*Chapter 1*

# COMPUTER-AIDED DESIGN FOR NEXT-GENERATION QUANTUM COMPUTING SYSTEMS

*Alireza Shafaei , Mohammad Javad Dousti*, and Massoud Pedram*
Department of Electrical Engineering,
University of Southern California, Los Angeles, CA

**PACS** 05.45-a, 52.35.Mw, 96.50.Fm.

**Keywords:** Physical Design, Quantum Computing, Compiler, Fault-tolerant Quantum Computation, Quantum Mapper.

**Abstract**

A computer-aided design (CAD) flow for large-scale quantum circuits is presented in this chapter. The proposed flow along with other stand-alone tools are implemented in a tool suite called Next-generation Quantum Computing Systems (NQCS) Toolbox, which receives as input a physical machine description (PMD), a quantum error correction (QEC) code, a quantum control (QC) protocol, as well as a quantum algorithm, and produces a machine control language (MCL) code and physical resource requirements. NQCS Toolbox comprises of several stand-alone tools to perform required tasks, and interactions between these tasks are achieved through well-defined interfaces. This modular design approach simplifies the task of developing and ensuring the correctness of the overall design flow. By taking advantage of hierarchical designs in quantum circuits, NQCS is able to efficiently handle large-scale quantum benchmarks.

*E-mail address: dousti@usc.edu

# 1. Introduction

Quantum computing offers the potential of dramatically better performance on certain problems compared to classical computing. As a result, substantial research effort has been invested in discovering high-level algorithms that could operate on a quantum computer [1–4], and in developing a wide variety of physical technologies that might make the construction of a quantum computer possible [5]. However, less attention has been paid to the *computer-aided design* (CAD) aspects of quantum computing.

One way to view the implementation of a quantum algorithm on a realistic quantum computer is as a quantum physics experiment that is under the control of a classical computer. The experimental apparatus consists of several parts. The *quantum core* contains the physical qubits whose quantum-mechanical properties are used to accomplish the computation. Classical control and detection systems serve to initialize, manipulate, and read out the state of the quantum core. A real-time classical computer directs the quantum experiment by issuing a sequence of instructions to the control and readout electronics, to be performed on the quantum core. The instruction sequence is generated from a high-level *quantum program* by an off-line compilation process. Similarly, result verification and output post-processing are also done off-line.

A realistic, non-ideal quantum computer is subject to *noise* and faces numerous limitations and constraints. For example, the quantum core components are sensitive to environmental disturbances and to errors in the control and detection systems. If ignored, either of these effects can rapidly lead to computational error. The rate at which errors occur limits the length of computations that can be carried out. In addition, the technology used to implement the quantum computer will be subject to design constraints on parallelism, geometry, bandwidth, etc., that further narrow the options for implementing ideal quantum algorithms.

There is strong theoretical evidence that arbitrarily long quantum computations could be carried out on realistic quantum computers if the error rates per gate and per time step could be reduced below a level known as the "fault-tolerant accuracy threshold" [6]. This process, known as *fault-tolerant quantum computation*, relies in part on the ability of *quantum error-correcting codes* to protect the quantum information used in the computation from random disturbances.

Using currently known techniques, implementation of fault-tolerant quantum computation will require massive computational resources. This work thus proposes a CAD flow for quantum information processing that receives as input a *physical machine description* (PMD), a *quantum error correction* (QEC) code, a *quantum control* (QC) protocol, as well as a quantum algorithm, and produces different artifacts including but not limited to a *machine control language* (MCL) code, and physical resource requirements. The proposed CAD flow is called Next-generation Quantum Computing Systems (NQCS). This flow along with other stand-alone tools are implemented in a tool suite which will be referred to as the NQCS Toolbox in the rest of the chapter.

NQCS Toolbox comprises of several stand-alone tools to perform required tasks. These tools interact with each other through well-defined interfaces. This feature allows the tool developers to work on individual tools independently, and any new tool can be simply added to the whole flow in the future. Moreover, NQCS users have the flexibility of augmenting

the interface files when and if needed. To finish, this modular design approach simplifies the task of ensuring the correctness of the overall design flow.

As a proof of concept, NQCS Toolbox uses *Scaffold Compiler* [7] as the quantum compilation tool to generate HF-QASM of the given quantum algorithm, and QUFD [8] to provide the latency and MCL of fault-tolerant gates for the given PMD, QEC, and QC. Furthermore, Squash 2 [9] is adopted as the quantum physical design tool which translates the HF-QASM to the final circuit MCL, and reports the final physical resources.

The remainder of this chapter is organized as follows. A list of key definitions required for the rest of this chapter are provided in Section 2. NQCS requirements are explicitly described in Section 3. Section 4 analyzes functional requirements and provides detailed discussions on the key components in the design flow. Interface specifications are presented in Section 5. Verification methods and performance measures are explained in Section 6 and Section 7, respectively. Finally, recommendations for further reading are presented in Section 8.

## 2. Key Definitions

The following terms have been used throughout this chapter.

**Physical Machine Description (PMD)** A description for the underlying quantum physical machine technology, which include the qubit de-coherence super-operator, control response functions, inter-qubit connectivity, geometric constraints (e.g., nearest neighbor interactions), qubit movement constraints, crosstalk, and detector parameters.

**Quantum Control (QC)** A protocol to control the quantum dynamics of the quantum hardware.

**Native instruction** A quantum gate that belongs to the instruction set of a PMD. (e.g., $R_x$, $R_y$, and $R_z$, which are equal to the rotation of the quantum bit around the X-, Y-, and Z-axes by an arbitrary angle).

**Basic operations** Quantum gates that are used in fault-tolerant quantum computation. The standard universal set of the basic operations includes H, T, CNOT, S, X, Y, and Z gates. Quantum circuits are built out of basic operations. However, to realize a given quantum circuit in the given PMD, each basic operation should be implemented by using native instructions (e.g., $T = R_z(\pi/4)$ and $H = R_y(\pi/2)R_z(\pi)$, up to a global phase).

**Quantum Gate Set (QGS)** A universal set of quantum gates to be used in a fault tolerant quantum computation. The standard universal set includes T, S, H, X, Y, Z, and CNOT. Basic single-qubit measurements of X and Z are also included in the QGS.

**Primitive cells** PMD of a quantum circuit fabric consists of a 2D array of identical *primitive cells* (or cell for short). Each cell contains some sites for generating/initializing qubits, reading them out, performing operations on one or two physical qubits, and resources for moving information about qubits.

**Tile** Dealing with the 2D array of primitive cells is very cumbersome and unwieldy. So in practice we build another 2D array of super-cells (which we call *tiles*). Each tile comprises of an $n \times n$ array of primitive cells. We can thus deal with mapping a quantum algorithm to this tiled architecture. An example of a $2 \times 2$ tiled architecture where each tile consists of a $4 \times 4$ cell is shown in Figure 7. The layout consists of tiles where each tile stores a single logical qubit and ancillae to enable all single-qubit logical gates. To execute a CNOT gate (as a two-qubit operation), target and control logical qubits should initially be collocated.

**Universal Logic Block (ULB)** ULB is analogous to a Configurable Logic Block in an FPGA device, which means it can implement any of a set of target functions.

**Computing Universal Logic Block (CULB)** A ULB that is capable of performing one- and two-input fault-tolerant gate operations on one or two logical qubits. It can also perform quantum error correction on a single logical qubit.

**Memory Universal Logic Block (MULB)** A ULB that can store several idle logical qubits. It periodically performs QEC on the logical qubits (one after another) in order to maintain their fidelity.

**Scaffold** A high-level programing language that is used to describe a quantum algorithm. More information about the Scaffold and its associated compiler can be found in [10] ans [7], respectively.

**Quantum Assembly Language (QASM)** A gate-level programming language for describing a quantum circuit [11].

**Hierarchical Fault-tolerant Quantum Assembly Language (HF-QASM)** An intermediate representation between the front-end and back-end parts of the NQCS Toolbox. This representation is a variant of QASM with hierarchy added and gate types restricted to fault-tolerant gates.

**Extensible Markup Language (XML)** XML is a simple and powerful language, which has been used extensively to design textual databases. It is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design rules are specified with XML schema files. There are many XML development resources and solutions, XML parsers, etc., which are quite helpful for us.

**Mini languages** A set of XML-based specifications that describe the PMD, the QC protocol, and the Quantum Error Correction (QEC) code. To describe the encoding/decoding circuity of the QEC code, a quantum circuit description language (e.g. QASM) is also required.

**Machine description and protocol libraries** After processing Mini languages, the machine description and protocol libraries are created, which include: (i) the realization of each gate in QGS using native instructions of the given PMD by considering the QC protocol, and (ii) encoding/decoding circuits for the given QEC code.

**Quantum Instruction Dependency Graph (QIDG)** Graph representation of a quantum circuit, in which nodes represent (assembly) instructions and edges capture data dependency between operands of the instructions. This graph is used in scheduling, placement and routing steps of the quantum physical designer.

**Quantum Universal logic block Factory Designer (QUFD)** A tool designed by NQCS team that generates a library of MCL codes for fault-tolerant quantum gates.

**Quantum Physical Designer (QPD)** Main tool of the back-end which generates MCL of the quantum circuit through scheduling, placement, and routing.

**Machine Control Language (MCL)** A low-level textual language which encodes simple control instructions that would be delivered to the classical real-time computer. The set of instructions is therefore necessarily tightly coupled to the physical machine description. The classical computer would then translate the machine control language instructions into technology-dependent commands (e.g., lasers, detectors, magnets, RF-signal generators, etc) for execution by the specific hardware components that control the qubits and quantum gates. The machine control language may include items such as qubit movement primitives, readout primitives, control primitives, qubit addressing, and pulse shapes. The MCL is analogous to the classical Instruction Set Architecture (ISA).

**Resource Calculation (RC)** A major task in the Toolbox that calculates the physical resources (i.e., the number of physical qubits, the number of physical gate operations, and circuit delay) needed to implement a quantum algorithm.

**Multiple-Control Toffoli (MCT)** A Toffoli gate with multiple controls.

**Enhanced Functional Flow Block Diagram (EFFBD)** An EFFBD represents the behavior for a system or components of a system. The purpose of the EFFBD is to indicate the sequential relationship of all functions that must be accomplished by a system. EFFBDs show the same tasks identified through functional decomposition and display them in their logical, sequential relationship at different levels of a hierarchy.

**Satisfiability (SAT)** In computer science, SAT is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. SAT was the first known example of an NP-complete problem.

## 3. NQCS Toolbox Requirements

The NQCS Toolbox requirements can be divided into functional and interface requirements as discussed below.

