Investigating Quantum Cause-Effect Graphs

Noah Oldfield
Simula Research Laboratory and
University of Oslo
Oslo, Norway
noah@simula.no

Tao Yue
Simula Research Laboratory
Oslo, Norway
tao@simula.no

Shaukat Ali
Simula Research Laboratory
Oslo, Norway
shaukat@simula.no

ABSTRACT

Cause-effect graphs have shown promising results in identifying relations among causes and effects of classical software systems, followed by designing effective test cases from them. Towards this end, we investigate the use of cause-effect graphs for quantum programs. Classical cause-effect graphs apply classical logic (e.g., AND, OR) to express these relations, which might not be practical for describing similar relations in quantum programs due to quantum superposition and entanglement. Thus, we propose an extension of cause-effect graphs, where quantum logic inspired functions (e.g., Hadamard) and their generalizations are defined and applied. Moreover, we present a metamodel describing various forms of cause-effect graphs. Finally, we demonstrate a possible method for generating test cases from a quantum cause-effect graph applied to a Bell state quantum program. Lastly, the design and utility of the resulting testing method is discussed, along with future prospects for general quantum cause-effect graphs.

CCS CONCEPTS

• **Theory of computation** → *Quantum computation theory*; • **Software and its engineering** → **Design languages**; **Software design engineering**.

KEYWORDS

cause-effect graphing, quantum software testing, quantum software engineering, quantum program specification

ACM Reference Format:
Noah Oldfield, Tao Yue, and Shaukat Ali. 2022. Investigating Quantum Cause-Effect Graphs. In *The 3rd International Workshop on Quantum Software Engineering (Q-SE'22), May 18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3528230.3529186

1 INTRODUCTION

In the field of quantum computing, calculations are performed on quantum computers by exploiting properties of quantum mechanics such as superposition and entanglement. Quantum software engineering, as an emerging research area, is concerned with creating practical and cost-effective solutions for developing quantum software systems that should be executed fully or partially on quantum computers. Similar to classical software system development, a quantum software engineering life-cycle is also expected to analyze requirements, design, develop, test, deploy and maintain quantum software systems, as discussed by Zhao [22].

Particularly, a number of methods and tools (e.g., [19], [5], [8], [2]) has been proposed for quantum software verification and validation. In some of these approaches, test cases are derived from a kind of specification where the relations between the inputs and outputs of a quantum program are identified. However, in quantum software testing, there does not yet exist a method for the systematic development of such specifications. This motivates the study of cause-effect graphs [12], which have shown to be effective with the deriving of effective test cases of classical software systems, in addition to identifying incomplete specifications. In this paper, we focus on applying both classical cause-effect graphs and extending them for supporting quantum software testing.

Classical cause-effect graphs use classical logic functions, including AND, OR, NOT, to capture relations between the causes and effects within a program specification. It is then natural to study the development of an extension how quantum programs may instead benefit from the quantum logic gates such as Hadamard and Pauli gates (X, Y, Z). These gates will be referred to as *quantum logic functions* or just *functions* in the context of quantum cause-effect graphs. In this paper, we propose the very first quantum cause-effect graph notations for identifying and capturing cause and effect relations within quantum programs to support specification construction, and in supporting quantum software testing. We demonstrate that both classical and our newly proposed quantum cause-effect graph notations can be applied to develop cause-effect graphs at various levels of abstractions. Moreover, to systematically capture concepts related to various cause-effect graphs, we also present a conceptual model implemented in the form of a metamodel as a Unified Modeling Language (UML) class diagram.

We organize the paper as follows: Section 2 presents the works from the literature that are related to our work. Section 3 presents the background relevant for our work in addition to a running example that will be used to demonstrate our ideas. Section 4 presents our metamodel, Section 5 describes classical cause-effect graphs, and Section 6 presents quantum cause-effect graphs. Finally, we present discussion in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

Given their base in quantum mechanics, understanding quantum software may pose challenging for software engineers. Moreover, current quantum programming support is at a lower level of abstraction, e.g., at the quantum circuits level, making it difficult for software engineers to efficiently understand, test, debug, and maintain

Q-SE 2022, May 18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9335-5/22/05…$15.00
https://doi.org/10.1145/3528230.3529186

quantum software. As a result, the software engineering community, in general, has been increasingly realizing challenges related to building and testing reliable quantum software [10, 11, 21, 22].

Quantum software modeling aims at raising the level of abstraction to an appropriate level [1, 3, 4, 14]. There are proposals to use the Unified Modeling Language (UML). For example, a work in [14] proposes to use UML for modeling circuits. An idea of Quantum UML, along with principles for supporting quantum software modeling, was presented in [1]. Moreover, in [3, 4], questions about whether UML is sufficient and domain-specific languages are needed were discussed. In addition, in [4], the authors argued the need for model-driven quantum software engineering. Our work well aligns with these literature, but focuses on a particular modeling notation, i.e., cause-effect graphs, for the purpose of facilitating quantum software testing.

Quantum software testing is gaining a lot of attention, as evidenced by the increasingly number of methods being published, e.g., those based on devising coverage criteria [2], property-based testing [7], search-based testing [18], fuzz testing [15], combinatorial testing [17], and differential testing [16]. Several of them come with dedicated tools such as Quito [19], QuSBT [18], QSharpCheck [7], and Muskit [9]. In this paper, we take a different yet complementary perspective for test case design based on cause-effect graphs.

3 BACKGROUND AND RUNNING EXAMPLE

We introduce basic quantum computing concepts (Section 3.1) and a running example (Section 3.2). Details about cause-effect graphs are provided Section 4.

