Partial report verification/testing

# Verification

Verification of the core engine is an important factor in ensuring the quality of the implementation. Although some parts of the system have undergone automated self-testing based on unit tests, the overall performance needs to be benchmarked against well-established results. Quantitative results should objectively show the fitness of the system.

This being said, verification of the current system as a whole is a quite challenging task. The reason is that this is the first full MBQC simulator that allows simulation of MCalc patterns. The lack of such software means that only parts of the whole could be automatically tested. Nevertheless several solutions exist that could aid the testing process.

Some of the methods that are presented in this section use already existing software that has been independently verified. They are based on the assumption that the third party software is producing correct results. Those methods could be in theory automated to produce consistent outputs and reliable benchmark for accuracy. This would guarantee that the components that are undergoing the aforementioned procedures are free from bugs in the context of the data that is being analysed. No testing could possibly guarantee absolutely bug free performance unless all possible inputs/output pairs of the algorithm are known in advance. This is impractical in the context of a simulator.

## Symbolic Algebra System

The mathematical system for symbolically manipulating algebraic expressions lies at the core of the whole project. All of the components rely on its correct behaviour. The way simplifications are done could introduce a large number of bugs. Luckily, testing of this system is simple enough and is quite reliable to rule out any possible causes of error.

A unit test **MathAutoTest** was specifically designed to perform an automated test onto the whole of the existing system, making sure that algebraic simplifications do not result in unexpected results. The idea is based on the fact that there is a simple way to perform the same computation that the algebraic system is supposed to undertake by explicitly calculating the results in parallel using ordinary floating point arithmetic. The result of the algebraic system is then compared with the expected result and any large discrepancies will make the automated unit test to fail.

In order to achieve this there is an array of **MathsItem**s. The *generateComplexMathItem* method creates a random **MathsItem** and puts it into a random place in the array. The way it is done is, it picks at random a mathematical function – *exp*, *sqrt*, *fract*ion, *Im* or a *mathematical symbol* or a *number* and uses the classes in the system to create a **MathsItem** object that represents it. If the resulting **MathsItem** is a mathematical function, a randomly chosen element from the array is used as an argument. If instead it is a symbol, it is picked from a predefined pool of available symbols that have some initial values (so that evaluation is possible). The newly formed **MathsItem** is modified by adding or multiplying it with a randomly chosen element of the array. The value of the function is compared before and after the simplification. If any differences are found, an error is thrown.

You could find below a sample pseudo code of the verification process:

Array of *MathsItems* **items**;

*Repeat* **N** times

Mathematical function **f** = *random* between (*exp*, *fract*, *sqrt*, *im*, *number*, *symbol*)

*If* (**f** is *exp*, *fract*, *sqrt* or *im*)

argument of **f** = random element from **items**

*If* (*random* number between [0, 2) == 0)

**f** = **f** + random element from **items**;

*else*

**f** = **f** \* random element from **items**;

*put* **f** in a random place in **items**;

*Complex* **result1** = **f**.evaluate();

**f**.simplify();

*Complex* **result2** = **f**.evaluate();

*If* (**result1** != **result2**)

*throw* an Exception(“TEST FAILED”);

Where **N** is the number of iterations to be run. This algorithm can generate arbitrary mathematical expressions because of the recursive nature of the generation process. To demonstrate its working, a sample possible output for the algorithm for **items** = {1, 1}, **N** = 5 and pool of symbols {*a*, *b*} would be:

1. **items** = {exp(1)+1, 1} *// exp is randomly generated, 1 is randomly picked for argument, it is then added to 1 and is put as first element of* ***items***
2. **items** = {exp(1)+1, a} *// a is randomly generated, it is multiplied by 1 and is put as second element of* ***items****. Note the simplification has turned a \* 1 into simply a*
3. **items** = {b \* (exp(1) + 1), a} *// b is generated and multiplied by the first element and set as first element*
4. **items** = {b \* (exp(1) + 1), a + sqrt(b \* (exp(1) + 1))} *// sqrt is generated, b \* (exp(1) + 1) is picked as an argument. It is then added to a and put as second element*
5. **items** = {Im( b \* (exp(1) + 1) ) \* (a + sqrt(b \* (exp(1) + 1))), a + sqrt(b \* (exp(1) + 1))} *// Im is generated, it got its argument from the first element and it was multiplied by the second element and put as first element of* ***items****.*

