Delta Debugging for Property-Based Regression Testing of Quantum Programs

Gabriel Pontolillo
Department of Informatics, King's College London
London, United Kingdom
gabriel.pontolillo@kcl.ac.uk

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

ABSTRACT

Manually debugging quantum programs is a difficult and time-intensive process. In this paper, we introduce an automated debugging technique, based on delta debugging and property-based testing, for quantum programs. Our technique automatically identifies the changes made within an update to a quantum program that cause a property-based regression test to fail. To evaluate our technique, we inject faults and semantic preserving changes into three quantum algorithms. We discuss the viability and efficacy of our approach after measuring the percentage of faults and semantic preserving changes. Our results indicate that our method has a high true positive (called sensitivity) and true negative rate (called specificity) and is robust in terms of the amount of changes introduced to the program. Moreover, the sensitivity of the method increases significantly with the number of properties. While the specificity remains stable when increasing the number of properties and inputs.

ACM Reference Format:
Gabriel Pontolillo and Mohammad Reza Mousavi. 2024. Delta Debugging for Property-Based Regression Testing of Quantum Programs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Motivation

We are witnessing increased availability of powerful quantum computing facilities as a service; also there are increasingly promising prospects of applying quantum computing in fields such as material- and drug discovery [1], scheduling [2], and optimisation [3, 4]. With these promising prospects comes an inherent challenge of quality assurance of complex quantum programs. Quantum programs and programming frameworks are becoming increasingly complex and this complexity calls for novel and rigorous testing and debugging frameworks. In particular, there is very little available in terms of debugging tools and techniques for quantum programs [5] and it is unknown how fault localisation can be supported in this context.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper, we address this gap and investigate, to our knowledge, the first application of delta debugging, an automated debugging technique, to quantum programs. We augment our proposed technique with property-based testing [6], which is an established testing approach recently extended to the domain of quantum programs [7].

Applying (delta) debugging to quantum programs turns out to be very challenging due to various reasons:

(1) the input- and output-space of a quantum program are both infinite, each qubit taking values from an infinite Bloch sphere; as a consequence, determining whether the observed failure is the same as the original one is challenging;
(2) unlike traditional debugging, observing the intermediate state causes a state in superposition or entanglement to collapse, permanently altering the state [5];
(3) the input-space of quantum programs is not always just a valuation of qubits, some algorithms [8, 9] take an oracle as an input and this complicates finding the process of simplifying the deltas and finding the root cause; and
(4) the dimension of the input-space increases exponentially with the number of inputs; in particular, entanglement poses serious problems in debugging and makes it difficult to simplify test inputs.

We hypothesise that property-based testing provides a convenient technique to mitigate some of these challenges. Namely, one can rely on properties to generate complex, yet meaningful inputs, and tell apart different types of failure by considering which properties have been violated and how they have been violated.

To simplify the experimental setup, we apply delta-debugging to a regression testing scenario, where the root cause for failure is to be found among a (varying) number of introduced changes. Given the computational cost of debugging, this simplified setting allowed us to manage the time needed for our experiments. We explore some future directions to generalise our approach in the conclusions.

1.2 Research Questions

To evaluate our hypothesis and the proposed setup, we consider a benchmark of three quantum algorithms [8]. We inject behaviour changing faults [9], along with semantic preserving changes and analyse the outcomes of automated debugging on the resulting regression to answer the following research questions:

RQ1: *Is delta debugging effective at locating faults in quantum program updates?*
We further analyse this research question in terms of two sub-questions: RQ1.1 pertaining to the sensitivity (true positive

rate), and RQ1.2 pertaining to specificity (true negative rate), as specified further below:

RQ1.1: *Does the output of delta debugging include the actual failure-inducing change (sensitivity)?*

RQ1.2: *Does the output of delta debugging exclude the semantic-preserving changes (specificity)?*

For these research questions, we are interested to see how the two metrics change with the amount of introduced changes.

RQ2: Does the effectiveness of the technique correlate with the *number of properties* evaluated within the property-based test function?

Similar to the previous question, we divide effectiveness in terms of sensitivity and specificity as follows:

RQ2.1: *Does the sensitivity of delta debugging correlate with the number of evaluated properties?*

RQ2.2: *Does the specificity of delta debugging correlate with the number of evaluated properties?*

RQ3: Does the effectiveness of the technique increase with the *number of inputs* generated per property-based test?

Again, we divide effectiveness in terms of sensitivity and specificity as follows:

RQ3.1: *Does the sensitivity of delta debugging correlate with the number of generated inputs for test?*

RQ3.2: *Does the specificity of delta debugging correlate with the number of generated inputs for test?*

By posing these research questions, we establish whether our proposed approach is effective in finding and isolating the root causes among the rest of changes that do not contribute to the initially-observed failure (RQ1). Also, we study how fault-isolation improves by adding more properties (RQ2) and whether generating more inputs leads to be better fault-isolation (RQ3).

1.3 Contributions

The contributions of the paper are summarised below:

- We develop an integrated technique for delta debugging and property-based testing, which is applicable to regression tests on Qiskit programs.
- We optimise the developed technique for quantum-specific phenomena, e.g., by reusing the delta debugging steps for multiple properties.
- We evaluate our developed technique on three fundamental algorithms and analyse its true positive rate (sensitivity) and true negative rate (specificity) by varying the number of changes, properties, and inputs.

