RuntimeSpeQ: Specifying Runtime Monitors for Quantum Programs

Flavio Melinte-Citea
Department of Informatics, King's College London
London, United Kingdom
flavio.melinte_citea@kcl.ac.uk

Mohammad Reza Mousavi
Department of Informatics, King's College London
London, United Kingdom
mohammad.mousavi@kcl.ac.uk

Abstract

With the growing maturity of quantum computing, robust verification techniques become essential. In this paper, we introduce RuntimeSpeQ: a specification language for the runtime behaviour of quantum programs. We then propose a scheme to translate RuntimeSpeQ into runtime monitors, exploiting two types of runtime assertions. We evaluate our approach in terms of fault detection and equivalence checking capabilities as well as reusability of specifications. The results show promising runtime fault detection capabilities, with several faults detected before the final steps of the programs. Our experiments show that our monitors found 40% of faults in erroneous implementations of Quantum Phase Estimation before the final result was checked, and 90% of faults for Grover's Algorithm. They also show no false negatives in equivalence checking. Also, across four different implementations of one case study, we found that we could reuse the same specification structure by only making minor changes, such as changing the indices pertaining to qubits or execution steps, showing the promise of reusability.

ACM Reference Format:
Flavio Melinte-Citea and Mohammad Reza Mousavi. 2026. RuntimeSpeQ: Specifying Runtime Monitors for Quantum Programs. In *Proceedings of 7th International Workshop on Quantum Software Engineering (Q-SE 2026).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation

Quantum computing has a proven theoretical advantage over classical computing in terms of computational complexity [1]. For the theoretical bounds to be realised, we are awaiting the arrival of powerful and scalable fault tolerant quantum computers. However, the practical advantages of quantum computing are likely to be demonstrated earlier using powerful hybrid architectures that integrate quantum computing and classical (high-performance) computing [2]. Such hybrid architectures typically involve repetitive calls to a compact parametrised quantum circuit using parameters that are determined and improved in a classical optimisation loop.

Both for future fault-tolerant (larger and deeper) quantum computations and current hybrid computations, we need runtime mechanisms that can monitor the behaviour of the computation and, if indicated, perform adaptations and adjustments. This vision has a long tradition in classical computing, in terms of runtime monitoring, runtime verification, and runtime adaptation.

Q-SE 2026, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.2 Problem Definition

The area of runtime monitoring is, however, very much under-researched in the domain of quantum computing. While there are some early research ideas about the implementation and placement of runtime assertions [3–6], to our knowledge, there are no full-fledged runtime monitoring or runtime verification frameworks for quantum circuits (and beyond that for hybrid architectures). We refer to the Related Work section for a more detailed analysis of available results.

In this paper, we bridge this identified gap and come up with a language for runtime monitoring of quantum circuits and provide a prototype implementation of it for some small case studies. The main objectives of the paper are thus: 1) to design a language for runtime monitoring of quantum circuits that allows for reusable and flexible specification of monitors; 2) a way of implementing such monitors from their specifications, and 3) to evaluate the effectiveness of the proposed framework in terms of fault detection and reusability of our monitoring specifications across different implementations.

We envisage that by integrating existing ideas and frameworks from classical runtime monitoring, our framework will provide a solid foundation for runtime monitoring and adaptation of hybrid quantum-classical architectures, as well as future fault tolerant quantum computations.

1.3 Research Questions

To evaluate the effectiveness of our approach, we propose the following research questions and answer them through a designed experiment that is carried out using our prototype implementation:

RQ1: *How successful is our approach at finding faults during runtime?*
To ensure that our monitors find faults during runtime, as that is their main purpose, we measure the percentage of faults detected, as well as whether they are capable of early diagnosis.

RQ2: *Can the generated monitors identify equivalent programs?*
Beyond fault detection, we want to know how generalisable the verification results are over implementations that are equivalent, relative to the specification. That is, if two programs satisfy the same specification, do they both pass the verification? This is useful for checking different implementations, such as optimised versions of the same algorithm. For this purpose, we are interested in the rate of false positives

across equivalent implementations (relative to a specification).

RQ3: *How easily can specifications be adapted for different implementations of the same algorithm?*
Finally, we explore the flexibility of our specification language. For practical purposes, specifications need to be able to be adapted and reused with ease. The design of the language needs to be abstract enough to go beyond specific implementation details, and expressive enough to capture semantic variations. To this end, we will compare different variations of the same algorithm to evaluate how specifications need to be adapted to be satisfied.

1.4 Structure

The remainder of the paper is structured as follows: We review the related work in Section 2 in order to position our research. We describe the RuntimeSpeQ language in 3, as well as how monitors can be generated from specifications. Section 4 details how we designed the experiments to address our research questions. The results of those experiments are covered in Section 5. Finally, we conclude the paper and discuss ideas for future work in Section 6.