3.1 Quantum Computing

The fundamental unit of information in quantum computing is the *qubit* and is represented as a two-component *state vector* $|\psi\rangle$ in a Hilbert space.[13] For our purposes, the following representation in the computational basis is sufficient:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where $\alpha, \beta \in \mathbb{C}$ are the *probability amplitudes* and $\{|0\rangle, |1\rangle\} = \{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\}$ are the computational basis vectors.

For larger state spaces consisting of d qubits $q_1, q_2, \cdots, q_d$, the general state vector consists of basis states $|q_1, q_2, \cdots, q_d\rangle$ where $q_k = 0, 1$ for $k = 1, 2, \cdots, d$. We can write a general linear combination for a state space of d qubits with the 2^d basis vectors of the state space as $|\psi\rangle = \sum_{j=1}^{2^d} P_j\,|j\rangle$. Here the basis vectors $|j\rangle$ are mapped from the compact decimal representation and into binary strings [13]. For instance, if $d = 3$ then $|2\rangle = |010\rangle$. Several single two-component qubits form larger composite systems such as $|010\rangle = |0\rangle \otimes |1\rangle \otimes |0\rangle$ by applying the *tensor product* $\otimes$ [13].

Furthermore, when the state vector is measured to yield a program output state $|j\rangle$, it will be measured with probability $p_j = |P_j|^2$ from the normalized probability distribution of the state vector, thus we require $\sum_j^{2^d} |P_j|^2 = 1$.

Table 1: Inputs and outputs of the Bell states program

Input ($\|q_1 q_2\rangle$)	Output
$\|00\rangle$	$\frac{1}{\sqrt{2}}\,(\|00\rangle + \|11\rangle)$
$\|01\rangle$	$\frac{1}{\sqrt{2}}\,(\|01\rangle + \|10\rangle)$
$\|10\rangle$	$\frac{1}{\sqrt{2}}\,(\|00\rangle - \|11\rangle)$
$\|11\rangle$	$\frac{1}{\sqrt{2}}\,(\|01\rangle - \|10\rangle)$

3.2 Running Example: The Bell States Circuit and Code

As a running example, we shall use the common two-qubit circuit (see Figure 1) that takes the input state $|q_1 q_2\rangle$, and yields the respective Bell state as outputs. The circuit applies a Hadamard gate to the first qubit (q_1), before applying a CNOT using the first qubit as a control bit, creating an *entanglement* [13] between q_1 and q_2.

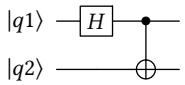

Figure 1: Bell states circuit

The latter is a well known property of the Bell states, i.e., representing entangled qubits, implying that measuring one of the qubits also provides correlated information about the unmeasured one.

4 METAMODEL FOR CAUSE-EFFECT GRAPH

The metamodel, i.e., conceptual model, of our approach is presented in Figure 2 as UML class diagrams. With this metamodel, we aim to provide an overall view of the context, capture the important concepts of the classical and quantum cause-effect graph notations, and illustrate their usages for test case design.

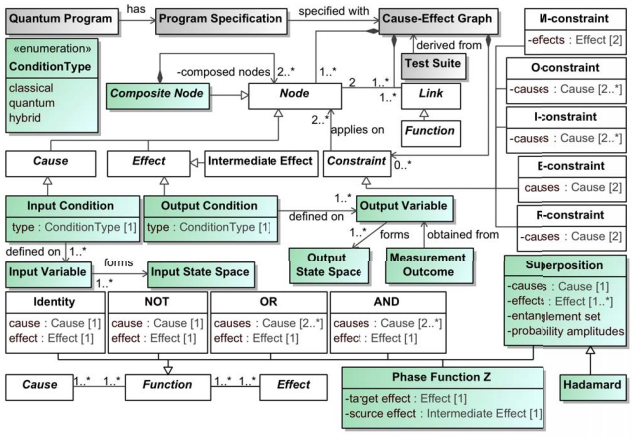

Figure 2: Metamodel. Concepts highlighted with green are newly introduced; while those with the white background are from the original cause-effect graph definitions.

As shown in Figure 2, a *Cause-Effect Graph* models the *Program Specification* of a *Quantum Program*, based on which test suites

could be derived either manually or automatically. A *Cause-Effect Graph* is composed of a set of nodes and links. In a classical cause-effect graph, each cause is captured as a node; similarly, each effect is also captured as a node. Additionally, it is possible to construct a cause-effect graph with *Intermediate Effect* nodes. Each link represents a cause-effect relation, named *Function*. Classically, a function represents a classical logic function (e.g., logical *AND*). While in the quantum domain, a function can also represent a quantum logic function (e.g., Hadamard).

We consider a *Cause* as an *Input Condition* defined on a set of *Input Variable*s. These variables all together form the *Input State Space*. An input condition therefore describes a subset of the input state space. For instance, in our running example (Section 3), an input condition is $|00\rangle$ (see Table 1), i.e., the two qubits are initialized as 0. Similarly, an *Effect* is an *Output Condition* defined on one or more *Output Variable*s. The example shown in Table 1 corresponds to $|00\rangle$ as $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, telling that there is the 0.5 probability to observe 00 and 0.5 probability to observe 11 when qubits are measured given that the qubits are initialized as $|00\rangle$. These variables consequently form the *Output State Space*. When they are measured, we obtain *Measurement Outcome*s. For our running example, measuring $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ leads to either 00 or 11.