A lab package is available online, comprising the implemented technique, its documentations, generated data, and its analysis [10].

1.4 Structure

The remainder of this paper is structured as follows. We review the related work and position our research in Section 2. In Section 3, we define our proposed techniques and its various steps. We specify the experimental setup to answer our research questions in Section 4 and analyse and discussed the outcomes of our experiments in 5. Finally, we conclude the paper and present the directions of future work in Section 6.

2 RELATED WORK

Delta Debugging. The initial idea [11] [12] as well as the detailed elaboration of delta-debugging in Python [13] are the starting points of our research. We re-used some of the publicly available code for delta debugging [13] and extended it to work with our property-based tests and quantum circuits.

Debugging Quantum Programs. Metwali and van Meter [5] developed a tool that facilitates debugging quantum programs by allowing for dividing the circuits into parts and inspecting the intermediate outcomes to triangulate the fault. While their tool is meant to be used for manual debugging (potentially facilitated by automated tests), our technique is aimed to automate the process of debugging. The tool developed by Matwali and van Meter can be useful in the future development of our technique where we will triangulate faults by modifying the intermediate states.

Li et al. [14] propose the insertion of projection-based runtime assertions within quantum programs for the identification and localisation of faults. The quantum state can be verified in various locations within the circuit by inserting projective assertions as an alternative to statistical assertions. Projective assertions have potential to be used for the verification of postconditions within property based tests and can be used as a future extension of our technique.

Testing Quantum Programs. Wang et al. propose QuCAT [15], a tool for the combinatorial testing of quantum programs. Combinatorial testing, such as the one proposed in QuCAT, could be applied for the 'smart' generation of test inputs, as opposed to the random generation applied for our property based tests, though it would only be useful if the precondition heavily limits the input space.

Wang et al. [16] propose QuanFuzz, a grey box approach to fuzz testing quantum programs. QuanFuzz first analyses the code to identify key areas of the program, particularly measurement operations on qubits. This idea can be used to improve the input generation and replace our random inputs for property-basded testing.

Honarvar et al. [7] took the first step in applying property-based testing to quantum programs. They provide QSharpCheck, a property based testing framework for the Q# language. We take a similar approach for the property based tests used within the property based test oracles, though we go beyond their basic approach in order to simultaneously apply multiple property-based tests along with a statistical correction to control the family-wise error rate.

Tao et al. [17] propose Gleipnir for the calculation of error bounds of quantum programs on noisy hardware. Their methodology computes a new error metric: the $(\hat{\rho}, \delta)$-diamond norm which constrains the input state to generate tighter error bounds, compared to the unconstrained diamond norm. The constrained diamond norm error metric, may have applications for the verification of post-conditions within property based tests, where the constraints of the generated input states are known.

3 PROPOSED TECHNIQUE

Our approach can be summarised as an integration of property-based testing and delta debugging for quantum programs. Properties are an essential part of our approach, because they are used

as test oracles, not only for distinguishing failing and passing runs but also for distinguishing different failures from each other.

We make recursive calls to the delta debugging algorithm, which is initialised with a passing and failing quantum program (represented by the empty set of changes $D1$, and the difference between the passing and failing programs $D2$, respectively) and an auxiliary parameter specifying the granularity of the search initially set to 2 (the coarsest granularity, see below for more information).

At each delta-debugging call, we perform the following four steps:

(1) Calculate the difference $\Delta = D2' \backslash D1'$ between the currently passing $D1'$ and failing $D2'$ sets of deltas.
(2) Split Δ into n subsets: $\Delta_0 \ldots \Delta_n$
(3) For each subset of deltas Δ_i, apply the property based test oracle on $D1' \cup \Delta_i$ and $D2' \backslash \Delta_i$, generating intermediate circuits, and testing them to identify whether the same failure as $C2$ is observed.
(4) Prepare the next recursive delta-debugging call.

Below, we spell out all the key functions required for the initial, and following recursive calls to the delta debugging function.

3.1 Circuit serialisation

The quantum circuits are serialised to a list of quantum circuit instructions (explained below), this is so we can apply a diffing algorithm to identify the deltas between two circuits $C1, C2$ (pre-update, post-update respectively). As of Qiskit 0.45.0, quantum circuits are stored as a list-like object of quantum circuit instructions, such as quantum gates, barriers, and measurements. When a gate is added to a circuit in Qiskit, it is appended to the end of the list.

Semantically equivalent circuits or sections within circuits may generate different deltas due to the order that the quantum gates are inserted. These different representations hamper the delta-debugging process and their effect is comparable to (a large number of) semantic preserving changes within the failing update, which we investigate in **RQ1** (see Sec. 5.1, Figs. 3, 4 for the effect of such changes in the effectiveness of our approach).

3.2 Delta Generation and Application

To apply delta debugging, we need a set of passing deltas $D1$ (initially $\emptyset$, pertaining to the passing program), and a set of failing deltas $D2$ (initially $diff(C1, C2)$, the difference between the initially passing and failing circuit). We utilise a simple diffing algorithm from [18], including their function definitions (with minor modifications).