2 Related Work

2.1 Runtime Verification

This research is based on the long-standing tradition of classical runtime verification [7–9]. Our specification language is inspired by stream runtime verification languages such as LOLA [10] and Striver [11], as well rule-based languages such as EAGLE [12]. These languages are aimed at maximising reuse while providing sufficient expressiveness, which aligns with our design goals.

2.2 Quantum Software Testing and Verification

Many classical software verification techniques have been adapted to quantum programs. For example, fuzz testing, introduced by Wang et al. through QuanFuzz [13], a quantum fuzz testing tool that uses branch coverage to guide test input generation. Such a tool could be used to generate inputs for runtime monitors.

Some testing techniques that rely on property specification include property-based testing and metamorphic testing. Initial work on quantum property-based testing was done by Honarvar et al. with QSharpCheck [14], a tool for the Q# language. QuCheck [15] is an improved version targeting the Qiskit language. Abreu et al. [16] proposed a metamorphic testing approach which encodes metamorphic relations into circuits in Qiskit. Independently, Paltenghi and Pradel [17] developed their own approach, MorphQ. In evaluating our approach, we use some earlier datasets and approaches for property-based testing [15].

Runtime verification has its roots in model checking [18], so they share some similarities, including property specification. Feng et al. [19] propose a model checking approach for quantum programs. They introduce *quantum Markov chains* as a model for quantum computations and *quantum computation tree logic* to specify properties. While for efficiency reasons, we use runtime assertions, described below, in our monitoring rules, we expect more sophisticated logical specifications (and potentially their temporal extensions) can be embedded into our rule-based specifications.

2.3 Quantum Runtime Assertions

Various approaches have been proposed for implementing runtime assertions for quantum programs. These assertions can be used to implement quantum runtime monitors.

The first is Stat by Huang and Martonosi [3], who propose statistical assertions using “quantum breakpoints” to make measurements at various points in the program; they use statistical analysis on measurement outcome distributions to determine whether assertions hold.

Another approach, introduced by Zhou and Byrd [20], makes use of ancilla qubits and controlled operations to obtain information indirectly about relevant qubits, avoiding the use of Huang and Martonosi’s “quantum breakpoints” which stop the computation.

Li et al. make use of projective measurements in their Proq [5] assertion scheme, which similarly avoids stopping the computation early. It relies on the fact that projecting into a subspace will leave the state unchanged if the state is already in that subspace.

More recently, Oldfield et al. introduced Bloq [21], a runtime assertion scheme based on expectation value measurements of Pauli operators. Their contribution is complementary to those of this paper, and our approaches could potentially be integrated in the future.

For the purposes of our experiments, we implemented our monitors using a combination of Proq and Stat. This combination gave us a lot of flexibility regarding the types of assertions we could implement, as Proq allows for precise equality assertions, while Stat allows for more general ones about probability distributions. However, we designed our specification language to be abstract enough to allow for monitors to be implemented in different assertion schemes. This opens up possibilities to extend our implementation in the future and gain more efficiency.

3 Language Overview

RuntimeSpeQ is a specification language that describes the runtime behaviour of quantum programs at a high level. Specifications in the language only consider the states at each logical “step” in the program.

We consider a program to be composed of a series of such “steps”, which may correspond to an arbitrary number of operations in the underlying circuit. The first and final steps correspond to the initial and final states of the system, while the rest are determined by the placement of barriers. As such, the second step will correspond to the first barrier in the circuit, the third step will correspond to the second barrier, and so on.

A *program trace* is a sequence of states produced by a program, each corresponding to a step in the program. A trace is said to satisfy a specification if every rule defined in the specification holds at every step in the program trace. For simplicity, we will say that a program satisfies a specification if all of its possible traces satisfy it. We consider two programs to be *equivalent* under a specification if they both satisfy that specification.

A RuntimeSpeQ specification is made up of three types of expressions:

- Input variable definitions
- Constant definitions
- Rule definitions

Input variables are variables used in the rule definitions, which depend on the specific instance of the algorithm being specified. For example, the number of qubits can be provided as an input variable, or the expected output of the algorithm.

Constants, on the other hand, have the same value across different executions of the program and different monitoring sessions.

Rules are Boolean expressions made up of a *trigger* and a *monitor*. At every step where the trigger condition holds true, the monitor must also hold for the whole rule to hold true.

3.1 Syntax

Specifications are structured as follows:

$$\begin{array}{ll} T_{v_1} & v_1 \\ & \dots \\ T_{v_n} & v_n \\ T_{c_1} & c_1 = x_1 \\ & \dots \\ T_{c_n} & c_n = x_n \\ & t_1 \longrightarrow m_1 \\ & \dots \\ & t_n \longrightarrow m_n \end{array}$$

where

- Each v_x is an input variable of type T_{v_1}.
- Each c_x is a constant of type T_{c_1}, and x_n is its corresponding value.
- Each pair t_x, m_x are Boolean expressions and represent the *trigger* and the *monitor* for the rule $t_x \longrightarrow m_x$.

