]	PASS
T.7.d	$-4\pi/16$ rotation around Y	[0.15,0.25]	[0.1933,0.2091]	PASS

Continues on next page...

Id	Details	Probability of success		Test Result
		Expected	Actual	
T.7.e	$-5\pi/16$ rotation around Y	[0.20,0.30]	[0.2577,0.2751]	PASS
T.7.f	$-6\pi/16$ rotation around Y	[0.30,0.40]	[0.3509,0.3697]	PASS
T.7.g	$-7\pi/16$ rotation around Y	[0.35,0.50]	[0.4307,0.4501]	PASS
T.7.h	$-8\pi/16$ rotation around Y	[0.45,0.55]	[0.4877,0.5073]	PASS
T.7.i	$-9\pi/16$ rotation around Y	[0.55,0.65]	[0.5777,0.5969]	PASS
T.7.j	$-10\pi/16$ rotation around Y	[0.65,0.75]	[0.6678,0.6862]	PASS
T.7.k	$-11\pi/16$ rotation around Y	[0.70,0.80]	[0.7332,0.7504]	PASS
T.7.l	$-12\pi/16$ rotation around Y	[0.75,0.85]	[0.7816,0.7976]	PASS
T.7.m	$-13\pi/16$ rotation around Y	[0.80,0.90]	[0.8310,0.8454]	PASS
T.7.n	$-14\pi/16$ rotation around Y	[0.80,0.95]	[0.8578,0.8712]	PASS
T.7.o	$-15\pi/16$ rotation around Y	[0.85,1.00]	[0.8712,0.8840]	PASS
T.8	State $\|+\rangle$ preparation and measurement in Y	[0.40,0.50]	[0.4507,0.4620]	PASS
T.9	State $\|1\rangle$ preparation in small rotations around X and measurement in Z	$\geq$ 0.92	0.9210	PASS

6.1.4 Qubit Initialization: a Failed Test Case

As we saw in section 2.2.2, tests at the integration level are usually run following a systematic approach, where the components of a system are gradually integrated. If small groups of components are already verified, then we can proceed to test a more complex or bigger integration, without needing to test again the already validated behaviors. Having this in mind, it was striking to find that the only test case that failed was the simplest one, the one that served as the basis for all the other test cases: the initialization of a qubit.

This test case simply initializes a qubit in $|0\rangle$, and then measures it in the Z basis. These two operations are further used as building blocks for creating test cases that verify other operations. It is clear that testing any single-qubit gate requires, always, the initialization and measurement of a qubit. We cannot test a quantum gate in isolation. Hence, if the test that verifies these basic operations do not pass, we can suspect that the results obtained from the more complex test cases are wrong. It is then imperative to know the root cause of the failure.

Sadly, in our case, it is not possible to follow the process for debugging failures in quantum programs outlined in section 3.4.2. This is because the QuTAF was not used for the execution of the test cases, but only for the analysis of the test outcomes and the generation of the test results. Hence, there is no test execution log available for scrutiny. This also means that the source of the problem cannot be an incorrect or incomplete test case definition. On the other hand, we do have the outcomes of the measurements performed by the test. So, we can definitely perform checks on the way the test result is derived from them.

Before moving forward, though, it is important to recall what does it mean for a test to fail. Broadly speaking, we say that a test fails if the probability distribution of the outcomes is different than the probability distribution expected by the test. In the QuTAF we make a more precise formulation, and say that the test fails if the probability of success calculated from the test outcomes p_{calc}, is strictly less than the expected probability of success p_{exp}, i.e., if $p_{calc} < p_{exp}$. So, a failing test case does not necessarily mean that the qubit initialization is not working at all. This is confirmed by checking the results table, and observing that the calculated success probability is actually close to the expected one, but certainly not higher.

One probable source of error can be the way this probability of success is calculated from the measurement outcomes data. For the failing test case, this calculation is done by using the Binomial Inverse Survival Function (ISF), which specifies a minimum number of expected successful observations k_{min} given an expected probability of success, a significance level (α), and the total number of observations, as detailed in 4.3.1. In our case, the expected probability of success is 0.91, an $\alpha = 0.05$ is chosen, and we perform a total of 30000 repetitions. Hence, the k_{min} is:

$$k_{min} = Z(\alpha, n, p) = Z(0.05, 30000, 0.91) = 27381 \tag{6.1}$$

This number is compared with the actual number of successful outcomes observed, which in this case is $k_{obs} = 26861$. Thus, given that $k_{obs} = 26861 < k_{min} = 27381$, the test is correctly marked as a fail. Given the high number of repetitions, relaxing the confidence level, i.e., increasing the α, makes no difference in the final verdict for the test.

Now, in section 4.3.2 we saw there is another way of calculating the probability of success: the binomial proportion confidence interval. Even when this is an approximation of the true Binomial probability distribution, the fact that we have 30000 observed outcomes, gives us confidence that the approximation is going to be accurate. For calculating the binomial proportion confidence interval we can choose different approximation methods. However, as we can see in Table 6.3, with all the available methods, the confidence interval calculated from the data is in accordance with the value obtained using the Binomial ISF. From these results, we can then be sure that the probability of success was indeed correctly calculated.

On the other hand, we know that the testing of a quantum program is a random process. As such, it can happen that the test fails simply because the test realizations were just *unlucky*. Nevertheless, this uncertainty can easily be reduced by increasing the number of observations made. For the failing test case that concerns us, having 30000 observations allows us to calculate binomial proportion confidence intervals for the probability of success, with a high confidence. Remember that setting $\alpha = 0.001$ is equivalent to a confidence level of 0.999, which means that we can be 99.9% sure that the *true* probability of success of the test lies within the calculated interval. As observed in Table 6.3, the confidence intervals calculated from the outcomes of the failed test, never go beyond 0.9012.

After discarding sources of error coming from the QuTAF or the testing assumptions, the only possibility left is that the failure is indeed caused by a malfunctioning quantum node. Upon further discussion with the team in charge of the NV-based quantum device, the reason for the failure was found to be a miscalibration of the quantum hardware.

During the execution of the test cases, there were different calibrations applied to the quantum device. Due to various experimental constraints like laser power drifts, or possible ice forming in the lab set up, the values can change over the course of the days. A first calibration was done some days before the tests were run, and the failed qubit initialization test was the first one to be run. Subsequent calibrations were done in between the test runs, which explains why the rest of the tests did not fail. Finally, the fact that during the calibrations the probability of success for the initialization of a qubit was not maximized, as it was not the scope of the integration tests, further explains the failure observed.

Tab. 6.3: Binomial proportion confidence intervals for the probability of success of the failed qubit initialization test. The intervals are calculated from the observed measurement outcomes, using two different significance levels (α). The different approximation methods are detailed in [30].

Approximation Method	**Confidence Interval**	
	$\alpha = 0.05$	$\alpha = 0.001$
Normal	[0.891903, 0.898830]	[0.889552, 0.901181]
Agresti-Coull	[0.891852, 0.898780]	[0.889407, 0.901041]
Clopper-Pearson	[0.891848, 0.898808]	[0.889435, 0.901098]
Wilson	[0.891852, 0.898780]	[0.889408, 0.901039]
Jeffreys	[0.891865, 0.898792]	[0.899452, 0.901082]

The fact that the QuTAF was capable of identifying this problem is something that we cannot stress enough. It proves that our test automation framework for quantum applications is certainly capable of uncovering tangible and real issues in the functioning of the quantum devices tested. It also gives us the confidence to continue with the next use case, where specific failure scenarios will be simulated and verified using the QuTAF.