You could actually see that even for a small number of elements and iterations the algorithm produces a set of complicated random expressions. This means that the test is being conducted with completely arbitrary inputs. The Unit test has an **N** = 10 and number of elements in items = 20. The unit test actually runs the whole algorithm 10 additional times producing 100 different expressions which are being tested.

The core of the test is comparing the value of a random expression before and after simplification. This will pick up all possible errors and bugs that could exist in the symbolic algebra manipulation library. This is because the bugs could be divided into two subcategories.

* Simplification bugs – the resulting expression after simplification is not mathematically equivalent to the initial expression. This would yield a difference in the evaluated output of the expressions and will be detected as an assertion error and the test will fail.
* Evaluation bugs – assuming the simplification is correct, the two resulting expressions would be mathematically equivalent. It will be very likely, that they will have different representations. If the evaluation system is malfunctioning the resulting expression after simplification will not yield the same result as the original one therefore the test will fail.

Since the unit test is not failing for any number of runs, it could be assumed that the mathematical system can handle an arbitrary expression and simplify it correctly. Therefore the correctness of the algebraic system could be proven for the random examples that have been generated during testing.

## Experimental Data Analysis

The first approach to verifying the fitness of the simulation itself is to actually use real experimental data and compare it with the predictions done by the simulator. The experimental data that is currently available is taken from an experiment on optical implementation of quantum computing. The choice of data source is rather useful since there already exists a third party software that is specifically developed to analyse this particular experimental setup. Furthermore, the available software has been tested and provides a good benchmark for performance and accuracy.

### Optical Implementation of Quantum Computing

The experimental setup is described in details in *Diagnosing Optical Implementations of Quantum Computing* by Flaviu Cipcigan. In a nutshell, a laser produces photons which pass through a series of optical filters. Each filter applies a certain quantum operation to the photons. The original stream is split into several photon beams. Detectors at the end of the optical paths measure the arrival of individual photons. Most of the events are discarded, since the probability of entangling photons is quite low. When photons arrive at the same time in all of the detectors, a certain assumption could be made about the particles and their quantum states. Events that have such coincidences are recorded and the number of photons arriving at those states is stored into a csv file.

The quantum “algorithm” implemented could be neatly described in terms of MCalc operations onto an initial system called a “lab cluster”. A simulation of this lab cluster with a series of measurements could be done. This will change the state of the cluster. The probability of obtaining this particular state could be calculated and it directly relates with the relative number of photons that are expected to trigger the simultaneous events at the detector. The predicted probability and the actually obtained one could be compared and the quality of the comparison will prove the quality of the simulation algorithm.

The lab cluster contains 4 qubits. They start in an initial state

Where *n* depends on the type of experiment that is being performed and is recorder in the filename of the csv file.

A series of 4 measurements are made, which could be either an X, Y, M, P or Z measurement. Each set of measurement corresponds to a row in the csv file. For each of them, events are recorded for all of the possible 4 branches thus generating 16 columns for each row.

In order to obtain the probability of a particle ending up in a certain branch when a certain sequence of measurements is applied, the state of the lab cluster is used. As mentioned previously in the report, the probability of a branch is the norm squared of the resulting vector.

### Simulation

In order to implement a simulator tailored for the optical quantum computation experiment, several additional entities needed to be added to the already existing system.

**LabCluster** – a class that extends SystemMatrix and allows for initialization of the pool of bits to a certain lab cluster. Once an instance of this class is created, measurements could be directly applied to it and the probability could be extracted using the inherited methods from SystemMatrix.