### 3.1. Functional Requirements

The NQCS Toolbox provides the following functional requirements:

1. The NQCS Toolbox allows the quantum programmer to write the quantum algorithm in Scaffold language providing quantum data types, quantum operations, classical operations, and appropriate control flow structures. Scaffold is discussed in [10].

2. The NQCS Toolbox enables

   (a) PMD expert to describe a new PMD architecture, along with error rates and latencies for various quantum instructions,

   (b) QEC expert to describe a new quantum error correction code, and

   (c) QC expert to describe a new quantum control protocol.

   The above items are provided by the use of Mini languages.

3. The NQCS Toolbox provides an efficient quantum compiler for the quantum programmer. Quantum compiler is discussed in [7].

4. The NQCS Toolbox generates the machine description language (i.e., MCL code) for the quantum programmer given the Scaffold program, QC, QEC code, and PMD specification. MCL generation is discussed in Section 4.5.

5. The NQCS Toolbox includes standalone tools for calculating physical resources for the given Scaffold program, QC protocol, QEC code, and PMD specification. Resource calculation is discussed in Section 4.6.

6. The NQCS Toolbox provides a set of software utilities such as a timing simulator and the NQCS quantum programming environment. Timing simulator is discussed in Section 4.5.1.

7. The NQCS Toolbox supports a mechanism to analyze the correctness of algorithm implementations in Scaffold. Verification is discussed in Section 6.

### 3.2. Interface requirements

The NQCS Toolbox provides the following interface requirements:

1. The input to the whole design flow is a quantum algorithm written in Scaffold [10].

2. The input to the front-end tool suite is a program in Scaffold, and the output is HF-QASM.

3. The input to the quantum physical design (back-end) is HF-QASM, and the output is MCL code.

4. Inputs to the resource calculation tool are HF-QASM as well as error rate and delay information extracted from protocol libraries, and the output is physical resources (# of physical qubits, # of physical gates and their types, # of clock cycles).

5. The input to the Mini language processor is a set of XML and QASM files for describing (PMD, QC, QEC), and the output is protocol libraries.

6. The input to the library designer is a set of protocol libraries, and outputs are MCLs for gates, tile design and the 2D title architecture.

7. The NQCS Toolbox provides an integrated development environment, in the sense that a quantum algorithm written by a quantum programmer in Scaffold will be transformed into the final MCL after applying several automated tools including a quantum compiler, a synthesizer, etc.

Interface requirements are verified by monitoring and validating (as much as possible) inputs and outputs of the front-end and back-end tools, resource calculator, Mini language processor, and library designer.

Based on the above interface requirements, the following stand-alone tools are available in the NQCS Toolbox:

- Quantum compiler
- Mini language processor
- Reversible logic synthesizer
- Physical designer
- Verifier(s)
- Integrated Development Environmnet (IDE) and visualizers

# 4. Functional Requirements Analysis and Specification

Functional analysis is the process of identifying, describing, and relating the functions a system must perform in order to fulfill its goals and objectives. Functional analysis is logically structured as a top-down hierarchical decomposition of those functions. Several techniques are available to do functional analysis. The primary functional analysis technique is the *functional flow block diagram* technique. These diagrams show the network of actions that lead to the fulfillment of a function. This section uses Enhanced Functional Flow Block Diagram (EFFBD) representation, which is based on [12, Appendix F], to depict functional decomposition of the NQCS Toolbox.

## 4.1. Enhanced Functional Flow Block Diagram (EFFBD)

The purpose of the EFFBD is to indicate the sequential relationship of all functions that must be accomplished by a system. EFFBDs show the same tasks identified through functional decomposition and display them in their logical, sequential relationship at different levels of a hierarchy. For example, the entire tool chain flow of a quantum circuit designer can be defined in a top level EFFBD, as shown in Figure 1. Each block in the first level diagram can then be expanded to a series of functions, as shown in the second level diagrams in Figure 3 to Figure 13. EFFBDs use the following symbols and notation:

- Functions are represented as rectangular boxes (blocks) with an associated label number. Functional (control) flow is shown by head-flat arrows, whereas data flow overlay to capture data dependencies is represented by head-filled arrows and elongated ovals.

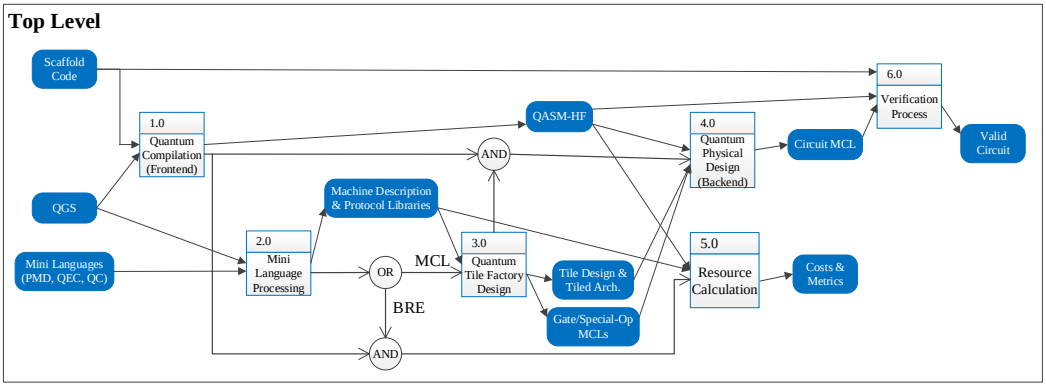


Figure 1. EFFBD of NQCS top level. For EFFBD notation please refer to Section 4.1.

- Each function receives a unique label number that can be used as identification, and is shown in the top-left corner of the function. The numbering scheme establishes identification and relationships that carry through all the diagrams and facilitate traceability from the lower levels to the top level.
- Lines connecting functions indicate functional flow and not lapsed time or intermediate activity.
- Diagrams are laid out so that the flow direction is generally from left to right. The diagrams show both input (e.g., "Mini language Processing") and output (e.g., "Quantum Physical Design"), thus facilitating the definition of interfaces and control processes.
- Each diagram contains a reference to other functional diagrams to facilitate movement between pages of the diagrams.
- Gates are also used, including "AND", "OR", "Go or No-Go", sometimes with enhanced functionality, including exclusive OR gate (XOR), iteration (IT), repetition (RP), or loop (LP).
- A circle is used to denote a summing gate when AND/OR is present. As expected, AND indicates parallel functions and all conditions must be satisfied to proceed (i.e., concurrency) whereas OR indicates that alternative paths can be satisfied to proceed (i.e., selection).
- "Go or No-Go" decision blocks are shown by a diamond. The "Go" path from the decision block is executed if the condition is met; otherwise, the "No-Go" path is executed. These symbols are placed adjacent to lines leaving a particular function to indicate alternative paths.

An EFFBD specification of a system is complete enough that it is executable as a discrete event model, capable of dynamic as well as static validation. EFFBDs provide freedom to use either control constructs or data triggers or both to specify execution conditions for the system functions.

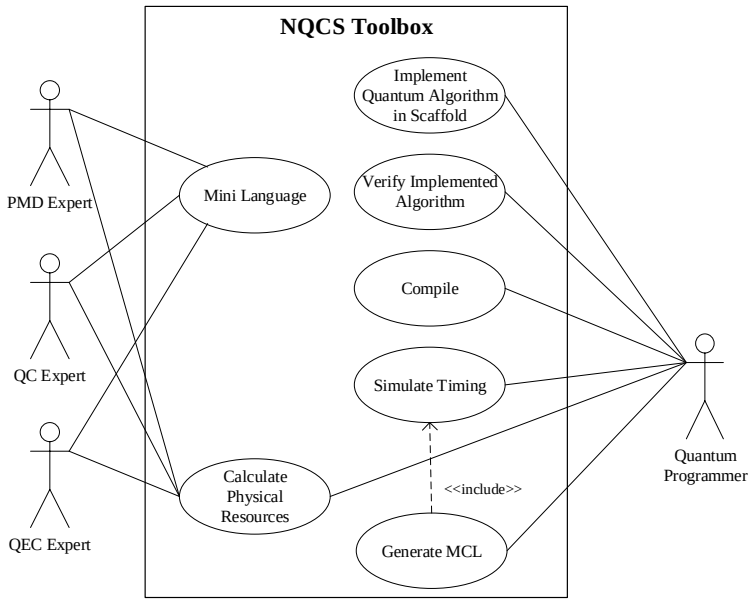


Figure 2. Use-case diagram of the NQCS Toolbox.

## 4.2. The NQCS Toolbox

The top- (first-) level diagram of the NQCS Toolbox is depicted in Figure 1, which is then expanded to second-level diagrams in Figure 3 through Figure 13. NQCS, at the top-level view, performs the following two major tasks:

- **Design automation** for a quantum compiler, which maps a quantum algorithm into an MCL code, and
- **Resource calculation**, which reports the physical resources that are sufficient to implement a quantum algorithm on a target PMD.

These tasks consider QEC to protect qubits against noise, and QC to increase the fidelity. As a result, a combination of quantum algorithm, PMD, QEC code, and QC protocol acts as the input to each task. More accurately, the primary inputs to NQCS are classified as follows:

- A quantum algorithm written in Scaffold [10].
- Mini languages, which are a set of XML-based specifications that describe the PMD, the QC protocol, and the QEC code.
- A universal set of quantum gates to be used in a fault tolerant quantum computation (previously referred to as basic operations.) This universal set is referred to as QGS and includes T, S, H, X, Y, Z, and CNOT [6]. For the given QEC code, the exact implementation of each of the gates in the QGS is given as a QASM file. We also include basic single-qubit measurements of X and Z in the QGS, since they are a necessary part of a fault-tolerant quantum computing scheme. In addition to

the standards gates, new gates can also be added to this set as long as an efficient, fault-tolerant implementation for the new gates exists in the given QEC code, and implementation costs are properly considered in the synthesis step.

To perform quantum computation fault-tolerantly, quantum gates should be chosen from the QGS to build quantum circuits. To this end, NQCS adopts the following approach (cf. Figure 1):

1. "1.0 Quantum Compilation (Front-end)" in Figure 1, receives Scaffold code and the QGS, compiles the code, and synthesizes it to a circuit consisting of gates from QGS. This intermediate representation is called HF-QASM.