We first generate the longest common sub-sequence (LCS) matrix $f(i, j)$, where $i = ||C1||$, $j = ||C2||$, containing the lengths of the LCS of circuit instructions between the serialised circuits $C1, C2$.

$$f(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ 1 + f(i-1, j-1), & \text{if } C1_{i-1} = C2_{j-1} \\ max(f(i-1,j), f(i,j-1)), & \text{otherwise} \end{cases}$$

We then utilise the LCS matrix to generate the list of diffs between the two serialised circuits $C1$, $C2$:

$$diff(C1, C2) = diff'(||C1||, ||C2||)$$

$$diff'(i,j) = \begin{cases} \epsilon, & \text{if } i = 0 \text{ and } j = 0 \\ Insert(i, C2(j-1)).diff'(i, j-1), & \text{if } i = 0 \\ Remove(i, C1(i-1)).diff'(i-1, j), & \text{if } j = 0 \\ diff'(i-1, j-1), & \text{if } C1_{i-1} = C2_{j-1} \\ Insert(i, C2(j-1)).diff'(i, j-1), & \text{if } f(i-1,j) \leq f(i,j-1) \\ Remove(i, C1(i-1)).diff'(i-1, j), & \text{if } f(i-1,j) > f(i,j-1) \end{cases}$$

The diff algorithm iterates through the indexes i, j, checking if the circuit instructions at $C1_{i-1}$, and $C2_{j-1}$ are equal, or the end of *either* ($i = 0, j = 0$) or *both* ($i = 0$ and $j = 0$) serialised circuits has been reached. If none of the above conditions are met, adjacent elements in the LCS matrix are compared to take the path that outputs the smallest number of deltas.

Next, the function $apply(\Delta, c)$ receives a serialised to a list of quantum instructions c that describe a quantum circuit (typically C1) and applies an ordered set of deltas Δ to generate an intermediate circuit.

$$apply(\Delta, c) = apply'(\Delta, c, 0)$$

$$apply'(\Delta, c, n) = \begin{cases} c, & \text{if } \Delta = \epsilon \\ apply'(\Delta', c', n+1), & \text{if } \Delta = Remove(i).\Delta' \wedge n = i, c = g.c' \\ g.apply'(\Delta, c', n+1), & \text{if } \Delta = Remove(i).\Delta' \wedge n < i, c = g.c' \\ g.apply'(\Delta', c, n), & \text{if } \Delta = Insert(i, g).\Delta' \wedge n = i \\ g.apply'(\Delta, c', n+1), & \text{if } \Delta = Insert(i, g).\Delta' \wedge n < i, c = g.c' \end{cases}$$

Where:
$\Delta = \epsilon | Remove(i).\Delta' | Insert(i, g).\Delta'$
$c = \epsilon | g.c'$

g represents a circuit instruction.
i represents an index to insert the circuit instruction before, or the index of the circuit instruction to remove.

$apply(D2, C1) = C2$ and $apply(\emptyset, C1) = C1$ holds.

3.3 Property-Based Testing

Once we calculate the deltas, we can re-run the tests for different subsets of deltas and localise the faults by focusing on smaller deltas that feature the same failure.

Determining whether a failure is the **same as the initially-observed failure** is vital to our technique. When testing intermediate circuits during delta debugging, **different failures** may be observed that cause the same property to be violated. As a consequence, we may not simply return a "failure" whenever a property does not hold, we must first check that the failure is the *same as the initially-observed failure, as observed in* $C2$.

We classify two failing circuits as featuring the same failure when the same distribution of outputs is be observed when executing and measuring their outputs for all tested properties. We compare the distribution of outputs by applying the Fisher's exact test to the observed X, Y, and Z basis measurements on all qubits.

The identification of a property failure is contained within the *property based test oracle*; using the oracle and the following verification step, we distinguish between different failures.

The $test(\Delta)$ function, used in the delta-debugging call in Section 3.4, applies our property-based tests oracle as follows.

$test(\Delta) = testOracle(\Delta, P, n)$

$$testOracle(\Delta, P, n) = \begin{cases} PASS, & \text{if } PropertyBasedTests(\Delta, P, n) = \emptyset \\ verify(R, \Delta), & \text{if } PropertyBasedTests(\Delta, P, n) = R = \{(i, p)\} \neq \emptyset \end{cases}$$

Where $PropertyBasedTests(\Delta, P, n)$ is a function that applies (using the *apply* function in Section 3.2) Δ onto $C1$ to create an intermediate circuit and tests the intermediate circuit using the set of properties P, generating n inputs per property.

The $PropertyBasedTests(\Delta, P, n)$ function returns a set of pairs of inputs i that failed a property p $\{(i, p)\}$. If two properties were to fail, one with one input, and one with two inputs, we would get: $\{(|\phi\rangle, EqualProperty), (|\psi\rangle, PhaseProperty), (|\Phi\rangle, PhaseProperty)\}$.

$$verify(R, \Delta) = \begin{cases} FAIL, & \text{if } \forall (i, p) \in R : measure(\Delta, p, i) \approx measure(D2, p, i) \\ INCONCLUSIVE, & otherwise \end{cases}$$

Where $measure(\Delta, p, i)$, initialises the state to i, applies Δ to $C1$ to create the intermediate circuit, as well as potential modifications on the circuit caused by the property based test p, and measures the output states (as defined by p). This state is then compared against the output of the original failure, given the same input and modifications caused by the property based test ($measure(D2, p, i)$ creates the original failing circuit).