**ZM** – a Z measurement in the basis of . This literally means that a Z measurement in branch 0 will take only the coefficients of the current qubit that are in the state, setting the others to 0. A measurement in branch 1 will take only the coefficients in the state.

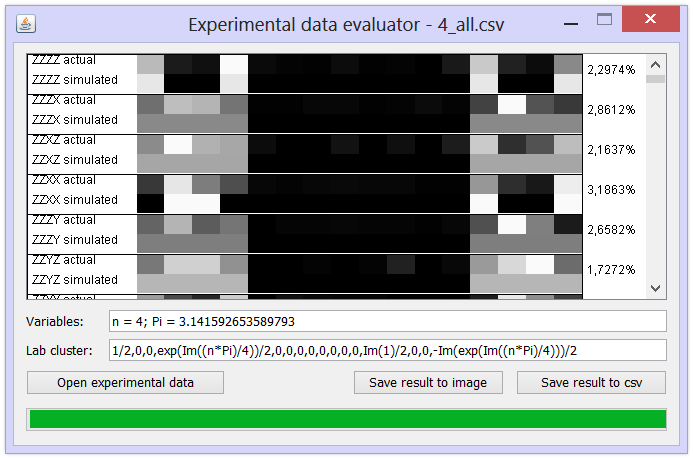
**WorkSheet, TableStorage** – for parsing the csv and storing the original experimental probabilities for each pair of measurements and branches.

**ExperimentVisualizer, CSVExporter** – for converting predicted output to images and csv files for further analysis and comparison with the original data.

**ExperimTest** – performs the tests by loading a file. It takes an ASCII version of the initial cluster state and runs an experiment. Some sanity checks of the results and inputs is also done and warnings are issued in cases in which total probability does not add up to 1 or input is not normalized. A callback ensures that updates are being passed to the original invoker, allowing for user friendly features like a progress bar to be rendered on the screen. It then returns a result that contains a CSV file and an image as a result of the simulation.

A GUI interface was developed for intuitively interacting with the simulation. It allows for defining a custom lab cluster and setting values for variables. It then shows the visual representation of the analysis on screen and gives option to save it as an image file or as a csv file. Also a progress bar is shown while the analysis is taking place so the user is aware of the status of the computation.

This configuration allows for flexibility in the simulation. If a new set of experiments is generated (that follows the same input csv format), the analysis could be easily adapted by changing the initial state of the lab cluster. The system can handle an arbitrary number of qubits as well (i.e. the lab cluster could in theory hold more than 4 qubits) as long as the number of qubits in the csv file matches with the number of qubits in the lab cluster.



Simulation output

Cluster definition

Input/output

### Output

Figure 1 – GUI of the tool for experimental data evaluation and comparison

The output from the simulator could be best visualized using a grayscale mapping[[1]](#footnote-1). The mapping is between the minimum (black) and maximum (white) probability in each row between the actual and the simulated data.

min

max

How to interpret a “row” from the generated result image:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Measurement sequence | Branches | | | | | | | | | | | | | | | | Average residual error |
| 0 0 0 0 | 0 0 0 1 | 0 0 1 0 | 0 0 1 1 | 0 1 0 0 | 0 1 0 1 | 0 1 1 0 | 0 1 1 1 | 1 0 0 0 | 1 0 0 1 | 1 0 1 0 | 1 0 1 1 | 1 1 0 0 | 1 1 0 1 | 1 1 1 0 | 1 1 1 1 |



The row above means that for each cell, which is illuminated, the lab cluster needs to be initialized and 4 measurements must be done (in this case a Y measurement, followed by three consecutive Z measurements). The first column represents all measurements resulting in a projection to the 0 branch. The second column means the final Z measurement is resulting in a projection to the 1 branch while the other 3 result to the 0 branch.