2. The HF-QASM representation is then translated by "4.0 Quantum Physical Design (Back-end)" to a physical quantum circuit language, i.e., the MCL description. However, prior to the physical design, an MCL code for each gate in the QGS is generated as follows:

   (a) After processing Mini languages (cf. function 2.0 in Figure 1), the *machine description and protocol libraries* are created, which include: (i) the realization of each gate in QGS using native instructions of the given PMD by considering the QC protocol, and (ii) encoding/decoding circuits for the given QEC code.

   (b) "3.0 Quantum Library Design" uses the data set generated in (a) in order to produce the internal design of a tile, the 2D tiled-architecture of the PMD, and MCL codes for QGS members and also the *special operations* (or, special-ops for short). Special-ops are defined in this chapter as the *hop operation* (i.e., one step move operation in up, down, left or right directions), and *syndrome extraction* for the QEC code.

3. In addition to the back-end, which generates the MCL code, "5.0 Resource Calculation" computes the *costs and metrics* for the quantum algorithm that include the number of physical qubits, number of physical gate operations, and the circuit delay (i.e., number of physical time steps to execute the quantum algorithm).

4. Verification shows proof of compliance with the Toolbox requirements where validation shows that Toolbox accomplishes its major tasks. "6.0 Verification Process" handles these cases and will be discussed in Section 6.

Based on the above discussion, it can be seen that the NQCS uses a *meet-in-the-middle approach* to generate the MCL code. On one hand, the top-down development in the front-end tool suite breaks the Scaffold code into gates that are chosen from the QGS. On the other hand, the bottom-up development in the library designer tool generates optimized MCL codes (in terms of gate or qubit counts) for each gate of the QGS. These two paths meet each other in the back-end, where high-level gates are scheduled, placed and routed on the 2D tiled architecture, and then each high-level gate as well as move operation are replaced by the appropriate MCL code in order to build the circuit MCL. Note that the path that goes through Mini language processing and quantum library design tools is an offline process that executes independently of the quantum algorithm.

A key benefit of the meet-in-the-middle approach is that the efficiency of the top-down (forward) synthesis approach and the accuracy of the bottom-up physical design approach are simultaneously realized. Additionally, the design tool scalability is ensured by hiding from the algorithm and code developers the physical design details of implementing logic operations (and moves) in a given PMD for given QC and QEC protocols. Ease of verification of the design by separating front-end transformations from gate library verification is another benefit.

The NQCS toolbox relies on a set of well-defined *use-cases*, whose purpose is to capture system functions that affect the interface between the system and the outside world. Well-written use-cases offer the analysis, development, testing, and documentation teams an invaluable guidebook. Precisely, a use-case is a formalized story that describes how someone procedurally interacts with a proposed software system. In the rest of this section, we develop use-case diagrams for various components in the NQCS Toolbox. These components are those that an expert quantum programmer/researcher can utilize in order to write a given quantum algorithm and subsequently transform the written program into a machine-level code for a given combination of PMD, QEC, and QC. The rest of this section provides detailed descriptions for functions of the top-level diagram along with the appropriate use-cases.

## 4.3. Front-end

A given quantum algorithm is written in Scaffold [10]. The specification of Scaffold allows two kinds of modules for describing a quantum algorithm: (1) *normal modules* which are written with quantum variables and gates, along with classical or quantum control structures, and (2) *Classical-to-Quantum-Gate* (Classical code To Quantum Gate (CTQG)) modules. CTQG modules are a key feature of Scaffold that allows a programmer to express a desired quantum functionality using classical programming. Rather than viewing everything in terms of quantum gates, CTQG modules allow programmers to describe the functionality of some parts of the algorithm from a higher perspective. They are written as classical C-like code, with the intent of being synthesized to reversible logical circuits that are compatible with a quantum circuit. Normal and CTQG modules are suitable for specifying the quantum and oracle parts of the algorithm, respectively. These two parts are distinguishable by the Scaffold grammar. In addition to the quantum and classical modules, there are various control structures that allow the programmer to identify the control flow of Scaffold code. More details are available in Table 1.

After parsing the Scaffold code (cf. function 1.1 of Figure 3), function 1.2 of Figure 3 first analyzes the Scaffold code, identifies normal and CTQG modules, and forwards them along with the required control structures through the appropriate path in the design flow to the corresponding compilation/synthesis tools. CTQG modules are synthesized by a *reversible logic synthesis* tool (cf. function 1.4 of Figure 3), e.g., Reed-Muller Decision Diagram Synthesis (RMDDS) [13]. The normal modules are fed into the *quantum compilation* tool (cf. function 1.3 of Figure 3), which executes in parallel to the reversible synthesis path. The tool is responsible for synthesizing quantum logic gates into QGS. In the first step (not shown in the figure), the quantum compiler unrolls the loops, evaluates conditions, and interprets the gate sequence and control path described by the programmer. In

Table 1. The detailed description for the "Implement Quantum Algorithm in Scaffold" use-case .

| Use-case name | Implement Quantum Algorithm in Scaffold |
|---|---|
| Related requirements | Requirement 1 in Section 3 |
| Goal in context | The NQCS Toolbox provides an integrated development environment to specify a quantum algorithm and subsequently transform and optimize the quantum program into the machine-level code. Scaffold is the NQCS quantum programming language. |
| Preconditions | Must have a detailed specification of the algorithm, specifying the different parts in quantum circuit or high-level (e.g., oracle, etc.) flow. |
| Successful end conditions | A new Scaffold program is developed. |
| Fail end conditions | - |
| Primary actors | Quantum Programmer |
| Secondary actors | - |
| Trigger | The quantum programmer starts writing the Scaffold program. |
| Main flow | Action |
| | (Step #1) Quantum Programmer studies the algorithm in detail.<br>(Step #2) Quantum Programmer decides on which parts to specify using classical code and which parts to write at quantum gate level.<br>(Step #3) Quantum Programmer uses the available language features and syntax to implement the specifications.<br>(Step #4) Quantum Programmer must be aware of the decision's implications and possible implementation costs (e.g., large non-reversible logics that use a lot of ancilla to be made reversible, arbitrary rotations, etc.) |

the second step (again not shown), the quantum compiler implements arbitrary rotations in terms of QGS gates by using the Solovay-Kitaev algorithm [14]. In this case, a sequence of machine instructions must be generated to approximate the arbitrary rotation to the required precision. Analysis in [15] shows that static code generation can, in some cases, lead to a terabyte of code to realize needed rotations. Further, some rotation angles are unknown until the runtime, requiring dynamic code generation. Dynamic code generation, however, can result in significant tradeoffs in terms of the execution time overhead versus code quality (code size and rotation precision). Moreover, dynamic code generation will be performed on classical (non-quantum) computing resources, which may or may not have a clock speed advantage over the target quantum technology.[1] In [15], the design space formed by these trade-offs of dynamic versus static code generation is explored. They also introduced several techniques to provide smoother trade-offs for dynamic code generation and evaluate the viability of options in the design space. Note that this step produces complex quantum gates such as multiple-control Toffoli gate. The output of the second step is Hierarchical

[1]For example, although modern classical processors run at gigahertz clock speeds, operations on trapped ions run at kilohertz speeds, whereas superconducting qubits run at gigahertz speeds.

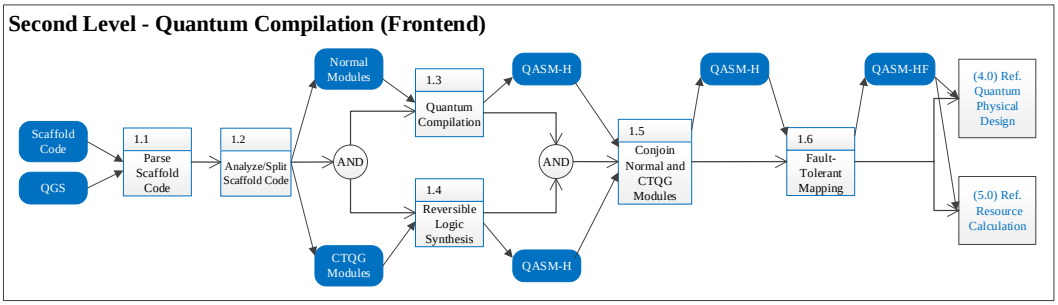


Figure 3. EFFBD of NQCS front-end.

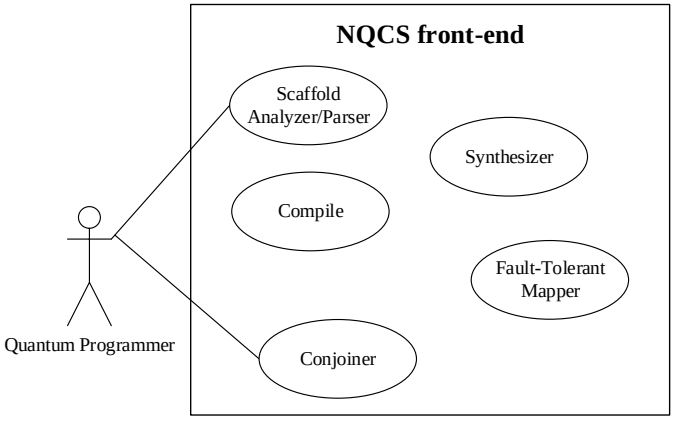


Figure 4. NQCS front-end use-case diagram.

Quantum Assembly Language (H-QASM) (see Section 5).

Compiled normal and CTQG modules are merged together by a conjoiner tool (cf. function 1.5 in Figure 3). The *conjoiner* is responsible for putting the codes together and resolving hybrid, nested instantiations, viz., instantiation of a classical module inside a quantum. To manage the size of output, the conjoiner avoids flattening the output and maintains the hierarchical structure (instantiation of modules inside the other modules). Circuit optimizations, similar to [16] and [17], may also be performed by the conjoiner tool to further optimize the quantum circuit. Finally, a *fault-tolerant mapping* tool (cf. function 1.6 in Figure 3) takes the output of the conjoiner, and maps it to gates in the QGS. The output of the fault-tolerant mapper is the HF-QASM representation. Based on the above discussion, the frond-end use-case diagram is shown in Figure 4, and the front-end actions are summarized in Table 2.

## 4.4. Library Designer

The purpose of the library designer is to design the internal structure of a tile and the 2D tiled architecture, and to generate a library of gate models.[2] The library designer comprises

[2]This model includes MCL code, delay, and error rate for each QGS, move operation, and MCL code for applying syndrome extraction.

