Testing Quantum Programs using Q# and Microsoft Quantum Development Kit

Mariia Mykhailova[a] and Mathias Soeken[a]

[a] *Microsoft Quantum, Redmond, United States*

Abstract
As the quantum software matures, the quantum codebases grow both in size and in complexity, and so does the task of testing them and verifying their correctness.
In this paper we show how to test and validate several common types of quantum programs written in the quantum programming language Q# using the tools provided by the Microsoft Quantum Development Kit. Our approach uses multiple simulators and library tools for program testing and resource estimation, performing these steps before running the programs on quantum hardware. The demonstrated approach is illustrated with code examples from the Quantum Katas, a collection of quantum programming tutorials that provide immediate feedback for the learner's solution.

Keywords 1
Quantum programming, Quantum software engineering, Quantum software testing, Q#, Microsoft Quantum Development Kit

1. Introduction

Quantum program validation is an important branch of quantum computing and quantum programming. As quantum software matures, quantum codebases grow both in size and in complexity, and so does the task of testing them and verifying their correctness. The quantum software industry should take advantage of the lessons given by classical software engineering and incorporate program testing early in the quantum software development process. Ideally development of unit tests and end-to-end tests should happen in parallel with the development of the main quantum program, following a test-driven development process.

Testing quantum programs presents a new challenge compared to testing classical software, which arises from the fundamental differences between quantum and classical computing. A variety of approaches have been proposed, from full-state simulation of the programs [1] to formal verification [2] and interactive theorem provers [3].

In this paper we show how to test and validate several common types of quantum programs written in the quantum programming language Q# [4] using the tools provided by the Microsoft Quantum Development Kit [5]. Our approach uses multiple simulators and library tools to perform testing and resource estimation before running the programs on quantum hardware.

This paper is structured as follows: Section 2 offers an overview of the general quantum software development workflow. Section 3 introduces Microsoft Quantum Development Kit, which offers the tools for each step of quantum software development. Section 4 briefly covers the Quantum Katas [6], the collection of programming tutorials and exercises that use unit testing extensively to provide immediate feedback to the learner. Section 5 dives into several common types of quantum programs and the ways to implement unit tests for them using the QDK, using the code snippets from the Quantum Katas as an illustration.

Proceedings of the 2nd Quantum Software and Engineering Workshop (QSET'21), October 19, 2021, Virtual conference
EMAIL: mamykhai@microsoft.com (A. 1); mathias.soeken@microsoft.com (A. 2).
ORCID: 0000-0002-1700-4687 (A. 1)

2. General quantum software development workflow

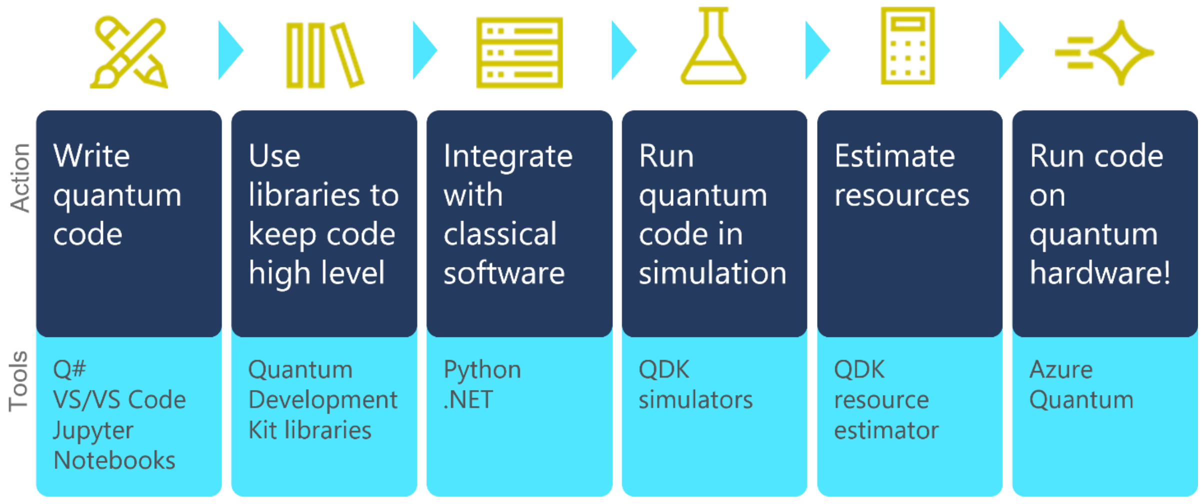
Quantum software development for a reasonably complex quantum application typically goes through the following stages:

1. **Identifying the algorithm to be applied and implementing its quantum portion.** At this stage the developer should take advantage of high-level features of the chosen programming toolkit to focus on the high-level logic of the code, rather than on low-level implementation details.
2. **Integrating it with the classical portion of the algorithm.** Quantum algorithms are typically used as parts of hybrid workflows that combine them with classical libraries and tools, using each type of tools to solve the problems for which they are best suited. For example, quantum chemistry tools rely on classical chemistry simulation packages such as NWChem to formulate the inputs to the quantum portion of the algorithm [7].
3. **Running quantum code in simulation.** Quantum simulators – classical programs that simulate certain aspects of behavior of a quantum system – allow the developer to run their quantum programs to solve a small instance of the problem, and to test their correctness.
4. **Estimating the required resources.** This stage allows to estimate the hardware resources (such as the number of qubits and the circuit depth) required to execute the quantum program for a larger instance of the problem to figure out whether hardware execution is possible, or to optimize the algorithm even if it does not fit within the specifications of the current hardware.
5. **Running the program on quantum hardware.**

Ideally the programming toolkit used for quantum software development should provide tools for all stages of this workflow to enable smooth transitions and to reduce the number of errors introduced between stages.

3. Microsoft Quantum Development Kit overview

Microsoft's Quantum Development Kit (QDK) is a set of open-source tools designed for quantum software development at scale. Let's take a quick look at its components and their matching to the workflow we've seen in Section 2.



Action
Tools
Write quantum code
Q#
VS/VS Code
Jupyter Notebooks
Use libraries to keep code high level
Quantum Development Kit libraries
Integrate with classical software
Python
.NET
Run quantum code in simulation
QDK simulators
Estimate resources
QDK resource estimator
Run code on quantum hardware!
Azure Quantum

Figure 1: Quantum software engineering workflow and QDK tools supporting each step

The most recognizable element of the QDK is Q#, a high-level quantum programming language [4]. Q# is a domain-specific programming language, meaning that it designed specifically for expressing quantum programs: it lacks a lot of functionality of general-purpose programming languages, such as file system access or database access, but it natively implements a lot of quantum programming patterns, such as qubit management, automatic generation of adjoint and controlled variants of quantum operations, repeat-until-success loops, conjugations, and others. A lot of other tasks common in quantum programming languages, such as gate synthesis, auxiliary qubit management, and quantum-specific optimizations, are performed by the compiler automatically without the developer's involvement.

The Q# programming language is augmented with the QDK libraries, which include the standard libraries, implementing patterns that are common for quantum programs, and the domain-specific libraries, offering higher level algorithms such as chemistry and numerics. The standard libraries offer a variety of tools, from small convenience routines such as ControlledOnInt function to larger building blocks such as state preparation, quantum Fourier transform, and phase estimation. They also include tools for developing unit tests, which we'll discuss in more detail in Section 5.

Q# programs can be easily integrated with Python and .NET host programs, which allows developers to take advantage of all classical libraries and tools developed in these ecosystems in the past.

The QDK includes a variety of simulators, from the full-state simulator that imitates a quantum system perfectly to the Toffoli simulator convenient for working with reversible computations and the recently introduced experimental simulators designed to simulate noisy systems.