With classical cause-effect graphs, among causes or effects, one can specify constraints of five types (Figure 2). *e-constraint* can only be specified between two causes, indicating that the two causes can have one be true, but not both be true simultaneously. *I-constraint* can be applied to two or more causes, denoting that at least one of the causes must always be true, i.e., not being false at the same time. *O-constraint* can also be applied to two or more causes, denoting that one and only one of the causes must be true. *R-constraint* can only be applied on two causes, telling that for one cause to be true, the other must be true. In addition to these four types of constraints for causes, *M-constraint*, i.e., Mask constraint, is however only applicable to two effects, stating that if one effect is true, the other is forced to be false.

Moreover, the classical cause-effect graph notation also defines a set of functions for the purpose of distinguishing various cause-effect relations, represented as links in a graph. As shown in Figure 2, the notation defines four types of classical logical functions: *Identity*, *NOT*, *OR*, and *AND*. *Identity* tells that if the cause is true, then the effect must be true; otherwise the effect is false. *NOT* indicates that if the cause is true, then the effect is false; otherwise the effect is true. *OR* tells that if one of the causes is true, then the effect is true; otherwise the effect is false. *AND* denotes that if all the causes are true, then the effect is true; otherwise the effect is false. In addition, in this paper, we introduce quantum-specific functions from the quantum logic. This includes standard quantum logic functions, i.e., Hadamard (H) and Pauli Gate (e.g., Z). In addition, we define a generalized function, named *Superposition (ψ)*, which indicates that a cause transforms into a set of effects in superposition with arbitrary probability amplitudes. Still, quantum programs frequently rely on equal amplitude states. *H*, therefore, will be defined as a specialization of ψ, with equal probability amplitudes.

Table 2: Mapping of the metamodel to cause-effect graphs

Concept(s)	Description
Input Variable and Output Variable	An input/output variable is a qubit, a set of qubits, or other high-level quantum variables of quantum data types, e.g., QInt [6].
Cause and Input Condition (Classical, Probabilistic, and Phase-based)	A cause is an input condition defined on one or more input variables (e.g., classical logical operations on qubit values without superposition).
Cause and Input Condition (Quantum)	A cause is an abstract true/false statement with an associated probability and phase. The cause is mapped onto an input bitstring.
Intermediate Effect (Classical, Probabilistic, Phase-based)	An intermediate effect is the intermediate state of values of variables before the transformation to the values of output variables. An intermediate effect may have only values available for a subset of variables or all.
Effect and Output Condition (Classical)	An effect is an output condition defined on output variables (e.g., classical logical operations on qubit values after measurement.)
Effect and Output Condition (Probabilistic)	An effect is an output condition defined on output variables and their associated probabilities of being observed (e.g., classical logical operations on qubit values before or after measurements together with probabilities)
Effect and Output Condition (Phase)	An effect is an output condition defined on output variables, their associated probabilities of being observed, and expected phase angles (e.g., classical logical operations on qubit values before measurements together with probabilities and phases)
Effect and Output Condition (Quantum)	An effect is an abstract true/false statement with an associated probability and phase. The effect is mapped onto an output bitstring.

Classical, *probabilistic*, *phase* and *quantum* are short for classical, probabilistic, phase-based and quantum cause-effect graphs, respectively. *All* represents all the four.

5 CLASSICAL CAUSE-EFFECT GRAPHS FOR QUANTUM PROGRAMS

The three versions of classical cause-effect graphs for quantum programs at three different levels of abstractions, i.e., *classical*, *probabilistic*, and *phase-based*, will be discussed in Section 5.1-Section 5.3. We call these *classical* cause-effect graphs since we capture causes, effects, and their relation with classical logic (e.g., with $\wedge$). While quantum cause-effect graphs also include the use of quantum logic functions. We will discuss such functions in a separate section (Section 6). Table 2 maps the metamodel (Figure 2) to all four types of cause-effect graphs. In the rest of this section, we provide an example for each type of cause-effect graphs, which describes the Bell state circuit (Section 3).

5.1 Classical Cause-Effect Graph

We analyse a quantum program as a black-box, where the input/output states $|q_1 q_2\rangle$ are considered after measurement. To this end, a classical cause-effect graph identifies relations between causes and effects in terms of classical logical relations among input/output variables of the quantum program. For the classical cause effect graphs we shall assume a cause node to represent a single qubit q_1 or q_2 in either the $|0\rangle$ or $|1\rangle$ state.

Such a classical cause-effect graph for the Bell states program is shown in Figure 3. It has two causes (C_1 and C_2) and two effects (E_1 and E_2). The node for cause C_1 represents the true or false statement q1=1, whereas C_2 represents q2=1. The causes are also shown in rows 4 and 5 of Table 3. The effects of this graph, i.e., E_1 and E_2, can be represented in two ways as shown in rows 6-9 in Table 3. In the first case, E_1 is captured as a condition, where either *q1 is 0 and*

q2 is 0 or *q1 is 1 and q2 is 1* (row 6). Alternatively, the E_1's condition can also be captured as *q1 equals to q2* (row 8). Similarly, the two representations of E_2 are shown in rows 7 and 9 in Table 3. ‘

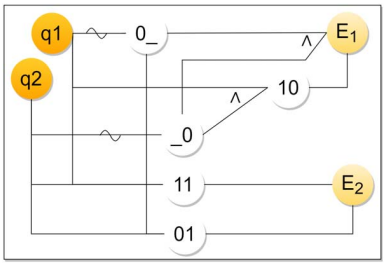

Figure 3: Classical and probabilistic cause-effect graphs for the Bell states quantum program. C_1 ($q1 = 1$) and C_2 ($q1 = 2$) are the two causes, where E_1 (e.g., $q1 \equiv q2$) and E_2 (e.g., $q1 \not\equiv q2$) (see Table 3) are the two effects. $\sim$ represents the logic NOT function, whereas $\wedge$ represents the logic AND function (Figure 2 in Section 4).