RuntimeSpeQ can access both the current *step* number in the trace and its corresponding state. The current step is accessed through a special *step* variable. Meanwhile, the state of a qubit at the current step can be accessed through *state[x]*, where x is the index of the given qubit. In the case of a register, the syntax is *state[x, y]*, where x and y represent the (inclusive) lower and (exclusive) upper bounds of the register, respectively.

The state at a step different from the current one can be accessed through the offset operator **@**, where *step[x]@t* refers to the state of the qubit with index x at step t. This t can either be a number or an arithmetic expression based on the *state* variable. If an offset leads to a step lower than 0, the state at step 0 will be accessed instead. Likewise, an offset beyond the trace length will access the final state.

There are various kinds of assertions that can be made within rules. This includes:

- Equality assertions between states: $s_1 = s_2$ and $s_1 \neq s_2$. It should be noted that states are only asserted to be equal up to a global phase.
- Assertions about the current step number (e.g. *step* = t).
- Probability assertions, using the syntax *prob(s, x)* to refer to the probability of measuring value x when in state s (e.g. $|prob(s, x) - p| \leq \epsilon$), as well as *most(s)* to refer to the most probable measurement outcome (e.g. $most(s) = x$)
- The following general assertions about states: *classical(s)* which checks whether s is a classical state or not, and *uniform(s)* which checks whether s is a uniform superposition state.

We started off with this simple set of assertions by analysing a set of canonical benchmarks from the literature, also in our past research. We plan to extend this set and we will keep our design extensible with new assertion types and assertion monitoring mechanisms (presented in the next section).

3.1.1 Examples. The following is an example of a specification with a single rule:

$$true \longrightarrow \text{state}[0] = |0\rangle$$

This specification states that, at every step of the program, the state of the qubit at index 0 will be equal to $|0\rangle$ (up to a global phase). Here, $state[0]$ refers to the state of the qubit at index 0 at the current step. The program in Figure 1 satisfies the specification, as the state of the qubit at index 0 will remain $|0\rangle$ throughout the computation.

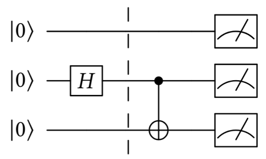

Figure 1: Example quantum circuit

Because specifications only describe the behaviour at each barrier, the program in Figure 2 will also satisfy the specification. This is because the program has three steps: at step 0, the state of the system is $|000\rangle$; at step 1 it becomes $|010\rangle$, and at step 2 it goes back to $|000\rangle$. Thus, relative to this, albeit limited, specification, this program is equivalent to the program in Figure 1.

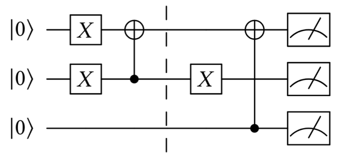

Figure 2: Example quantum circuit

The following is another example specification:

$$\begin{array}{lll} (\text{step} = 0) \vee (\text{step} = 2) & \longrightarrow & \text{state}[0, 2] = |000\rangle \\ (\text{step} = 1) & \longrightarrow & \text{state}[0, 2] = |010\rangle \end{array}$$

This specification states that the state of the system at steps 0 and 2 will be equal to $|000\rangle$, and $|010\rangle$ at step 1. Thus, the program in Figure 2 will also satisfy it, but not the program in Figure 1.

3.2 Generating Monitors

Our approach to generating monitors is to translate rules into runtime assertions. RuntimeSpeQ was designed to allow translation into various different assertion schemes, but in this section we will only cover the two that were used for our experiments: Stat [3] and Proq [5].

Stat is able to make general assertions about the state of a system by analysing probability distributions post-measurement. This is achieved using "quantum breakpoints", which are points in the program where the computation is stopped to perform measurements. The drawback of this approach is that only one assertion can be verified per execution, so the required number of system runs increases for each assertion needed.

The two types of Stat assertions that are of interest to us are classical and superposition assertions. Classical assertions verify whether a state is a specific classical state, while superposition assertions assert that the state is in a uniform superposition.

Classical assertions verify whether the probability distribution at the location of the assertion is unimodal with a peak at a specific value. For example, asserting that the state at a specific step is the classical state $|00\rangle$ requires: 1) making multiple measurements at that step; 2) obtaining a probability distribution from those measurements; 3) performing a goodness-of-fit test with the hypothesis that the distribution is unimodal with a peak at $|00\rangle$, and 4) verifying that the p-value is large enough.

