Quacee: A Mechanism for Specifying Quantum Circuits and Controlling Quantum Computations, with Quantum Computer Simulation Capability

Katherine Hudek

Abstract

In this paper, Quacee, a new quantum programming language, was designed, developed, and demonstrated. In addition, a simulator of a quantum computer was created and used to test Quacee functionality. As research progresses towards the goal of programmable universal quantum computers able to support a wide range of quantum algorithms, there is a corresponding need for mechanisms to describe the desired computations and program the quantum computers. Quacee is a new quantum programming language designed to meet this need. It has been designed to support the quantum circuit model and is independent of the physical manifestation of quantum hardware. Quacee supports a wide range of standard quantum gates, the ability to specify new custom gates, and a number of higher-level circuit operations. Quacee is easy to use and was demonstrated by using it to program a number of quantum algorithms.

Quantum computing offers the promise of addressing extremely large and complex problems, many of which are beyond the scope of classical computing today. Applications include improvements to the drug discovery process, dramatically reducing the cost and time of drug development. In the longer term, personalized drugs based on a patient's DNA could become cost effective and widespread. Other application areas include weather modeling, cybersecurity, physics modeling, logistics, chemical reaction modeling, and others.

NOTE: This is an updated version of the paper which now incorporates the recent addition of a quantum computer simulator capability.

1 Introduction

Quantum computing is an emerging field in which new types of hardware and methods of computation are being developed which utilize the effects of quantum mechanics. Quantum mechanics describes the behaviors of subatomic particles at the quantum level, which are quite different from the behaviors observed in the macroscopic world. Among these differences are superposition and entanglement. Superposition is the ability of a particle to exist in more than one state at a time, until it is measured. This can be seen in a photon, which could be in a superposition of polarity states (e.g., vertical, horizontal) until it is measured, or an electron in a superposition of spin states (e.g., up, down) until it is measured. Entanglement enables an observer to learn some information about one quantum particle by taking measurements of another, no matter how far apart they are at the time of measurement.

Classical digital computer architecture is based on the *bit*, which is represented as either a logical zero or one. A quantum computer, however, utilizes a *qubit*, which can represent a logical zero, a logical one, or arbitrary superpositions of the logical zero and logical one states. Different types of hardware using different physical systems are being created and tested in the lab. New algorithms are being developed to take advantage of these characteristics. A useful feature of quantum computing is the ability to simultaneously represent all possible input combinations via superposition, whereas a classical computer would have to handle each combination separately. For example, with n inputs, a quantum computer can process 2^n combinations in parallel, creating an equal superposition of all possible inputs at the start of a computation. Quantum computing uses constructive interference to boost the probability of the correct result upon measurement, and destructive interference to lessen the likelihood of incorrect results upon measurement.

In the future, quantum computers could be used to efficiently tackle complex problems, many of which are beyond the scope of classical computing. Applications include improvements to the drug discovery process, dramatically reducing the cost and time of drug development. In the longer term, personalized drugs based on a patient's DNA could become cost effective and widespread. Other application areas include weather modeling, cybersecurity, physics modeling, logistics, image processing, chemical reaction modeling, and others.

Researchers are working towards the goal of creating general-purpose universal quantum computers capable of performing a wide variety of quantum computations. With new quantum hardware and computational methods, there is a need for a general means of specifying and controlling quantum computers and their computations. [1][2][3]

Within this field, there has been some experimentation in creating quantum programming languages based on existing classical computing languages. Examples include: QCL[4], based on C; the Q language[5], based on C++; LanQ[6], based on C; and Quipper[7], based on Haskell.

The remainder of this paper is organized as follows. Section 2 contains background material which includes a brief discussion of mathematical notation, quantum bits, quantum gates and quantum circuits. Section 3 describes the requirements and design of Quacee, a new language for programming quantum computations, provides a listing of its varied functions, and also describes the simulator of a quantum computer developed as a part of this research. Section 4 provides results, describing how Quacee meets the requirements and demonstrating its use when applied to a number of quantum algorithms, as well as using the quantum computer simulator to validate Quacee output. Finally, Section 5 provides a summary and conclusion.

2 Background

2.1 Quantum Mechanics Notations and Linear Algebra Concepts

In quantum mechanics, a mathematical notation known as the Dirac notation is widely used. Named after Paul Dirac, a pioneering theoretical physicist, it utilizes the symbols |, >, and < to represent column or row vectors of complex numbers. It is also commonly known as the bra-ket notation.

A ket, represented by a number or symbol between a | and a > (such as `|X>`), represents a column vector of complex numbers. A bra, represented by a number or symbol between a < and a |, represents a row vector of complex numbers. For a given ket (column vector of complex numbers), the corresponding bra is its complex transpose (row vector of complex conjugates of the corresponding ket entries). The inner product between two vectors can be expressed as `<X|Y>`. As used in quantum computing, the vectors are normalized such that the sum of the squares of the absolute value of each complex element is 1. [8]

In quantum mechanics, the concept of a unitary matrix is significant, especially in the domain of quantum gates. Some key characteristics [3] of a unitary matrix (U) are:

- Its inverse equals its conjugate transpose ($U^{-1} = U^{\dagger}$)
- Both its inverse and its conjugate transpose are unitary
- For a fixed column, the sum of the squares of the row absolute values equals one ($\sum_i |U_{ij}|^2 = 1$ for a fixed j)
- For a fixed row, the sum of the squares of the column absolute values equals one ($\sum_j |U_{ij}|^2 = 1$ for a fixed i)

Another key mathematical concept in quantum computing is the tensor product (also known as the Kronecker product) of matrices, which uses the symbol $\otimes$. It takes each element of the first matrix and uses it as a scaler which is multiplied by the entire second matrix and these results are combined into a larger matrix. Thus, if R_1 and C_1 are the row and column dimensions of the first matrix, and R_2 and C_2 are the dimensions of the second matrix, the dimensions of the resulting matrix are (R_1 x R_2) rows by (C_1 x C_2) columns. An example showing a tensor product of a 2 x 2 matrix A with a 2 x 2 matrix B is shown in equation 1.

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{21}b_{22} \end{bmatrix} \tag{1}$$

2.2 Quantum Bits (Qubits)

Recall from Section 1, a quantum system can exist in a superposition of states. Physically, a qubit can be represented by a two-state quantum system, such as a photon (e.g. vertical or horizontal polarization) or an electron (e.g. spin up or down). At the level of qubits, however, the exact physical representation is not a concern. This is similar to the situation in classical computing where one works with boolean operations and bits without a concern for the precise voltage levels necessary to represent a logical 1 in a piece of hardware.