The $measure(\Delta, p, i)$ function returns a triple containing the (x, y, z) basis measurements, '$\approx$' is checked by applying a statistical test of equality on the two distributions of measurements.

To assert equality, we apply Fisher's exact test to each qubit's X, Y, and Z basis measurements individually. This implementation is imperfect and is reminiscent of the projection-based assertions in the literature [14]; this typoe of test is not capable of distinguishing between states such as: $|\Phi^+\rangle$ and $|\Phi^-\rangle$. We opted for this approach in our experiments to enhance scalability and reduce the computational overhead of performing full quantum state tomography on multiple qubit states.

3.4 Delta Debugging

Once all the deltas between the passing and failing circuit are identified, and the property based test oracle is defined with all of the circuit's properties to evaluate, we may call the delta debugging function to localise the changes Δ_{fault} that cause the initial failure $C2$. We utilise the delta debugging algorithm [12, 13], as specified below.

$dd(D1, D2) = dd'(D1, D2, 2)$

$$dd'(D1', D2', n) = \begin{cases} dd'(D1', D1' \cup \Delta_i, 2), & \text{if } \exists i =\in \{1, \ldots, n\} \cdot test(D1' \cup \Delta_i) = FAIL \\ dd'(D2' \backslash \Delta_i, D2', 2), & \text{if } \exists i =\in \{1, \ldots, n\} \cdot test(D2' \backslash \Delta_i) = PASS \\ dd'(D1' \cup \Delta_i, D2', max(n \tilde{} 1, 2)), & \text{if } \exists i =\in \{1, \ldots, n\} \cdot test(D1' \cup \Delta_i) = PASS \\ dd'(D1', D2' \backslash \Delta_i, max(n \tilde{} 1, 2)), & \text{if } \exists i =\in \{1, \ldots, n\} \cdot test(D2' \backslash \Delta_i) = FAIL \\ dd'(D1', D2', min(2n, |\Delta|)), & \text{if } n < |\Delta| \\ (D1', D2'), & \text{otherwise} \end{cases}$$

The conditions within the algorithm simultaneously maximise the passing set of deltas ($D1$), whilst minimising the failing set of deltas ($D2$).

The algorithm works by evaluating various subsets of deltas, using a binary search, in order to isolate the failure causing deltas. The subsets of deltas are evaluated with a $test(\Delta)$ function (specified in the previous section), which may return a *pass*, *fail* or *inconclusive* result. Inconclusive results are returned by the test function if the observed failure is different to the failure observed in $C2$. The results of the test function are cached on completion, if the same subset of deltas need to be tested again by the algorithm, the cached result is returned instead.

The test function we apply is identified in Section 3.3, as the *property-based test oracle*, which aims to distinguish between different failures, and thus more accurately steer the delta debugging algorithm.

Once the passing and failing set are respectively maximised and minimised and no further progress is possible, the relative complement is taken between the sets, which yields the isolated fault $\Delta_{fault} = (D2 \backslash D1)$.

For further information on delta debugging, please refer to Zeller's work [12]

4 EXPERIMENT DESIGN

Our research questions (**RQ1-3**) require an assessment of the *effectiveness* of the technique when varying the number of *semantic preserving changes injected* (**RQ1**), the *number of properties* (**RQ2**), and *number of inputs per property* (**RQ3**). We measure the effectiveness of delta debugging through the measurement of two variables:

- *Percentage of faults identified (sensitivity):* Given by the percentage of delta debugging outputs that include the inserted fault .
- *Percentage of semantic preserving changes removed (specificity):* Given by the percentage of inserted deltas that are omitted from the delta debugging output.

In the remainder of this section, we first introduce the subject systems that were used to design our experiments. Then we describe the experimental setup and finally, we specify the experiment design for each and every research question.

4.1 Subject Systems

We used three commonly used quantum programs as our subject systems, Quantum Fourier Transform (QFT), Quantum Teleportation (QT) and Quantum Phase Estimation (QPE), which we briefly discuss below.

Three faulty implementations were tested for each subject system, and an array of experimental groups from the below tables was prepared to evaluate the different research questions:

of Changes
2
4
8
16

of Inputs
1
2
4

of Properties
1
2
3

The independent variables assigned to each experimental group determined the experimental setup for delta debugging. Delta debugging was applied with: a correct implementation of a quantum algorithm, a faulty implementation of the same quantum algorithm (injected with the defined amount of semantic preserving changes, specified below), and a property based test oracle using the defined

number of properties and inputs per property based test. Each experiment was repeated 50 times, recording the number of detected faults and semantic preserving changes in the delta debugging output.