6.2 Simulating Fail Scenarios

In this section we showcase the capabilities and limits of the proposed QuTAF by exploring how it performs when executing testing scenarios purposefully designed to fail. We use the `NetSquid` package in order to simulate a quantum device that closely resembles the quantum hardware from the previous use case. This is done so that we can compare the test results from the simulations and those obtained from the real quantum device, if needed. It also ensures that the parameters chosen for the simulations are indeed physically attainable.

We first describe the configuration of the simulated quantum node in section 6.2.1. Then, an example of what a simulated test run looks like is presented in section 6.2.2. From this example we derive some key parameters used for the test runs of the failure scenarios explored. The first actual fail scenario is given in section 6.2.3. This scenario is based on our experience with the failed qubit initialization test case from the first use case. So, its failure premise is that the quality of the quantum device is worse than expected. Test cases for the quantum depolarizing and quantum dephasing error models are simulated and discussed. Finally, section 6.2.4 contains the test cases and results for the scenario where a quantum programming mistake is simulated: the application of an extra rotation gate inside a loop.

6.2.1 Configuration of the Simulated Quantum Node

As mentioned in section 5.3.2, the `NetSquid` package provides various options for the noise models that can be applied to the qubits and quantum operations. From these, we will choose those that allow us to replicate, as closely as possible, the quantum hardware from the networked quantum node tested for use case A). The quantum device in question is based on Nitrogen-Vacancy centres in Diamond (NV), and has a single electron qubit. The noise model used for the electron qubit is the `T1T2NoiseModel`, while the qubit initialization operation is subject to the `DepolarNoiseModel` where the rate represents the probability that the qubit will depolarize with time. The single qubit gates are deemed as perfect, while the quantum measurements use a classical probabilistic error model where `p0` and

Tab. 6.4: Physical parameters used for the simulation of the NV-based quantum device. Based on [5].

	Error		Duration
	Parameter	Value	
Electron qubit	T_1 time	2.86 ms	-
	T_2^* time	1.0 ms	-
Electron qubit initialization	Depolarization prob.	0.025	2 μs
Single qubit gate	-	-	5 ns
Qubit measurement	Prob. of error $\|0\rangle$	0.05	3.7 μs
	Prob. of error $\|1\rangle$	0.005	

p1 are the probabilities of incurring in an error when measuring a $|0\rangle$ or a $|1\rangle$, respectively.

The actual parameters used for the error models above, shown in Table 6.4, are taken from Table 6 of the Appendix D.2 in [5]. Here, Dahlberg et al. use the NetSquid package to accurately simulate a similar quantum node based on NV centres. For a detailed discussion about the physics of NV-based platforms, and the reasons behind the values and error models chosen, we refer the reader to the same Appendix D of the cited work.

Besides the physical parameters of the simulated quantum device, we also need to define the parameters related to the tests executed as part of the failure scenarios. The two outstanding test parameters are: the number of repetitions or executions of each test case, and the chosen significance level (α) for calculating and comparing the probabilities of success. Given that the goal of these tests is to validate the workings of the QuTAF and explore its limits, we will actually use different values for these two parameters to see if, and how, they impact the test results.

6.2.2 Scenario 1: Base case

This first test scenario is a basic qubit initialization and measurement test, where a qubit is first set to $|0\rangle$ and then measured in the Z basis. It allows us to showcase what a test run with a *vanilla* configuration, i.e., one that uses the parameters and values shown in Table 6.4 with no modifications, looks like. From these parameters, we calculate that the expected probability of success for the test is 0.93. Also, the test is configured to perform 10,000 repetitions and uses a significance level of $\alpha = 0.05$ for estimating the probability of success from the actual outcomes.

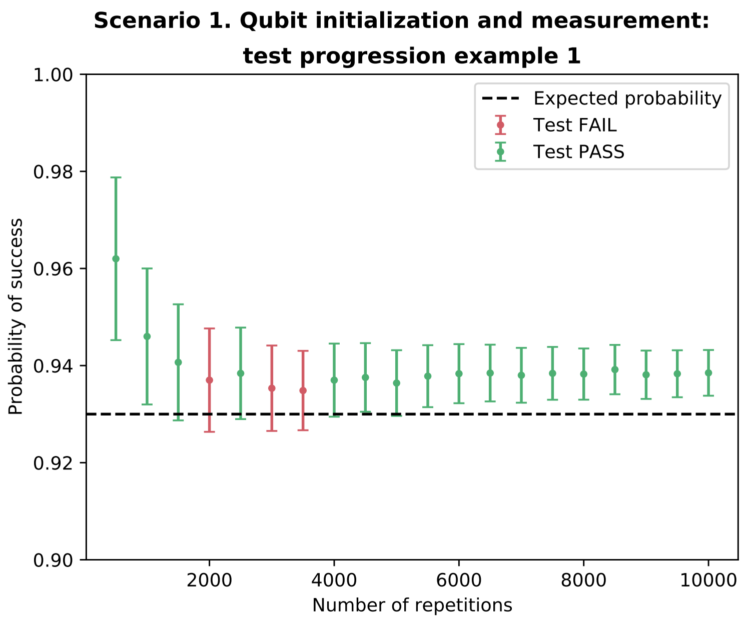


Fig. 6.3: Simulated qubit initialization test - test progression 1: test results and estimated success probabilities shown every 500 repetitions.

Figure 6.3 shows what the result of the test would be if it was stopped after 500, 1000, 1500, and so on, repetitions. The error bars correspond to the estimated interval for the probability of success, calculated with a 95% confidence. As we can see, the interval narrows down as the number of repetitions increases, even when the significance level, or confidence level, is not changed. This confirms our claim that a way of having more confidence on a test is to simply increase the number of repetitions performed.

Now, looking at Figure 6.3 it seems that after around 4000 repetitions, the estimated probability of success does not fluctuate anymore. So, we could get the impression that we only need to repeat our tests that much in order to get a meaningful result. However, we have to remember that testing quantum applications is a random process, and as such, we can not base our conclusions on a single test run.

In Figure 6.4 we find another realization of the same qubit initialization test. Again, the test result and the estimated confidence intervals for the probability of success are taken after every 500 repetitions. There, we see that if we stopped the test after anywhere between 500 and 6000 repetitions, the result of the test would be a pass. However, the test outcomes between 6500 and 7500 are such that make the test fail. This brings up the question of how much repetitions are actually needed in order to have a valid test result.

In order to answer this question, we perform a series of 50 test realizations of the same test with the same parameters. Each test realization consist of 10000

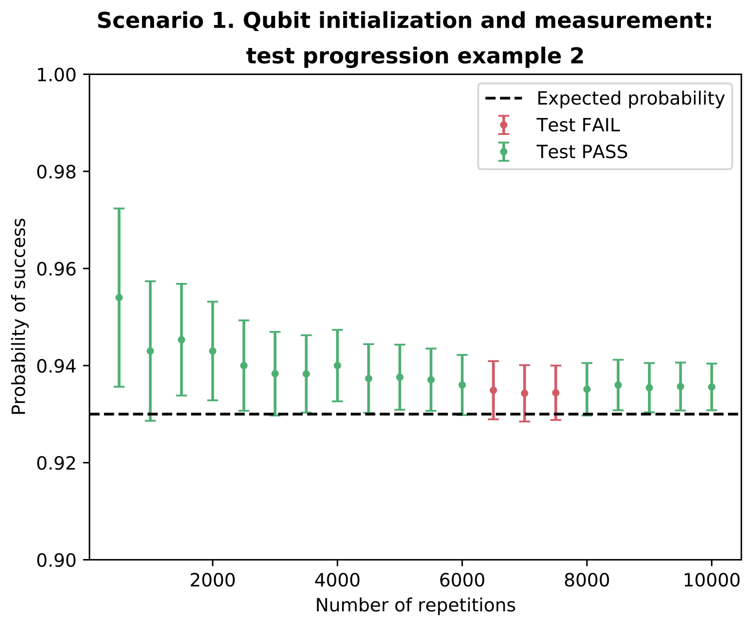


Fig. 6.4: Simulated qubit initialization test - test progression 2: test results and estimated success probabilities shown every 500 repetitions.

repetitions. Again, we stop the tests after every 500 repetitions and count how many of the tests result in a pass, and how many in a fail. The results are shown in Figure 6.5. As expected, the number of passed tests increases as we perform more test repetitions.