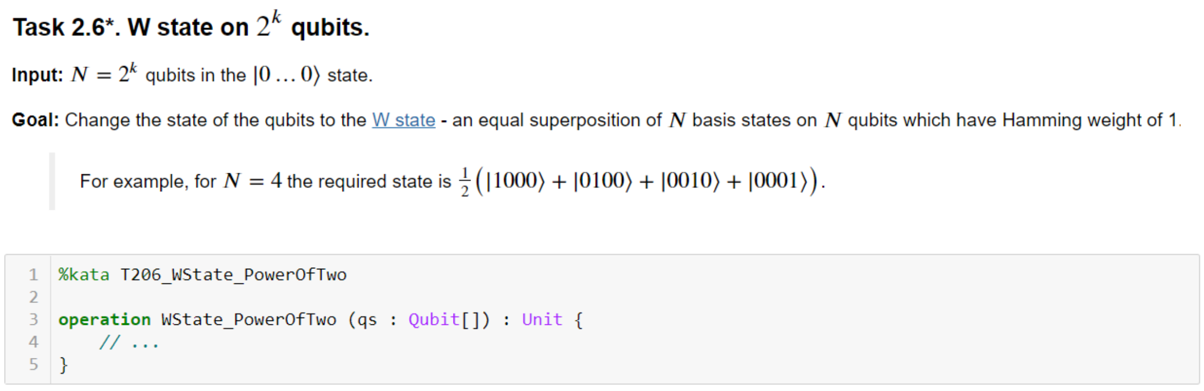
Two of the simulators shipped as part of the QDK, resources estimator and trace simulator, perform the task of resources estimation and, for the latter, several program validation tasks which can be performed without full program simulation, such as checks for the program applying operations to qubits that have already been released.

Finally, the QDK can be used to submit jobs to quantum hardware via Azure Quantum.

All tools in the Microsoft Quantum Development Kit use the same Q# code for all steps of the workflow.

4. The Quantum Katas: programming tutorials with feedback

The Quantum Katas [6] are an open-source project which aims to aid learning quantum computing and Q# programming. Each kata is a sequence of programming tasks which cover one topic (e.g., quantum measurements) or several related topics (e.g., quantum oracles and Grover's search algorithm). The tasks in the katas are purely practical; each task describes a problem and asks the learner to write a fragment of Q# code that would implement the solution.



Task 2.6*. W state on 2^k qubits.

Input: $N = 2^k$ qubits in the $|0\dots0\rangle$ state.

Goal: Change the state of the qubits to the W state - an equal superposition of N basis states on N qubits which have Hamming weight of 1.

> For example, for $N = 4$ the required state is $\frac{1}{2}\big(|1000\rangle + |0100\rangle + |0010\rangle + |0001\rangle\big)$.

```
%kata T206_WState_PowerOfTwo

operation WState_PowerOfTwo (qs : Qubit[]) : Unit {
    // ...
}
```

Figure 2: An example of state preparation task from the Quantum Katas

The key element of the Quantum Katas is immediate automated feedback that they provide, enabling effective self-paced learning even without access to other sources of feedback (such as an instructor). Each kata includes a testing framework which validates the tasks' solutions as soon as they are written. The testing framework is implemented as a series of unit tests, one per task. Each unit test uses the full-state quantum simulator included in the QDK and the techniques described in Section 5 to simulate a quantum program that sets up the inputs required by the task, runs the solution, and processes the results. This enables the learner to solve the katas on a classical computer without access to quantum hardware and to get fast and reliable feedback on their solutions.

5. Testing quantum programs with the QDK

Our approach to testing is based on several assumptions and requirements defined by the nature of the Quantum Katas project:

1. Each code fragment that needs to be tested ("solution") performs a task for which we have a "reference solution" – the known correct implementation of this task. The unit tests don't always rely on it, but often comparing the solution with the reference solution is the easiest way to implement a test.
2. The solution should be evaluated based on whether what it does matches the task description, rather than on the exact path it takes to accomplish the task. Consider, for example, the following task: given a pair of qubits in the $|00\rangle$ state, prepare one of the Bell states $|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. There are multiple sequences of gates that will do this, but all of them are acceptable, as long the learner's solution ended up with the qubits in the right state.
3. The tests need to be written completely in Q#. Q# code can be easily integrated with .NET or Python, allowing for unit tests that combine quantum code with arbitrary classical processing, but the testing framework used by the Quantum Katas requires the tests to be written in Q# alone.

