Budapest University of Technology and Economics

Quantum Software for Adding Two Numbers: Testing and Verification Approach

Laith Soub[1]

Department of Networked Systems and Services

lalsoub@hit.bme.hu

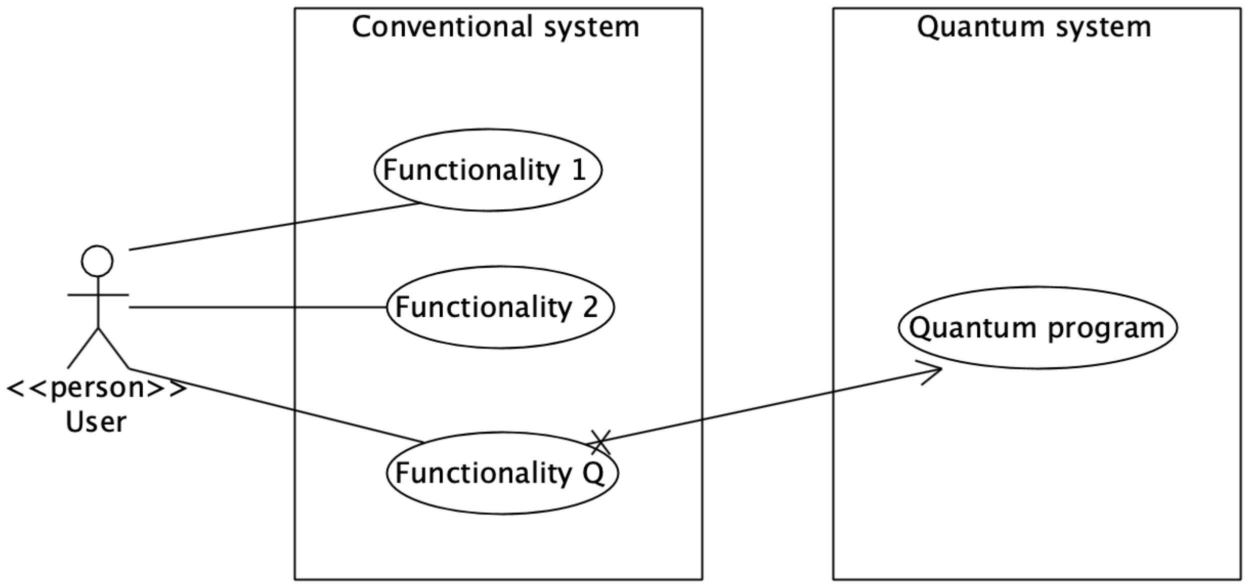
1- Introduction

Quantum computers operates at temperatures close to absolute zero. One among the characteristics of calculus created with quantum computers is that the uncertainty of the results, that aren't always the same: in reality, a similar program executed twice might turn out completely different outcomes with different possibilities. As the range of executions grows up, the program output that would be considered the actual output is going to be the most repeated one. This uncertainty introduces fascinating challenges in quantum program testing. the case is completely different if the quantum program is executed on a machine, that turn out fixed results.

In classical software engineering, a test case consists of [1] “specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective […]”. One of the desired characteristics of test cases is their “repeatability”: this is, the verdict of a test must be always the same. The “verdict” is the result of the test execution, usually “Pass” (if the test has not found any error in the system under test, the SUT) or “Fail” (if the system has shown a behavior different than the expected one). So, a test case has three parts: (1) specification of the initial situation, (2) execution of operations on the SUT and (3) an oracle, which compares the actual and expected outputs, so determining the verdict.

Thus, executing a test case on a quantum computer actually requires executing the same test case a number of times and then, to compare the most repeated output to the expected output. As well as this adaptation is required, quantum software testing requires both the adaptation of other techniques, as well as the creation of new ones.

In practice, quantum programs provide quantum functionalities to conventional programs via some kind of interface. In Figure 1, the *Functionality Q* calls the *quantum program*, which runs on a quantum computer, to perform some specific calculus.



Going back to the three-parts structure of a test case and recalling that the uncertainty requires that every test case must be executed several times: (1) the initial situation of a quantum test case will set up the initial status of the qubits, (2) as in conventional testing, the quantum circuit will be executed, and (3) finally, the test suite will save the obtained result, in order to calculate, in a further step, which is the most probable actual result.