Furthermore, it is also clear that if we want to be 100% confident in the result of a test, then we must perform more repetitions. As we can see, from the 50 test realizations, in 4 occasions the test was deemed as failed even after doing 10000 repetitions. Alas, due to time and resource constraints, no further explorations with a higher number of repetitions were conducted.

Finally, Figure 6.6 shows the success probabilities grouped by passed and failed tests averaged over 50 test realizations, as they would be reported if the test was stopped after those many repetitions. It is interesting to observe that the error bars of the success probabilities from the passed tests are about the same size, no matter the number of repetitions so far executed. On the other hand, it is clear that the failed tests have a high variability when a few repetitions have been conducted, but they converge to a value that is less than 0.93 as the number of repetitions increases. The expected probability of success is indicated with the black line, and we can see that though some of the error bars of the passed tests fall below the line, the dot that marks the calculated average, is always above 0.93.

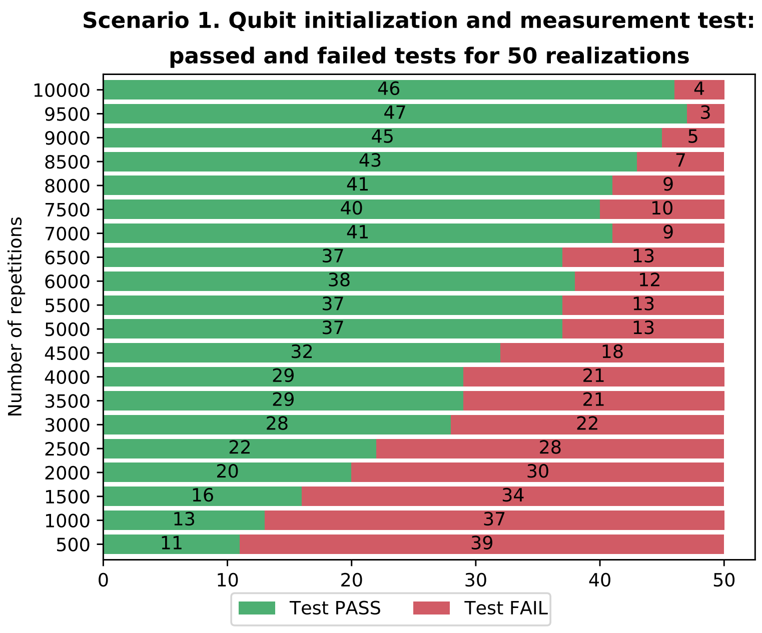


Fig. 6.5: Number of passed and failed qubit initialization tests, from a total of 50 test realizations.

6.2.3 Scenario 2: Defective Quantum Devices

In this scenario we explore how issues in the quality of the quantum device affect the test results reported by the QuTAF. We already know that the QuTAF is capable of detecting deficiencies in the quantum hardware, as it correctly did for the failed test case from use case A) discussed in section 6.1.4. However, we do not know if it can properly detect all types of errors observed in a quantum device, or how sensitive it is to them. In this section, we give an answer to these questions.

The test case run in this scenario is the same simple qubit initialization test. We inspect the sensitivity of the QuTAF to the deficiencies in the quantum hardware by applying, independently, the quantum depolarizing and quantum dephasing noise models to the quantum initialization operation. The parameters for the decoherence of the memory qubit (T_1 and T_2^*), and for the probability of error of the measurement, are left as in Table 6.4. The results shown below are obtained following a similar approach as in scenario 1: by performing 50 test realizations each consisting of 10000 repetitions. We report, again, the average probability of success as well as the number of passed and failed tests.

6.2.3.1 Scenario 2. a) Qubit initialization: depolarizing

We know that the quantum device being simulated is a noisy quantum node based on NV-centres. This means that we are already applying noise to the qubits and

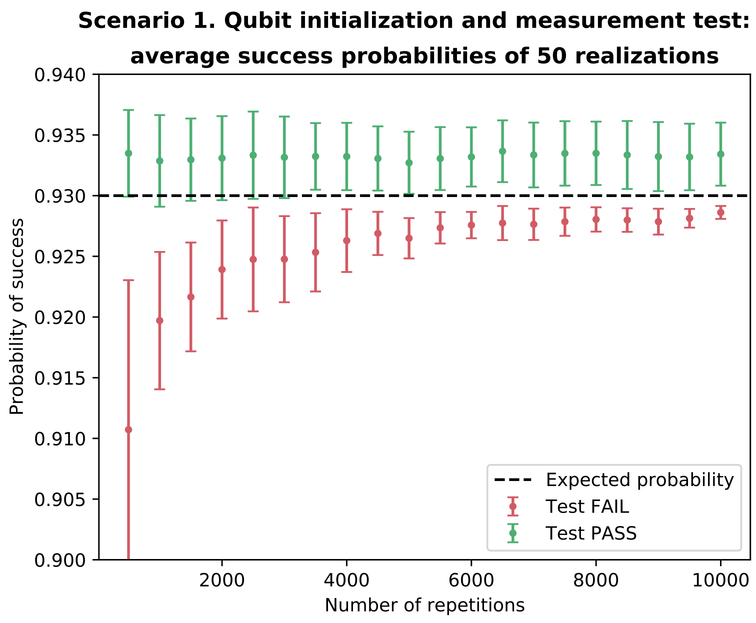


Fig. 6.6: Probability of success of the qubit initialization tests simulated with the *vanilla* configuration. The average and standard deviation from a total of 50 test realizations is shown.

quantum gates, including the quantum initialization operation. According to Table 6.4, in our simulations the initialization of the electron qubit is subject to a quantum depolarizing noise that is applied with a probability of 0.025. From this, the expected probability of success of the qubit initialization test is calculated to be 0.93. In this failure scenario we want to answer the following question: if we change the depolarizing probability of the initialization operation to a different value, how different are the results reported by the QuTAF?

For this, we run test cases where we increase the depolarizing probability from 0.005 to 0.045, almost the double of the 0.025 probability value used for the *vanilla* simulations. In the base case in scenario 1, we saw that 46 out of the 50 realizations passed the qubit initialization test. So, given that a lower value for the depolarizing probability means a better qubit initialization operation, we would expect all the test cases up to 0.020 to pass. Then, also, we would expect the QuTAF to report a fail for those tests where the depolarization probability is higher than 0.025.

Figure 6.7 shows the number of passed and failed tests from a total of 50 test realizations, simulated with different depolarizing probabilities. As expected, all the test cases with a depolarizing probability ≤ 0.2, pass all the time. Also, with the *vanilla* 0.025 depolarizing probability, we get the same number of pass and fail as in scenario 1 (see the results in Figure 6.5 for 10000 repetitions). Finally we see that as we increase the value for the depolarization probability, in other words, when the quality of the quantum device gets worse and worse, then the QuTAF reports more and more failed tests. In quantitative terms, increasing the depolarization probability by 0.5%, from 0.025 to 0.030, results in almost a 4-fold increase in the number of failed tests, from 4 to 15. Thus, we can conclude that the QuTAF is indeed sensitive to the quantum depolarization noise.