Superposition assertions work in a similar way, except the hypothesis is that the distribution is uniform.

Proq can make precise assertions about system states through the use of projective measurements. Projective measurements can be implemented using a rotation to the computational basis, a measurement, then a rotation back to the asserted basis. For example, asserting that qubit is in state $|+\rangle$ requires: 1) applying a Hadamard gate; 2) making a measurement; 3) applying the Hadamard gate again to restore the state (if it was indeed $|+\rangle$), and 4) verifying whether the measurement result was 0. The advantage is that multiple assertions can potentially be verified in a single run, but at the cost of increased circuit depth.

The rest of this section will cover how RuntimeSpeQ rules can be translated into a combination of Stat and Proq assertions, with some extensions to both of these schemes where required.

Equality assertions. Asserting that a system state is equal to a state vector $|\psi\rangle$ can be done by Proq, as long as a circuit to produce $|\psi\rangle$ is provided. In the case where two system states are being compared (e.g. $state[x]@5 = state[x]@3$), a segment from the circuit under test can be used (e.g. the segment of the circuit up to step 3).

Proq cannot, however, assert that two states are *not* equal. Unless the states are orthogonal to each other, the projective measurement will modify the state of the system, and the program will have to be stopped. Because of this, we propose combining Proq with Stat to implement this type of assertion. This can be done by performing Stat's classical assertion on the result of a projective measurement. For example, if we want to assert that the state at a specific step is different from $|+\rangle$, we can add a Hadamard gate, and then a Stat classical assertion for the value 0. If the state is indeed different from $|+\rangle$, this assertion should fail.

Probability assertions. Although Stat makes assertions about probability distributions, rather than single probabilities, the same idea of quantum breakpoints can be applied for probability assertions. In this case, if we make an assertion about the probability of some outcome at a specific state (e.g. $\text{prob}(\text{state}[0,n], x) \geq y$) we make multiple measurements at the inserted breakpoint, then estimate the probability based on those measurements. As for assertions relating to the most probable outcome (e.g. $\text{most}(\text{state}[0,n]) = x$), we can just retrieve the outcome that was measured the most amount of times.

General assertions. The *uniform* assertion corresponds to the superposition assertion in Stat, which checks for a uniform distribution. The *classical* assertion in RuntimeSpeQ can be implemented similarly to Stat's classical equality assertion, except instead of checking that the distribution is unimodal with the peak at a specific point, the peak point is unspecified.

Step assertions. By default, the trigger of a rule is verified for every step in the program. For example, the rule

$$\text{uniform}(\text{state}[0,n]) \longrightarrow \text{state}[0,n] = |0\rangle$$

will require first verifying at which steps the state of the system is in a uniform superposition; only after that can the assertion corresponding to the monitor be verified. Step assertions can be used to determine in which steps to insert assertions. The rule

$$(\text{step} > 0) \wedge (\text{step} < 4) \longrightarrow \text{state}[0,n] = |0\rangle$$

will insert equality assertions at steps 1, 2 and 3. Using the @ operator can override this. An extreme example would be

$$(\text{step} > 0) \wedge (\text{step} < 4) \longrightarrow \text{state}[0,n]@5 = |0\rangle$$

where the only assertion generated will be inserted at step 5.

4 Experiment Design

4.1 Subject Systems

To evaluate our research questions, we picked two algorithms for our experiments: Quantum Phase Estimation (QPE) and Grover's Algorithm. We chose QPE because it is the cornerstone of much of the current NISQ and future fault-tolerant algorithms that are likely to deliver short and long term quantum advantage. QPE is a challenging algorithm for verifying runtime properties, as its main step modifies each qubit once, and it relies on phase kickback, which results in runtime states that are vastly different and difficult to verify. The latter was chosen as an algorithm with multiple iterations, with some expected progress between them, which makes for a good case study for evaluating runtime monitors, due to its extended runtime behaviour.

For each of the two algorithms, we wrote a specification, which we then manually translated into assertions. These assertions were evaluated against various Qiskit implementations of each algorithm. We tested each assertion separately, 30000 times per implementation tested. The steps after translation are all automated and can be found in our lab package [22]. We ran our experiments using Qiskit 2.2.1, the Qiskit Aer Simulator, and Python 3.10.12, on an Ubuntu 22.04 laptop.

4.1.1 *Quantum Phase Estimation.*

Quantum Phase Estimation (QPE) is an algorithm for estimating the phase θ of a unitary U, with respect to an eigenvector $|\psi\rangle$ with eigenvalue $e^{2\pi i\theta}$.

Given U and $|\psi\rangle$, the algorithm will output $2^n\tilde{\theta}$, where n is the number of qubits used for the estimation, and $\tilde{\theta}$ is an estimate of θ with error ϵ.