Table 2. The detailed description for the "Compile" use-case .

| Use-case name | Compile |
|---|---|
| Related requirements | Requirement 3 in Section 3 |
| Goal in context | The Scaffold program is compiled into HF-QASM representation. |
| Preconditions | The Scaffold program should pass the verification step. |
| Successful end conditions | A new HF-QASM is generated. |
| Fail end conditions | Compiler generates a list of errors including the location and type of the error. |
| Primary actors | Quantum Programmer |
| Secondary actors | - |
| Trigger | The quantum programmer asks the NQCS to compile the Scaffold program. |
| Main flow | Action |
| | (Step #1) The Quantum Programmer asks the NQCS Toolbox to compile the Scaffold program.<br>(Step #2) The Scaffold program is parsed, and split into normal and CTQG modules (cf. function 1.2 in Figure 3).<br>(Step #3) CTQG modules are synthesized to H-QASM using a reversible logic synthesis tool.<br>(Step #4) Normal modules are compiled into H-QASM using the Quantum Compiler.<br>(Step #5) CTQG and normal H-QASM files are conjoined to produce the final H-QASM representation.<br>(Step #6) The H-QASM representation is transformed into the HF-QASM representation.<br>(Step #7) HF-QASM serves as input to other tools including the quantum physical design tool or the resource calculation tool. |

of two stand-alone tools:

1. The *Mini language processor*, shown in Figure 5: The input to the Mini language processor is a set of XML-based "Mini languages", which describe the PMD, the QC protocol, and the QEC code. The Mini Language processor produces a QASM description file for each gate of the QGS using the native instructions of the PMD.

2. The *quantum tile factory designer*, shown in Figure 6: This tool implements each gate of the QGS in a fault-tolerant manner, attaches the syndrome extraction circuit to the output(s) of the gate, and places the resultant quantum circuit into the architectural tile. The MCL code for the QGS gate being fully realized in the tile is then generated.

After appropriate high-level transformations, the quantum circuit is represented as a QIDG (function 3.1 in Figure 6), in which nodes represent (assembly) instructions and edges capture data dependency between operands of the instructions. To capture the quantum no-cloning theorem, which forbids fan-out in quantum circuits, a pre-processing step is implemented on the QIDG to resolve the multiple read dependencies between instructions

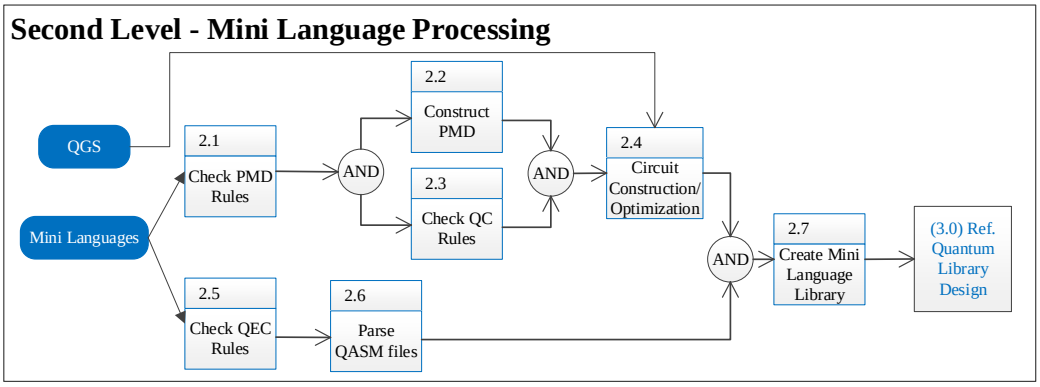


Figure 5. EFFBD of NQCS Mini language processor.

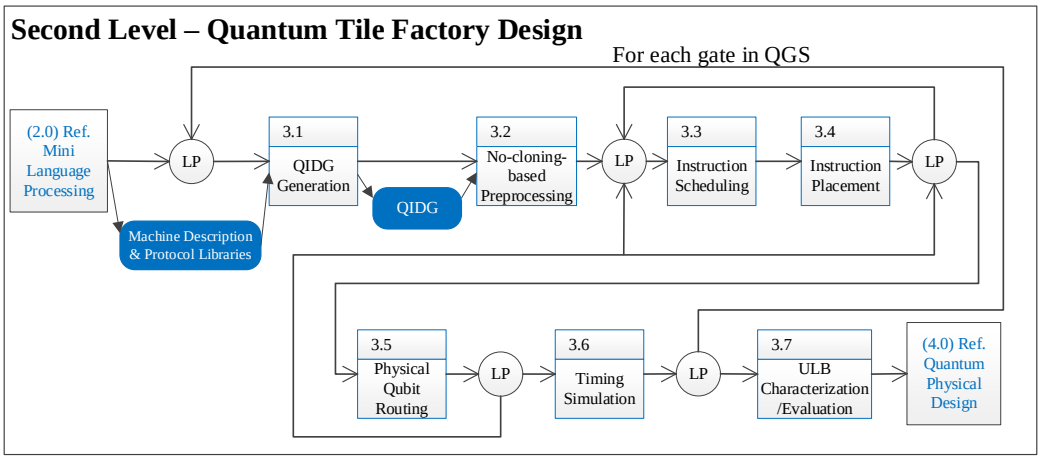


Figure 6. EFFBD of NQCS quantum tile factory designer. LP denotes a loop in the EFFBD.

(function 3.2 in Figure 6). In the next step, the instructions are scheduled (function 3.3 in Figure 6) and placed (function 3.4 in Figure 6) on the quantum fabric, and the qubits are routed (function 3.5 in Figure 6). Scheduling, placement, and routing problems in the context of quantum computing are defined as:

- **Scheduling**: Given a QIDG, the scheduling problem is to reduce the resource pressure (e.g., the number of concurrent instructions) while minimizing the total latency of the circuit.
- **Placement**: Given a total ordering of the instructions on the circuit (as the output of the instruction scheduling step), the placement problem is to assign physical (logical) qubits and instructions (operations) to the cells (tiles) such that the communication time (or total latency of the computation) is minimized.
- **Routing**: Given the position of each qubit and the instruction schedule, the qubit routing problem is to minimize the total routing time.

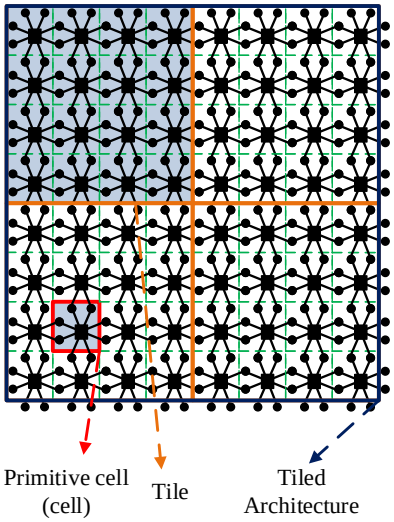


Figure 7. A $2 \times 2$ tile-based architecture in superconducting PMD, where each tile is a $4 \times 4$ cell.

In the last step, the optimal size of the tile is determined such that various fault-tolerant realizations of required quantum instructions can be implemented in any tile in the fabric with acceptable latency and without any resource waste. We refer to this tile as Universal Logic Block (ULB). When this optimal tile size is known, the tool generates the MCL code for each gate of the QGS. For circuits coded with block QEC, the proposed algorithms have been implemented as a quantum Computer Aided Design (CAD) tool called Quantum Universal logic block Factory Designer (QUFD) [8]. Timing simulation (cf. function 3.6 of Figure 6) is used in QUFD tool to verify the generated MCL code.

The quantum tile factory designer is responsible for designing two types of high-level (architectural) ULBs.

- **Computing Universal Logic Block (CULB)**: This tile is capable of performing fault-tolerant gate operations on one or two logical qubits. It can also perform quantum error correction on a single logical qubit. The type of error correction is specified as an input by the QEC code.

- **Memory Universal Logic Block (MULB)**: This tile can store several idle logical qubits. It periodically performs QEC on the logical qubits (one after another) in order to maintain their fidelity. This is based on the fact that idle qubits need to be refreshed approximately once every 100 operation cycles in concatenated codes [18]. Therefore, applying QEC on idle qubits in a round-robin manner is adequate.

To achieve a regular tiled architecture for the front-end tool suite, width and height of the MULB should be the same as those of the CULB. Hence, the number of idle logical qubits that the MULB can store is a function of CULB size, idle qubit de-coherence rate, and the delay of applying a QEC on a logical qubit. An example of the tiled architecture for the superconducting PMD is shown in Figure 7.

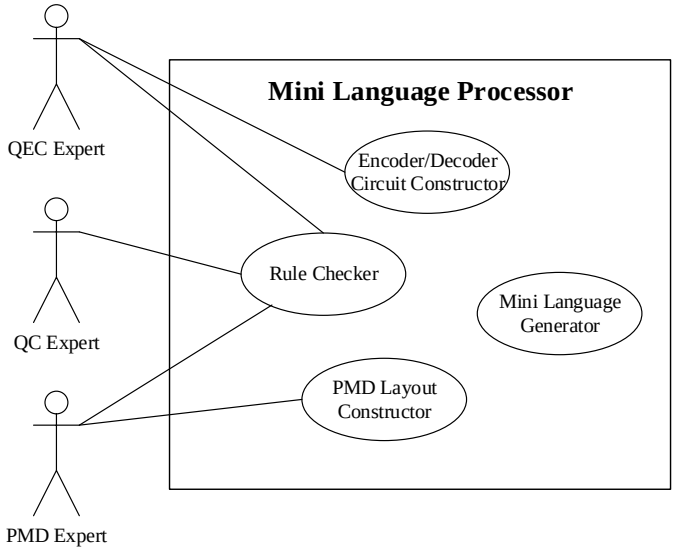


Figure 8. NQCS Mini language processor use-case diagram.

Table 3. The detailed description for the "Mini language" use-case.

| **Use-case name** | **Mini language** |
|---|---|
| Related requirements | Requirement 2 in Section 3 |
| Goal in context | A new PMD, QC protocol, or QEC code is described by the PMD expert, QC expert, or QEC expert respectively |
| Preconditions | All XML and QASM files are available, and are specified correctly. |
| Successful end conditions | A new PMD, QC protocol, or QEC code is integrated into NQCS. |
| Fail end conditions | The PMD, QC protocol, or QECC specification is rejected. |
| Primary actors | PMD expert, QC expert, or QEC expert |
| Secondary actors | - |
| Trigger | The PMD expert, QC expert, or QEC expert asks the NQCS to process the new PMD specification, QC protocol, or QEC code, respectively. |

### 4.4.1. Mini language processor

Specifying PMD, QC and QEC in a simple language format provides the ability to modify/extend the existing PMD, QC and QEC protocols. It also provides the flexibility to capture new PMDs, QC, or QEC protocols even after the design of the quantum Toolbox. Considering these objectives, we need to devise a descriptive language for PMD, QC and QEC to be used in this process. The designed language should be easy to use by the PMD, QC and QEC developers (Figure 8 and Table 3). Yet, it should be expressive, concise, and robust. For these reasons, we have opted to use the XML language to specify the PMD, QC and QEC protocols. In addition to XML, some type of quantum circuit description language is needed to specify the required QEC protocols viz. to provide a description of the encoding and decoding circuitry. QASM or Scaffold may be used for this purpose.