Quantum bits (known as qubits) are employed as building blocks of quantum computations. For comparison, a classical bit can only be in one logical state or the other (e.g. either 0 or 1) at

any given time. A qubit can be in one state or the other, but can also be in a superposition of states (e.g. existing as both logical 0 and logical 1) until measured, at which time it will collapse into either one state or the other and act as a classical bit.

A qubit's state space contains the two *computational basis states* ket-zero (`|0>`) and ket-one (`|1>`). In matrix notation:

$$|0> = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2}$$

$$|1> = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3}$$

Note that in the ket-zero matrix, by definition, there is a 1 in row 0 (rows are counted starting from 0), representing a qubit in state 0. In the ket-one matrix, there is a 1 in row 1, representing a qubit in state 1.

When using the ket notation for a qubit, the absolute value squared of the coefficient of a computational basis state ket is the probability that one would obtain that ket if the qubit was measured. As noted before, a qubit can be in state 0, state 1, or a superposition. In general, a qubit is represented as α`|0>` + β`|1>`, where α and β are complex numbers. These coefficients are sometimes known as amplitudes of a wave function representing the state of a system.

A qubit that is known with certainty will be in a logical state 0 when measured would have a coefficient $\alpha = 1$ and a coefficient $\beta = 0$, which is simply represented as `|0>`. A qubit that is known with certainty will be in a logical state 1 when measured would have a coefficient $\alpha = 0$ and a coefficient $\beta = 1$, which is simply represented as `|1>`.

A qubit in state $(1/\sqrt{2})$`|0>` + $(1/\sqrt{2})$`|1>` is in superposition and has a coefficient α of $1/\sqrt{2}$ and a coefficient β of $1/\sqrt{2}$. So, $|\alpha|^2 = 1/2$ and $|\beta|^2 = 1/2$ and thus, the probability of the qubit being zero if measured is 1/2 and the probability of the qubit being one if measured is 1/2. The qubit is in equal superposition.

A qubit in state $(\sqrt{3}/2)$`|0>` + $(i/2)$`|1>` is in superposition and has a coefficient α of $\sqrt{3}/2$ and a coefficient β of $i/2$ (where i^2 = -1). So, $|\alpha|^2 = 3/4$ and $|\beta|^2 = 1/4$ and thus, the probability of the qubit being zero if measured is 3/4 and the probability of the qubit being one if measured is 1/4. The qubit is in an unequal superposition.

When working with multiple qubits in a system, their general state is defined as the tensor product of each. For example, assume a system of three qubits with known states 0, 1, and 1 (since the states are known with certainty, the ket coefficients are 1). The qubits' general state would be expressed as `|0>` $\otimes$ `|1>` $\otimes$ `|1>`. This is typically expressed in the more compact notation `|011>`, where the tensor product is implied. [1][8][9]

The matrix representation is shown in equation 4. Note that `|011>` has a 1 in row index 3 (binary 011, counting from 0) and a 0 in every other row.

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \texttt{|011>} \tag{4}$$

2.3 Quantum Gates

In classical computing, a set of logic gates are used, such as AND, OR, NOT, NAND, and NOR gates. These perform boolean operations on classical bits. Each logic gate transforms its inputs into one or more outputs in a deterministic fashion according to the gate's definition.

In quantum computing, a different set of gates that perform different types of operations are used for computations involving qubits. Unlike classical gates, quantum gates can manipulate arbitrary superpositions of the computational basis states. Quantum gates are typically mathematically represented as unitary matrices where the row indices (row number counting from zero in binary) correspond to inputs and the column indices (column number counting from zero in binary) correspond to outputs. The matrix elements can be interpreted to represent the square roots of the probabilities of the input (row) evolving to the output (column) and these elements may be complex numbers. A set of typical quantum gates and their matrices is shown in Table 1. [3]

To illustrate, Figure 1 shows a closer look at the matrix representation of the CNOT gate (note that the matrix picture in the figure has been augmented with row and column indices in binary form):

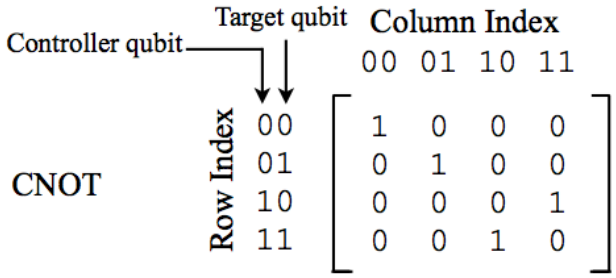

Figure 1: Controlled NOT Gate

The CNOT (short for CONTROLLED-NOT) is an important gate used in quantum computing. Its function is to negate the second qubit (the target) whenever the first qubit (the controller) is 1. Each row index corresponds to an input to the gate, and each column index represents the corresponding output; a 1 at the intersection of the row and column indicates that the input transitions to the output with probability $|1|^2 = 1$ (i.e., with certainty). For example, the row 00 has a 1 in column 00; thus, an input of 00 to a CNOT will yield an output of 00. Similarly, the row 01 has a 1 in column 01, and thus an input of 01 to a CNOT will yield an output of 01. The row 10 has a 1 in column 11, and thus an input of 10 will yield an output of 11. Since the first qubit was 1, the second qubit was negated, changing 10 to 11. Finally, the row 11 has a 1 in column 10; therefore an input of 11 will yield an output of 10.

Another important gate is the Hadamard. The Hadamard gate is one of the most useful gates in quantum computing, which maps computational basis states into superposition states and vice versa. Its matrix is defined in Table 1. When applied to computational basis state `|0>`, it results in $1\sqrt{2}$`(|0> + |1>)`, and when applied to computational basis state `|1>`, it results in $1/\sqrt{2}$`(|0> - |1>)`, as shown in equations 5 and 6. [3]

$$\texttt{H|0>} = 1/\sqrt{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = 1/\sqrt{2}\begin{bmatrix}1\\ 1\end{bmatrix} = 1/\sqrt{2}(\texttt{|0> + |1>}) \tag{5}$$

$$\texttt{H|1>} = 1/\sqrt{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}0\\ 1\end{bmatrix} = 1/\sqrt{2}\begin{bmatrix}1\\ -1\end{bmatrix} = 1/\sqrt{2}(\texttt{|0> - |1>}) \tag{6}$$

Table 1: Sample Quantum Gates And Their Matrix Representation

Hadamard Gate	$1/\sqrt{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
Pauli-X Gate	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
Pauli-Y Gate	$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
Pauli-Z Gate	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
Phase	$\begin{bmatrix}1 & 0\\ 0 & -i\end{bmatrix}$
CNOT	$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$
SWAP	$\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$
TOFFOLI	$\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{bmatrix}$
FREDKIN	$\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}$

2.4 Quantum Circuits

The circuit model of a computer is an abstraction of the computing process often used in classical computing. A computation is modeled as different types of Boolean logic gates acting on a binary input. Just as a classical computation can be broken down into a sequence of classical logic gates, a quantum computation can be broken into a sequence of quantum logic gates. [3]

A quantum circuit specifies a sequence of unitary operations and measurements which are applied to a multi-qubit state. A quantum circuit is organized in stages, with time progressing from left to right. Initially, the qubit inputs to the circuit are specified. The remainder of the circuit specifies the operations on each qubit at each stage. Quantum circuit operations include nothing (a no-op or identity), a quantum gate (e.g., Hadamard), or a measurement. A qubit could also serve as a controller for another gate (such as a CNOT), thereby affecting another qubit. Once a qubit is measured, it loses its superposition and collapses to either one state or the other.