Therefore for each row like this, 16 simulations are done using the following algorithm:

*For* each branch combination **B** in **bs** *do*:

*Initialize* new lab cluster

*For* **i** from 0 to size of **ms** do:

*Perform* measurement **ms**[**i**] taking a branch **B**[**i**]

*Calculate* probability of the cluster

For the example above, **ms** = {Y, Z, Z, Z}, **bs** = { {0,0,0,0}, {0,0,0,1}, {0,0,1,0} … {1,1,1,1} }

If we assume that we have the original probabilities in an array **orig** and the simulated probabilities in an array **sim**, then we could take the values **max\_val** = *max*( *max*( **orig** ), *max* ( **sim** ) ) and **min\_val** = *min*( *min*( **orig** ), *min* ( **sim** ) ). The colour for each cell is calculated as **colour** = (**current\_value** – **min\_val**) / ( **max\_val** – **min\_val**). The resulting value is mapped as **colour** = 0.0 means *black* and **colour** = 1.0 means *white*.

The average residual error is the average absolute difference between results obtained in the experiment and the simulated output for a given row. It gives a quantitative measure on the accuracy of the theoretical fit. It is reported in percent, which is the actual value multiplied by 100.

The relationship of actual values vs predicted vs residuals should be clear from clear from the figure above. The average value of the residual line is 0.0166 therefore the reported value as an average residual error for the row shown in Figure 1 – GUI of the tool for experimental data evaluation and comparisonis 1.6617%. You could also see that in this case **max\_val** = 0.149 and **min\_val** = 0.

### Results of verification

The verification has been done on 8 lab clusters, each with 225 measurement tuples, each with 16 possible branches. This totals to 28800 simulations for the available data. The average residual error is 3.68%. The best simulated experiment has a residual error of 0.8% while the worst one has an error of 10.8%.

49.83% of the rows have a value of the average residual error less than 2.96%. Only 27.61% of the data has an average residual error bigger than 3.95%. This could be clearly seen on the cumulative average residual distribution profile below. The plot shows the fraction of data rows which have an average residual error less than a certain value.

This clearly show that majority of the data could be explained using the simulation correctly. Here are two examples of a good and a bad theoretical prediction.



Figure 2 Good agreement with simulation



Figure 3 Disagreement with simulation

It could be seen that the actual data is quite noisy but the pattern that it follows could be observed. The reason for disagreement is the Y measurement and it is explained in detail in *Diagnosing Optical Implementations of Quantum Computing* by Flaviu Cipcigan.

A result of the *Diagnosing Optical Implementations of Quantum Computing*, a software was developed for specifically verifying the results of the above experiment. The software was run on the exact data set input files and the predicted

|  |  |
| --- | --- |
|  |  |

Figure 4 Comparison between output of the current project (on the left) and Mr Cipcigan's project (on the right)

The resulting data was compared and the two applications agreed completely on the predicted values. This could be seen on Figure 4 Comparison between output of the current project (on the left) and Mr Cipcigan's project (on the right). These are the first 5 measurements of the 7th lab cluster. They match completely as do all of the other predictions.

### Conclusion

It should be made clear that Mr Cipcigan’s software uses a completely different algorithm for generating the theoretical predictions. His software works exclusively with probabilities. The core of the current project is a full MBQC simulation. Furthermore, the current project uses symbolic algebra which could make printing the exact probability and lab cluster more human readable.

It is apparent from the results presented in the previous subsection that the output from the current simulator completely matches with the output from an already existing software. Furthermore, it could be successfully used to explain and analyze experimental data. Therefore this should guarantee the correctness of the implementation of several of the core systems – the symbolic mathematics manipulation and the MCalc and cluster operation. This is the essentially all of the required components for a full MBQC simulator.

It should be noted that the test above does not test the user-interaction part of the system. Systems that are not being used by the above test are the ASCII to MCalc translation and the graphical pattern to MCalc translation. Those systems would be targeted with the next verification technique.

## Semi-automated Procedure

Simulation of an arbitrary MBQC pattern was not possible before the development of this software. Therefore directly comparing the output of a random MBQC pattern with an already existing software solution is impossible. This makes the task of evaluating the accuracy of the ASCII and graphical to MCalc translation challenging.