The definition of a PMD is based on primitive cells that are specific to that PMD. The quantum fabric can be generated by replicating these primitive cells and considering links between the neighboring cells. In addition to these layout definitions, the set of native instructions that can be implemented in each of the PMDs is specified in an XML file. For the QC definition language, native instructions of each PMD are replaced with a series of native and control instructions to increase their fidelity. These replacements are defined using an XML file. To capture different QEC schemes (block vs. topological), we suggest different definition languages for different QEC schemes. These definition languages include the basic parameters specific to each coding scheme. In addition to these parameters, circuit implementation of basic operations, ancilla generation and parity check for different coding schemes are also specified using augmented QASM files. Moreover, the NQCS Toolbox considers two different paths for applying concatenated and topological QEC to a quantum circuit.[3] For EFFBD of the NQCS Mini language processor refer to Figure 5.

### 4.4.2. Circuit construction/optimization

A *circuit construction/optimization* tool [16] (cf. function 2.4 in Figure 5, and expansion in Figure 9) is used in the Mini language processor. This tool finds optimized quantum circuit realizations for compound gates (i.e., gates that require more than one native instruction to be realized in the given PMD). The optimization objective may be the number of native instructions or the execution time for the compound gate operation. The tool relies on a *default Operation-to-Instruction (O2I) mapping tool*, or *O2I mapper* for short, (function 2.4.1 in Figure 9) to compute and store in an O2I library the optimal realizations of each basic operation in terms of native instructions in the given PMD. Optimal realization means either the minimum number of native instructions or the shortest execution time. The native instruction set for this tool are: (i) One-qubit operations ($R_x$, $R_y$, $R_z$, $X$, $Y$, $Z$, $S$, $T$, $H$), and (ii) two-qubit operations (CNOT, CZ, Geometric, Controlled-Phase, Swap, iSwap). After mapping the circuit into native instructions that are supported by the PMD, some adjacent instructions may be redundant. Thus, the tool optimizes the circuit by using quantum identity rules (function 2.4.2 in Figure 9) until no more optimizations are possible. Figure 10 shows the construction of SWAP gate in various PMDs, obtained by the circuit construction/optimization tool [16].

## 4.5. Back-end

The back-end tool suite comprises of a Quantum Physical Designer (QPD) tool, which is shown in Figure 11 with its use-case model in Figure 12. The inputs to the QPD are the HF-QASM representation, which is the output of the front-end tool suite, and tile design/tiled architecture along with gate models, which are outputs of the library designer.

[3]The proposed mini-language processor can easily handle any modifications to the encoding/decoding circuitry for a supported QEC - regardless of whether it is concatenated or topological code. On the other hand, handling a new (and previously unseen) is a very challenging task - something that requires further research in order to assess its feasibility and if so, ways of achieving it. We anticipate that a totally new QEC will require changes to the NQCS toolbox that go beyond the mini-language processor and touch on other aspects of the toolbox including the front- and back-end tools in potentially significant ways.

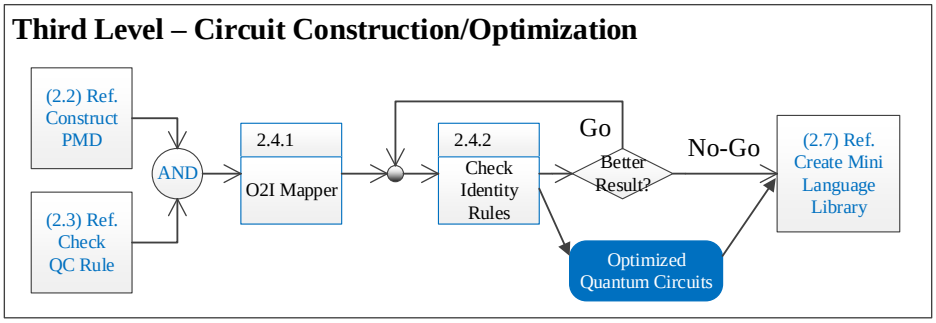


Figure 9. NQCS circuit constructor/optimizer.

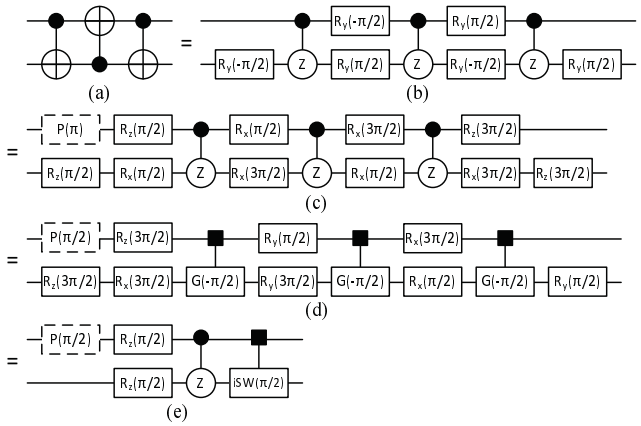


Figure 10. SWAP gate construction in (a) photonics, (b) neutral atom, (c) quantum dot, (d) trapped ion, and (e) superconducting PMDs [16].

The HF-QASM representation is translated into a QIDG, which is subsequently used during scheduling, placement, and routing steps (see next paragraph). After scheduling, placement, and routing steps, a new HF-QASM is generated, which includes information about the operation locations (the placement coordinates of the tile in the quantum fabric where each operation takes place) as well as qubit movements (the steps that specify how logical qubits are routed between tiles). By substituting fault-tolerant quantum gates and move operations with respective MCL codes, this HF-QASM is translated into the circuit MCL code. Applying syndrome extraction to this circuit based on operation, movement, and stall error rates is also the responsibility of back-end when generating the final circuit MCL code.

Scheduling, placement, and routing problems for the back-end are defined in the same way as those for the library designer. However, the back-end deals with logical qubits and operations, as opposed to the library designer that deals with physical qubits and operations. The library designer receives the physical realization of a basic operation, schedules and places native instructions on the basic cells of the quantum fabric, and routes physical qubits between these cells in order to generate the MCL code for the given operation. On the other hand, the input to the back-end is a circuit potentially composed of a very large

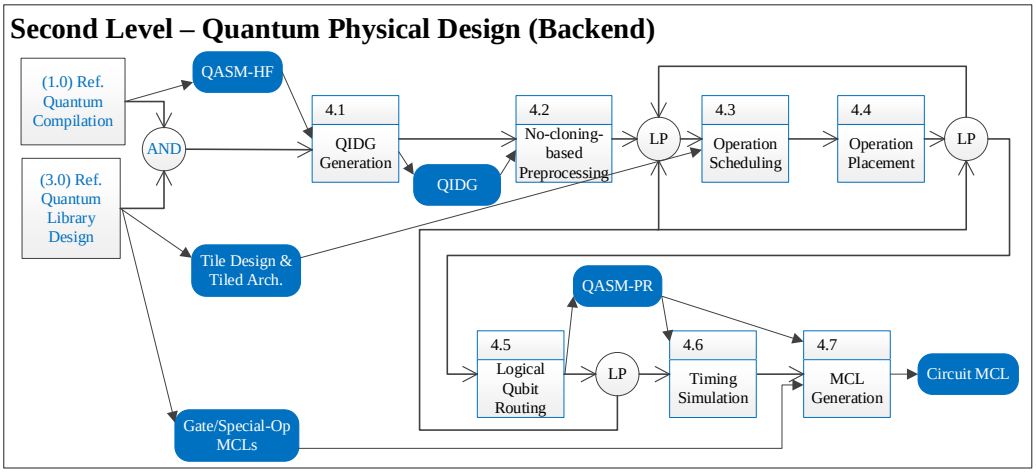


Figure 11. EFFBD of NQCS back-end.

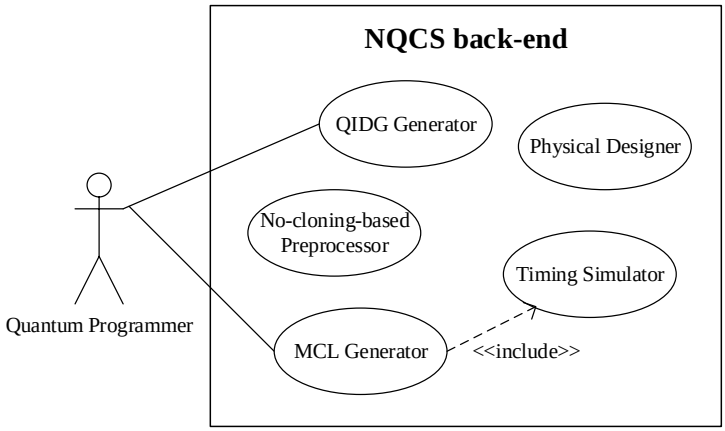


Figure 12. NQCS back-end use-case diagram.

number of basic operations on logical qubits (i.e., the HF-QASM representation). The MCL generation of the circuit in the back-end tool proceeds by scheduling and placing these operations on the 2D tiled architecture, routing logical qubits between tiles, and replacing various operations and moves with respective MCL codes. Another key difference arises from the scale (size) of the input file (problem). While the number of native instructions in a fault-tolerant implementation of a basic operation can vary from hundreds to thousands of instructions, a HF-QASM representation may contain (tens of) millions of operations [19]. To handle this huge input size and generate a scalable back-end, partitioning techniques on QIDG and tiled-architecture are necessary. Using partitioning techniques, the number of operations per partition can be limited to a reasonable number for placement and routing algorithms. The required steps for MCL generation and timing simulation are summarized in Table 4 and Table 5.