Figure 2 shows a pictorial representation of a quantum circuit which implements the Deutsch algorithm (discussed in Section 3.1 and programmed using Quacee in Section 4.1.1). The inputs are defined as two qubits, one initialized to `|0>` and the other to `|1>`. In stage 0 (numbering from 0), they each pass through a Hadamard gate (H), in stage 1 they pass through a two-input black box (an unknown two-input, two-output unitary operation), in stage 2 they each pass through a Hadamard gate, and finally in stage 3 the top qubit is measured.

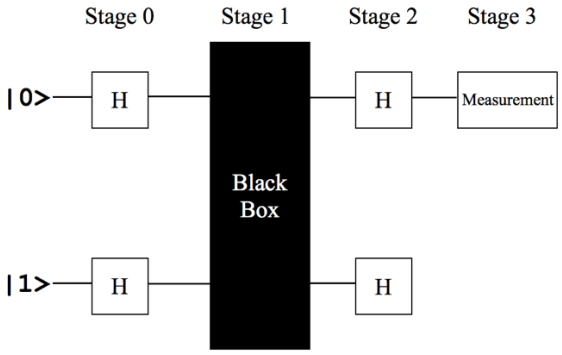

Figure 2: Example Quantum Circuit (Deutsch Algorithm)

When specifying a quantum algorithm, one could represent it algebraically, which would be useful to mathematicians but is not as useful for instructing a computer. One also could express it pictorially, which is useful for a human, but doesn't scale to large circuits and is not useful for a computer. There is a need for an extension of a generic programming language, enhanced with the ability to specify quantum computations in terms of quantum circuits. This method would scale nicely to larger circuits and is useful to both humans and computers [1]. This is the need that Quacee is designed to address.

Figure 3 shows the likely mode of operation for the first generation universal quantum computers. The quantum circuit model for quantum programming features a quantum computer acting together with, but under the control of, a classical computer. A classical computer would be able to perform regular classical computations and also specify desired quantum computations. The classical computer would send a description of the needed quantum computations, in the form of a quantum circuit with inputs, gates, and measurements, across the interface to the quantum computer. The quantum computer would then perform the operations specified in the circuit and return any requested measurements to the classical machine. This is the mode of operation that Quacee is designed to support.

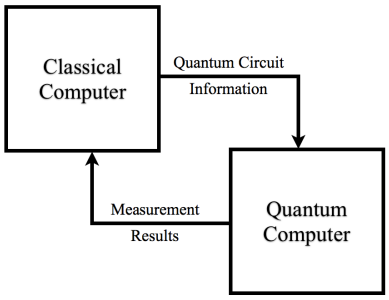

Figure 3: Quantum Computing Mode Of Operation

While universal quantum computers are not yet generally available, researchers have developed an abstract model of their operation; this model is known as the quantum circuit model. The quantum circuit abstracts away the details of exactly how the quantum operations would be implemented on different physical devices (specific hardware which affects quantum state such as photon polarization, electron spin, etc.), and instead focuses on the essential capabilities and operations of quantum computing in terms of qubits and quantum gates. As shown in Figure 4, Quacee is designed to write to an abstract quantum circuit interface, which represents a general universal quantum computer and supports the mode of operation shown in Figure 3. When universal quantum computers become available from different companies, they will likely have different product-specific instruction sets for implementing the quantum circuit model. The quantum circuit representation of the computer would need to be translated or encoded into these different instruction sets to run on a particular machine, but this would not affect programs written in Quacee, which interfaces at the abstract quantum circuit model level.

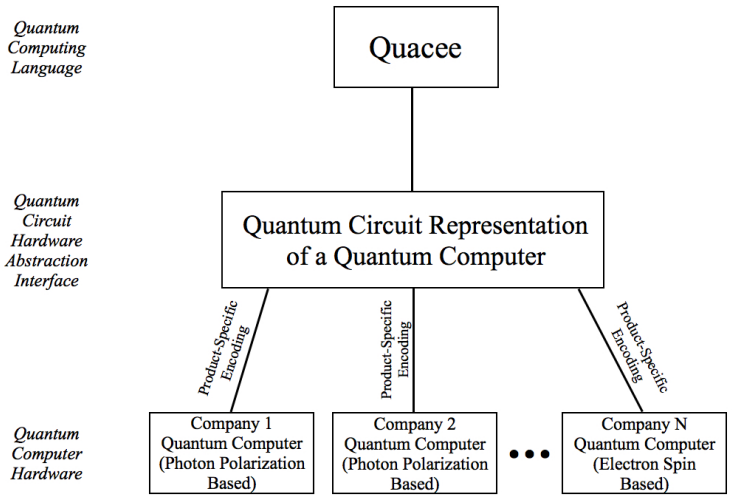

Figure 4: Quantum Computing Interface

3 Design

3.1 Quantum Algorithms

In designing a quantum programming language such as Quacee, it is important to understand the characteristics of the desired quantum computations. Quantum computations are used to implement quantum algorithms. An investigation of different quantum algorithms was performed, including the Deutsch, Quantum Fourier Transform, and a recent algorithm by Cai et. al. [10] for solving systems of linear equations and these were then used as programming test cases to verify Quacee capabilities. Quantum algorithms are an area of active research, and there is a repository of algorithm-related papers maintained by the National Institute of Standards and Technology at http://math.nist.gov/quantum/zoo.

The Deutsch algorithm will now be described in detail to provide background in understanding the later Quacee code example of Section 4.1.1. The Deutsch algorithm was first used to show a definite advantage of quantum computing over classical computing. The formulation described below is consistent with the description offered in [8].