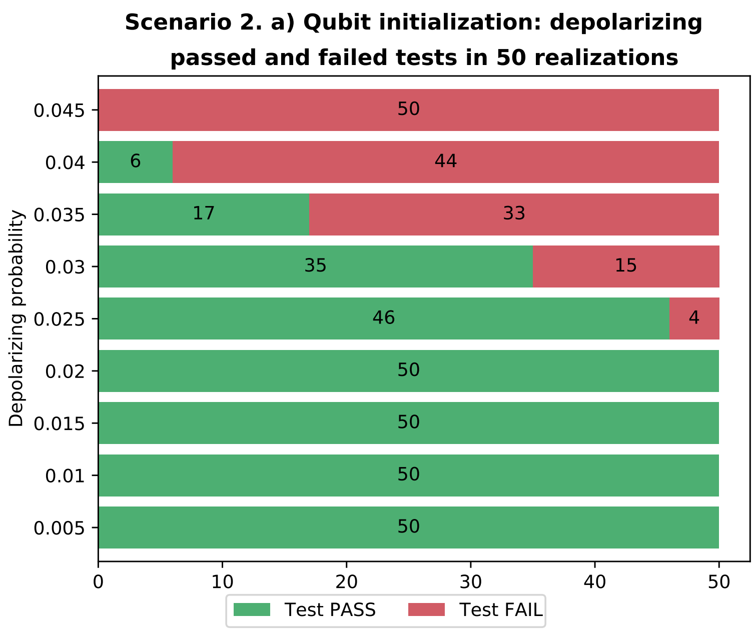


Fig. 6.7: Number of passed and failed qubit initialization tests, from a total of 50 test realizations, while increasing the initialization depolarizing probability.

6.2.3.2 Scenario 2. b) Qubit initialization: dephasing

The other type of noise that we explore is the quantum dephasing noise, applied again to the qubit initialization operation. We use the same methodology as in a), where the values are varied from 0.005 to 0.045, but they correspond to the probability of applying the quantum dephasing noise. In figure 6.8 we see the failed and passed tests in 50 test runs, for each of the simulation dephasing probabilities.

As observed, for each of the tested dephasing probabilities, all the 50 test runs pass. This can be explained in two ways: either the probability values chosen are too low that they do not affect the quality of the quantum initialization operation, or the QuTAF is incapable of detecting this type of quantum error. In order to know the root cause, we can look at Figure 6.9 where the average probability of success from the 50 test runs is shown, and the error bars denote the standard deviation of the estimated probabilities. Immediately, we note that the average of the estimated probability does change as the dephasing probability is increased, which means that the values do affect the performance of the quantum device. However, these values fluctuate around 0.945, and that is why all the tests pass.

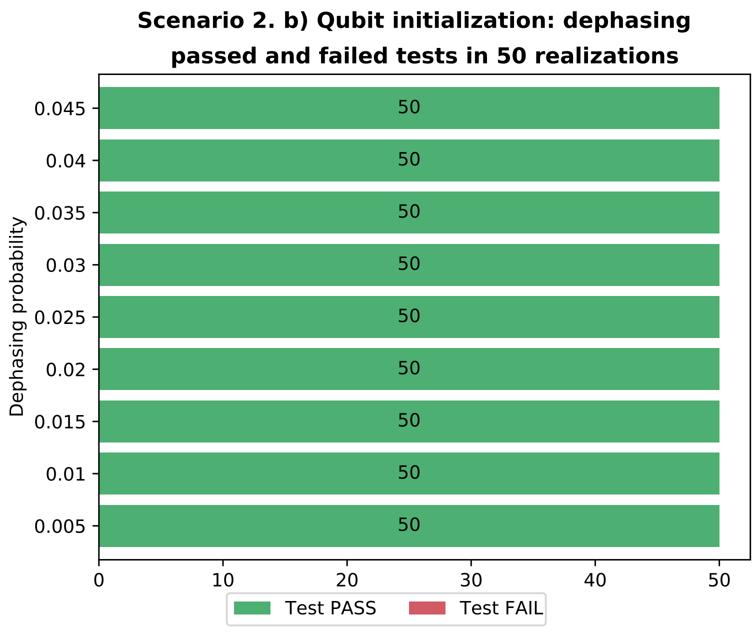


Fig. 6.8: Number of passed and failed qubit initialization tests, from a total of 50 test realizations, while increasing the initialization dephasing probability.

The alternative is, then, that the QuTAF is not capable of detecting the quantum dephasing noise introduced to the system. This is exactly the root cause, and it has to do with the way the quantum dephasing channel noise acts on the quantum states. As described in section 2.1.6.2 applying the quantum dephasing noise to a quantum state changes its phase, but crucially, it does not alter the magnitude of the Z component. Given that the test is always measuring the qubit in the computational basis, then this change in the phase goes unnoticed. The fluctuation in the success probabilities that we observe in Figure 6.9, can be attributed to the decoherence suffered by the qubit and the noise in the quantum measurement operations.

Thus, we conclude that when using the QuTAF for testing applications run in systems dominated by the quantum dephasing noise, we need to run test cases that do quantum measurements in basis different than the computational basis. If no other quantum measurements are supported by the SUT, then the test results reported by QuTAF can not be taken as conclusive.

6.2.4 Scenario 3: Quantum Programming Errors

Besides the intrinsic quality deficiencies of the quantum device, a quantum application can also behave incorrectly due to errors introduced in the quantum program itself. Think of a quantum application that initializes a qubit in $|0\rangle$, and applies small quantum rotations in a loop in order to produce a different state that is measured in Z. Now, imagine that the programmer makes a common mistake and starts the loop

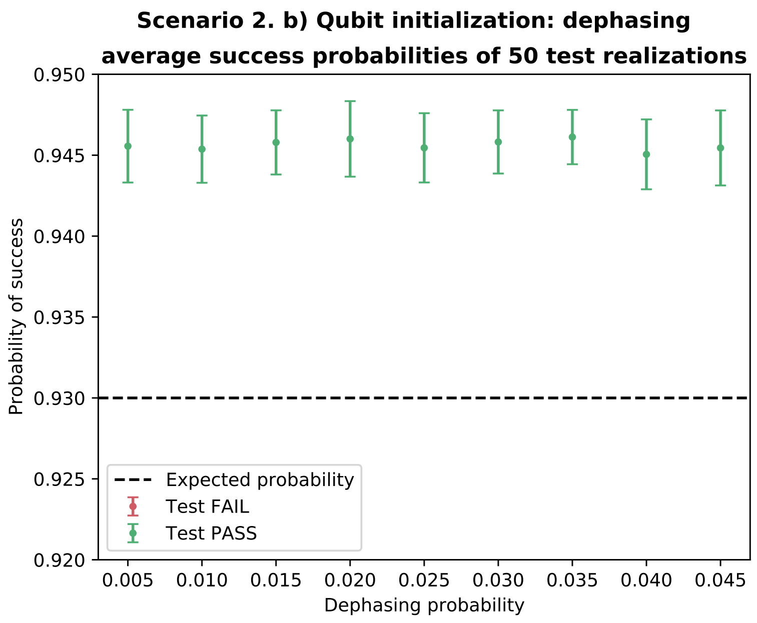


Fig. 6.9: Probability of success of the qubit initialization tests simulated with various depolarization probability values. The average and standard deviation from a total of 50 test realizations is shown.

in 0 where it should have been 1, or uses a $\leq$ instead of a $<$, effectively applying one extra rotation to the qubit. If we test this quantum program, then we would expect it to fail, as it has a *bug* in it. In the test cases presented below, we explore the ability of the QuTAF to detect these types of quantum programming errors.