2- White box testing

In white box testing, the tester is interested in knowing which "fragments" of the program are executed by the test cases. Thus, a program is completely tested if the test cases have gone through all those "fragments". Moreover, if the test cases do not reveal errors, then the probability that the program is right is very high. The problem here is in the granularity of the "fragment". To measure it, there exist a number of coverage criteria [2], such as *statements, branches, conditions, decisions, modified condition-decision…* Thus, if a test suite runs all the statements of the SUT, it is said that its coverage is 100%. Obviously, some coverage criteria are more powerful than others: running all the *methods* is a coarse criterion that is *subsumed* by, for example, *statements* because if a test suite runs all the program statements, it runs also all its methods. Modified condition-decision (MC/DC) is one of the most powerful coverage criteria. It is fulfilled when every decision (understood as a set of conditions: for example, *if A>B or B<C* is a decision with two conditions, *A>B* and *B<C*) is executed in its two branches (true and false) thanks to the individual contribution of each individual condition. For this example, since the decision is composed by two conditions separated by *OR*, a complete test suite should lead the decision to the *true* branch thanks to *A>B* and to *B<C* (first two rows of Table 1), and to the *false* branch also thanks to each individual condition: since the operator is *OR*, this requires to do *false* both individual conditions, as shown in the third row.

A>B	B<C	A>B or B<C	Determinant condition
true	false	true	A>B (B<C has no influence)
false	true	true	B<C (A>B has no influence)
false	false	false	Both conditions influence

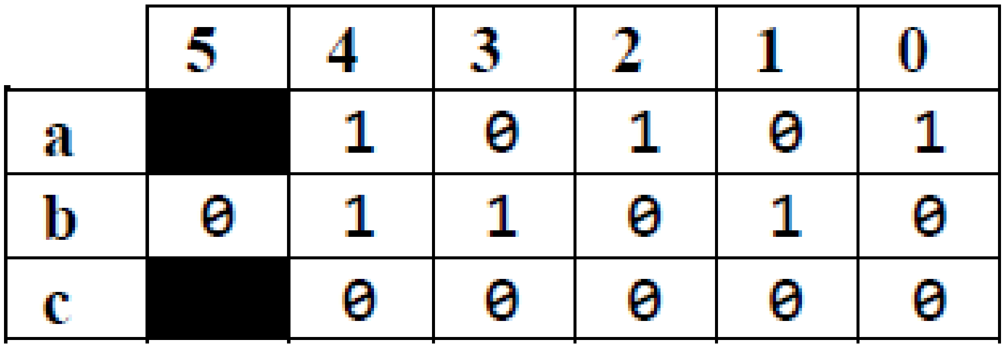
An additional white-box coverage criterion is based on mutation testing (**Mutation Testing** is a type of software **testing** in which certain statements of the source code are changed/**mutated** to check if the **test** cases are able to find errors in source code. The goal of **Mutation Testing** is ensuring the quality of **test** cases in terms of robustness that it should fail the **mutated** source code.). In mutation testing, the tester creates a set of mutants, which are copies of the SUT. Each copy contains a syntactic change that, with the adequate test case, may produce an output different than the SUT's output. Thus, the mutant can be seen as a faulty version of the original program. In this way, a mutation-based testing process finishes when the test suite has found all the faults inserted in the mutants and none in the original one. Some of the mutants produced are *functionally equivalent*, what means that it is impossible to find a test case that finding the inserted fault (in practice, this means that the change introduced in the mutant is not a fault, but a code optimization or de-optimization). The coverage is measured in terms of the *mutation score*, which is the percentage of faults discovered (excluding equivalent mutants). Artificial faults are inserted by means of mutation operators. A mutation operator introduces some type of fault: some of the most typical mutation operators (see Table2) are AOR (Arithmetic Operator Replacement), ROR (Relational Operator Replacement) or UOI (Unary Operator Insertion). As it is seen, the artificial faults reproduce errors that a *competent programmer* could commit [3, 4].

`if (a>b)` `return a+b;`	AOR	`if (a>b)` `return a*b;`
		`if (a>b)` `return a-b;`
	ROR	`if (a==b)` `return a+b;`
	UOI	`if (-a==b)` `return a+b;`

There are many different types of mutation operators for reproducing many different types of faults. There are also specific operators for specific programming languages (operators on the use of pointers in C and C++, for example) and specific technologies, such as mobile software. Depending on the goodness of the mutation operators used, a given mutation score may subsume other coverage criteria. Mutation testing is quite suitable for quantum testing.