The Deutsch algorithm involves a black box which computes a simple function. It takes two qubits as inputs and produces two qubits as outputs. The black box implements one of four functions, (f_1, f_2, f_3, f_4) shown in equations 7, 8, 9, and 10 where $(\bar{y})$ indicates NOT-y, and the $\oplus$ is an exclusive OR or equivalently addition modulo 2.

$$f_1(x, y) = (x, y) \tag{7}$$

$$f_2(x, y) = (x, \bar{y}) \tag{8}$$

$$f_3(x, y) = (x, x \oplus y) \tag{9}$$

$$f_4(x, y) = (x, x \oplus \bar{y}) \tag{10}$$

The functions are grouped into two sets: f_1 and f_2 are in Set 1, and f_3 and f_4 are in Set 2. The goal of the problem is to determine from which set (Set 1 or Set 2) the function that the black box implements is drawn, in as few queries as possible.

To implement the Deutsch algorithm to solve this problem, a quantum circuit as shown in Figure 2 would be used. In this circuit, if the Black Box was using functions f_1 or f_2, then the output if measured would be 01 with certainty; if the function was f_3 or f_4, then the output if measured would be 11 with certainty. With quantum computing, only one query and one measurement is required. In this algorithm, only the top-level bit varies with the set. Quantum computing allows one to make a single query (and also take a single measurement of the top qubit) to determine which set the Black Box function was drawn from. If measured to be a 0, a function from Set 1 was used; if measured to be a 1, a function from Set 2 was used. With a classical system, it has been shown that one would have to make a minimum of two queries to determine which set the Black Box function was drawn from. This Deutsch algorithm demonstrated a definite advantage of quantum computing over classical computing in some areas.

Following is a step by step mathematical evaluation of the Deutsch algorithm. Results are calculated for each possible case of the Black Box function. The unitary matrices corresponding to each possible function based on the function definitions are as follows:

$$f_1 \text{ corresponds to matrix } U_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

$$f_2 \text{ corresponds to matrix } U_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{12}$$

$$f_3 \text{ corresponds to matrix } U_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{13}$$

$$f_4 \text{ corresponds to matrix } U_4 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

At the start of the algorithm, the top qubit is `|0>` and the bottom qubit is `|1>`. Recall from Section 2.2, the input state is defined by the tensor product `|0>` $\otimes$ `|1>`. The input state is therefore `|01>`, and its matrix form is shown in equation 15.

$$\texttt{|01>} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{15}$$

After the input, apply two Hadamards to the circuit state in Stage 0 (counting from 0), as shown in equation 16 (note that as one moves right across the circuit, the matrix operations are mathematically applied to the left):

$$\Big[H \otimes H\Big] \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1/2 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{16}$$

This result at Stage 0 is common to all cases regardless of the Black Box function.

From this point forward, each Black Box case will be discussed separately. Assuming the Black Box is implementing f_1, matrix $U1$ is applied in Stage 1 to the Stage 0 result.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} (1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}) = 1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \tag{17}$$

Equation 17 shows the result of Stage 1. In Stage 2, apply two Hadamards (H $\otimes$ H) to the result of Stage 1:

$$(1/2 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix})(1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}) = 1/4 \begin{bmatrix} 1-1+1-1 \\ 1+1+1+1 \\ 1-1-1+1 \\ 1+1-1-1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{18}$$

Equation 18 gives the result at Stage 2. As shown, destructive interference reduces the probability of some results and constructive interference increases the probability of other results. In this

case, the amplitudes of |00>, |10>, and |11> are reduced to zero, and the amplitude (square root of the probability) of |01> is raised to 1. The probability of measuring 01 is $|1|^2 = 1$. At Stage 3, the measurement of the top qubit will be 0 with certainty and a measurement of the bottom qubit would be 1 with certainty.

Now the cases of the Black Box implementing the remaining three functions will be examined.

Picking up at Stage 1 and assuming the Black Box implements f_2, the effect of Stage 1 ($U2$) is

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} (1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}) = 1/2 \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \tag{19}$$

The effect of Stage 2 ($U2$) is

$$(1/2 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix})(1/2 \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}) = 1/4 \begin{bmatrix} -1+1-1+1 \\ -1-1-1-1 \\ -1+1+1-1 \\ -1-1+1+1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \end{bmatrix} \tag{20}$$

The probability of measuring 01 is $|-1|^2 = 1$. The measurement at Stage 3 will be 0 with certainty.

Picking up at Stage 1 and assuming the Black Box implements f_3, the effect of Stage 1 ($U3$) is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} (1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}) = 1/2 \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \tag{21}$$

The effect of Stage 2 ($U3$) is

$$(1/2 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix})(1/2 \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}) = 1/4 \begin{bmatrix} 1-1-1+1 \\ 1+1-1-1 \\ 1-1+1-1 \\ 1+1+1+1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{22}$$

The probability of measuring 11 is $|1|^2 = 1$. The measurement at Stage 3 will be 1 with certainty.

Picking up at Stage 1 and assuming the Black Box implements f_4, the effect of Stage 1 ($U4$) is

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} (1/2 \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}) = 1/2 \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \tag{23}$$

The effect of Stage 2 ($U4$) is

$$(1/2 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix})(1/2 \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}) = 1/4 \begin{bmatrix} -1+1+1-1 \\ -1-1+1+1 \\ -1+1-1+1 \\ -1-1-1-1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} \tag{24}$$

The probability of measuring 11 is $|-1|^2 = 1$. The measurement at Stage 3 will be 1 with certainty.

As shown mathematically in the equations above, if the Black Box function is from Set 1 (f_1 or f_2) the top qubit will be 0 with certainty, while if the function is from Set 2 (f_3 or f_4), the top qubit will be 1 with certainty. Only one query (and one measurement) is required to answer the question.

The Deutsch algorithm is a simple example but other algorithms that solve more complex real-world problems are being developed. Many of these are collected at the National Institute of Standards and Technology (http:// math.nist.gov/ quantum/ zoo).

As more algorithms are developed and quantum computers become available, there will be a need for a mechanism to instruct quantum computers to perform the correct operations to execute these algorithms. Quacee is designed to address this need.

3.2 Requirements

When creating a design, it is prudent to establish a set of requirements to be satisfied. Quantum programming languages are an area of active research, and in [2] a set of five initial requirements have been proposed. In the design of Quacee, these requirements have been accepted and further expanded. The Quacee design requirements are listed below.

Completeness: the language must support a set of quantum gates sufficient to express desired quantum circuits.

Extensibility: (1) the language must extend and interwork with a classical computing language; (2) it should also allow the user to define and use new quantum gates.

Separability: the quantum and classical parts of the system should be separable; it should be possible for the classical computations to operate on a classical computer, and the quantum computations on a quantum computer.

Expressivity: the language should allow the user to operate not only at the detailed gate level, but also at higher levels of abstraction such as the circuit level.