Before delving into the results, it is important to mention that the simulations performed in all the cases used the *vanilla* configuration from Table 6.4. Also, since this simulations are based in the quantum device of the networked quantum node from use case A), the single-qubit gates were made to operate in the same way as in the mentioned device. In particular, it limits the angles of the rotation gates, where the full range of rotation 2π is divided into 32 steps. Then, the angles can only be multiples of $\frac{2\pi}{32}$, i.e., they take the values $[\frac{\pi}{16}, \frac{2\pi}{16}, \ldots, \frac{15\pi}{16}]$.

6.2.4.1 Scenario 3. a) Rotation around X to $|1\rangle$

In this test case we rotate the qubit around the X axis to prepare $|1\rangle$. Since we are limited to rotation angles of $\frac{\pi}{16}$ steps, then the correct number of rotations to apply would be 16. The expected probability of success of the program, when correctly preparing and measuring a $|1\rangle$, is 0.98. In order to see how sensitive the QuTAF is to the quantum programming errors, we simulate tests that apply 14, 15, 16, 17, and 18 rotations. We again perform 50 test runs and report the number of passed and failed tests, along with the average success probabilities.

Figure 6.10 shows that when the wrong number of rotations are used, the 50 tests executed as part of the test run, fail. This is a clear indication that our QuTAF is indeed able to detect this type of bugs in a quantum application. Interestingly, not all of the tests for the non-failure case, where 16 rotations are performed, succeed. However, the proportion of failed and passed tests (6/44) is in line with what we observed when testing the qubit initialization (4/46).

Since in our simulations the quantum gates are practically error free, the two extra failed tests that we see in Figure 6.10 can be a result of the decoherence suffered by the qubit while the 16 rotations are applied. Also, by looking at the average probability of success reported in Figure 6.11, we attest that the failed tests for the 16 rotations have an average that is still closer to the expected 0.98 value, than the other tests that failed due to an incorrect number of rotations being applied.

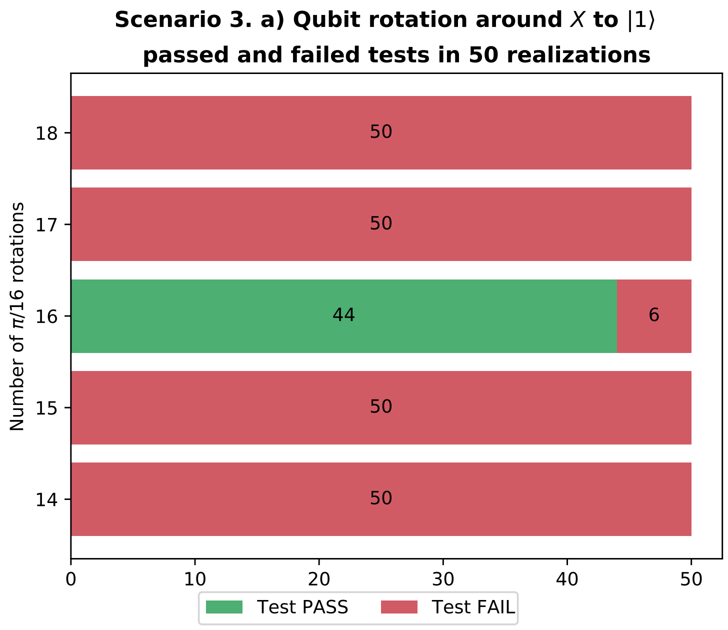


Fig. 6.10: Number of passed and failed tests for the rotation around X to $|1\rangle$, from a total of 50 test realizations. Steps of 14, 15, 16, 17, and 18 $\frac{\pi}{16}$ angles are shown.

6.2.4.2 Scenario 3. b) Rotation around X to $|i\rangle$

Here we also test the R_x quantum rotation gate, but instead of $|1\rangle$, we prepare the superposition state $|i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$. The right amount of rotations is 8, as we are preparing an equal superposition of $|0\rangle$ and $|1\rangle$. The expected probability of success is specified as an interval, and it is calculated to be $[0.45, 0.5]$. We inspect again the sensitivity of our QuTAF by simulating cases where 6, 7, 8, 9, and 10 rotations are applied.

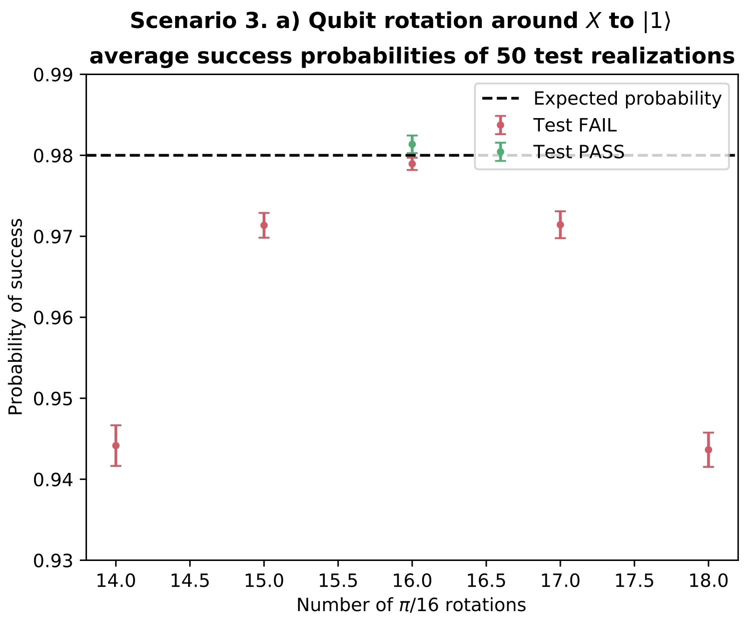


Fig. 6.11: Probability of success of the rotation around X to $|1\rangle$ tests. The average and standard deviation from a total of 50 test realizations is shown.

Once more, we find that all the tests where a wrong number of rotations is applied are correctly signaled as failed, as reported in Figure 6.12. Interestingly, unlike the previous failure test case, here all the tests succeed when the correct number of rotations are applied. One reason for this could be that the estimated success probabilities do not vary as much as in the failure test case a). But this is quickly discarded by comparing the standard deviations from both scenarios. For a), the standard deviation of the average probability of success for the non-failure test case is ≈ 0.001086. While for b), it is ≈ 0.005111. Thus, the estimated success probabilities deviate more from the mean in this case.

By looking at Figure 6.13, we can derive the true reason why all the tests with 8 rotations pass. As it is evident, the calculated expected interval for the probability of success of $[0.45, 0.5]$, is too big. This makes all the tests for the non-failure test case to pass. Still, even when the expected interval is bigger than necessary, it is important to note that none of the actual failure tests is incorrectly deemed as a pass. This means that our QuTAF will correctly identify bugs in a quantum program, even if the specified expected interval was overestimated.