4.1.1 Quantum Fourier Transform (QFT). QFT and its inverse are commonly applied in other algorithms, e.g., those using quantum phase estimation [19], such as Shor's algorithm [20] and quantum counting [21]. The algorithm performs the transformation $|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{\frac{2\pi i j k}{N}} |j\rangle$ where n is the number of qubits in $|j\rangle$ and $N = 2^n$ is the number of qubits in $|j\rangle$. The same transformation is applied as the discrete fourier transform, though instead of transforming a vector of complex numbers, the amplitudes of state $|j\rangle$ are transformed. To implement this in a circuit, the Hadamard gate is applied to each qubit, followed by controlled phase shifts.

We identified, formally specified, and coded three properties of QFT for our property-based tests: 1) linear shift, 2) phase shift, 3) identify after applying reverse QFT on QFT. Due to space limitations, we refer to the lab package [10] for their detailed specification.

4.1.2 Quantum Teleportation (QT). The quantum teleportation protocol transmits an arbitrary and unknown single qubit quantum state $|\psi\rangle$ by sending two classical bits of information. A potential application for quantum teleportation is within quantum communication [22]. To achieve this, a qubit pair is first entangled and shared between two parties (sending and recieving). The sending party performs a sequence of operations followed by a computational basis measurement their half of the entangled pair and the qubit containing the quantum state to teleport. The computational basis measurements are sent to the receiving party, which determine the operations to apply on their half of the entangled pair, after the operations are applied, the receiver's qubit assumes the $|\psi\rangle$ state.

The quantum teleportation implementation that we use transmits the qubit from register one into register three, and applies controlled Pauli-X and controlled Pauli-Z gates to abstract the sending and receiving of classical information, and conditionally applying the respective Pauli gates.

We specified three essential properties of QT: 1) identify of input and output, 2) application of unitaries, 3) appending zero bit strings.

4.1.3 Quantum Phase Estimation (QPE). Phase estimation algorithm is applied as a component within multiple quantum algorithms [19], such as Shor's algorithm and quantum counting, The algorithm evaluates the phase θ of a unitary operator U with respect to an eigenvector $|\psi\rangle$ with eigenvalue $e^{2\pi i\theta}$. The algorithm works by initialising and applying $H^{\otimes n}$ on the upper register with n estimation qubits, and initialising the lower register to $|\psi\rangle$. For each qubit in the upper register $\{q_0, q_1, \ldots, q_{n-1}\}$, the unitary $U^{2^{n-i}}$ is controlled by qubit q_i, and applied to the lower register. Finally, the inverse of the quantum fourier transform ($QFT\dagger$) is applied to the upper register, which is then measured in the computational basis to receive an estimate of θ.

The implementation of the QPE algorithm used in our case study assumes a fixed unitary U, though our three properties are not dependent on this specific unitary: 1) application to eigenvectors with the same eigenvalue, 2) to eigenvectors with different eigenvalues, and 3) initialisation with eigenstate.

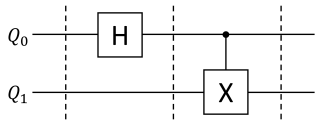

Figure 1: Bell state circuit

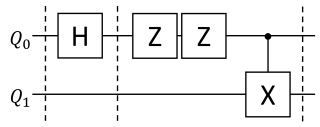

Figure 2: Bell state circuit with a Z-identity injected

4.2 Experimental Setup

We applied the delta debugging algorithm, based on [13], modifying the Python implementation to work with our implementation of diff [18], as well as the property-based test oracles that we propose.

For each case study and their faulty implementations, all remaining independent variable combinations from Section 4.1 were concurrently executed, and repeated 50 times using a Python multiprocessing pool, and stored in a CSV format.

We ran our experiments using Qiskit 0.44.2 and Python 3.11 on a Windows 11 desktop (Ryzen 7 5700x, 16 GB RAM, RTX 3060 Ti).

4.3 RQ1: Effect of Changes

We are particularly interested in measuring how the effectiveness *scales with an increasing number of changes introduced in an update.* This is to gain insight on whether the technique will be effective at locating faults when small or large changes are made to quantum programs.

When performing our experiments on the subject systems, we used the "most complete" application of the technique (three properties evaluated, with four inputs generated per property). Based on that we varied the number of semantic preserving changes injected into the failing implementation of the algorithms, and recorded the percentage of faults identified, as well as the percentage of semantic preserving changes removed

Semantic preserving changes were injected into the circuit by randomly selecting a location within the circuit before, or after an already-present gate at any quantum register (represented by the dotted lines in Fig. 1). A circuit identity, consisting of a pair of identical Pauli gates is then inserted in the chosen location. This process is repeated until the required number of gates ~~is~~ are injected into the circuit. An example that injects two semantic preserving changes can be seen in Fig. 2, a pair of Pauli-Z gates are inserted after the original Hadamard gate in register Q_0.

4.4 RQ2: Effect of Properties

Answering this research question gives us insight whether there is value in increasing the number of properties to evaluate within the property-based test oracle, or if a single property within the oracle is sufficient for the technique to be effective at locating faults.

We performed experiments evaluating one, two, and three properties within the property based test oracle, and measured the percentage of faults identified and changes removed, aggregating the results across all faults.

4.5 RQ3: Effect of Inputs

Our property based test oracle employs multiple individual property based tests that each generate multiple inputs to test. RQ3 seeks to identify the effects of generating different amounts of inputs to evaluate within the property based tests.