### Procedure

A possible approach is to use more than one already existing software packages and pipe their inputs and outputs together. The method proposed below, in theory, could be fully automated to generate a good test. The reason the automation hasn’t been done at this point is that one of the third party software does not have an automated input system but rather relies on a GUI.

The software packages that have been used are:

* Software[[2]](#footnote-2) developed by Einar Pius that translates quantum circuits into MCalc description. The software is written in C++.
* Quantum Circuit Simulator[[3]](#footnote-3) by Davy Wybiral that simulates quantum circuits. The software is web based and written in JavaScript.

The way the procedure works:

Random Quantum Circuit

QCircuit to MBQC translator (ParallelQC)

Web Based Circuit Simulator

The MBQC simulator that is being checked

Data 2

Data 1

The Random Quantum Circuit could be literally a random generated one. It does not need to represent anything physically meaningful. The verification method will produce two sets of data **Data 1** and **Data 2** for each Quantum Circuit. Those two sets could be compared in order to evaluate the correctness of the implementation.

### Implementation

There is a user interface built in the GUI that automates the translation from a quantum circuit to MCalc and evaluates the pattern using the MBQC simulator engine for all possible inputs. It also generates a nice ASCII image analogous to the way the web-based quantum simulator would display the quantum circuit.

The input quantum circuit can contain the following gates ZZ, H, J(pi), J(pi/2), J(pi/4) and J(pi/8). The input is compatible with the input of Einar Pius’ quantum circuit to MCalc pattern translator. When the “Evaluate” button is pressed, the input circuit is parsed and transformed into an array of **QCGate** objects. Each of those objects represents a quantum circuit gate and the order of the elements in the array is the order in which the gates act on the input qubits.

This representation is important so that the input quantum circuit could be presented into a compatible format for the web-based quantum circuit simulator. The simulator does not use the same format as Einar’s translator. Furthermore, it uses a different set of gates. Luckily the six possible gates in the input that we already mentioned could be translated into compatible sequence of gates that the web-based translator supports. Each of the six gates has a unique equivalent representation. The representations of those gates from the input **QCGate** array is concatenated together in order to form the ASCII web-friendly representation of the input quantum circuit.

The input is saved to a temporary file and the **ParallelQC** executable is invoked with the file as an argument. The verbose output is analyzed and the resulting MCalc description is stored. It is then parsed into a **MCalcDescription** object. This object is used to run a simulation using the symbolic algebra system. A **SystemMatrix** is generated that contains the output of the MBQC simulation.

All possible inputs (for the specific number of qubits) are generated, and the **SystemMatrix** is reevaluated for each of the possible input values. A note should be made that since symbolic algebra is involved, once the **SystemMatrix** is obtained in the previous step, a new simulation is not necessary for each input. The outputs are then shown on the screen.

There are, though, some incompatibilities in some MCalc descriptions that are generated by the **ParallelQC** translator. The MCalc model limits the **n** input qubits to always be the first **n** qubits in an MBQC pattern. This is not always the case in **ParallelQC** since the Circuit to MBQC translation is not what the software was originally designed for, and this translation is used internally (in order for the MBQC pattern to be obtained, a verbose mode should be enabled for **ParallelQC**). Therefore some patterns generated by **ParallelQC** might be incompatible with the MBQC simulator. Furthermore, some patterns outputs could contain a large number of qubits which may be bigger than what is practical to simulate (as discussed in the beginning of the report). Therefore if an incompatible pattern is introduced, the test suit will try its best to give a meaningful error message to the user.

The automatic quantum circuit pattern generation tries to generate completely random patterns that are always compatible with the MBQC simulator. As a start, two random numbers are picked – the number of gates and the number of qubits in the circuit. In order to ensure that the generated MBQC pattern is possible to simulate, the maximum number of qubits is limited to 3 and the maximum number of gates is limited to 5. The gates are generated using a completely random distribution (using the Java **Random** class). The qubits they act on are also randomly generated with the only restriction being that the qubit ids must be getting bigger in ids, starting from the 1st one. A note should be made that this decision does not mean that only a specific subset of MBQC patterns are isolated. All of the resulting MBQC patterns would be still random. The only limit this restriction poses is that the input qubits in the resulting MBQC pattern are always the first **n** qubits (so the convention is followed).