3- Adding two numbers

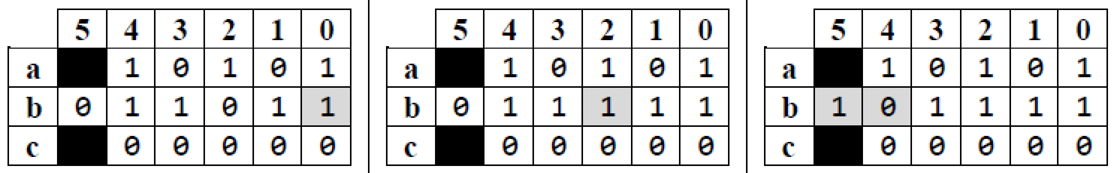
In quantum programming, the programmer writes her/his programs directly using bits (actually qubits) and logic gates (actually quantum gates). Thus, the simple operation of adding two numbers requires to work at so low abstraction level.
Anagolum [6] presents an example for adding two integer numbers using the IBM's QISkit simulator. The program code (and its corresponding quantum circuit) reproduces the process used by a classical computer (carry bits, etc.). The example needs two qubytes for saving the initial numbers, one additional qubyte for the carry bits and one classical byte (to save the state of the read qubits). The program code, translated into OpenQASM, appears in Figure 3: it reserves 5 qubits for the first binary number *(qreg a)*, 6 for the second one *(qreg b)*, 5 for the carry bits *(qreg c)* and 6 classical bits (*creg d)* for saving the results. Initially all qubits are set to 0. Some of them (0, 2, 4 of *a* and 1, 3, 4 of *b*) are set to 1 via de *x* gate. Thus, the input numbers are those in Figure 3.



	5	4	3	2	1	0
a	■	1	0	1	0	1
b	0	1	1	0	1	0
c	■	0	0	0	0	0

Then, the remaining code in the left column is composed by five blocks of three statements. Each block performs the following:
1. Via the *ccx* gate, if *a[i]=1* and *b[i]=1*, then it changes the value of *c[i+1].*
2. With *cx*, if *a[i]=1*, then it changes the value of *b[i].*
3. The third statement changes *c[i+1]* if *b[i]=1* and *c[i]=1.*

In these blocks of statements, the qubits change according to Figure 4.



	5	4	3	2	1	0
a	■	1	0	1	0	1
b	0	1	1	0	1	1
c	■	0	0	0	0	0

After the 1st block

	5	4	3	2	1	0
a	■	1	0	1	0	1
b	0	1	1	1	1	1
c	■	0	0	0	0	0

After the 3st block (the 2nd changes nothing)

	5	4	3	2	1	0
a	■	1	0	1	0	1
b	1	0	1	1	1	1
c	■	0	0	0	0	0

After the 4th block (the 3rd changes nothing)

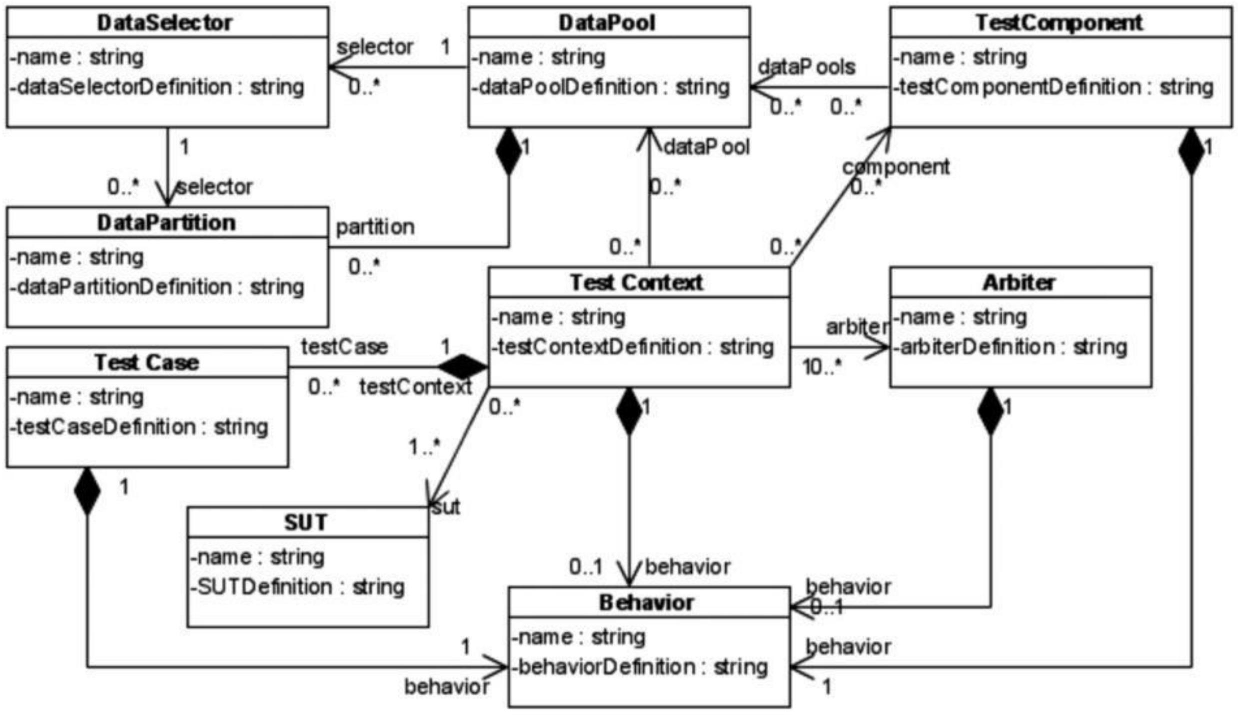
```
OPENQASM 2.0;
include "qelib1.inc";
qreg a[5];
qreg b[6];
qreg c[5];
creg d[6];

x a[0];
x a[2];
x a[4];

x b[1];
x b[3];
x b[4];

ccx a[0], b[0], c[1];
cx a[0], b[0];
ccx b[0], c[0], c[1];

ccx a[1], b[1], c[2];
cx a[1], b[1];
ccx b[1], c[1], c[2];

ccx a[2], b[2], c[3];
cx a[2], b[2];
ccx b[2], c[2], c[3];

ccx a[3], b[3], c[4];
cx a[3], b[3];
ccx b[3], c[3], c[4];

ccx a[4], b[4], b[5];
cx a[4], b[4];
ccx b[4], c[4], b[5];
```

```
cx c[4], b[4];

ccx b[3], c[3], c[4];
cx a[3], b[3];
ccx a[3], b[3], c[4];
cx c[3], b[3];
cx a[3], b[3];

ccx b[2], c[2], c[3];
cx a[2], b[2];
ccx a[2], b[2], c[3];
cx c[2], b[2];
cx a[2], b[2];

ccx b[1], c[1], c[2];
cx a[1], b[1];
ccx a[1], b[1], c[1];
cx c[1], b[1];
cx a[1], b[1];

ccx b[0], c[0], c[1];
cx a[0], b[0];
ccx a[0], b[0], c[1];
cx c[0], b[0];
cx a[0], b[0];

measure b[0] -> d[0];
measure b[1] -> d[1];
measure b[2] -> d[2];
measure b[3] -> d[3];
measure b[4] -> d[4];
measure b[5] -> d[5];
```