The number of inputs generated for each property based test was varied between one, two, and four between experiments. As we are interested in the impact of varying the number of inputs on the "best case" implementation of delta debugging, we analysed the data where we evaluated all three properties included in the oracle.

4.6 Inserting a minimal fault

Three failing implementations of each quantum program were created by inserting a minimal fault into the correct implementation of the circuit. The insertion of the fault was manual, and consisted in the insertion or removal of a quantum gate. Mutation testing techniques for quantum programs also include mutation operators that *replace* quantum gates [23] [24]. We did not include any faults involving the replacement of quantum gates, as they are comprised of both an insertion, and removal delta, of which; one or both deltas may be the minimal fault. To keep the experiments fair, we controlled for the number of faults within the failing implementations of the quantum circuits, and thus omitted the replacement of gates when creating the faulty implementations.

5 RESULTS

5.1 RQ1: Is delta debugging effective?

We divided this question into the following sub-questions pertaining to the sensitivity and specificity of the method, respectively, which are analysed below.

RQ1.1: *Does the algorithm include the actual faults?* We are not only interested in analysing sensitivity, but also its variation against an increasing amount of changes. The results in Figure 3 suggest that the sensitivity of our proposed technique is very high and is robust against increasing changes. In this figure, the percentage of faults identified remains stable across all three algorithms (QT, QFT and QPE) when increasing the number of semantic preserving changes injected (respectively 86%, 97.3%, 91.3% at N=2 to 85.3%, 92.7%, 88% at N=16. When considering data for all three algorithms, grouping the different numbers of semantic preserving changes injected, the median percentage of faults identified was 96%, with a standard deviation of 11.9% (min 58%, max 100%). The Pearson correlation coefficient was -0.08, with a p-value of 0.62, suggesting no significant correlation between the number of semantic preserving changes injected and percentage of faults identified.

RQ1.2 *Does the algorithm exclude semantic-preserving changes?* No strong correlation between the the percentage of semantic preserving changes removed and the number of semantic preserving changes was observed (see Fig. 4; Pearson correlation coeff. 0.10, p-value 0.56). Quantum Teleportation showed an increasing trend

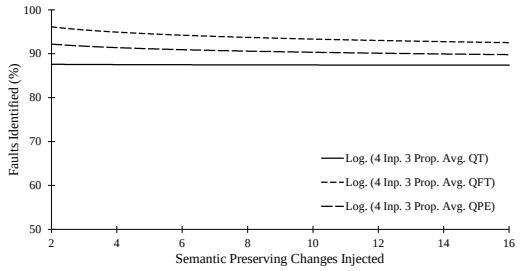

Figure 3: Faults identified with increasing semantic-preserving changes

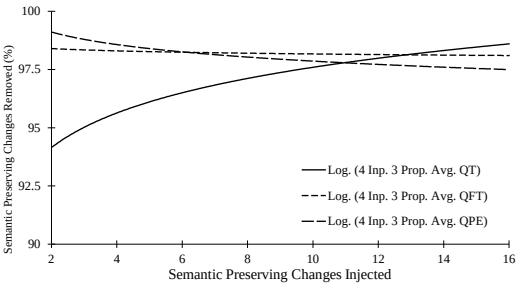

Figure 4: Semantic-preserving changes removed with increasing semantic-preserving changes

starting at a comparatively lower 93.7% of semantic preserving changes removed at N=2, eventually overtaking QFT and QPE. The median percentage of semantic preserving changes was 98.5%, with a standard deviation of 3.4% (min 81%, max 100%). Overall, the specificity of the technique remains constant as the number of semantic preserving changes increases, though it should be noted that the *absolute value* of semantic preserving changes present within the output also increases.

> Delta debugging has shown robustness with respect to debugging small and large sets of changes (updates), evidenced by showing a *constant level of sensitivity and specificity* when increasing the number of semantic preserving changes.

5.2 RQ2: Does effectiveness correlate with the *number of properties*?

Similar to the previous case, we report effectiveness in terms of sensitivity and specificity below.

RQ2.1. Does sensitivity correlate with the number of properties? In Fig. 5, we see a significant increase in the percentage of detected faults with respect to the increasing number of properties evaluated within the property based test oracle (between 55.7% and 67.2% at N=1, and 87.5% and 94.3% at N=3). The median (96%, 96%, 64%) and standard deviation (11.9%, 19.3%, 23.4%) of the percentage of faults identified for three, two, and one, properties showed a respective decrease to the variance in the output across all three case studies, as more properties are evaluated within the oracle. This trend is statistically verified using the Pearson's correlation

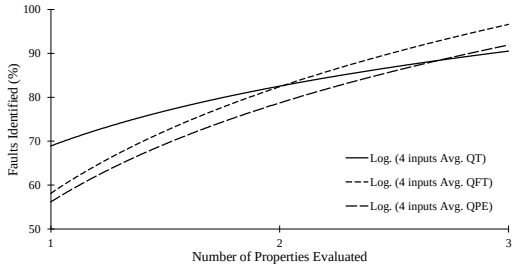

Figure 5: Faults identified as number of properties within oracle increases

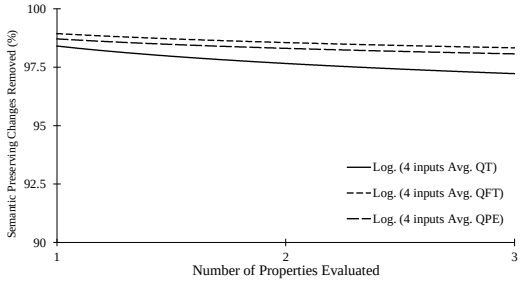

Figure 6: Semantic preserving changes removed as number of properties within oracle increases

test (Pearson's correlation coeff. 0.55, p-value 4.69E-10, showing a moderate correlation).