### Usage

From the Tools menu, the user could open the Quantum Circuit Evaluator. They need to choose the path to the ParallelQC executable which is Einar Pius’ quantum circuit to MCalc pattern translator. For each test the user would like to run, they need to follow the simple procedure:

* Generate a random Quantum Circuit with the “Generate random” button (or alternatively, load it from file or manually type it in)
* Press “Evaluate”
* Copy the web-compatible description of the quantum circuit into the web-based quantum circuit simulator.
* Use the web-based quantum circuit simulator to evaluate the circuit for a sample input.
* Compare the output of the web-based simulator with the output generated from the same input in the MBQC simulator.

The procedure is semi-automatic because there is no easy way of interfacing the web-based quantum simulator automatically. Therefore the user would need to manually input the circuit into the web-based simulator. It is worth noticing that the user is a passive element in the testing procedure; they do not do any computation, but rather act as a link that merely transfers output from one component to the other component. If there was an easy API to access the web-based simulator, the whole procedure could have been fully automated.

Figure 5 The MBQC simulator (on the left) and the QC Web simulator (on right) result comparisonshows a sample quantum circuit that has been using the procedure above. The similarity between the ASCII based web-friendly circuit description and the corresponding representation inside the web simulator is clearly visible. You could also notice that the outputs from both simulators for the input completely match for the provided 9 significant figures. The user could utilize the web-based simulator to change the input (this is done by clicking on the beginning of the line representing the input qubits) and compare the generated output with the output from the MBQC simulator. In this case the verifications showed a complete agreement between the results obtained by the two simulators.

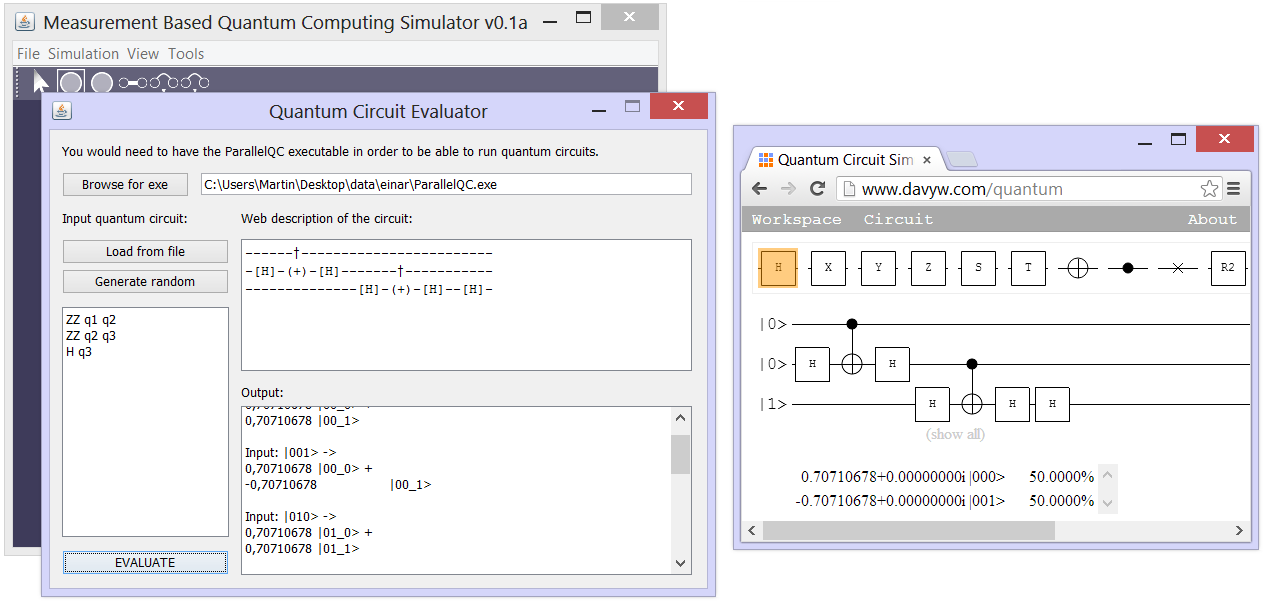


Figure 5 The MBQC simulator (on the left) and the QC Web simulator (on right) result comparison