Table 4. The detailed description for the "Generate MCL" use-case .

| Use-case name | Generate MCL |
|---|---|
| Related requirements | Requirement 4 in Section 3 |
| Goal in context | Machine Control Language (MCL) is generated. |
| Preconditions | The Scaffold program should be compiled with no error. |
| Successful end conditions | MCL for the quantum algorithm is generated. |
| Fail end conditions | An error message is generated. |
| Primary actors | Quantum Programmer |
| Secondary actors | - |
| Trigger | The quantum programmer asks the NQCS to generate the MCL code. |
| Included Cases | Simulate Timing |
| Main flow | Action |
| | (Step #1) For each QGS operation as well as special-ops an MCL is generated.<br>(Step #2) [include::Simulate Timing] Timing of each MCL is verified by timing simulation.<br>(Step #3) The HF-QASM representation is translated into MCL code based on a 2D tiled architecture and MCL codes from step 1.<br>(Step #4) [include::Simulate Timing] Timing of the MCL code is verified by timing simulation. |

### 4.5.1. Timing Simulation

After generating the MCL code by the back-end tool suite or in the case of constructing MCL code for each fault-tolerant quantum gate by the Quantum Universal logic block Factory Designer (QUFD), scheduling, placement, and routing steps are visualized by a timing simulation tool. The main function of timing simulation is to extract timing information, i.e., starting execution time of each instruction, latency of routing qubits, and the delay of the quantum circuit. As a byproduct, timing simulation may work as a verifier which examines the correctness (and possibly the efficiency considering different metrics) of scheduling, placement, and routing solutions. Table 5 summarizes the description.

## 4.6. Resource Calculation (RC)

The RC function, as shown in Figure 13 with use-case diagram in Figure 14, has been automated in the NQCS Toolbox. After compiling the Scaffold code, the HF-QASM is automatically analyzed to produce the number of basic operations, and the estimated parallelism and communication cost for routing the qubits (cf. function 5.1 in Figure 13). The error rate and delay information for basic operations are also automatically extracted from the machine description and protocol libraries (cf. function 5.2 in Figure 13). The number of gates and the time needed to perform quantum error correction that yields a desired error rate is expressed as a series of recursive functions in the automatic Error Correction Code (ECC) analysis tool (cf. function 5.3 in Figure 13). If the error threshold is met, the

Table 5. The detailed description for the "Simulate Timing" use-case.

| Use-case name | Simulate Timing |
|---|---|
| Related requirements | Requirement 6 in Section 3 |
| Goal in context | Latency of a PMD-based specification (i.e., MCL of a QGS operation, Special-op, or the circuit) is determined. Circuit will be visualized for possible actions from quantum programmer. |
| Preconditions | MCL code is generated successfully. |
| Successful end conditions | Timing information and other related information for the MCL code is displayed. |
| Fail end conditions | An error message is displayed. |
| Primary actors | Quantum Programmer |
| Secondary actors | - |
| Trigger | The quantum programmer asks the NQCS to run the timing simulator. |

*ECC level/distance calculation* tool (cf. function 5.4 in Figure 13) computes the concatenation level (in case of concatenated codes), or the code distance (in case of surface codes) based on (i) the number of basic operations that comes from function 5.1, (ii) the gate error rates that come from function 5.2, and (iii) the functions that come from function 5.3. Finally, a *resource calculation* tool (cf. function 5.5 in Figure 13) combines these outputs, and reports the number of physical qubits, number of physical gate operations, and number of clock cycles required to execute the quantum algorithm. However, in the case that the error rate of a quantum gate (after applying the QC protocol) is higher than the error threshold of the QEC code, the given combination of PMD, QC, and QEC is not allowed (cf. function 5.6 in Figure 13). The required steps for PMD are summarized in Table 6.

As discussed in [19] in detail, a suite of Octave scripts has been provided that automatically generates the resource estimates for a cross product of algorithms, PMDs, and error correction techniques. For each quantum algorithm, the number of logical gates of each type, the parallelization factor for these gates (how many gates of each type can be safely scheduled in parallel), and the number of logical qubits are needed. Another required input is information about each combination of PMD and control protocol. The required information is the time of each gate type, the error of the worst gate, and error rate per unit of time. For more information, please refer to [19].

# 5. Interface Specification

N2 diagram (or N $\times$ N interaction matrix) [12] is used in this section to identify interactions or interfaces between major functions of the NQCS Toolbox from a system perspective. This diagram is typically used to develop system interfaces. The N2 diagram can be taken down into successively lower levels to the component functional levels. In addition to defining the interfaces, the N2 diagram also pinpoints areas where conflicts could arise in interfaces, and highlights input and output dependency assumptions and requirements.

The top-level N2 diagram for the NQCS Toolbox is depicted in Figure 15. In this diagram, various NQCS functions are placed on the diagonal. The remainder of the squares

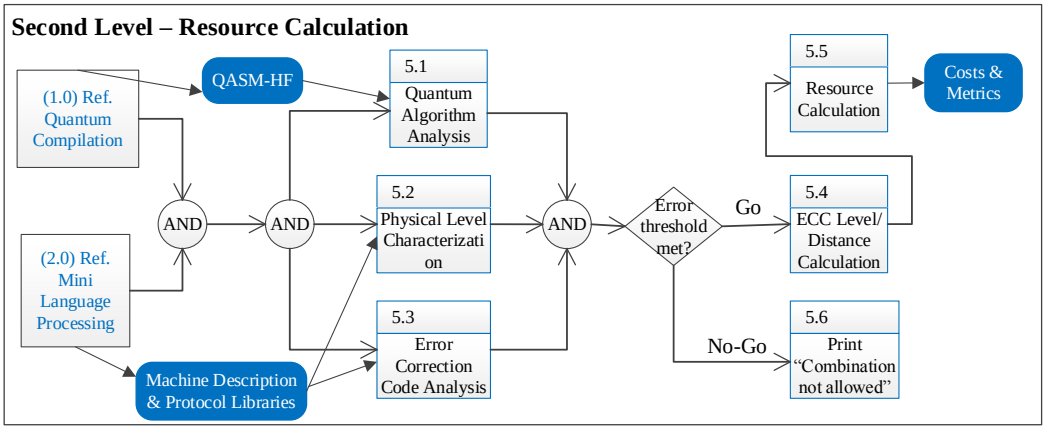


Figure 13. EFFBD of NQCS Resource Calculation (RC).

Table 6. The detailed description for the "Calculate Physical Resources" use-case .

| Use-case name | Calculate Physical Resources |
|---|---|
| Related requirements | Requirement 5 in Section 3 |
| Goal in context | Physical resources for a given HF-QASM, PMD, QC, and QEC are calculated. |
| Preconditions | HF-QASM is generated, and PMD, QC protocol and QEC code are described. |
| Successful end conditions | Physical resource calculation is displayed. |
| Fail end conditions | An error message ("combination not allowed") is displayed. |
| Primary actors | Quantum Programmer |
| Secondary actors | PMD Expert, QC Expert, QEC Expert |
| Trigger | The Quantum Programmer asks the NQCS to launch the resource calculator. |
| **Main flow** | **Action** |
| | (Step #1) Scaffold program is compiled into HF-QASM in order to find the number of basic operations, and the estimated parallelism and communication cost.<br>(Step #2) Error rate and delay information of basic operations are extracted from PMD and QC protocols.<br>(Step #3) Error threshold is extracted from QEC code.<br>(Step #4) If the error threshold is met, concatenation level or the code distance is computed, and then the number of physical qubits, number of physical gate operations, and number of clock cycles required to execute the quantum algorithm are reported.<br>(Step #5) Otherwise, the given combination of PMD, QC, and QEC is not allowed. |

in the N x N matrix represents the interface inputs and outputs. A blue geometric shape at the intersection of a row and a column contains a description of the interface between the two functions represented on that row and that column. A letter "L" ("P") at the intersection

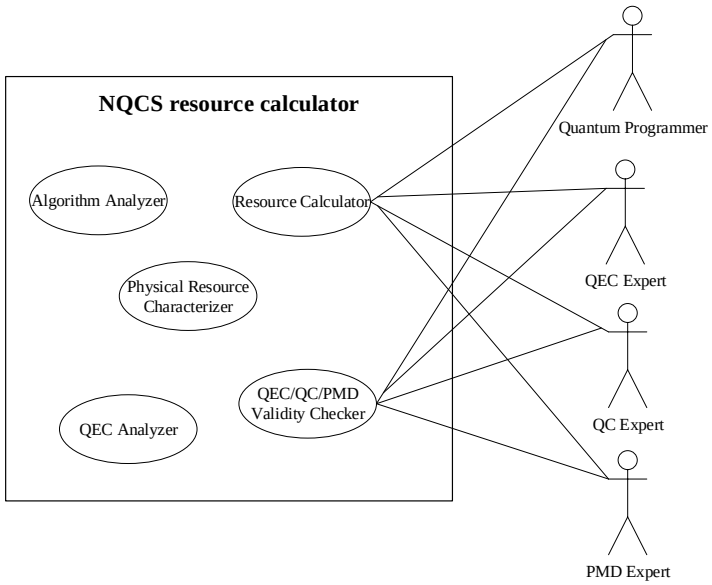


Figure 14. NQCS Resource Calculation (RC) use-case diagram.

of a row and a column indicates an interface that acts on logical (physical) qubits/gates between the two functions. For example, the Fault-Tolerant (FT) mapper has a logical interface, which is called HF-QASM, with the resource calculation tool. Where no shape appears, there is no interface between the respective functions.

The rest of this section explains the main interfaces of the NQCS Toolbox in more details. The interface examples are for demonstration purposes only.

1. **Scaffold code**: This language is developed as a Quantum Programming Language (QPL). The details of Scaffold are presented in a separate document [10]. As a working example, we can consider the Scaffold code shown in Figure 16, which is the cat state preparation circuit.

2. **H-QASM**: This representation is the output of the reversible logic synthesis tool as well as the quantum compiler, and the input to the fault-tolerant mapper. The main difference between H-QASM and the standard QASM [11] is the ability to use hierarchical design, which is important for complex circuits. More precisely, H-QASM adds the following features to the standard QASM representation.

   - An entity called “module” which defines a quantum sub-circuit. The sub-circuit then can be reused several times.
   - Any module or gate can be repeated several times by indicating the “repeat” keyword before the respective module/gate.

   Although modular design simplifies the design process, the aforementioned features are vital to maintain the code size reasonable. Figure 17(b) shows the H-QASM grammar.