Independence: the language must be independent of the physical manifestation of a quantum machine and suitable for use with a variety of future quantum computers. It should operate at the quantum circuit level of abstraction, which is above and independent of differing physical hardware implementations.

Interactivity: the programmer should be able to create and modify quantum programs interactively and dynamically without a separate edit file, compile file, run program cycle.

3.3 Quacee

Quacee (**QUA**ntum **C**omputing **E**lucidation **E**xtension) has been designed to meet the requirements listed in Section 3.2 for a new quantum computing language. Quacee provides an extension to the Common LISP language, adding over 30 new function primitives which allow one to easily specify quantum circuits and the desired quantum computations.

Quacee allows the user to specify desired quantum circuits and inputs, working at both the quantum gate level as well as higher levels. The user has the flexibility to specify any stage of the circuit in any order. It validates that qubits are not used in a quantum manner after measurement, since after they are measured, they lose their superposition and act as regular bits (although they could still be used as controllers in controlled gates). Quacee also allows the user to create custom gates and validates that any custom gates specified with a transition matrix conform to the correct constraints (i.e. they must be unitary). The user can specify desired operations at a detailed gate level or at the higher level of circuits, save and load prior circuits, add circuits together to build more complex circuits, validate new circuits, and perform other functions discussed below.

The subsections which follow provide a high level summary of the capabilities provided by Quacee version 1.1.

3.3.1 Initialization Functions

Quacee provides functions which allow one to create quantum circuits for evaluation and establish initial qubit inputs, either from arbitrary lists or ordered by 0s or 1s. If the user knows the dimensions (number of qubits and stages) of the circuit in advance, they may specify it during circuit creation, but it is not mandated and Quacee will dynamically grow the circuit size as needed as it is being defined.

- `qc_create_circ`
- `qc_init_circ`
- `qc_add_qubits`
- `qc_init_qubits_by_list`
- `qc_init_qubits_0first`
- `qc_init_qubits_1first`

3.3.2 Applying Gates

Quacee provides functions which allow one to specify quantum state evolutions of the qubits by lower level gate operations using any of 23 supported standard quantum gates. Of note: the phase-shift and controlled phase-shift functions natively support angles of π, $\pi/2$, $\pi/4$, $\pi/8$, $\pi/16$, $\pi/32$, as well as allow the user to specify an arbitrary denominator.

In addition to support for standard gates, Quacee also provides the functionality to support customized gates. The user is able to specify a custom gate with any number of inputs. A unitary matrix must be provided to describe the operation of this gate. Quacee also supports controlled custom gates, with any number of 0-controllers (where the effect occurs when the controlling qubits are 0) and/or 1-controllers (where the effect occurs when the controlling qubits are 1). As with the general custom gate, a unitary matrix must be provided to describe the operation of the gate. For all custom gates, the supplied unitary matrix will be checked for validity before use.

- `qc_apply_hadamard`
- `qc_apply_pauliX`
- `qc_apply_pauliY`
- `qc_apply_pauliZ`
- `qc_apply_phshift`
- `qc_apply_cphshift`
- `qc_apply_meas`
- `qc_apply_swap`
- `qc_apply_cnot`
- `qc_apply_zcnot`
- `qc_apply_toffoli`
- `qc_apply_fredkin`
- `qc_apply_no_op`
- `qc_apply_customgate`
- `qc_apply_ctrld_customgate`

3.3.3 Higher Level Operations

Quacee provides functions which allow one to specify operations at a higher level. These include functions that allow one to move up from the gate level and instead use circuits as higher level building blocks, combining circuits together. The user may use circuits defined within a given program or others saved from prior program executions or other sources.

- `qc_add_circ_to_circ`
- `qc_save_circ_to_file`

- `qc_read_circ_from_file`

In addition, a set of 32 starter circuits are provided which create equal superpositions of n bits, with n ranging from 1 to 32. These are useful for algorithms which utilize the technique of beginning a circuit with a collection of ket-zero (`|0>`) inputs each passing through a Hadamard gate, creating an exponential set of states using a polynomial set of inputs. These circuits use the naming convention SuperposN.qcirc, where N ranges from 1 to 32, and can be read into the program using the `qc_read_circ_from_file` function. They can be used as a starter circuit or to conveniently add Hadamards to another circuit at an arbitrary offset.

Because of Quacee's support for saving and reading individual circuits to and from files, users can establish and share useful collections of quantum circuits to build upon and combine when writing quantum programs.

Other functions allow one to specify multiple gates at a time and validate a given circuit.

- `qc_add_mult_hadamards`
- `qc_validate_circ`

3.3.4 Output Functions

Quacee provides functions which allow one to output a quantum circuit in a variety of forms. These include `pretty_print` functions which output the quantum circuit in a human-friendly, ASCII-like picture. It also provides a function that translates the quantum circuit information into a dot file format which can be read by an external open-source program called graphviz to produce a graphical representation in common image formats, such as jpg or png. If Graphviz is installed on the classical computer, the `qc_output_circ_graph` function will automatically create the dot file translation and invoke Graphviz to generate a picture of the circuit. Gates with multiple inputs are shown on multiple lines (one for each qubit) with the common associations shown either by an arrow (for controlled gates in the graphical representation) or otherwise by showing a common gate ID for the gate type.

- `pretty_print_by_qubits`
- `pretty_print_qubit_path`
- `pretty_print_by_stages`
- `pretty_print_circ`
- `qc_output_circ_graph`

In the future, when universal quantum computers become available, Quacee would be used to send the desired quantum circuits to a quantum computer for evaluation as in Figure 3. To demonstrate this function, Quacee is equipped with two placeholder functions.

- `print_circ`
- `qc_exec_circ`

The function `print_circ` outputs the circuit data in a format that is not so easily readable to humans but is easily parsed by a computer. The function `qc_exec_circ` is a stand-in for the process of handing the quantum circuit data to a future quantum computer and awaiting results; in addition to some explanatory text, it outputs an easily parsed representation of the circuit data that would be sent to the quantum computer and the measurements expected as a result.

Note that the desired quantum computation is defined by a quantum circuit and Quacee includes the capabilities required to specify the desired circuit, including inputs and gate types, along with their matrix representations, to be applied to qubits at every stage.

3.3.5 Quantum Computer Simulator

As universal quantum computers are not currently available, a simulator functionality was also developed to simulate the operations of a future quantum computer when given Quacee quantum circuit output. It simulates evolving the qubits' state through the circuit, performing the mathematical operations described in Section 3.1. It computes the tensor product of the qubit input matrix representations, calculating the initial state of the system. It then iterates through the circuit stage by stage, computing the tensor product of each gate's transition matrix representation to generate a system transition matrix for that stage. It multiplies the current state matrix of the system by the system transition matrix, creating a new state matrix of the system and evolving the qubits throughout the circuit. The simulator also provides the capability of tracing the qubit system state as it evolves through the circuit, and tracing a joint probability matrix of the qubits as they evolve through the circuit.