RQ2.2. Does specificity correlate with the number of properties? Unlike in fault identification, we see little change in the specificity of delta debugging when the number of properties are increased (see Fig. 6; between 98.5% and 98.9% at N=1, and 97.4% and 98.2% at N=3). This trend can be seen across all three case studies (median 98.3%, 98.8%, 99.1%, standard deviation 3.4%, 2.7%, 1.9%). The slight decline in specificity that can be seen in the graph is not verified by statistical test (Pearson's correlation coeff. -0.13, p-value 0.18.

> The sensitivity of delta debugging increases with the number of properties, while its specificity remains constant.

5.3 RQ3: Does effectiveness increase with the *number of generated inputs*?

We divide our results for this research question into the results concerning sensitivity and specificity.

RQ3.1. Does sensitivity correlate with the number of generated inputs? Increasing the number of inputs per property showed a weak positive correlation with percentage of faults identified (Pearson's correlation coef. 0.14), though not enough to be statistically significant (p-value 0.16). The results for quantum teleportation fell marginally with an increase to inputs generated per property, which showed a different trend compared to QFT and QPE (see Fig. 7). The median (96%, 93%, 94%) and standard deviation (11.9%, 16.5%, 22.8%) for four, two, and one inputs generated, showed little change to the median values, though the standard deviation for the percentage of faults identified increased significantly.

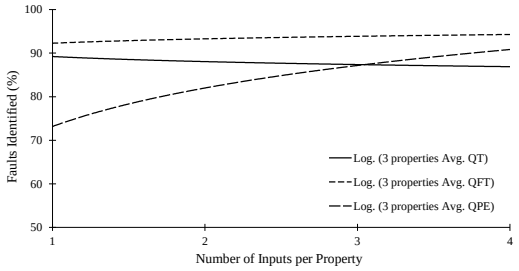

Figure 7: Faults identified as number of inputs generated per property increases

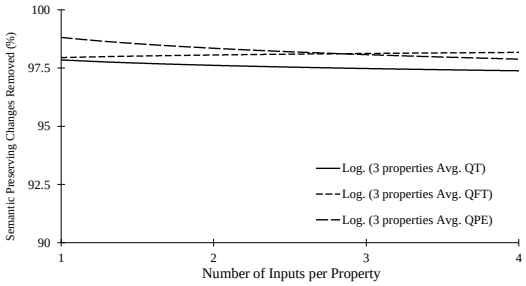

Figure 8: Semantic preserving changes removed as number of inputs generated per property increases

RQ3.2. Does specificity correlate with the number of generated inputs? The percentage of fault preserving changes removed portrayed and even weaker correlation (see Fig. 8) with corresponding to an increase to the number of inputs per property (Pearson's correlation coef. -0.05, p-value 0.58) displaying the potential to identify more faults without the cost of removing less semantic preserving changes. The median (98.3%, 98.5%, 98.9%) and standard deviation (3.4%, 3.1%, 2.6%) values remained constant across the different numbers of inputs generated for each property based test.

> No statistically significant correlation between the the number of inputs and the effectiveness of the technique was observed.

5.4 Threats to validity

Below we provide a number of threats to the generaliseability of the reported analysis.

- **Fault and semantic preserving changes injection**: Manual fault injection is prone to bias. Our utomated method for injecting semantic preserving changes may not represent changes that a person would make when modifying a circuit. Both of these threats may be mitigated by considering real code evolution.
- **Limited input domain for property-based tests**: Some property based tests do not take advantage of the infinite input domain, for our current implementation of property based testing, we only seek to modify the input states of each quantum circuit, rather than inserting different oracles within (for example QPE).

For oracle algorithms, the input to oracle algorithms is generally the oracle itself. It is possible to randomly generate oracles as input for such algorithms, We were able to apply delta debugging to QPE because part of its input is an eigenstate of the controlled unitary, and we maintained the same oracle between the passing and failing circuits.
Relaxing these assumptions would require more complex generators.

6 CONCLUSIONS

In this paper, we proposed an integration of property-based testing and delta-debugging for automated debugging of quantum programs. We evaluated the effectiveness of our proposed technique using a benchmark of three quantum algorithms and nine failing implementations of those algorithms, created by inserting a single fault into a correct implementation. Our delta debugging approach was applied using a correct implementation of the quantum algorithm, alongside a faulty implementation that was injected with a failure-inducing fault along with a varying number of semantic preserving changes. We measured the sensitivity of the technique, i.e., whether the failure-inducing fault was included in the output and specificity, i.e., whether the semantic-preserving changes were excluded. Our evaluation considered a varying number of semantic-preserving changes, properties, and inputs. The proposed technique has a robust sensitivity and stable specificity with the increasing number of semantic-preserving changes.