### Results

50 random quantum circuits were generated. 9 of them contained 1 gate, 11 of them – 2, 12 – 3, 8 – 4 and 10 – 5 gates. Of them 13 had 1 qubit, 19 had 2 qubits and 18 had 3 qubits in their quantum circuit representation. This means that 2 × 13 + 4 × 19 + 8 × 18 = 246 outputs were obtained from both simulators and compared. All of them completely agreed up to the 9th decimal digit, which is the limiting precision of the web-based simulator.

The procedure uses software packages which were already properly tested. Although the verification is semi-automatic, the user does not take part in any data processing therefore the user is not a factor in the results obtained. The agreement in the outputs can prove the reliability of the MBQC simulator that is undergoing the testing. Furthermore, the results show clearly that at least 50 pseudo-randomly generated MBQC patterns could be simulated correctly.

The components that have been tested using the procedure are the ASCII MCalc parser, the symbolic mathematics engine and the quantum operator system. The evidence show that under the testing conditions, they behave as expected and produce consistent results. The component that has not been tested using this procedure is the Graphical GUI to ASCII Mcalc translation process.

## Manual Testing

Some components that comprise the human – computer interaction part of the project could not be tested procedurally. For this testing, special patterns were input manually to verify the correctness of the implementation. This was the main method that was used during the early stages of the development since most major bugs could be detected this way. It does not prove completeness, nor quality but it gives certain guarantees that widely known cases are being simulated correctly.

Other aspects, specifically the quality and the intuitiveness of the GUI could be also evaluated. F feedback was collected from people already familiar with the concept of MBQC patterns. They were able to use the graphical interface to set the input for the simulation and obtained results which were consistent with the manual theoretical predictions.

Some of the patterns that were tested are quite computationally intensive. They are extremely tedious to be simulated with pen and paper. A good example is the Figure 7 Swap example which if simulated using pen and paper would mean manipulation of 256 coefficients and keeping track of 256 bra-kets and performing 6 measurements with the corresponding corrections. Nevertheless the outputs of the circuits that have been tested is known. The simulator is successfully able to deliver the expected output for all of them. Below are examples of some of the MBQC patterns that were manually simulated and verified.

|  |  |
| --- | --- |
| Figure 6 Identity | Figure 7 Swap |

Other patterns that have been verified using this method are two versions of CNOT, CNOT+Hadamards and the quantum teleportation example.

An advice was made by Einar Pius to mention this small note. A special care should be made to the way outputs are numbered. This could be a source of confusion since the resulting bra-ket is reported in ascending order of the qubit id. When handling the examples manually, this is usually not noted and if not enough care was taken it may look like the output from the simulator has its inputs swapped. In such a case take a closer look in the input MBQC pattern and check whether the ids of the output qubits are correct.

Overall the manual testing resulted in those famous quantum computing patterns being run correctly. This would mean that the system as a whole is working as expected for the provided examples. The testing has included all of the components of the simulator (symbolic algebra, ASCII Mcalc parser and quantum operations), including the graphical to MBQC translator.

1. Flaviu Cipcigan, Diagnosing Optical Implementations of Quantum Computing [↑](#footnote-ref-1)
2. Einar Pius, Automatic Parallelisation of Quantum Circuits Using the Measurement Based Quantum Computing Model [↑](#footnote-ref-2)
3. <http://www.davyw.com/quantum> [↑](#footnote-ref-3)