4 Results

As detailed below, Quacee fulfills the requirements specified in Section 3.2.

Completeness: Quacee supports a set of quantum gates sufficient to express desired quantum circuits. It handles over 23 standard quantum gates, including the Hadamard, Pauli-X, -Y, -Z, Controlled Not, Various Phase-shifts, Toffoli, and the Fredkin. If a gate other than one of the natively supported standard gates is desired, Quacee offers the capability to create a custom gate to suit the user's needs.

Extensibility: Quacee supports both aspects of this requirement. It is written in Common LISP and acts as a quantum extension to the language while seamlessly interworking with it. As noted earlier, Quacee also allows the user to define and use new quantum gates if they so desire.

Separability: The quantum and classical parts of the system are separable but also interworkable. Referring to the mode of operation shown in Figure 3, classical computations operate on the classical computer. Quacee also runs on the classical computer, and is used to specify the desired quantum computations in the form of quantum circuits. The quantum circuit data can then be passed to the quantum computer for evaluation, with measurement results returned to the controlling computer. Quacee-based operations and general LISP-based operations are interoperable.

Expressivity: Quacee allows the user to operate not only at the detailed gate level, but also at higher levels of abstraction. It provides functions which allow one to operate at the circuit level, saving, reusing, and combining circuits as building blocks. In addition, a circuit-level validation function is provided. Multiple-gate functions are also supported.

Independence: Quacee is independent of the physical manifestation of a quantum machine and suitable for use with a variety of future universal quantum computers which support the quantum circuit model. Quacee allows the user to easily and flexibly specify quantum circuits and output them in a compact but easily parseable format suitable for use by a universal quantum computer.

Interactivity: With Quacee, the programmer is able to create and modify quantum programs interactively and dynamically without a separate edit file, compile file, run program cycle. Quacee inherits LISP's interactive development environment: if desired, one can develop, test, and interact with individual functions via the REPL (read-evaluate-print-loop). One can also follow a more conventional development style (editing a file, loading, and testing).

4.1 Quacee Results for a Selected Set of Quantum Circuits

In this section, a selected set of quantum circuits are illustrated using Quacee.

4.1.1 Deutsch Algorithm Circuit

A section of sample code is given below which demonstrates how Quacee can be used to create a quantum circuit representation of a quantum algorithm, in this case the Deutsch algorithm discussed earlier (see Figure 2 and Section 3.1). This annotated example was written to illustrate several of Quacee's features. Within this code sample, functions at the individual gate level, multiple gate level, and circuit level are used. In addition, support for custom gates is illustrated. This example also demonstrates interworking between classical language functions and the quantum language functions.

In the function below, a mixture of Common LISP and Quacee is used to generate four different quantum circuits representing the Deutsch algorithm applied to each of the four possible hidden black box functions (defined by four different unitary matrices). First, the parts of the circuit that are independent of the black box are established. Then, the circuit is repeatedly augmented with each of the four different black box possibilities in a loop.

```
(defun deutsch_circ ()
  (let*
       ;; set up an initial circuit with two qubits and
       ;; a vector with the 4 deutsch algo unitaries for black box use
      ((new_circ (qc_create_circ "Deutsch" :est_qubits 2 :est_stages 1))
       (da_unis (vector *U1* *U2* *U3* *U4*)))
    ;; initialize input to ket 01,  |01>
    ;; then add the initial Hadamard gates with one function
    ;; then add the later two Hadamards illustrating reading in a
    ;;    saved circuit along with a circuit level addition operation
    ;; then finally add a measurement gate for the top qubit
    (setf (circ_info-qubits_init_state new_circ) (qc_init_qubits_0first 1 1))
    (qc_apply_mult_hadamards (list 0 1) 0 new_circ)
    (qc_add_circ_to_circ (qc_read_from_file nil "Superpos2.qcirc")
                         new_circ 0 2)
    (qc_apply_meas 0 3 new_circ)
    ;; the shell of a deutsch algo circuit is now established
    ;; loop through placing different functions in the black box as a custom gate
    (dotimes (i 4 t)
      (setf (circ_info-id new_circ) (format nil "DeutschCircF~a" (1+ i)))
      (qc_apply_customgate (list 0 1) 1 new_circ (format nil "BlackBoxU~a" (1+ i))
                           (svref da_unis i))
      ;; deutsch circuit complete. could hand off to quantum computer at this point
      (qc_exec_circ new_circ)
      ;; also pretty_print and output a graph
      (pretty_print_circ new_circ)
      (qc_output_circ_graph new_circ))))
```

Going line by line, at the start of the function, a new quantum circuit is created, along with a vector of the unitary matrices which will be used in creating the custom black box. The unitary matrices *U1*, *U2*, *U3*, and *U4* were defined elsewhere using regular Common LISP, and are used to represent each of the possible hidden black box functions f_1, f_2, f_3, and f_4. The qubits are initialized to |0> and|1>, which starts the circuit off in the |01> state. Next, multiple Hadamard gates are added to the circuit using one function. Deferring the black box to the later loop, a circuit-level operation is used to copy an available two-Hadamard circuit that was saved to

the disk earlier onto the circuit being built at Stage 2, after the black box. Then, an individual measurement gate for the top qubit is added at the end of the circuit. Finally, using classical language looping methods, the function iterates through all four possible matrices for the black box, adding a custom Black Box gate to complete the circuit, and calling the placeholder exec function (plus the `pretty_print` and the output graph function) for each possibility.

Samples of the output of running the program are shown in Figures 5, 6, 7, and 8. Figure 5 shows output from `qc_exec_circ` the first time through the loop (with the Black Box implementing f_1). `qc_exec_circ` serves as a placeholder for the act of handing off the quantum circuit to an attached universal quantum computer using the mode of operation depicted in Figure 3.

```
qc_exec_circ: if there was an attached quantum computer,
the following quantum circuit would be given to it for evaluation --
[Qubit_0000] |0>:,(HADAMARD_GATE[##ID-0.0##]),(BlackBoxU1[##ID-0.1##]),(HADAMARD_GATE[##ID-0.2##]),(MEASURE_GATE[##ID-0.3##])
[Qubit_0001] |1>:,(HADAMARD_GATE[##ID-1.0##]),(BlackBoxU1[##ID-0.1##]),(HADAMARD_GATE[##ID-1.2##]),(NO_OP_GATE[##ID-1.3##])

Also, the following measurements would be expected as a result:
Qubit_0000 measured in stage 3

1 measurements would be expected as a result.
```

Figure 5: qc_exec_circ example for Deutsch algorithm (f_1)

Figures 6 and 7 show output graphs of the Black Box implementing f_2 (second time through the loop) and f_4 (fourth time through the loop), respectively. Based on the mathematical analysis in Section 3.1, a quantum computer should return a measurement of 0 for the first case (f_2) and a measurement of 1 for the second case (f_4).