We have identified the following avenues for future work:

- Extending the context of automated debugging beyond regression testing by using various inputs to a faulty circuit as the context;
- Demonstrating the impact of the verification step within the property based test oracle;
- Exploring other approaches for the verification of property-based tests, such as the application of distance metrics;
- Generating faulty implementations by inserting multiple realistic faults [25], and evaluating;
- Extending the technique to allow for the automated generation of oracle circuits; and
- Assessing the influence of noise on the technique.

Acknowledgments. The authors have been partially supported by the 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, grant reference EP/W032635/1. Also King's Quantum grants provided by King's College London are gratefully acknowledged.

REFERENCES

[1] J. L. Tilly, "Methods for variational computation of molecular properties on near term quantum computers," Ph.D. dissertation, University College London, 2022.
[2] D. Amaro, M. Rosenkranz, N. Fitzpatrick, K. Hirano, and M. Fiorentini, "A case study of variational quantum algorithms for a job shop scheduling problem," *EPJ Quantum Technology*, vol. 9, no. 1, p. 5, 2022. [Online]. Available: https://doi.org/10.1140/epjqt/s40507-022-00123-4
[3] X. Liu, A. Angone, R. Shaydulin, I. Safro, Y. Alexeev, and L. Cincio, "Layer vqe: A variational approach for combinatorial optimization on noisy quantum computers," *IEEE Transactions on Quantum Engineering*, vol. 3, no. 01, pp. 1–20, jan 2022.
[4] T. V. Le and V. Kekatos, "Variational quantum eigensolver with constraints (vqec): Solving constrained optimization problems via vqe," 2023.
[5] S. Metwalli and R. V. Meter, "A tool for debugging quantum circuits," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2022, pp. 624–634.
[6] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L.-A. Fredlund, V. Gulias, J. Hughes, and S. Thompson, "Property-based testing-the protest project," in *Formal Methods for Components and Objects: 8th International Symposium*. Springer, 2010, pp. 250–271.
[7] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in Q#," in *ICSE'20: 42nd International Conference on Software Engineering, Workshops*. ACM, 2020, pp. 430–435. [Online]. Available: https://doi.org/10.1145/3387940.3391459
[8] G. Pontolillo and M. R. Mousavi, "A multi-lingual benchmark for property-based testing of quantum programs," in *3rd IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2022, Pittsburgh, PA, USA, May 18, 2022*. IEEE, 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3528230.3528395
[9] J. Campos and A. Souto, "Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments," in *2nd IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2021, Madrid, Spain, June 1-2, 2021*. IEEE, 2021, pp. 28–32. [Online]. Available: https://doi.org/10.1109/Q-SE52541.2021.00013
[10] G. Pontolillo, "Delta Debugging for Property-Based Regression Testing of Quantum Programs," 1 2024. [Online]. Available: https://figshare.com/articles/software/Delta_Debugging_for_Property-Based_Regression_Testing_of_Quantum_Programs/25075154
[11] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.
[12] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
[13] A. Zeller, "Reducing failure-inducing inputs," in *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2023, retrieved 2023-11-11 18:05:06+01:00. [Online]. Available: https://www.debuggingbook.org/html/DeltaDebugger.html
[14] 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
[15] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Qucat: A combinatorial testing tool for quantum software," 2023.
[16] J. Wang, F. Ma, and Y. Jiang, "Poster: Fuzz testing of quantum program," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 466–469.
[17] R. Tao, Y. Shi, J. Yao, J. Hui, F. T. Chong, and R. Gu, "Gleipnir: Toward practical error analysis for quantum programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2021, p. 48–64. [Online]. Available: https://doi.org/10.1145/3453483.3454029
[18] F. Hartmann, Dec 2020. [Online]. Available: https://florian.github.io/diffing/
[19] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1017/CBO9780511976667
[20] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, Oct. 1997. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172
[21] G. Brassard, P. HØyer, and A. Tapp, *Quantum counting*. Springer Berlin Heidelberg, 1998, p. 820–831. [Online]. Available: http://dx.doi.org/10.1007/BFb0055105
[22] Q.-C. Sun, Y.-L. Mao, S.-J. Chen, W. Zhang, Y.-F. Jiang, Y.-B. Zhang, W.-J. Zhang, S. Miki, T. Yamashita, H. Terai, X. Jiang, T.-Y. Chen, L.-X. You, X.-F. Chen, Z. Wang, J.-Y. Fan, Q. Zhang, and J.-W. Pan, "Quantum teleportation with independent sources and prior entanglement distribution over a network," *Nature Photonics*, vol. 10, no. 10, p. 671–675, Sep. 2016. [Online]. Available: http://dx.doi.org/10.1038/nphoton.2016.179
[23] D. Fortunato, J. CAMPOS, and R. ABREU, "Mutation testing of quantum programs: A case study with qiskit," *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–17, 2022.
[24] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A mutation analysis tool for quantum software testing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1266–1270.
[25] P. Zhao, Z. Miao, S. Lan, and J. Zhao, "Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs," *Journal of Systems and Software*, vol. 205, p. 111805, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002005