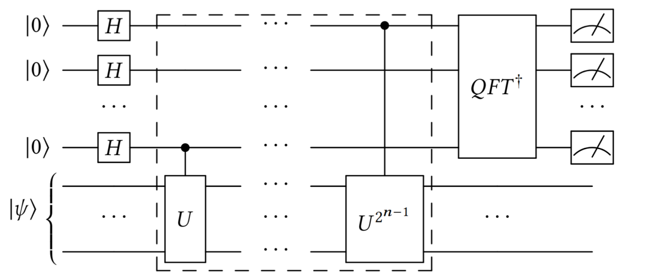

Figure 3: An implementation of Quantum Phase Estimation

Figure 3 shows the schematic for an implementation of QPE. The steps of the algorithm are as follows:

(1) The top register is initialised to $|0\rangle^n$, while the bottom register is set to $|\psi\rangle$.
(2) Hadamard gates are applied to the top register, bringing it to the state $|+\rangle^n$.
(3) For each qubit q_j in the top register, the controlled version of $U^{2^{n-j-1}}$ is applied, with q_j as the control and the bottom register as the target. Due to phase kickback, this will have an effect on the top register, but it will remain in a uniform superposition. As for the bottom register, since $|\psi\rangle$ is an eigenvector of U, the state of the register will remain equal to $|\psi\rangle$ (up to a global phase).
(4) The inverse of the Quantum Fourier Transform is applied to the top register, producing an estimation of $2^n\theta$.

Given this description of the algorithm, we wrote the following specification of the algorithm:

$$\begin{array}{lll}
\text{int } n, m & & \\
\text{real } \theta, \epsilon & & \\
\text{state } |\psi\rangle & & \\
\textit{true} & \longrightarrow & \text{state[n, n + m]} = |\psi\rangle \\
(\text{step} = 0) & \longrightarrow & \text{state[0, n]} = |0\rangle^n \\
(\text{step} = 1) & \longrightarrow & \text{state[0, n]} = |+\rangle^n \\
(\text{step} = 2) & \longrightarrow & \text{uniform(state[0, n])} \\
(\text{step} = 3) & \longrightarrow & |\frac{\text{most(state[0, n])}}{2^n} - \theta| \leq \epsilon
\end{array}$$

In this specification, n is the size of the top register, while m is the size of the bottom register, and ϵ is the error bound. $|\theta\rangle$ is assumed to be known for the instance, but the final rule could be rewritten to use some known estimate $\tilde{\theta}$ instead.

4.1.2 Grover's Algorithm. Grover's Algorithm is an algorithm that solves the problem of unstructured search. The idea is as follows: there is a set S of solutions, and a function f, such that $f(x) = 1$ if $x \in S$, and 0 otherwise. The algorithm takes as input a unitary $U_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle$, and has a very high probability of returning a value from S as output.

Figure 4 shows the general structure of Grover's algorithm. Not shown is the additional register that may be required depending on the choice of U_f. The steps of the algorithm are:

(1) All qubits are initialised to $|0\rangle$.
(2) Hadamard gates are applied to all qubits, changing their state to $|+\rangle$.

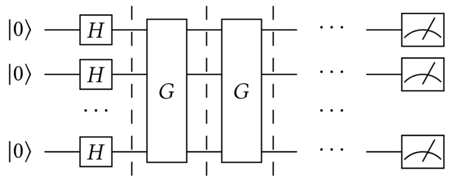

Figure 4: An implementation of Grover's algorithm

(3) The Grover iteration is applied (shown as G in Figure 4). It consists of two steps:
  (a) Apply the unitary U_f. This "marks" all solution states within the superposition with a negative phase.
  (b) Apply the "amplitude amplification" subroutine. This increases the amplitude of "marked" states, and thus their likelihood of being measured.
(4) Repeat step 3 for a number of iterations. This number depends on the size of the search space, as well as the size of S.
(5) Measure all qubits. The outcome will have a high probability of being an element from S.

Based on the above description of the algorithm, we came up with the following specification for the special case $S = \{x\}$:

$$\begin{array}{lll}
\text{int } n, x, k & & \\
(\text{step} = 0) & \longrightarrow & \text{state}[0, n] = |0\rangle^n \\
(\text{step} = 1) & \longrightarrow & \text{state}[0, n] = |+\rangle^n \\
(\text{step} > 1) & \longrightarrow & \text{prob}(\text{state}[0, n], x) > \\
& & \text{prob}(\text{state}[0, n], x)@(\text{step - 1}) \\
(\text{step} = k + 1) & \longrightarrow & \text{most}(\text{state}[0, n]) = x
\end{array}$$

Here n is the number of qubits, x is the singular element in S, and k is the number of iterations. This specification could be adapted to cases where S has more elements by adding a corresponding rule for each element, as well as rewriting the final rule into:

$$(\text{step} = k + 1) \longrightarrow \text{f}(\text{most}(\text{state}[0, n])) = 1$$

4.2 Mutation Analysis

To answer **RQ1** and **RQ2** we made use of mutation analysis, a technique for evaluating the effectiveness of test suites. The idea behind this technique is to apply small changes to a program to generate "mutants". The test suite is then evaluated against each mutant to obtain a *mutation score*, which represents the percentage of mutants "killed" (which means mutants that failed the tests). The higher this mutation score, the more effective the test suite is.