Using the cause-effect graph in Figure 3, one can derive test cases by applying a typical test case generation process with decision tables [12]. Table 4 shows such a decision table. As shown in the table, two test cases (TC_1 and TC_2) are needed to activate E_1 and E_2 based on the two identified causes, i.e., C_1 and C_2. For example, TC_1 initializes the quantum program with both C_1 and C_2 initialized as 0, and as a result, we will observe E_1 (i.e., both *q1* and *q2* having the same value, i.e., either 0 or 1).

Based on the input and output specifications (1) described in the background section (see Section 3) for the bell states program, with test cases generated from the classical cause-effect graph, we can only identify faults that lead the program to produce wrong outputs. This is due to the reason that we execute a test case exactly one time and observe corresponding output after the measurement. For example, as described in 1, given an input 00, we shall only observe 00 or 11. However, if we observe 10 or 01, then a program is failing with wrong output. This type of test oracle was defined in an existing work on testing quantum programs [18].

5.2 Probabilistic Cause-Effect Graph

In case of probabilistic cause-effect graphs, as compared to classical cause-effect graphs, each effect is extended with an associated probability of being observed. For our running example, the probabilistic cause-effect graph will look the same as the classical cause-effect graph as shown in Figure 3. The only difference is in the specification of the effects E_1 and E_2, where each effect has an associated probability of being observed given one or more causes were executed. Rows 10 and 11 show the specifications of E_1 and E_2, where a probability of $p = 0.5$ is specified with each effect. For example, *E1* states that we shall observe both *q1* and *q2* equal to 0 with *0.5* probability, whereas both *q1* and *q2* as *1* with *0.5* probability.

The test cases generated corresponding to this probabilistic graph based on the decision table are the same as for the classical cause-effect graph as shown in Table 4. The difference will be in the execution of the test cases. Since we have probabilities associated

Table 3: Cause and effects for the Bell states example for the three types of classical cause-effect graphs

Concept/Label	Details
Input variables	Qubits q1 and q2
Output variables	Qubits q1 and q2
C_1 (Classical, Probabilistic, Phase)	$q1 = 1$
C_2 (Classical, Probabilistic, Phase)	$q2 = 1$
E_1 (Classical)	$\neg q1 \wedge \neg q2 \oplus q1 \wedge q2$
E_2 (Classical)	$\neg q1 \wedge q2 \oplus q1 \wedge \neg q2$
E_1 (Classical)	$q1 \equiv q2$
E_2 (Classical)	$q1 \not\equiv q2$
E_1 (Probabilistic)	$(\neg q1 \wedge \neg q2 \wedge p = 0.5) \wedge (q1 \wedge q2 \wedge p = 0.5)$
E_2 (Probabilistic)	$(\neg q1 \wedge q2 \wedge p = 0.5) \wedge (q1 \wedge \neg q2 \wedge p = 0.5)$
E_1 (Phase)	$(\neg q1 \wedge \neg q2 \wedge p = 0.5 \wedge \theta = 0°) \wedge (q1 \wedge q2 \wedge p = 0.5 \wedge \theta = 0°)$
E_2 (Phase)	$(\neg q1 \wedge \neg q2 \wedge p = 0.5 \wedge \theta = 0°) \wedge (q1 \wedge q2 \wedge p = 0.5 \wedge \theta = 180°)$
E_3 (Phase)	$(q1 \wedge \neg q2 \wedge p = 0.5 \wedge \theta = 180°) \wedge (q1 \wedge q2 \wedge p = 0.5 \wedge \theta = 0°)$
E_4 (Phase)	$(q1 \wedge \neg q2 \wedge p = 0.5 \wedge \theta = 0°) \wedge (q1 \wedge q2 \wedge p = 0.5 \wedge \theta = 0°)$

Classical, *probabilistic*, *phase* and *quantum* are short for classical, probabilistic, phase-based and quantum cause-effect graphs, respectively. *All* represents all the four. θ represents an exclusive or. p represents probability.

Table 4: Decision table for classical and probabilistic cause-effect graphs for the Bell States program. The *Action* column represents the labels for the causes and effects, whereas TC_1 and TC_2 are two test cases.

Action	TC_1	TC_2
C_1	0	1
C_2	0	1
E_1	1	0
E_2	0	1

with each effect, we will need to execute the corresponding test case a specified number of times. Based on the execution results, we need to check whether an effect was observed similar to the ones specified in the cause-effect graph using relevant statistical tests. If not, then the test case is failed with certain probability. Such a process for checking failures in the context of quantum program testing has already been applied in existing literature [2, 7, 18, 20].

5.3 Phase-based Cause-Effect Graph

Phase-based cause-effect graphs extend probabilistic cause-effect graphs with phase information. As a result, effects are associated with both probabilities and phase information. In this case, for our running example, the phase-based cause-effect graph is presented in Figure 4. In this case, we have four effects (E_1-E_4 , Table 3), which are distinguishable because of the phase information. For example, one can see that E_1 and E_2 have the similar specification except that for E_2 $\theta = 180$ when both *q1* and *q2* are *1*.