5.1. Writing unit tests in Q#

A Q# unit test is an arbitrary Q# operation or function that takes no arguments, returns Unit (i.e., has no return), and is marked with @Test attribute which defines the simulator used to run this unit test. Here is an example of a unit test which verifies that the function `ClassicalIdentity` implements the classical function $f(x) = x$ which takes an integer between 0 and 1 as an input. The `Fact` library function checks that the Boolean value passed as the first argument is true and throws an exception if it's not.

```
namespace Tests {
    open Microsoft.Quantum.Diagnostics;

    @Test("QuantumSimulator")
    function TestClassicalIdentity() : Unit {
        for i in 0 .. 1 {
            Fact(ClassicalIdentity(i) == i,
                $"Incorrect function return for input {i}");
        }
    }
}
```

You can read more about creating Q# test projects in the Q# documentation[?].

5.2. Testing state preparation routines

The task of preparing the Bell state $|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ is an example of a state preparation task, which is typically formulated as follows: given a qubit or several qubits in the $|0\rangle$ state, prepare the specified state on them.

Q# doesn't allow the program to have direct access to the amplitudes of the quantum state vector, since that doesn't match the physical reality of a quantum-mechanical system. Instead, to test this task, we can use the full state simulator provided by the QDK and the `AssertAllZero` library operation from the Microsoft.Quantum.Diagnostics namespace which does nothing if all qubits of the given array are in the $|0\rangle$ state and fails otherwise. Since the task asks the learner to prepare a non-zero state $|\Phi^-\rangle$, rather than the $|00\rangle$ state, we'll need to use one more tool: the "reference solution" – a unitary transformation R which is known to transform the state $|00\rangle$ into the state $|\Phi^-\rangle$:

$$R|00\rangle = |\Phi^-\rangle$$

We need to check whether the learner's solution U transforms the state $|00\rangle$ into the state $|\Phi^-\rangle$ too:

$$U|00\rangle = |\Phi^-\rangle$$

We can rewrite this check by applying adjoint of the reference solution $R^\dagger$ to both sides:

$$R^\dagger U|00\rangle = R^\dagger|\Phi^-\rangle = R^\dagger R|00\rangle = |00\rangle$$

We see that if we start with the $|00\rangle$ state and apply first the learner's solution and then the adjoint of the reference solution, we'll end up with the $|00\rangle$ state again if and only if the learner's solution did indeed prepare the $|\Phi^-\rangle$ state. Importantly, the results of this check don't depend on the way the solution prepared the required state; it doesn't even have to be a unitary transformation for the check to work.

The code for this test will look as follows:

```
@Test("QuantumSimulator")
operation TestPrepareBellStatePhiMinus () : Unit {
    use qs = Qubit[2];
    PrepareBellStatePhiMinus(qs);
    Adjoint PrepareBellStatePhiMinus_Reference(qs);
    AssertAllZero(qs);
}
```

The reference solution can be implemented specifically for the task, or it can use a library operation, such as `PrepareArbitraryStateD` operation from the Microsoft.Quantum.Preparation namespace.

5.3. Testing implementation of a general unitary transformation

Another common type of tasks are tasks which ask the learner to implement a unitary transformation. This can include unitary transformations of a special form, such as quantum oracles, but generally the unitary transformations can have arbitrary form.

To test this type of tasks, we can compare it to the reference implementation using the Choi–Jamiołkowski isomorphism [8] [9] to reduce the comparison of two unitaries to a comparison of a quantum state to the all-zeros state, which we've seen in the section 5.2. Let's look at how it works for the case of single-qubit unitaries (multi-qubit unitaries follow the same principle).

1. We start by preparing a state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$
2. Apply the learner's solution that implements the unitary U to the second qubit:
$$\frac{1}{\sqrt{2}}(|0\rangle \otimes U|0\rangle + |1\rangle \otimes U|1\rangle)$$
This state carries all information about the effects of the unitary on all basis states, and thus, all the information about the unitary itself (up to a global phase).
3. Now apply the adjoint of the reference implementation R to the second qubit:
$$\frac{1}{\sqrt{2}}\left(|0\rangle \otimes R^\dagger U|0\rangle + |1\rangle \otimes R^\dagger U|1\rangle\right)$$
If the unitaries R and U are the same, their effects on the state are going to cancel out for both basis states $|0\rangle$ and $|1\rangle$, and we'll get the state $|\Phi^+\rangle$ with which we started. If the unitaries are different, their effects either on the $|0\rangle$ state or on the $|1\rangle$ state will not cancel out, and we'll end up with a different state.
4. To check whether we ended up with the original state $|\Phi^+\rangle$, we can use the trick from the previous section: apply the adjoint of the routine used to prepare it from the $|0\rangle$ state and compare the result to the $|0\rangle$ state.