Some mutants are impossible to kill, as they do not introduce any faults. These are known as *equivalent* mutants. The rest are *non-equivalent*.

To run our experiments, we used a set of equivalent and non-equivalent mutants from previous work [15].

For **RQ1**, we had 10 non-equivalent mutants for each algorithm, originally generated using QMutPy [23]. We used these mutants to get a mutation score for each rule, as well as an overall mutation score for the whole specification. These mutation scores represent how effective each rule (and the specification itself) is at detecting faults.

As for **RQ2**, we used a separate set of 5 equivalent mutants per algorithm, originally generated by inserting random gate identities in the original program. We also ensured the changes did not violate the respective specification. Similarly to the non-equivalent mutants, we obtained a mutation score for each rule, as well as the whole specification; however, in this case, a lower mutation score is the desired outcome, as equivalent mutants should not be killed by a correctly implemented monitor.

4.3 Program Variants

As defined in section 2, we consider two programs to be equivalent under a specification if they both satisfy that specification. Because of this definition, different implementations of the same algorithm are not guaranteed to be equivalent. We refer to such implementations as *functionally* equivalent if they always have the same output, regardless of whether they are equivalent relative to the same specifications.

To answer **RQ3**, we produced 4 different "variants" of Quantum Phase Estimation, each one of them being functionally equivalent to our original implementation, but not equivalent relative to the specification we defined. We then modified our specification of QPE for each variant separately, and quantified the number of changes required.

5 Results

5.1 RQ1: Are our runtime monitors successful at finding faults?

Quantum Phase Estimation. Table 1 shows the mutation score for each rule in our QPE specification. An immediate observation is that the mutation score increases from one rule to another. Indeed, every mutant killed by rule 3 was also killed by rule 4, and every mutant killed by rule 4 was also killed by rule 5. This suggests that, at least for this algorithm, rule 5 would have sufficed to kill the same amount of mutants.

Rules	**Mutation Score**
Rule 1	0%
Rule 2	0%
Rule 3	30%
Rule 4	40%
Rule 5	70%
Total	**70%**

Table 1: Mutation scores for QPE non-equivalent mutants

This appears to suggest that errors propagate easily in this algorithm. In classical testability terms, the circuit is non-squeezy [24], i.e., it effectively propagates errors through the computation. It could be linked to the reversibility of unitary operations and the lack of mid-circuit measurements. However, our experiments do not take into account how errors propagate under noisy conditions.

Furthermore, our approach still has the advantage of being able to identify the general locations of faults. For example, 30% of mutants were killed by rule 3, which verifies the correct application of Hadamard gates at the beginning of the algorithm, meaning that the faults happened very early in the program.

Another observation is that no mutants were killed by the first two rules. This is because none of our mutants introduced any faults that could be identified by these rules. With a larger and more diverse set of mutants, these results could potentially differ.

Grover's Algorithm. Our results for Grover's algorithm, shown in Table 2, are immediately more promising. Our first observation is that the third rule, which verifies whether each iteration increases the amplitude of the solution state, killed more mutants than the final rule, which only verifies the final output. This suggests that, unlike with the previous algorithm, some faults that can be detected early may not be identified in the final result.

This could be due to the highly probabilistic nature of Grover's algorithm. However, our results could potentially differ if we set a lower bound for the probability of the solution state being measured.

The other important observation is that not every mutant killed by rule 4 was killed by rule 3. This shows that neither rule 3 nor rule 4 would have been sufficient to kill every mutant.

Based on these results, as well as the high overall mutation score, we can reasonably say that our approach is effective at finding faults.

Rules	**Mutation Score**
Rule 1	0%
Rule 2	20%
Rule 3	90%
Rule 4	70%
Total	**100%**

Table 2: Mutation scores for Grover non-equivalent mutants

5.2 RQ2: Can our runtime monitors identify equivalent programs?

For every rule specified for each algorithm, our experiments yielded a mutation score of *0%* for equivalent mutants. Based on the high mutation score obtained from the non-equivalent mutants for the previous research question, this appears to indicate that our method can successfully identify equivalent programs, relative to some specification.