Table 5 shows the list of test cases generated using a decision table. In this case, we will need four test cases (TC_1 to TC_4). The test execution process will be similar as for the probabilistic cause-effect graph except that in addition to probabilities, we also need to check the observed phases with the ones specified in the cause-effect graph. However, comparing phases is not a straightforward task since measuring a qubit results in losing such information. When executing a test case in a simulator, one can check phases by

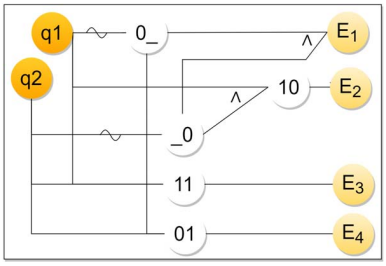

Figure 4: Phased-based cause-effect graph for the bell states quantum program. C_1 ($q1 = 1$) and C_2 ($q2 = 1$) are the two causes, whereas E_1-E_4 (see Table 3) are the four effects. $\sim$ represents the logic NOT function, whereas $\wedge$ represents the logic AND function.

reading state vectors; however, on a real quantum computer it is not possible to directly read phase information without changing to a known measurement basis. There exist some mechanisms such as phase estimation [6], which may be employed, but it requires additional investigation.

Table 5: Decision table for the phase-based cause-effect graph. The *Action* column represents the labels for causes or effects, whereas TC_1 to TC_4 are the four test cases.

Action	TC_1	TC_2	TC_3	TC_4
C_1	0	1	1	0
C_2	0	0	1	1
E_1	1	0	0	0
E_2	0	1	0	0
E_3	0	0	1	0
E_4	0	0	0	1

6 QUANTUM CAUSE-EFFECT GRAPH

A quantum program is inherently probabilistic and contains the quantum properties of superposition and entanglement. It is thus natural to consider cause-effect graphs that incorporate these quantum properties. To do this, the classical cause-effect graph notation needs to be extended with appropriate fundamental quantum symbols that operate between causes and effects using quantum logic.

6.1 Notations

Inspired by quantum logic gates, we propose graphical notations that are in partial correspondence to particular unitary transformations in the circuit model. The *Superposition* function, represented by ψ, transforms a cause C into N effects that are in superposition with arbitrary probability amplitudes $P_j = e^{i\theta} p_j$. Here $p_j \in [0, 1]$ is the probability for measuring the respective effect and the angle-parameter θ defines the phase factor $e^{i\theta}$. Note that in the case of a single and direct link between a cause node and an effect node, the probability amplitude may change, but the probability does not. For instance, the reversal of the sign of the amplitude would not affect the probability represented by the amplitude.

Furthermore, due to the presumed abundance of the scenarios where the probability amplitudes are all equal $P_1 = P_2 = \cdots = P_N$, we define this by the special case $\psi \rightarrow H$ and refer to H as the *Hadamard* function. These two newly introduced functions allow the effects of a graph to contain the phase and probability information of states compactly.

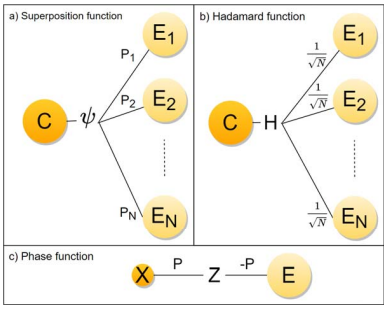

Figure 5: a) Superposition function ψ - Cause C is mapped onto a superposition of the effects $E_1, E_2, \cdots, E_N$ with probability amplitudes $P_1, P_2, \cdots, P_N$; b) Hadamard function H - Cause C is mapped onto an equal superposition of the effects $E_1, E_2, \cdots, E_N$ with probability amplitudes $P_1 = P_2 = \cdots = P_N = \frac{1}{\sqrt{N}}$; c) Phase function Z - Intermediary effect X is mapped onto the effect E by reversing the sign of the probability amplitude.

To reverse the sign of the phase on a link in a quantum cause-effect graph, we introduce the phase function Z, which should be applied between an intermediary effect and a final effect by rotating the phase angle θ by a factor π in the complex plane, $e^{i\theta} \rightarrow e^{i(\theta+\pi)}$.

In addition, we introduce the *entanglement set* $A = \{q_1, q_2, \cdots, q_k\}$, where the collection of qubits $q_j \in A$ represents a subset of qubits from the total state space that are entangled in a particular superposition. Thus, we denote a superposition of effects which includes an entanglement of a specific set of qubits by ψ^A. Also, we define the notation, H^A, for the special case of equal amplitudes. The purpose of set A is to compactly include the information about which qubits in the superposition are entangled.

6.2 Bell States Example

In this section, we present a quantum cause-effect graph of the Bell states circuit introduced in Section 3.

The procedure for drawing quantum cause effect graph is shown in Procedure 1. We write down the possible causes and effects of our program as nodes that represent the observable output and input states from Table 1, $\{|00\rangle, |11\rangle, |01\rangle, |10\rangle\}$. According to the defined input/output conditions from Table 2, we map the causes/effects as follows: C_1 and $E_1 \rightarrow 00$, C_2 and $E_4 \rightarrow 10$, C_3 and $E_3 \rightarrow 01$ and finally C_4 and $E_2 \rightarrow 11$. Thus, the nodes in the graph will be true or false corresponding to these mappings. We then go through one effect node at a time, using the classical logic and the newly introduced superposition and phase functions ψ and Z to create the quantum cause-effect graph. Effect E_1 is true if the input cause is either C_1 or C_2. In this case, E_1 and E_2 are also in superposition, which also contains entanglement between the two qubits. The entanglement is indicated by the H^A-function with the entanglement set $A = \{1, 2\}$ in the exponent. In this case where E_1 is caused by C_1, we can

now unambiguously determine the associated superposed state E_2 by following the connection along the amplitude P_1 to the P_2 connection to E_2. The other possibility is that E_1 occurs due to C_2, but in that case E_1 is in superposition with E_2 with the probability amplitude $P_4 = -1/\sqrt{2}$. The H-function provides, by definition, the probability amplitudes $+1/\sqrt{2}$. Thus we add the Z-function on the connection to the $E2$ node, reversing the sign.