The `AssertOperationsEqualReferenced` library operation from the Microsoft.Quantum.Preparation namespace implements this logic for comparing unitaries acting on arbitrary number of qubits:

```
@Test("QuantumSimulator")
operation TestCompoundGate () : Unit {
    AssertOperationsEqualReferenced(3, CompoundGate, CompoundGate_Reference);
}
```

5.4. Verifying the limits on the resources used by an operation

Sometimes the unit tests need to validate not only *what* the code does, but also *how* it does that. This is particularly important for projects such as the Quantum Katas, which need to check that the learner's solution not only arrives to the right answer, but also that it doesn't use any unintended shortcuts in the process. Such checks can include restricting the number of qubits used by the solution, the number of certain operations or measurements, or more complicated conditions such as the number of any operations acting on three or more qubits.

For example, during the testing of the Deutsch-Jozsa algorithm kata prototype one of the testers implemented the classical solution to the problem solved by the Deutsch-Jozsa algorithm (figuring out whether the given function is constant or balanced), applying the given quantum oracle to each of the basis states in turn to evaluate the function in a classical manner, rather than the Deutsch-Jozsa algorithm itself. The learner can be confused by the testing harness accepting such a solution, so the test had to be augmented with the extra check that ensured that the given quantum oracle was used exactly once.

As another example, learners who are getting started with reversible computation are often tempted to implement a quantum oracle for a classical function by measuring the input state, computing the value of the function for this input classically, and setting the state of the output qubit to this value. If the unit test for this kind of problems compares the effects of the quantum oracle on each basis state to the output of the classical function, it needs to additionally check that the implementation didn't use any measurements.

There are multiple approaches to verifying this kind of restrictions. For example, you could use the QDK resources estimator to create a separate test, written in C# or in Python, which would ignore all logical checks and focus on resource constraint verification alone.

The Quantum Katas use two approaches that allow to incorporate the resource constraint verification into the test code which checks the logical correctness of the solution, so that all testing code for each task is contained in a single unit test.

The first approach relies on two library operations from the Microsoft.Quantum.Diagnostics namespace, `AllowAtMostNCallsCA` and `AllowAtMostNQubits`. These operations assert that the Q# code between a call to this operation and a matching call to its adjoint uses at most the given number of calls to the given operation or allocates at most the given number of qubits, respectively. It is convenient to use them with the conjugation construct within … apply that is a part of Q# core language.

For example, task 1.2 of the Oracles tutorial asks the learner to implement a phase oracle for the N-bit function $f(x) = [x = 7]$ without using the phase kickback trick. The task is implemented as a unitary transformation, so the logical part of the test can be done using the approach described in section 5.3. The easiest proxy for the check that the learner's solution doesn't use phase kickback trick is the check that there are no extra qubits allocated on top of the 2N qubits required to use the `AssertOperationsEqualReferenced` library operation. The code for the check will look like this:

```
@Test("QuantumSimulator")
operation TestIsSevenPhaseOracle() : Unit {
    let N = 3;
    within {
        AllowAtMostNQubits(2*N, "You are not allowed to allocate extra qubits");
    } apply {
        // Perform the logic correctness check for the solution
        AssertOperationsEqualReferenced(N, Is7PhaseOracle, Is7PhaseOracle_Reference);
    }
}
```

The second approach involves implementing a custom simulator that would count each operation used by the Q# code and produce those statistics on demand. The Microsoft Quantum Development Kit provides API to extend existing simulators or build custom simulators from scratch for various purposes.