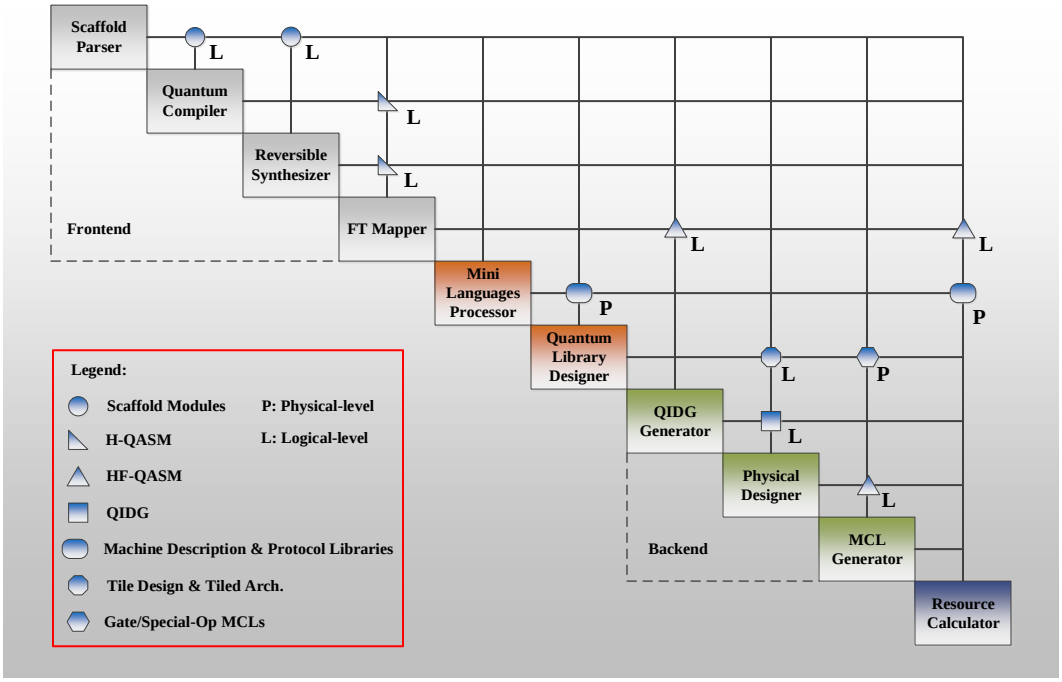


Figure 15. N2 Diagram of NQCS Toolbox.

```
#define N 4

module cat ( qbit *bit, const int n ) {
    H( bit[0] );
    for ( int i=1; i < n; i++ ) {
        CNOT( bit[i], bit[i-1] );
    }
}

int main () {
    qbit bits[N];
    cat( bits, N );
    return 0;
}
```

Figure 16. The Scaffold code for the cat state preparation circuit

3. **HF-QASM representation**: This file is the output of the fault-tolerant mapper. HF-QASM inherits all features of H-QASM; however, it limits the gate sets to only fault-tolerant gates. The grammar of HF-QASM is shown in Figure 17(c) [9]. HF-QASM allows the fault-tolerant mapper to define new modules (or gates) such as Toffoli gates. The resultant file should provide the implementation of all gates using only fault-tolerant gates (i.e., QGS). The HF-QASM code for the cat state preparation circuit generated by the Scaffold compiler is shown in Figure 18.

*name* → [**a**-**z**,**A**-**Z**]+[**a**-**z**,**A**-**Z**,**0**-**9**]*
*num* → [**1**-**9**][**1**-**9**]*
*whitespace* → **\n** | **\r** | **\t** |
*comment* → **#** .* (**\n** | **<EOF>**)

(a)

*start* → (*module*)* *main*
*main* → **module main {** *body* **}**
*module* → **module** *name* **(** *param_list* **) {** *body* **}**
*param_list* → *param* (**,***param*)*
*param* → **qubit** (*****)? *name*
*body* → (*def***;**)+ (*gate***;**)+
*def* → **qubit** ([*num*])? *name*
*gate* → (*one_qubit_gate* | *multi_qubit_gate* | *call*)
*one_qubit_gate* → (**H**|**X**|**Y**|**Z**|**S**|**S**$^\dagger$|**T**|**T**$^\dagger$|**Prep0**|**MeasX**|**MeasY**|**MeasZ**) (*arg***)**
*multi_qubit_gate* → ( **Toffoli (***ctrlArray***,** *arg***)** ) | ( **Fredkin ((***ctrlArray***,**)? *arg***,** *arg***)** )
*call* → *name* **(***call_list***)**
*call_list* → *arg* (**,** *arg*)*
*arg* → *name* | *name***[***num***]**
*ctrlArray* → *arg* (**,** *arg*)*

(b)

*start* → (*module*)* *main*
*main* → **module main {** *body* **}**
*module* → **module** *name* **(** *param_list* **) {** *body* **}**
*param_list* → *param* (**,***param*)*
*param* → **qubit** (*****)? *name*
*body* → (*def***;**)+ (*gate***;**)+
*def* → **qubit** ([*num*])? *name*
*gate* → (*one_qubit_gate* | *two_qubit_gate* | *call*)
*one_qubit_gate* → (**H**|**X**|**Y**|**Z**|**S**|**S**$^\dagger$|**T**|**T**$^\dagger$|**Prep0**|**MeasX**|**MeasY**|**MeasZ**) (*arg***)**
*two_qubit_gate* → **CNOT (***arg***,** *arg***)**
*call* → *name* **(***call_list***)**
*call_list* → *arg* (**,** *arg*)*
*arg* → *name* | *name***[***num***]**

(c)

Figure 17. (a) Definition of tokens. (b) H-QASM grammar. (c) HF-QASM grammar [9]. Regular expression is used to simplify the grammar specification. Note that the star character (*****) used for the *param* variable is a *terminal*. It determines whether a parameter is an array or not (similar to the C language).

4. **QIDG**: After processing the HF-QASM, a *quantum instruction dependency graph* (QIDG) is generated. Instruction dependencies avoid Write After Read (WAR), Write After Write (WAW), and Read After Write (RAW) hazards. In addition, the quantum no-cloning theorem adds another dependency to this graph that avoids multiple reads of a qubit at the same time. This graph is used in scheduling, placement and routing steps of the quantum physical designer.

```
module cat_4 ( qbit* bit ) {
    H ( bit[0] );
    CNOT ( bit[1] , bit[0] );
    CNOT ( bit[2] , bit[1] );
    CNOT ( bit[3] , bit[2] );
 }

module main (  ) {
    qbit bits[4];
    cat_4 ( bits );
 }
```

Figure 18. The HF-QASM code for the cat state preparation circuit

5. **Machine Description and Protocol Libraries**: Mini languages specify the target physical machine description, applied quantum control scheme and quantum error correction code in the quantum circuit. By fast processing of this information, it is possible to produce basic estimates about performance metrics of each gate in the quantum circuit. These estimates contain the number of native instructions needed to implement each gate and rough estimates of the total latency and error probability of their physical implementation. These estimates can be used to derive the baseline resource estimate of the quantum circuit.

6. **Tile Design and Tiled Architecture**: Quantum library designer examines different tile sizes and tile architectures to find the best tile architecture in terms of the quantum circuit performance metrics. The tile architecture specifies the size of the tiles, location of the tiles and topology of the whole fabric. It is used in the back-end for scheduling, placement and routing of logical qubits. Gate models specify the performance metrics, i.e., latency and error probability, of the required gates in the quantum circuit.

7. **MCL code**: Machine control language code specifies the lowest level control commands for the target quantum circuit.

## 6. Verification

Verification of a product shows proof of compliance with requirements — that the product can meet each shall statement as proven through performance of a test, analysis, inspection, or demonstration. More precisely, verification is answering to the following question: Does the Toolbox perform what it claims to do correctly?[4]. To manage the complexity and ensure scalability, we have opted to break the verification process of the NQCS Toolbox into smaller verification processes which includes “verification of algorithm to QPL”, “verification of Toolbox design requirements”, and “verification of QPL to MCL”. The difference

[4]On the other hand, validation of a product shows that the product accomplishes the *intended purpose* in the intended environment — that the product meets the expectations of the customer and other stakeholders as shown through performance of a test, analysis, inspection, or demonstration. Validation is answering to the following question: Is the designer claiming that the Toolbox does the right things?

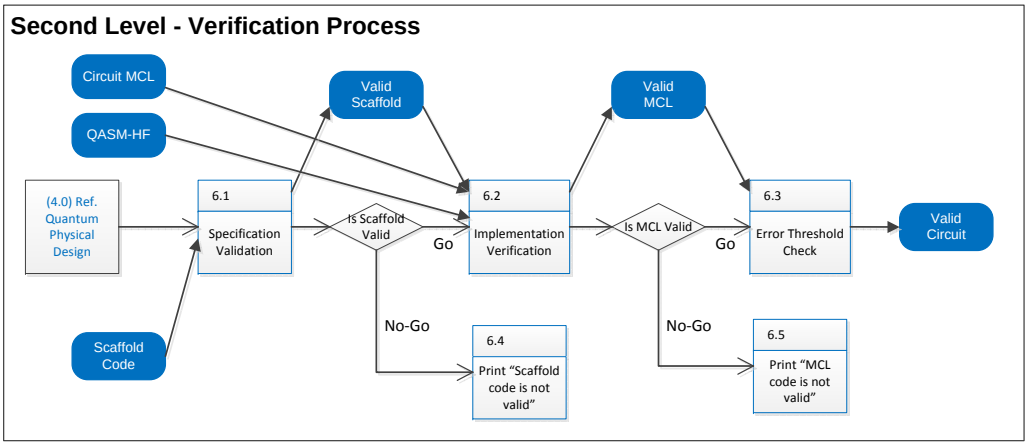


Figure 19. EFFBD of NQCS verification.

between the first and third tasks in verification arises from the potential source of errors in Scaffold code and the MCL code. More precisely, considering the manual process of coding a quantum algorithm, the main problem in Scaffold code can arise from software programming mistakes. In contrast, considering a given Scaffold code, errors in the MCL code are the result of the mistakes in the Toolbox design and tool suite. In addition to these mistakes, the MCL code should be verified to make sure that the error threshold of the applied QEC scheme is greater than the error probability of the PMD instructions considering the QC scheme. The EFFBD of NQCS Toolbox verification process is shown in Figure 19.

## 6.1. Verification of algorithm to QPL

Quantum algorithm to QPL Verification (Table 7) requires that we provide a proof that the QPL code correctly implements the desired algorithm. This is an extremely challenging task that does not lend itself to an exact solution. In general, the problem of automatic validation of a software program is intractable even in classical computing. Similarly, the problem of automatic validation of a Scaffold code versus a given quantum algorithm is intractable (and in fact un-decidable). A viable option is to run different benchmark circuits and compare the outputs with expected (golden) results. Another option is to ask the QPL programmer/user to provide a proof of the correctness for his code using formal methods (although this may be a huge demand imposed on the user).

Verification for larger quantum circuits will be done through inspection. When possible, arguments will be constructed to show that small-input verification is indicative of large-input cases. At the QPL level, we expect standard compiler techniques to emit code that maintains correctness at large scale. Such techniques have been shown to be behaviorally correct over the long history of compiler usage for large-scale classical programs and inputs. Since each algorithm is only one or two thousand lines of Scaffold code, we expect verification by inspection to be a reasonable approach.