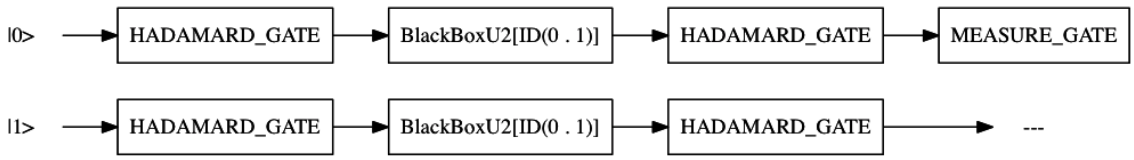

Figure 6: qc_output_circ_graph output example for Deutsch algorithm (f_2)

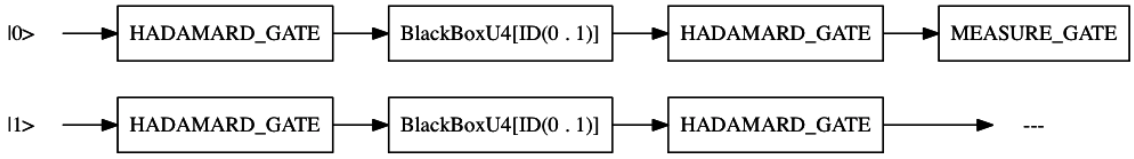

Figure 7: qc_output_circ_graph output example for Deutsch algorithm (f_4)

Finally, the results of the `pretty_print_circ` function (which creates an ASCII representation of the circuit) is shown in Figure 8 for the case of the Black Box implementing f_3 the third time through the loop.

```
CIRCUIT DeutschCircF3:

Qubit_0000  |0> ---| HADAMARD_GATE |------|   BlackBoxU3  |------| HADAMARD_GATE |------|  MEASURE_GATE |---
Qubit_0001  |1> ---| HADAMARD_GATE |------|   BlackBoxU3  |------| HADAMARD_GATE |------|   NO_OP_GATE  |---
```

Figure 8: pretty_print_circ_output example for Deutsch algorithm (f_3)

If the quantum computer simulator discussed in Section 3.3.5 is loaded, `qc_exec_circ` will call

the simulator, which will mimic the operations of a future universal quantum computer. Figure 9 shows the results of `qc_exec_circ` if the simulator is loaded. Based on the mathematical interpretation of the Deutsch Algorithm shown in Section 3.1, we would expect the measurement results of Qubit 0 for the 1st and 2nd functions to be 0, and the measurement results of Qubit 0 for the 3rd and 4th functions to be 1. As can be seen from the figure, the simulator results correspond with the expected mathematical results.

```
(deutsch_circ)

=========================================================
BEGIN SIMULATION
Simulating quantum computer operations
on the given quantum circuit DeutschCircF1....
---------------------------------------------------------
MEASUREMENTS at Stage 3
 Qubit 0 = 0
END SIMULATION
=========================================================

=========================================================
BEGIN SIMULATION
Simulating quantum computer operations
on the given quantum circuit DeutschCircF2....
---------------------------------------------------------
MEASUREMENTS at Stage 3
 Qubit 0 = 0
END SIMULATION
=========================================================

=========================================================
BEGIN SIMULATION
Simulating quantum computer operations
on the given quantum circuit DeutschCircF3....
---------------------------------------------------------
MEASUREMENTS at Stage 3
 Qubit 0 = 1
END SIMULATION
=========================================================

=========================================================
BEGIN SIMULATION
Simulating quantum computer operations
on the given quantum circuit DeutschCircF4....
---------------------------------------------------------
MEASUREMENTS at Stage 3
 Qubit 0 = 1
END SIMULATION
=========================================================
```

Figure 9: qc_simulate output example for Deutsch algorithm

4.1.2 Quantum Fourier Transform Circuit

The Quantum Fourier Transform (QFT) is a period-finding algorithm that can be used as a part of other algorithms. As a circuit, it can be used as a sub-circuit within a larger circuit. Quacee's `qc_add_circ_to_circ` function allows one to easily add a QFT sub-circuit to a larger circuit. A Quantium Fourier Transform circuit for three qubits consistent with the description in [9] was specified with Quacee and the resulting graph is depicted in Figure 10.

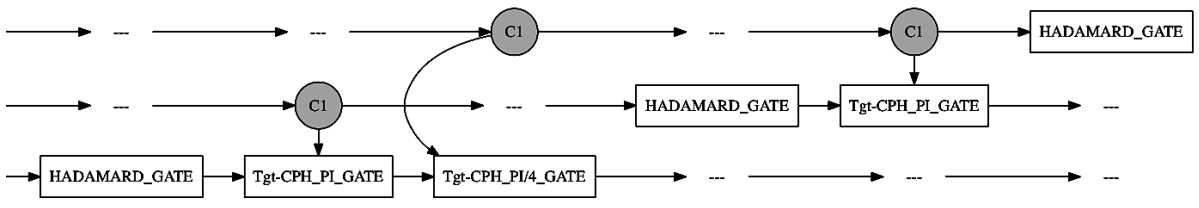

Figure 10: qc_output_circ_graph output example for programmed QFT circuit

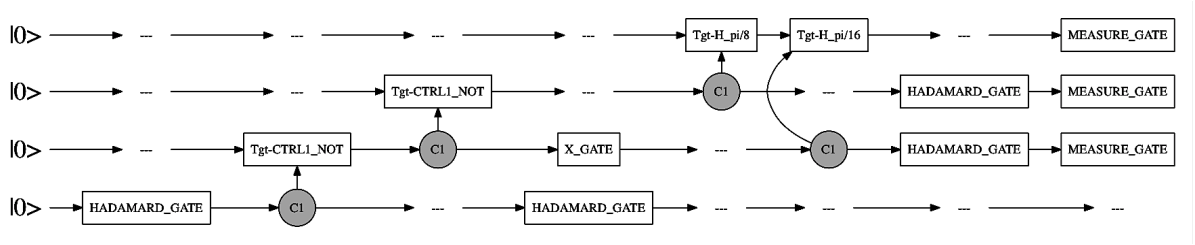

Figure 11: qc_output_circ_graph output example for $b3$ input of Cai circuit

4.1.3 Example Circuit to Solve System of Linear Equations

In Cai et. al. [10], the authors describe a quantum circuit providing a proof-of-principle demonstration of an algorithm for solving linear equations for three different input vectors $b1$, $b2$, and $b3$.