The custom simulator used in the Quantum Katas project CounterSimulator extend the standard full state simulator provided by the QDK with the resources-counting functionality. It tracks three kinds of resources: the maximum qubits allocated by the program, the number of times each individual operation has been called, and the total number of multi-qubit operations used by the program. Unlike the resources estimators included in the QDK, the operation counters allow to obtain the statistics for all operations, not just the primitive gates, which is very useful for higher-level logical checks, such as verifying the restrictions on the number of oracle calls.

This approach allows to implement more sophisticated checks. For example, task 6 of the JointMeasurements kata asks the learner to implement the CNOT gate using joint measurements for the input state of a special form $(\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle$. The test for this task will use the approach described in section 5.2 (since the operation does not have to implement the CNOT gate for all input states, the approach that verifies that the correct state has been prepared is more appropriate than the approach that verifies that the correct unitary has been implemented), but additionally it has to make sure that the solution doesn't use any multi-qubit operations other than the joint measurement operations (`Measure` or `MeasureAllZ`), since otherwise a solution consisting of the CNOT or Controlled X gates would achieve the formal goal without making the educational point.

The unit test will use three of the functions implemented by CounterSimulator: `ResetOracleCallsCount` which resets all counters of the simulator, `GetOracleCallsCount` which returns the number of times the given operation was called since the last reset, and `GetMultiQubitOpCount` which returns the number of times any multi-qubit operation was called since the last reset.

```
@Test("Microsoft.Quantum.Katas.CounterSimulator")
operation TestControlledXViaMeasurements() : Unit {
    // Allocate two qubits and prepare them in a state (α|0⟩ + β|1⟩) ⊗ |0⟩.
    use qs = Qubit[2];
    Ry(2.0 * 0.123, qs[0]);
    // Reset operation calls counters.
    ResetOracleCallsCount();
    // Apply the operation that needs to be tested.
    ControlledX(qs);
    // Get the number of multi-qubit operations that are not measurements
    // using CounterSimulator functionality.
    let nMultiQubitNonMeasurementOpCount = GetMultiQubitOpCount()
        - GetOracleCallsCount(Measure) - GetOracleCallsCount(MeasureAllZ);
    Fact(nMultiQubitNonMeasurementOpCount <= 1,
        "You are not allowed to use multi-qubit gates in this task.");
    // Continue the logical testing following the approach from section 5.1.
    // ...
}
```

You can find the complete CounterSimulator code in the Quantum Katas repository [10]. This approach can be customized to fit the exact needs of the unit test being implemented; for example, the functionality of counting the multi-qubit operations used by the program has been implemented to test the tasks of the JointMeasurements kata alone.

6. References

[1] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in Int'l Conf. on High Performance Computing, Networking, Storage and Analysis, 2017.

[2] M. Amy, "Towards Large-scale Functional Verification of Universal Quantum Circuits," in Quantum Physics and Logic, 2018.

[3] R. Rand, J. Paykin and S. Zdancewic, "QWIRE Practice: Formal Verification of Quantum Circuits in Coq," in Quantum Physics and Logic, 2017.

[4] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz and M. Roetteler, "Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL," in Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018). https://doi.org/10.1145/3183895.3183901.

[5] Microsoft, Q# and the Quantum Development Kit, 2021. URL: http://aka.ms/qdk.

[6] M. Mykhailova, "The Quantum Katas: Learning Quantum Computing using Programming Exercises," in {ACM} Technical Symposium on Computer Science Education, 2020.

[7] G. H. Low, N. P. Bauman, C. E. Granade, B. Peng, N. Wiebe, E. J. Bylaska, D. Wecker, S. Krishnamoorthy, M. Roetteler, K. Kowalski, M. Troyer and N. A. Baker, "Q# and NWChem: Tools for Scalable Quantum Chemistry on Quantum Computers," arXiv preprint arXiv:1904.01131, 2019.

[8] M.-D. Choi, "Completely positive linear maps on complex matrices," Linear Algebra and its Applications, vol. 10, p. 285–290, 1975.
[9] A. Jamiołkowski, "Linear transformations which preserve trace and positive semidefiniteness of operators," Reports on Mathematical Physics, vol. 3, p. 275–278, 1972.
[10] Microsoft, 2021. URL: https://github.com/microsoft/QuantumKatas/blob/main/utilities/Common/CounterSimulator.cs