Figure 6 shows the quantum cause-effect graph for the Bell states, invoking the mentioned quantum logic functions and probabilistic connections. The upper and lower four nodes are denoted by 1) and 2) respectively. In 1), the probability amplitudes are drawn explicitly, which are hidden in 2). However, the amplitudes in 2) can be inferred from the H and Z functions in the graph. This visibility option demonstrates that only certain scenarios might require an explicit listing of amplitudes. For instance, in Figure 6, effect E_1 (00) occurs with $|P_1|^2 = |P_2|^2 = 50\%$ probability and with a phase of +1 when the causes are either C_1 (00) or C_2 (10). However, the effect E_2 (11) occurs with $|P_3|^2 = |P_4|^2 = 50\%$ probability for both occurrences of C_1 and C_2. While on the other hand, only C_1 results in a phase of +1. Cause C_2 results in a phase of -1 when E_2 is measured. Thus, the graph's addition of the probability amplitudes on the connections, provides a method for creating several links to the same effect with arbitrary probabilities and phases.

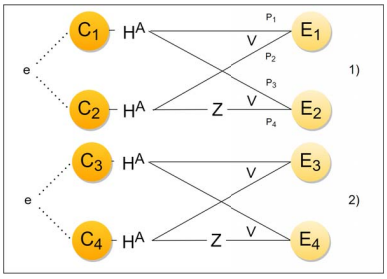

Figure 6: Quantum cause-effect graph for the Bell state program with four effects $E_1 - E_4$ with the corresponding probability amplitudes $P_1 = P_2 = P_3 = 1/\sqrt{2}$ and $P_4 = -1/\sqrt{2}$. The exclusive-OR constraint is denoted by dotted lines in conjunction with the symbol e. In 1) the probability amplitudes are explicit, while in 2) they are implicit.

6.3 Test Case Derivation

In order to derive test cases from the quantum cause-effect graph, we can first create a binary table (see Procedure 2) and then derive test cases from it, according to constraints which might relate to resources or other testing needs. Since quantum programs are fundamentally probabilistic, the table entry of the effects can be coupled with the probability amplitudes for the respective effect. Thus, an effect in a decision table is represented by a tuple (E_k, P_l) for all possible k, l combinations allowed by the nodes in the graph. For our running example we would have four possibilities for the upper four nodes in 1) from figure 6. The tuple states that the effect E_k is true with probability amplitude P_l, while the probability for measuring E_k is as usual given by the formula for p_j from section 3. With this definition, we can reproduce the classical table 4. So what additional information does the quantum graph provide? Since superposition, entanglement and phases are not directly measurable as true or false statements, like the output states of the Bell basis, we can instead define new test cases for these quantum features. By reading the quantum cause-effect graph from the uppermost effect E_1 towards the bottom effect node, we can identify two superposition test cases S_1 and S_2 due to the H-functions. We can then represent a test case for each superposition by adding a new row in the table, inserting a true binary value for the superposition test case ψ_1 and then ψ_2. For each respective superposition test case we then add true statements to the corresponding two effects that make up the superposition. For the first superposition S_1 we then add the TC_5-column and activate the tuples (E_1, P_1) and (E_2, P_3) according to the graph 6. For the entanglement we insert another row and represent this by ψ_1^A and ψ_2^A, where A is the entanglement set containing which qubits are entangled. Now, as with ψ_1 and ψ_2, we insert true statements and activate the underlying superpositions ψ_1, ψ_2 and their corresponding tuples since these are also true for the entangled case. The results of this procedure is represented in table 6, where we have the entries of the classical effects $E_1 - E_4$ in conjunction with the additional test cases for each instance of superposition along with the entanglements. All of which were read from the graph in a systematic way according to Procedure 2.

Table 6: Decision table for quantum cause-effect graphs for the Bell states program. Column *Action* represents cause and effect labels. There are three types of instances. Effects $E_1 - E_4$ are represented in tuples with their respective probability amplitudes $P_1 - P_4$. Finally there are quantum specific instances ψ_1, ψ_2 and ψ_1^A, ψ_2^A where $A = \{1, 2\}$ is the entanglement set containing the indices for the entangled qubits.

Action	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
C1	1	0	1	0	1	0	1	0
C2	0	1	0	1	0	1	0	1
E_1, P_1	1	0	0	0	1	0	1	0
E_1, P_2	0	1	0	0	0	1	0	1
E_2, P_3	0	0	1	0	1	0	1	0
E_2, P_4	0	0	0	1	0	1	0	1
ψ_1	0	0	0	0	1	0	1	0
ψ_2	0	0	0	0	0	1	0	1
ψ_1^A	0	0	0	0	0	0	1	0
ψ_2^A	0	0	0	0	0	0	0	1

Procedure 1 Steps for drawing a quantum cause-effect graph.

1: Draw and label nodes for the causes C_k and effects E_l for all $k, l = 1, 2, \cdots$ according to the quantum program input-output specification.
2: Add all superposition function nodes ψ and link them to the corresponding superposed effects, adding probability amplitudes P on the links. Label the weights in ascending order such that the uppermost link in the graph which connects to E_1 is weighted by P_1.
3: For a given superposition that contains entangled qubits, define a set $A =$ such that it contains the indices of the entangled qubits from the list of qubits $q_1, q_2, \cdots, q_d$, for the specific superposition.
4: Make appropriate modifications of the sign of the probability amplitudes using the Z function.
5: Perform the classical cause-effect procedure applying the OR, AND functions between the causes and effects.

Procedure 2 Procedure for obtaining the decision table reading a quantum cause-effect graph.