Quacee was used to program quantum circuits for this algorithm. An example of the output graph from `qc_output_circ_graph` for input $b3$ is shown in Figure 11. The input `|b>` is the bottom qubit. This circuit demonstrates the use of standard quantum gates such as the Hadamard, Pauli-X, CNOT, and others, as well as two custom controlled gates consistent with the definition in [10].

4.1.4 Example Circuit Illustrating the Variety of Gates Supported by Quacee

Figure 12 depicts an arbitrary Quacee demonstration circuit. This circuit was created to illustrate the wide range of gates that Quacee supports, showing all of the 23 standard quantum gate functions included in Quacee, as well as four example custom gates.

5 Summary and Future Work

The objective was to develop a new mechanism to describe desired quantum computations and program future universal quantum computers. First, an analysis of quantum computing was performed, including the representation of quantum bits (known as qubits), quantum gates for manipulating qubits, quantum algorithms, the quantum circuit model of quantum computations, and the mode of operation for future universal quantum computers supporting the quantum circuit model. Based on this analysis and recent research on quantum programming, a set of key requirements for quantum computing languages was established. A new quantum programming language called Quacee, which extends Common LISP, was designed and demonstrated to fulfill the requirements for completeness, extensibility, separability, expressivity, independence, and interactivity. The use of Quacee to specify quantum computations was illustrated by applying it to a selected set of quantum algorithms and creating corresponding quantum circuits for each. In addition, a simulator was developed to simulate the operations of a future universal quantum computer when given Quacee output; this simulator was used to test and verify Quacee's capabilities.

Future work could include adding support for automatically reversing portions of circuits.

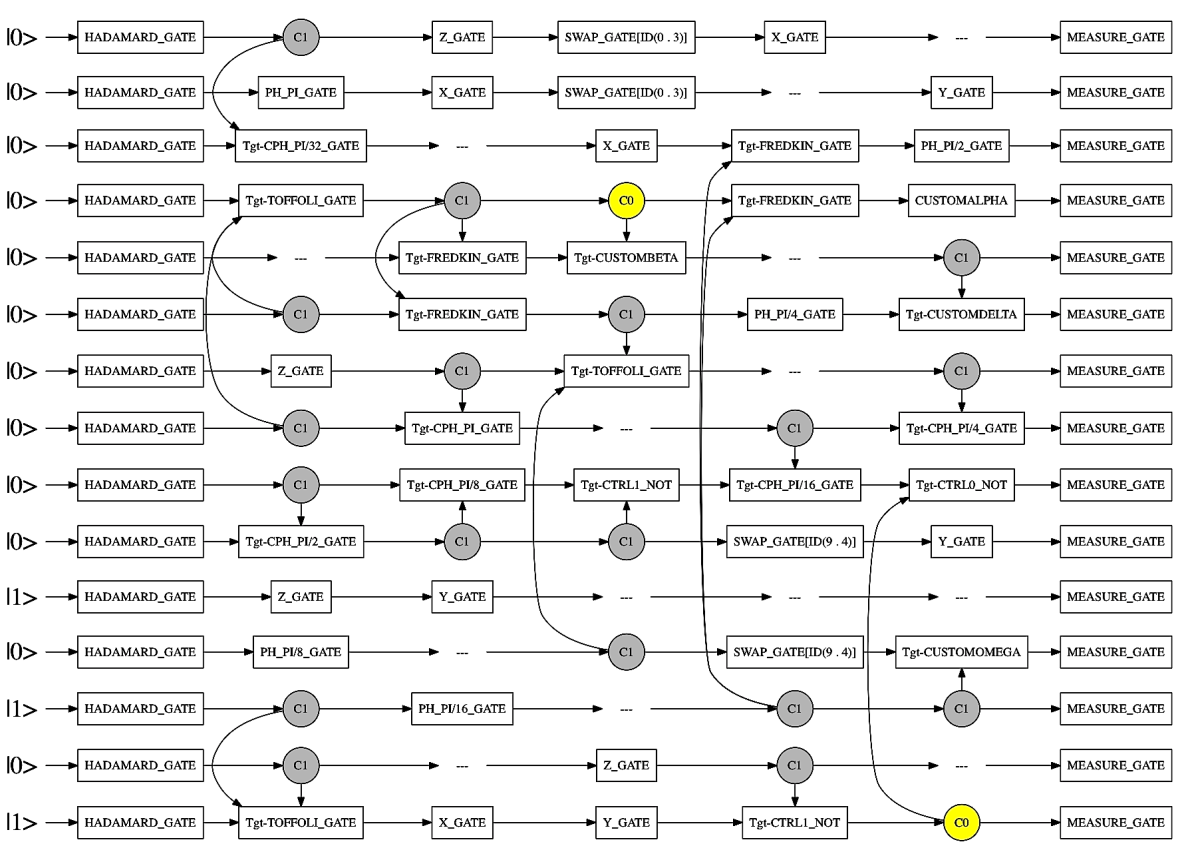

Figure 12: Output Graph - Quacee Demonstration Circuit

References

[1] Knill, E. et al. (2002). Quantum Information Processing: A Hands-On Primer. *Los Alamos Science* No. 27, pp. 2-37.

[2] Miszczak, J. A. (2011). Models of Quantum Computation and Quantum Programming Languages. *Bulletin of the Polish Academy of Sciences.* Vol 59, No. 3, pp. 305-324.

[3] Williams, C.P. (2011). Explorations in Quantum Computing. London, UK: Springer-Verlag.

[4] Ömer, B. (2003). Structured Quantum Programming. Ph.D. thesis, Vienna University of Technology.

[5] Bettelli, S., Serafini, L., and Calarco, T. (2003). Toward an architecture for quantum programming. *Eur. Phys. J. D* 25 (2), pp. 181-200.

[6] Mlnařík, H. (2008). Semantics Of Quantum Programming Language LanQ. *Int. J. Quant. Inf.* 6 (1, Supp.), pp. 733-738.

[7] Selinger, P. et. al. (2013). Quipper: A Scalable Quantum Programming Language. arXiv:1304.3390v1.

[8] Bacon, D. (2006). CSE 599d - Quantum Computing. University of Washington course notes, http://courses.cs.washington.edu/cse599d/06wi/.

[9] Dawar, A. (2008). Quantum Computing. University of Cambridge course notes, http://www.cl.cam.ac.uk/teaching/0809/QuantComp/notes.pdf.

[10] Cai, X.-D., et al. (2013). Experimental Quantum Computing to Solve Systems of Linear Equations. *Physical Review Letters* 110, 230501.