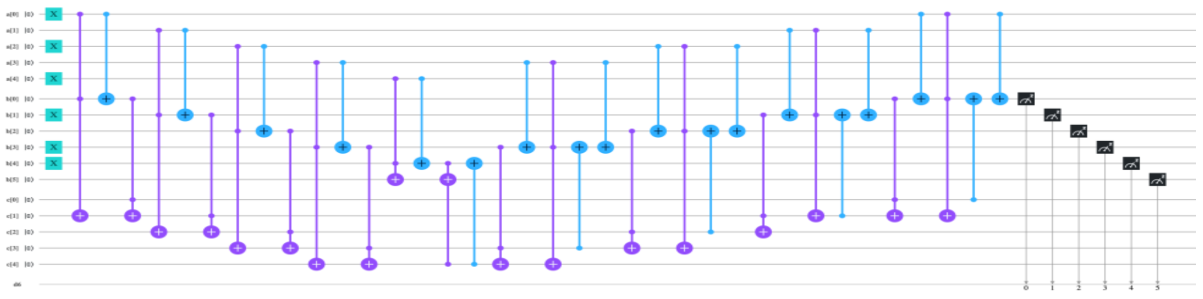
The code in the second column (blocks with five statements of gates *cx* and *ccx*) runs in a very similar way. The final statements (*measure)* read the qubit values and pass them to the classical register, *d.* As it is seen, working at bit level is hard and fault prone: the programmer may be specially tempted to copy and paste for changing later the indexes of the qubits (and forgetting it), she/he may use a wrong gate, write statements in a different order…
All these possible faults are suitable of being reproduced with mutation operators.

4- Model-based testing

In model-based testing (MBT), the test engineer models test cases with models, typically using UML. The UML specification includes the UML Testing Profile, an extension whose metamodel defines all the elements involved in tests definition (Figure 5).



Given a SUT described with a set of UML models, the tester may describe test cases using a model based on this profile. Then, she/he can get executable test cases, for example, with an automatic model-to-text transformation [8].
Figure 6 shows the quantum circuit corresponding to the program code of Figure 4: it includes a horizontal line for each qubit, as well as, in vertical, the corresponding representation of each operation executed. Actually, a circuit is a quite faithful model of the program, since each variable (the qubits) and program statement are completely represented in the circuit.
Then, also quantum testing may take advantage of classic MBT techniques for proposing novel strategies that lead to generate test cases from circuits. It should be researched, but maybe it is possible to tailor the UML testing profile to this new programming paradigm.


a[0] |0⟩
a[1] |0⟩
a[2] |0⟩
a[3] |0⟩
a[4] |0⟩
b[0] |0⟩
b[1] |0⟩
b[2] |0⟩
b[3] |0⟩
b[4] |0⟩
b[5] |0⟩
c[0] |0⟩
c[1] |0⟩
c[2] |0⟩
c[3] |0⟩
c[4] |0⟩
c6
X
0
1
2
3
4
5