1: Create the first column heading *Action* and add a column for each possible test case TC and label them in ascending order from TC_1.
2: In the *Action* column, add entries row-wise for the causes C, then add the effect-amplitude tuples (E_k, P_l), for all possible k, l. The tuple entries should be added such that k is fixed while adding new entries for each l. When all l entries are exhausted, increment k and repeat the process until all tuple combinations have been added to the table.
3: Add all superposition entries corresponding to the number of ψ functions in the graph, starting from the uppermost function which we denote by ψ_1.
4: If the superposition functions contain entanglement sets ψ^A, then add all such entries starting with ψ_1^A.
5: Input +1 starting from the (E_1, P_1) entry and add the +1 entry to the corresponding causes C in the table.
6: For the superpositions add +1 starting from the ψ_1 entry and add the +1 entry to the corresponding effect-amplitude tuples which construct the superposition. Finally, add +1 to the corresponding causes C in the table.
7: For the entanglements, add +1 starting from the ψ_1^A entry and repeat step 6 for the corresponding superposition.
8: Add 0 to all other entries in the table.

A full test case example of TC_5 from table 6 is given by:

- **Test case ID (Name):** TC_5 (TC5-Superpos1)
- **Description:** Test that the program output state is in superposition of E_1 and E_2 with amplitudes P_1 and P_3
- **Steps:** 1) x measurements in the Bell basis - 2) y measurements in the basis consisting of the Bell states
- **Expected Result:** Outputs of $50\% + err$ for $|00\rangle$ or $|11\rangle$ for 1) and $100\% + err$ for the respective basis state of 2).

A certain number of measurements x, y would provide a sufficient confidence in the verification of the superposition within the defined error parameter err.

7 DISCUSSIONS

We first discuss key differences between the application of classical and quantum cause-effect graphs for quantum programs (Section 7.1), followed by thoughts on the potential usages of cause-effect graphs (Section 7.2) and their the scalability (Section 7.3).

7.1 Classical vs. Quantum Cause-effect Graphs for Quantum Programs

In classical cause-effect graphs for quantum programs, we construct relations between causes and effects by treating quantum programs as black-boxes. Moreover, the relations consist of pure classical logic functions. Therefore, test case generation is the same as for classical software, except that for certain cases, the test cases will be executed multiple times to deal with the probabilistic nature of quantum programs. Based on such executions, passing and failing of test cases may be determined using appropriate statistical tests such as one-sample Wilcoxon Signed Rank test and Pearson's chi-square test employed by existing works [2, 7, 18]. Moreover, we need novel test strategies for test case design using classical cause-effect graphs.

The quantum cause-effect graphs defined in this paper are constructed from nodes representing single basis states. While the quantum features such as superposition and entanglement can be read from the functions within the graph itself by the links connecting the respective effects in the superposition. Thus, the effects can be considered before measurement (i.e., superpositions such as ψ_1, ψ_2 in Table 6) or after measurement such as the considered single state tuple (E_1, P_1). The quantum graphs thus provide classical effects in addition to quantum type effects. Both can be read from the graph, but they are represented in separate ways. This feature results from the usage of both quantum and classical logic. In order to derive test cases from a quantum cause-effect graph, we then have the following options: (1) If the causes and effects are all classical, test case generation can follow a process similar to what we described for classical ones; (2) If the causes are in superposition and effects are after measurement, when generating test cases, the causes shall include test setup mechanisms to set a quantum program in the specified causes in superposition, and the effects can be checked in a similar way as for the classical; (3) If the causes are classical, whereas the effects are before measurement, we need specific mechanisms to verify the superposition property of the effects. This is a new area of research and needs additional investigation; and (4) Causes and effects are in superposition instead of being in classical states. In this case, we need to set the program in superposition (e.g., with test drivers) and check effects in superposition. These topics are being investigated in the community.

When looking at causes in classical and quantum cause-effect graphs, we see two approaches. For the considered classical approach, a cause was defined as one qubit initialized as 1, whereas in the quantum case, a cause was defined as a set of binary values assigned to all qubits. Each representation as true or false statements has its benefits and drawbacks. For example, in the classical context, the number of causes will be less. For instance, for a quantum program with 20 qubits, we need 20 causes, whereas, for quantum, we need, at most 2^{20} causes (assuming no abstraction). However, for the classical context, even the causes are less, but the number of links will be increased as compared to quantum. Therefore, more investigation is needed to see which approach is most suitable for defining causes and effects.

7.2 Usages of Cause-effect Graphs

First, cause-effect graphs can help to build high-level specifications of quantum programs, currently in terms of inputs and outputs of a quantum program. In quantum software engineering, the literature still lacks methods for constructing program specifications; therefore, using cause-effect graphs can be considered as the very first initiative for helping the identification and specification of causes and effects by raising up the level of abstraction from quantum circuits. In the same way as the classical models are used, cause-effect graphs may help software engineers to reason, at a higher level abstraction than quantum circuits, input and output relations. This is important for understanding quantum programs. Nonetheless, our current specifications of causes and effects are dependent upon mappings onto the circuit level.

Second, we want to use quantum cause-effect graphs to systematically generate effective test cases. Currently, we only demonstrated

that we could use decision tables to generate the single test case TC_5 in sufficient detail in Section 6.3. However, additional work is needed to generate effective test cases using the quantum procedures 1 and 2, which is part of our immediate future work.