6.2.4.3 Scenario 3. c) Rotation around Y to $|1\rangle$

This third failure test case is similar to a). The state prepared is $|1\rangle$, but the rotation is done around the Y axis, instead of the X axis, i.e., the quantum gate applied is

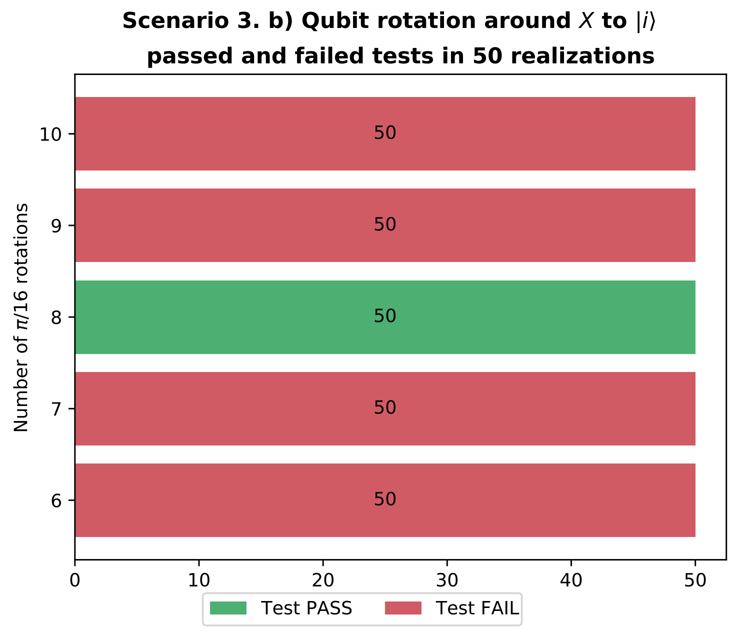


Fig. 6.12: Number of passed and failed tests for the rotation around X to $|i\rangle$, from a total of 50 test realizations. Steps of 6, 7, 8, 9, and 10 $\frac{\pi}{16}$ angles are shown.

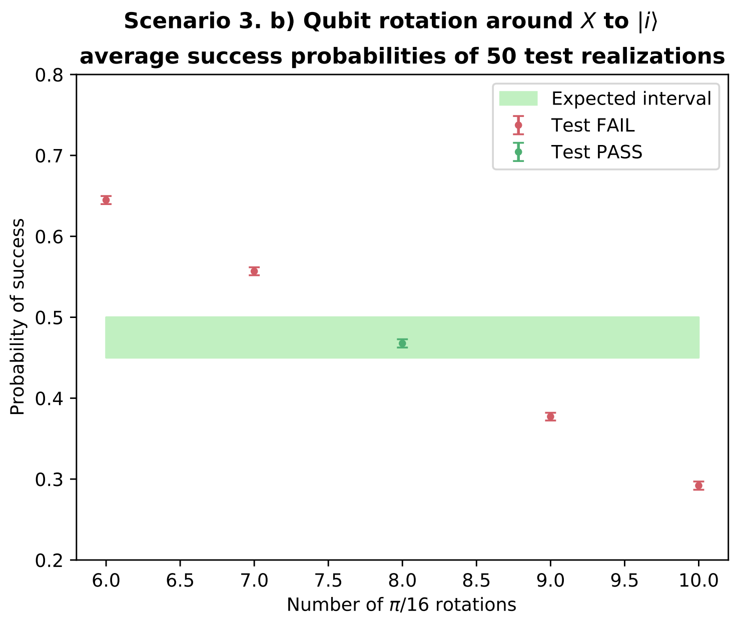


Fig. 6.13: Probability of success of the rotation around X to $|i\rangle$ tests. The average and standard deviation from a total of 50 test realizations is shown.

R_y. From the results shown in Figure 6.14, we arrive at the same conclusion as in the previous two test cases, namely that our QuTAF is again able to detect the bug of a wrong number of rotations applied.

However, in Figure 6.14, we note an important difference: the number of passed tests for the non-failure case is considerably lower than in a). If we compare the calculated average success probabilities from the two, i.e., we compare Figure 6.11 with Figure 6.15, it is hard to see a difference. Moreover, as the simulations of all the failure test cases use the same configuration, then the explanation for this would be that the value of 0.98 for the expected probability of success is wrongly calculated.

6.2.4.4 Scenario 3. d) Rotation around Y to $|+\rangle$

This failure test case is also the analogue of b), where the quantum gate used is the R_y instead of R_x. The interval for the expected success probability is again $[0.45, 0.50]$. The results observed in Figures 6.16 and 6.17, and the conclusions derived from said figures, are also in line with what has been commented already for b). Hence, no further discussion is necessary for this test case.

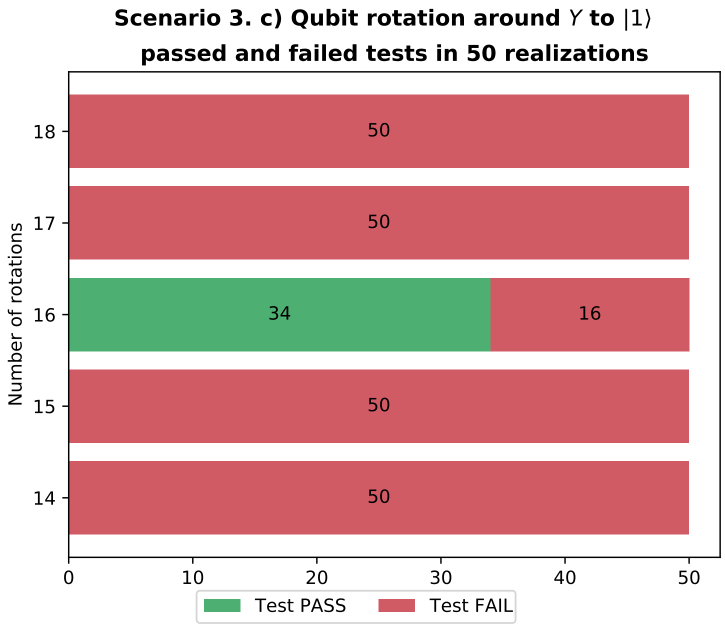


Fig. 6.14: Number of passed and failed tests for the rotation around Y to $|1\rangle$, from a total of 50 test realizations. Steps of 14, 15, 16, 17, and 18 $\frac{\pi}{16}$ angles are shown.

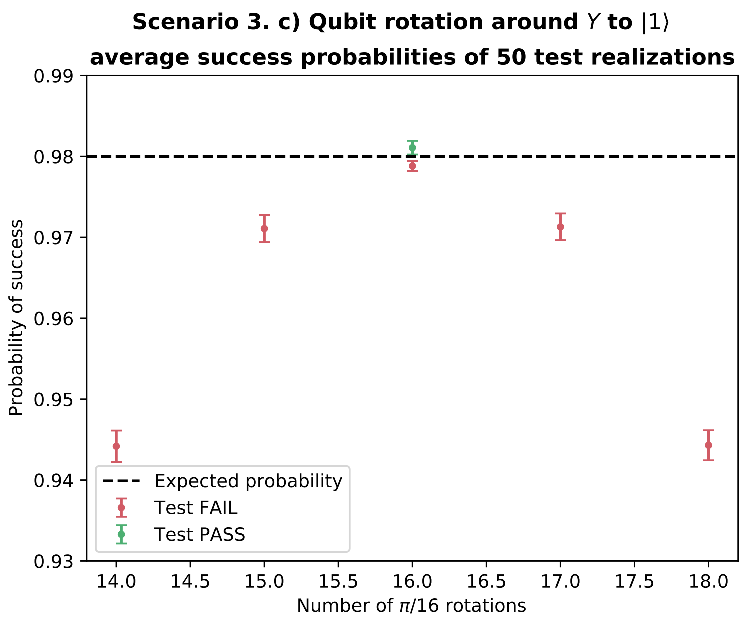


Fig. 6.15: Probability of success of the rotation around Y to $|1\rangle$ tests. The average and standard deviation from a total of 50 test realizations is shown.