5.3 RQ3: Can specifications be adapted to alternative implementations?

From the implementation of Quantum Phase Estimation shown in Figure 3, we produced 4 different variants by performing the following changes:

(1) Moving the Hadamard gates at the beginning of the algorithm inside the main section, right before their respective controlled gates.
(2) Moving the bottom register above the top register.
(3) Placing a barrier after each controlled gate.
(4) All of the above at the same time.

For the first variant, the only change that matters for the specification is the fact that the Hadamard gates are no longer applied before the first barrier. This means that, at that point the state of the top register should be $|0\rangle^n$. We can represent this with a single change in the specification:

$$(\text{step} = 1) \quad \longrightarrow \quad \text{state}[0, \text{n}] = |\mathbf{0}\rangle^n$$

The second variant introduces the most amount of changes into the specification, as the lower and upper bounds of each register are different, which introduces 5 changes, one per rule:

$$\begin{array}{lll} \textit{true} & \longrightarrow & \text{state}[\mathbf{0, m}] = |\psi\rangle \\ (\text{step} = 0) & \longrightarrow & \text{state}[\mathbf{m, n + m}] = |0\rangle^n \\ (\text{step} = 1) & \longrightarrow & \text{state}[\mathbf{m, n + m}] = |+\rangle^n \\ (\text{step} = 2) & \longrightarrow & \text{uniform}(\text{state}[\mathbf{m, n + m}]) \\ (\text{step} = 3) & \longrightarrow & |\frac{\text{most}(\text{state}[\mathbf{m, n + m}])}{2^n} - \theta| \leq \epsilon \end{array}$$

The third variant only adds more barriers, which has no effect on the state of the system at any point. However, our specifications depend on barrier placement, so this variant does require changes to the specification. Specifically, 2 changes are required, as the steps at which rules 3 and 4 are triggered are different:

$$\begin{array}{lll} (\text{step} = \mathbf{n + 1}) & \longrightarrow & \text{uniform}(\text{state}[0, \text{n}]) \\ (\text{step} = \mathbf{n + 2}) & \longrightarrow & |\frac{\text{most}(\text{state}[0, \text{n}])}{2^n} - \theta| \leq \epsilon \end{array}$$

The final variant combines the changes of all the previous variants, so its specification requires 8 changes in total:

$$\begin{array}{lll} \textit{true} & \longrightarrow & \text{state}[\mathbf{0, m}] = |\psi\rangle \\ (\text{step} = 0) & \longrightarrow & \text{state}[\mathbf{m, n + m}] = |0\rangle^n \\ (\text{step} = 1) & \longrightarrow & \text{state}[\mathbf{m, n + m}] = |\mathbf{0}\rangle^n \\ (\text{step} = \mathbf{n + 1}) & \longrightarrow & \text{uniform}(\text{state}[\mathbf{m, n + m}]) \\ (\text{step} = \mathbf{n + 2}) & \longrightarrow & |\frac{\text{most}(\text{state}[\mathbf{m, n + m}])}{2^n} - \theta| \leq \epsilon \end{array}$$

We observe that, despite all of the necessary changes, the structure of each rule remains the same. This suggests that our specifications are indeed easy to adapt and reuse. To establish the generalisability of our results, we plan to extend our experiments to a larger set of subjects in the future.

5.4 Threats to validity

We identify here some of the threats to the generalisability of our results:

- **Small pool of mutants and variants:** Our experiments only included 15 mutants per algorithm for the first two research questions, and 4 variants of QPE for the third. A larger, more diverse pool of mutants could lead to different results, as at least one rule was never violated by any of the mutants.
- **Small instances of algorithms:** For the sake of making our experiments run fast, we gave smaller inputs to our programs. We do not know how examples of increased size would affect our results, but it is possible that deeper circuits could reveal more errors during runtime.
- **Usage of barriers:** We defined the "steps" of a program in terms of the locations of barriers. This means that whether a program satisfies a specification or not depends on how barriers are placed. In practice, barriers often go unused or are used inconsistently, so this could represent an obstacle in the practicality of our approach. In the future, we will consider how to automatically split program executions into logical steps.

We do plan to mitigate these risks through more extensive case studies in an extended version of this paper, which will be prepared for an archival journal publication. We do believe the current prototype serves as a suitable proof of concept, and there is enough promise for the underlying ideas to be explored and expanded further.

6 Conclusions

In this paper, we introduced the specification language RuntimeSpeQ, and described how monitors for its specifications can be implemented. We used mutation analysis to evaluate how good our runtime monitors are at finding faults, as well as their usefulness in identifying equivalent programs. Finally, we produced different implementations of the same algorithm to verify how much the original specification needed to be changed for each.

We identify the following avenues for future work:

- Implementing our specification language as a DSL from which runtime monitors can be automatically generated.
- Exploring different assertion schemes, as well as efficient ways to implement and insert assertions.
- Adapting the language to hybrid architectures.
- Extending the language to allow for more types of assertions, and more sophisticated rules.
- Evaluating the effect of noise on the efficacy of runtime monitors.