7.3 Scalability

For quantum cause-effect graphs to be scalable, several directions need to be followed. First, we need to raise the level of abstractions by, e.g., grouping a set of qubits or merging a set of causes/effects/links. Second, we need to enable the reuse of cause-effect graphs, via nesting structures, of components such as the Bell states example we demonstrated in the paper, in larger quantum software applications such as quantum teleportation [13]. Similarly, we can reuse test cases designed for the Bell states with quantum cause-effect graphs for testing quantum teleportation programs.

The introduction of superposition function ψ provides several considerations for scalability: (1) Test cases for quantum superposition and entanglement can be read from the functions in the graph, whereas a classical graph would require nodes to represent such superposition instances as effects; (2) They provide links to include information about the probability and phase of a single state outcome; (3) They keep quantum superposition and entanglement as a part of the black-box quantum program. Thus the quantum graph arguably suits quantum programs more naturally by the integration of these quantum features.

8 CONCLUSIONS AND FUTURE WORK

Cause-effect graphs are designed to communicate relations among causes and effects, and have been used for test case design for classical software systems. In this paper, we investigated cause-effect graphs to capture relations between causes and effects of quantum programs. In particular, we studied the use of both quantum and classical logic functions to capture such relations. Based on the Bell states program, we analyzed various cause-effect graphs at different levels of abstractions. In the future, we plan to make our quantum cause-effect graph notations more complete, especially by adding concepts at a higher level of abstraction to improve the overall quality of quantum cause-effect graphs. We also want to enrich the quantum cause-effect graph notations with the complete quantum logic to capture complex relation among cause and effects. Additionally, we will carefully design a set of graphical notations, along with tool support, such that they can be understood and used more easily by end users. Also, adding support for automatic testing will allow more scalability. The main utility of quantum cause-effect graphs is suggested to allow additional test cases of superposition and entanglement to be included in a test suite. Thus, future work will aim to demonstrate this utility in a quantum test suite comparison.

REFERENCES

[1] C. A Perez Delgado and H. G. Perez-Gonzalez. 2020. Towards a Quantum Software Modeling Language. In *First International Workshop on Quantum Software Engineering (Q-SE 2020)*. https://doi.org/10.1145/3387940.3392183

[2] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. 2021. Assessing the Effectiveness of Input and Output Coverage Criteria for Testing Quantum Programs. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 13–23. https://doi.org/10.1109/ICST49551.2021.00014

[3] Shaukat Ali and Tao Yue. 2020. Modeling Quantum Programs: Challenges, Initial Results, and Research Directions. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software* (Virtual, USA) *(APEQS 2020)*. Association for Computing Machinery, New York, NY, USA, 14–21. https://doi.org/10.1145/3412451.3428499

[4] Felix Gemeinhardt, Antonio Garmendía, and Manuel Wimmer. 2021. Towards Model-Driven Quantum Software Engineering. *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)* (2021), 13–15.

[5] Alexandru Gheorghiu, Theodoros Kapourniotis, and Elham Kashefi. 2018. Verification of Quantum Computation: An Overview of Existing Approaches. *Theory of Computing Systems* 63, 4 (Jul 2018), 715–808. https://doi.org/10.1007/s00224-018-9872-3

[6] M. Gimeno-Segovia, N. Harrigan, and E.R. Johnston. 2019. *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Incorporated.

[7] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. 2020. Property-Based Testing of Quantum Programs in Q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 430–435. https://doi.org/10.1145/3387940.3391459

[8] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 150 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428218

[9] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. 2021. Muskit: A Mutation Analysis Tool for Quantum Software Testing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia). 1266–1270. https://doi.org/10.1109/ASE51524.2021.00150

[10] Andriy Miranskyy and Lei Zhang. 2019. On Testing Quantum Programs. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results* (Montreal, Quebec, Canada) *(ICSE-NIER '19)*. IEEE Press, 57–60. https://doi.org/10.1109/ICSE-NIER.2019.00023

[11] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2020. Is Your Quantum Program Bug-Free?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 29–32. https://doi.org/10.1145/3377816.3381731

[12] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2012. *The art of software testing* (3rd ed ed.). John Wiley & Sons, Hoboken and N.J.

[13] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, USA.

[14] Ricardo P'erez-Castillo, Luis Jim'enez-Navajas, and Mario Piattini. 2021. Modelling Quantum Circuits with UML. *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)* (2021), 7–12.

[15] Jiyuan Wang, Fucheng Ma, and Yu Jiang. 2021. Poster: Fuzz Testing of Quantum Program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 466–469. https://doi.org/10.1109/ICST49551.2021.00061

[16] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks.

[17] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Application of Combinatorial Testing to Quantum Programs. In *2021 IEEE 21th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. (to appear).

[18] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Generating Failing Test Suites for Quantum Programs With Search. In *Search-Based Software Engineering*, Una-May O'Reilly and Xavier Devroey (Eds.). Springer International Publishing, Cham, 9–25. https://doi.org/10.1007/978-3-030-88106-1_2

[19] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a Coverage-Guided Test Generator for Quantum Programs. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia). 1237–1241. https://doi.org/10.1109/ASE51524.2021.00144

[20] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a Coverage-Guided Test Generator forQuantum Programs. In *The 36th IEEE/ACM International Conference on Automated Software Engineering, Tool Demonstration*. IEEE/ACM.

[21] Tao Yue, Paolo Arcaini, and Shaukat Ali. 2022. Quantum Software Testing: Challenges, Early Achievements, and Opportunities. *the European Research Consortium for Informatics and Mathematics News* (2022).

[22] Jianjun Zhao. 2020. Quantum Software Engineering: Landscapes and Horizons. *CoRR* abs/2007.07047 (2020). arXiv:2007.07047

9 ACKNOWLEDGEMENT

This work is supported by Qu-Test (Project#299827) funded by the Research Council of Norway.