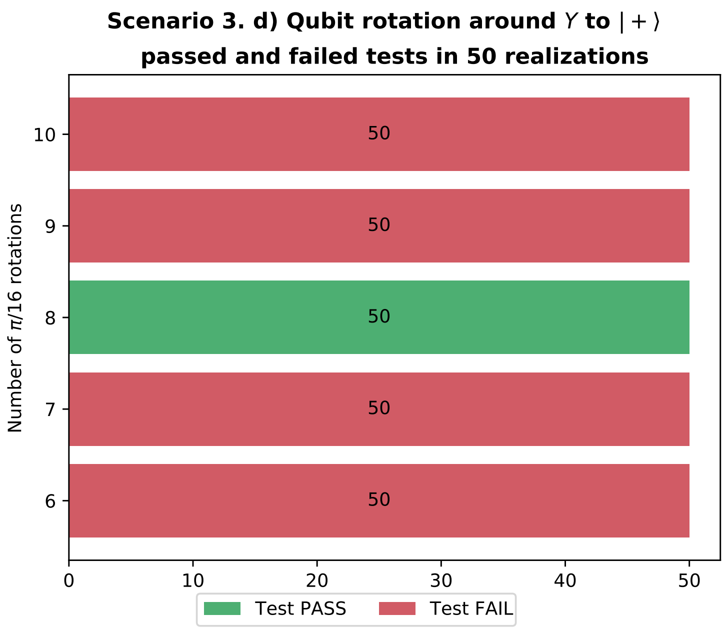


Fig. 6.16: Number of passed and failed tests for the rotation around Y to $|+\rangle$, from a total of 50 test realizations. Steps of 6, 7, 8, 9, and 10 $\frac{\pi}{16}$ angles are shown.

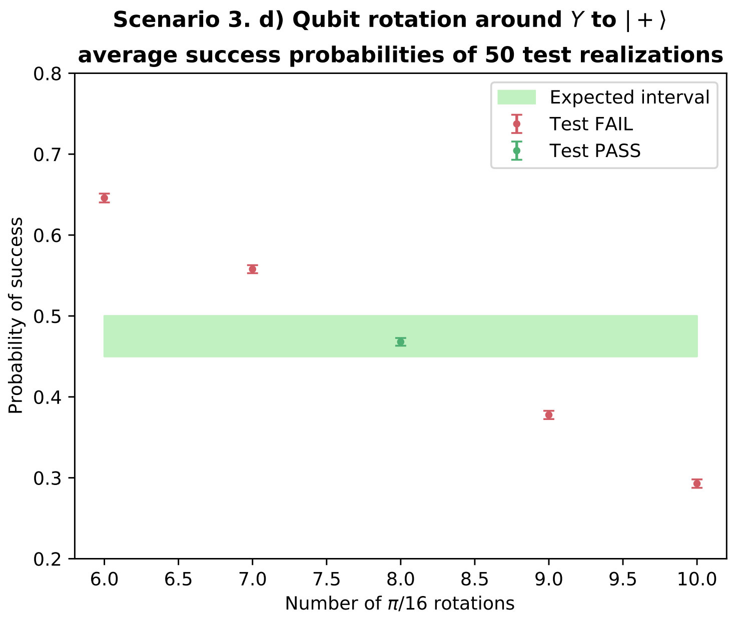


Fig. 6.17: Probability of success of the rotation around Y to $|+\rangle$ tests. The average and standard deviation from a total of 50 test realizations is shown.

7 Conclusion

It is good to have an end to journey toward; but it is the journey that matters, in the end.

— **Ursula K. Le Guin**
Writer

In this chapter we conclude this work by summarizing the key learnings gathered from the development of QuTAF, the first test automation framework for quantum applications. We also discuss the main results from using QuTAF for testing quantum applications in a real networked quantum node and a simulated quantum device. Finally, some future research opportunities for expanding our understanding about how to test quantum applications from the software engineering point of view, are listed.

7.1 Key Learnings

The most important takeaway is that testing a quantum application is a stochastic process. This is a consequence of the probabilistic nature of the quantum phenomena exploited for performing a quantum computation. We modeled this process as a series of i.i.d. Bernoulli random variables, where each trial succeeds with probability p and fails with probability $1 - p$. Since a quantum test has to be repeated several times, we concluded that test automation was the only practical solution for testing quantum software.

We saw then that the expected outcome of a quantum test is a probability distribution with a known Probability Mass Function, that needs to be compared with the distribution obtained from the individual test outcomes. For this, we used the Hypothesis Testing technique. More specifically, we used the Binomial Inverse Survival Function and the Binomial Proportion Confidence Interval for estimating the probability of success of the test realizations.

We also identified that the test definition language of a test automation framework for quantum applications should provide the ability to specify quantum circuits as part of the test case. Furter, the framework should translate this quantum program into the supported quantum instructions of the System Under Test (SUT). We saw that the keyword-driven approach allows us to easily define keywords for each hardware agnostic quantum instruction, and combine them into other keywords that represent the quantum circuits.

7.2 Results

The highlight from the integration tests run on a real quantum node using the QuTAF, is the failed qubit initialization test. This failure, which root cause was confirmed to be a miscalibration of the quantum device, confirmed that the framework is able to detect deficiencies in current state-of-the-art quantum hardware. Hence, a key result is that our QuTAF can indeed be used for testing quantum applications run on Noisy Intermediate-Scale Quantum (NISQ) devices.

Another result that we can derive from this use case is that, unlike the testing of classical software where the failure of the hardware is hardly considered, when testing quantum applications we must take into account the infidelities and inaccuracies of the quantum hardware. This is specially important when testing quantum applications in NISQ systems. It might, however, become irrelevant when full fault-tolerant quantum computers are available.

The main result obtained from using the framework to test simulated failure scenarios is that our QuTAF is not suitable for testing systems dominated by quantum dephasing noise. To be fair, this would be the case for any testing framework where the applications tested are run on quantum hardware that is limited to measuring in the computational basis. On the other hand, if the system's noise is better modeled by the quantum depolarizing channel, then we observed that our QuTAF is sensitive enough to detect small deviations in the quality of the hardware. Quantitatively, we saw that a 0.5% increase in the depolarizing probability resulted in almost a 400% increase in the number of failed tests.

7.3 Outlook

Even when the test automation framework proposed here was proved to be a viable and useful tool for testing quantum applications, while developing this work, we consciously left unanswered some valid and compelling considerations related to the modeling of the testing process, and the design of the QuTAF itself. These serve as great research questions for further exploring the topic of quantum software testing.

- Our framework focused on the functional testing of quantum applications. Then, the whole topic of how to perform non-functional tests on quantum applications remains open. These include testing the security, maintainability, reliability, or performance of quantum software.
- The testing of quantum applications can be modeled as a different stochastic process, for example a random walk. It would be interesting to compare if and how the test results are affected by using distinct random process models.
- Another assumption made while modeling the testing process was the i.i.d. one, i.e., that the random variables were independent and identically distributed. This is a hard assumption that can not be guaranteed during the execution of the tests. Then, another research venue is to use a model that does not assume that the outcomes of the tests are independent and have the same distribution. Similar methods like the ones proposed in [6], for the analysis of entanglement witnessing experiments with correlated noise, can be used.
- All the quantum applications tested with our QuTAF included the measurement of the qubit as part of the quantum program. However, this will not always be the case. Think of a quantum teleportation subroutine that teleports an arbitrary qubit to a remote quantum node. If we want to test this subroutine, we need to measure the qubit in order to verify that it was correctly teleported. The measurement operation in this case is, crucially, not part of the quantum application tested. Then, another interesting question is how to make the errors incurred during the measurement operation to be part of the test itself.