Acknowledgments

Mohammad Reza Mousavi has been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2, EPSRC project on Verified Simulation for Large Quantum Systems (VSL-Q), grant reference EP/Y005244/1 and the EPSRC project on Robust and Reliable Quantum Computing (RoaRQ), Investigation 009 Model-based monitoring and calibration of quantum computations (ModeMCQ), grant reference EP/W032635/1, ITEA/InnovateUK projects GENIUS, grant reference 600642, and ITEA/InnovateUK GreenCode, grant reference 600643.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.
[2] J. W. Z. Lau, K. H. Lim, H. Shrotriya, and L. C. Kwek, "Nisq computing: where are we and where do we go?" *AAPPS Bulletin*, vol. 32, no. 1, p. 27, Sep 2022. [Online]. Available: https://doi.org/10.1007/s43673-022-00058-z
[3] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. ACM, Jun. 2019, p. 541–553. [Online]. Available: http://dx.doi.org/10.1145/3307650.3322213
[4] J. Liu, G. T. Byrd, and H. Zhou, "Quantum circuits for dynamic runtime assertions in quantum computation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1017–1030. [Online]. Available: https://doi.org/10.1145/3373376.3378488
[5] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428218
[6] D. Rovara, L. Burgholzer, and R. Wille, "A framework for the efficient evaluation of runtime assertions on quantum computers," 2025. [Online]. Available: https://arxiv.org/abs/2505.03885

[7] K. Havelund and G. Rosu, “Synthesizing monitors for safety properties,” in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS ’02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 342–356.

[8] A. Bauer, M. Leucker, and C. Schallhart, “Runtime verification for ltl and tltl,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, Sep. 2011. [Online]. Available: https://doi.org/10.1145/2000799.2000800

[9] K. Havelund, “Runtime verification of c programs,” in *International Workshop on Formal Approaches to Software Testing.* Springer, 2008, pp. 7–22.

[10] B. d’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, “Lola: runtime monitoring of synchronous systems,” in *12th International Symposium on Temporal Representation and Reasoning (TIME’05).* IEEE, 2005, pp. 166–174.

[11] F. Gorostiaga and C. Sánchez, “Striver: Stream runtime verification for real-time event-streams,” in *International Conference on Runtime Verification.* Springer, 2018, pp. 282–298.

[12] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, “Rule-based runtime verification,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2004, pp. 44–57.

[13] J. Wang, M. Gao, Y. Jiang, J. Lou, Y. Gao, D. Zhang, and J. Sun, “Quanfuzz: Fuzz testing of quantum program,” 2018. [Online]. Available: https://arxiv.org/abs/1810.10310

[14] S. Honarvar, M. R. Mousavi, and R. Nagarajan, “Property-based testing of quantum programs in q#,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 430–435. [Online]. Available: https://doi.org/10.1145/3387940.3391459

[15] G. Pontolillo, M. R. Mousavi, and M. Grzesiuk, “Qucheck: A property-based testing framework for quantum programs in qiskit,” 2025. [Online]. Available: https://arxiv.org/abs/2503.22641

[16] R. Abreu, J. a. P. Fernandes, L. Llana, and G. Tavares, “Metamorphic testing of oracle quantum programs,” in *Proceedings of the 3rd International Workshop on Quantum Software Engineering*, ser. Q-SE ’22. New York, NY, USA: Association for Computing Machinery, 2023, p. 16–23. [Online]. Available: https://doi.org/10.1145/3528230.3529189

[17] M. Paltenghi and M. Pradel, “Morphq: Metamorphic testing of the qiskit quantum computing platform,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2413–2424.

[18] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS’07). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1567832608000775

[19] Y. Feng, N. Yu, and M. Ying, “Model checking quantum markov chains,” 2013. [Online]. Available: https://arxiv.org/abs/1205.2187

[20] H. Zhou and G. T. Byrd, “Quantum circuits for dynamic runtime assertions in quantum computation,” *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 111–114, 2019.

[21] N. H. Oldfield, C. Laaber, and S. Ali, “Bloch vector assertions for debugging quantum programs,” *arXiv preprint arXiv:2506.18458*, 2025.

[22] F. Melinte-Citea, “Runtimespeq experiments,” https://github.com/flavio-melinte/runtimespeq-experiments, 2025.

[23] D. Fortunato, J. Campos, and R. Abreu, “Qmutpy: a mutation testing tool for quantum algorithms and applications in qiskit,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 797–800. [Online]. Available: https://doi.org/10.1145/3533767.3543296

[24] D. Clark and R. M. Hierons, “Squeeziness: An information theoretic measure for avoiding fault masking,” *Information Processing Letters*, vol. 112, no. 8, pp. 335–340, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002001901200021X