With the advances in the development of quantum computing platforms and quantum algorithms, the need for mechanisms that allow to test those new applications will grow. The aim of this work was to serve as a first stepping stone in the path to building useful and attested tools that help with the testing, and development, of the future quantum applications.

Bibliography

[1] Scott Aaronson. *The Aaronson $25.00 Prize*. Oct. 2007. URL: `https://www.scottaaronson.com/blog/?p=284` (cit. on p. 17).

[2] Shiva Prasad Adirala. *Robot Framework Metrics Report*. Version 3.1.6. Mar. 14, 2020. URL: `https://github.com/adiralashiva8/robotframework-metrics` (cit. on p. 39).

[3] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Washington, DC, USA: IEEE Computer Society Press, 2014 (cit. on p. 11).

[4] Tim Coopmans, Robert Knegjens, Axel Dahlberg, et al. *NetSquid, a discrete-event simulation platform for quantum networks*. 2020. arXiv: `2010.12535 [quant-ph]` (cit. on pp. 35, 37).

[5] Axel Dahlberg, Matthew Skrzypczyk, Tim Coopmans, et al. "A Link Layer Protocol for Quantum Networks". In: *Proceedings of the ACM Special Interest Group on Data Communication - SIGCOMM '19* (2019). arXiv: 1903.09778, pp. 159–173. URL: `http://arxiv.org/abs/1903.09778` (cit. on pp. 37, 55).

[6] Bas Dirkse, Matteo Pompili, Ronald Hanson, Michael Walter, and Stephanie Wehner. "Witnessing Entanglement in Experiments with Correlated Noise". en. In: *Quantum Science and Technology* (Apr. 2020). arXiv: 2002.12400. URL: `http://arxiv.org/abs/2002.12400` (visited on Apr. 30, 2020) (cit. on p. 71).

[7] J. Eisert, D. Hangleiter, N. Walk, et al. "Quantum certification and benchmarking". In: *arXiv:1910.06343 [cond-mat, physics:quant-ph]* (Feb. 2020). arXiv: 1910.06343. URL: `http://arxiv.org/abs/1910.06343` (cit. on p. 16).

[8] Mark Fewster and Dorothy Graham. *Software test automation*. Addison-Wesley Reading, 1999 (cit. on p. 13).

[9] Steven T. Flammia and Yi-Kai Liu. "Direct Fidelity Estimation from Few Pauli Measurements". In: *Physical Review Letters* 106.23 (June 2011). arXiv: 1104.4695, p. 230501. URL: `http://arxiv.org/abs/1104.4695` (cit. on p. 16).

[10] The Robot Framework Foundation. *Robot Framework User Guide Version 3.2.2*. 2020. URL: `https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html` (cit. on p. 35).

[11] Gayatri Ghanakota. *Testing Frameworks*. URL: `https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/ghanakotagayatri.pdf` (cit. on p. 14).

[12]Alexandru Gheorghiu, Theodoros Kapourniotis, and Elham Kashefi. “Verification of Quantum Computation: An Overview of Existing Approaches”. en. In: *Theory of Computing Systems* 63.4 (May 2019), pp. 715–808. URL: http://link.springer.com/10.1007/s00224-018-9872-3 (visited on Mar. 3, 2020) (cit. on p. 1).

[13]Bernard Homès. *Fundamentals of software testing*. John Wiley & Sons, 2013 (cit. on p. 10).

[14]Zdenek Hradil. “Quantum-state estimation”. In: *Physical Review A* 55.3 (1997), R1561 (cit. on pp. 1, 16).

[15]Yipeng Huang and Margaret Martonosi. “QDB: From Quantum Algorithms Towards Correct Quantum Programs”. en. In: (2019). Artwork Size: 14 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number: 1.0, 14 pages. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10196/ (visited on Oct. 14, 2020) (cit. on p. 19).

[16]“IEEE Standard for Software Test Documentation”. In: *IEEE Std 829-1983* (1983), pp. 1–48 (cit. on p. 10).

[17]“IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (1990), pp. 1–84 (cit. on p. 10).

[18]Zhengfeng Ji, Anand Natarajan, Thomas Vidick, John Wright, and Henry Yuen. “Mip*= re”. In: *arXiv preprint arXiv:2001.04383* (2020) (cit. on p. 18).

[19]Wojciech Kozlowski, Axel Dahlberg, and Stephanie Wehner. “Designing a Quantum Network Protocol”. In: *arXiv preprint arXiv:2010.02575* (2020) (cit. on pp. 43, 44).

[20]Wojciech Kozlowski and Stephanie Wehner. “Towards large-scale quantum networks”. In: *Proceedings of the Sixth Annual ACM International Conference on Nanoscale Computing and Communication*. 2019, pp. 1–7 (cit. on p. 43).

[21]Benjamin Lévi, Cecilia C López, Joseph Emerson, and David G Cory. “Efficient error characterization in quantum information processing”. In: *Physical Review A* 75.2 (2007), p. 022314 (cit. on p. 1).

[22]Urmila Mahadev. “Classical verification of quantum computations”. In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2018, pp. 259–267 (cit. on p. 18).

[23]A. Miranskyy and L. Zhang. “On Testing Quantum Programs”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 2019, pp. 57–60 (cit. on p. 19).

[24]Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. en. ISBN: 9781107002173 9780511976667 Library Catalog: www.cambridge.org Publisher: Cambridge University Press. Dec. 2010 (cit. on p. 17).

[25]John Preskill. “Quantum Computing in the NISQ era and beyond”. en-GB. In: *Quantum* 2 (Aug. 2018). Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften, p. 79. URL: https://quantum-journal.org/papers/q-2018-08-06-79/ (visited on Mar. 4, 2020) (cit. on p. 1).

[26]Robert Rand. “Formally Verified Quantum Programming”. In: *Publicly Accessible Penn Dissertations* (Jan. 2018). URL: `https://repository.upenn.edu/edissertations/3175` (cit. on p. 18).

[27]Mark B Ritter. “Near-term Quantum Algorithms for Quantum Many-body Systems”. In: *Journal of Physics: Conference Series*. Vol. 1290. 1. IOP Publishing. 2019, p. 012003 (cit. on p. 1).

[28]R. Shaydulin, H. Ushijima-Mwesigwa, C. F. A. Negre, et al. “A Hybrid Approach for Solving Optimization Problems on Small Quantum Computers”. In: *Computer* 52.6 (2019), pp. 18–26 (cit. on p. 1).

[29]Ivan Šupić and Joseph Bowles. “Self-testing of quantum systems: a review”. In: *Quantum* 4 (Sept. 2020), p. 337. URL: `http://dx.doi.org/10.22331/q-2020-09-30-337` (cit. on p. 1).

[30]Stein Emil Vollset. “Confidence intervals for a binomial proportion”. In: *Statistics in medicine* 12.9 (1993), pp. 809–824 (cit. on pp. 29, 53).

[31]C. M. Wilson, J. S. Otterbach, N. Tezak, et al. *Quantum Kitchen Sinks: An algorithm for machine learning on near-term quantum computers*. 2019. arXiv: `1806.08321 [quant-ph]` (cit. on p. 1).

[32]Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. “Verification of quantum programs”. en. In: *Science of Computer Programming* 78.9 (Sept. 2013), pp. 1679–1700. URL: `http://www.sciencedirect.com/science/article/pii/S0167642313000774` (cit. on p. 18).

List of Figures

List of Tables

Colophon