In this way, we reduce the number of casual errors introduced by software programmers.

Table 7. The detailed description for the "Verify Implemented Algorithm" use-case .

| Use-case name | Verify Implemented Algorithm |
|---|---|
| Related requirements | Requirement 7 in Section 3 |
| Goal in context | To make sure that the Scaffold program satisfies all the requirements of the quantum algorithm. |
| Preconditions | The Scaffold program should be given. |
| Successful end conditions | The Scaffold program has a reasonable chance of correctly implementing the quantum algorithm. |
| Fail end conditions | The Scaffold program is not verified at all. |
| Primary actors | Quantum Programmer |
| Secondary actors | - |
| Trigger | The quantum programmer asks the NQCS to verify the Scaffold program. |
| Main flow | Action |
| | Debugging. |

## 6.2. Verification of Toolbox design requirements

Each Toolbox requirement (cf. Section 3) should be evaluated by one specific testing strategy to make sure that each requirement is properly addressed. Excluding the topics discussed in the other sections leads to the following items:

1. The NQCS Toolbox shall allow the quantum programmer to write the quantum algorithm in Scaffold providing quantum data types, quantum operations, classical operations, and appropriate control flow structures. This can be evaluated by "demonstration" — by expressing several quantum algorithms by Scaffold.

2. The NQCS Toolbox shall enable (1) PMD expert to describe a new PMD architecture, along with error rates and latencies for various quantum instructions, (2) QEC expert to describe a new quantum error correction code, and (3) QC expert to describe a new quantum control protocol. These requirements can be evaluated by "demonstration" — by handling different QCs, QECs, and PMDs.

3. The NQCS Toolbox shall provide an efficient quantum compiler for the quantum programmer. This can be checked by comparison of compiler outputs for several well-defined problem sizes with the expected outputs.

4. The NQCS Toolbox shall provide a set of software utilities such as a timing simulator as the NQCS quantum programming environment. This requirement can be verified by "demonstration" — (1) by manual output comparison for several specific small cases, and (2) by automatic output comparison for many large test vectors with known results.

## 6.3. Verification of QPL to MCL

The goal of this step is to ensure that the generated MCL code is a valid realization of the Scaffold code on the target PMD. We plan to divide this verification task into three

subtasks: (i) verification of the HF-QASM against the Scaffold code, (ii) verification of the FT gate library, and (iii) verification of the MCL code against HF-QASM.

- Different parts of the Scaffold code may follow different synthesis/compilation paths to be transformed into the HF-QASM. This motivates us to take different verification approaches for these parts.
  - Reversible circuits can be reduced to conventional AND-OR-NOT circuits for purposes of verification. As a result, for the classical modules and overall control flow, classical verification algorithms and tools similar to Mentor Graphics QuestaSim[5] can be used. Additionally, equivalence checking algorithms for reversible circuits can be used [20]. Equivalence checking is a common term in the published papers and is used to check two circuits (or a circuit vs. a truth-table).
  - For standard quantum modules such as quantum Fourier transform or realization of arbitrary rotations using the Solovay-Kitaev algorithm [14], it is possible to do the verification by construction because all of these quantum modules will be replaced by HF-QASM blocks by using well-known (textbook) algorithms that have analytical proof of correctness. As an alternative approach, recent verification algorithms [21], [22] for quantum logic can also be used.[6]
- Generating the FT gate library needs both verification and error simulation steps. Similar to the module responsible for generating FT gate library, related verification and error simulation steps can be performed once, and cached to be reused later on by the Toolbox as needed. It is, of course, impossible to simulate arbitrary quantum circuits, or there would be no need to build quantum computers; but large pieces of FT code can be simulated using efficient techniques.
- To verify the generated MCL code for each FT gate in the library against the transformation matrix of the FT gate, we will exploit the Gottesman-Knill theorem [6][7]. More precisely, we will use [23]. This simulation can be used for an adequate number of input (stabilizer) states to guarantee correct behavior with reasonably high probability. In a similar way, the classical modules (e.g., oracle functions) are essentially reversible classical circuits; they can be checked for random computational basis states to give probabilistic verification of their correctness.

[5] http://www.mentor.com/products/fv/questa/

[6] PP (Probabilistic Polynomial-Time) is the class of decision problems solvable by an NP (Nondeterministic Polynomial-Time) machine which gives the correct answer (i.e., 'Yes' or 'No') with probability $> \frac{1}{2}$. $\mathsf{P}^{\mathsf{PP}}$ (P with PP oracle) includes decision problems solvable in polynomial time with the help of an oracle for solving problems from PP. Quantum circuit simulation belongs to the complexity class $\mathsf{P}^{\mathsf{PP}}$ [22].

[7] According to the Gottesman-Knill theorem [6], quantum circuits exclusively consisting of the following components can be efficiently simulated on a classical computer in polynomial time:

- A state preparation N-circuit with initial value $|000...0\rangle$ — qubit preparation in the computational basis,
- Quantum gates from the Clifford group (Hadamard, Phase, CNOT, and Pauli gates),
- Measurements in the computational basis

- In addition to this quantum simulation tool, a Monte Carlo simulation framework can be developed to check if the applied QEC satisfies the maximum error probability and to determine the success rate of the gate. This can be done in a simplistic manner merely by generating random errors and estimating overall failure rates.
- Error probability or success rate of the gates can be used in error analysis of the final MCL code to check the total success rate of the quantum circuit. Verification of the MCL code against HF-QASM is not the target of our Toolbox because we expect that the number of FT gates to be too large to be simulated using classical computers.

# 7. Performance Measures

Performance measures are the non-functional requirements of the NQCS Toolbox. They are used to assess (i) the quality of results and (ii) the scalability of the NQCS Toolbox.

## 7.1. Quality of Results

We use the following performance measures to evaluate the outputs of the Toolbox.

- Total number of physical qubits: The total number of physical qubits required to implement an algorithm.
- Circuit depth: This measure reflects the total runtime of the algorithm considering the degree of parallelization in resulting circuits.
- Number of ancillae: Along with the number of main qubits, the number of ancillae required at each step is an important parameter for circuit evaluation.
- Runtime/area ratio of the ideal circuit to the encoded circuit: The ratio of runtime/area metric for the ideal circuit to that for the QEC-enabled circuit reflects the efficiency of the QEC code.
- Area-Delay-to-Correct-Result: This metric is introduced in [24] and is defined as follows:

$$ADCR = Area \times \frac{Latency}{P_{success}}, \tag{1}$$

where $Area$ is the area of the bounding box in the PMD in which the quantum algorithm is mapped to, $Latency$ is the delay of execution of the quantum algorithm, which is calculated by the timing simulator, and $P_{success}$ is the probability of getting the correct (error-free) answer.

In addition to the aforesaid metrics, there are other metrics that can be considered during different steps in order to make sure that our circuits meet the final constraints. Among these metrics, geometric constraints (e.g., nearest neighbor interaction [17]), and gate complexity (e.g., the average number of control qubits of each gate) are adopted in our Toolbox.

Furthermore, to be amenable to automation, the Toolbox can take into account the role of the researcher/user in various parts. One possibility is to provide the user access to manipulate the trade-offs between resources. As an example, our back-end tool allows the user to explore the circuit latency vs. ancilla count trade-off [25]. More specifically, physical ancilla qubits are precious resources in quantum computers. However, increasing the total ancilla budget may lead to a lower circuit latency and hence improved quality of results.

## 7.2. Toolbox Scalability

The Toolbox should be able to handle sufficiently large problem sizes. To better appreciate the scalability challenge, we mention a few notable results.

- For a reversible circuit with $n$ lines, where its optimal gate-count realization, needs $h$ gates from a library $\mathcal{L}$, an enumerative method may branch $h$ ways on each $\mathcal{L}$-gate. For example, assume that only multiple-control Toffoli gates exist in the library. For this simplified case, an exhaustive method examines $(n \times 2^{n-1})^h$ gates[8] to find an optimal circuit. Currently, the sharpest upper bound on the number of gates for a reversible circuit is $8.5n2^n + o(2^n)$ [26]. Therefore, a given circuit may need an exponential number of gates.

- The method in [27] uses a search-based method [28] to synthesize a given function. Therefore, it may not be able to find a result. Accordingly, NQCS Toolbox uses [13] which adopts decision diagrams to handle large functions. The early synthesis results on an Intel 3.1 GHz CPU with 4 GB of memory with a runtime limit of 600 seconds reveal interesting results [13]. We can increase the runtime limit to several days and improve processing power in order to enhance the quality of results and also to handle large-scale quantum circuits.

- The search space for quantum-logic synthesis is not finite, and circuits implementing generic unitary matrices require $\Omega(4^n)$ gates [29]. We use the Solovay-Kitaev theorem for compiling an arbitrary single-qubit gate into a sequence of gates from a fixed and finite set. The algorithm runs in $O(\log^{2.71}(1/\varepsilon))$ time.[9] Considering an exponentially number of gates leads to a runtime/memory limit on the classical computer assigned to run the algorithms. Therefore, the processing power of the underlying machine will limit the Toolbox.

- Quantum circuit simulation belongs to the complexity class $\mathsf{P}^{\mathsf{PP}}$. Hence, handling an arbitrary-size circuit is out of reach in general. For reversible circuits, SAT-based techniques lead to good results [20]. While the Boolean satisfiability problem is NP-complete, the state-of-the-art SAT solvers solve many practical problem instances,

[8] There are $\binom{n}{1}$ possible NOT gates and $\binom{n}{2}$ possible CNOT gates in which one of its two inputs can be the target output. Hence, the total number of $2\times\binom{n}{2}$ CNOT gates can be obtained. For a ($k$+1)-bit gate, $k \in (2,3,\cdots,n-1)$, there are $\binom{n-1}{k}$ possible gates when the target can be the $i$-th ($i \in [1,n]$) bit. Considering all possible bits as the target leads to $n \times \binom{n-1}{k}$ ($k$+1)-bit gates. Therefore, the total number of gates is $\binom{n}{1} + 2 \times \binom{n}{2} + n \times (\sum_{i\in(2\cdots n-1)} \binom{n-1}{i}) = n \times 2^{n-1}$.

[9] It produces as output a sequence of $O(\log^{3.97}(1/\varepsilon))$ quantum gates which is guaranteed to approximate the desired quantum gate to an accuracy within $\varepsilon > 0$.

i.e., a SAT instance can consist of hundreds of thousands of variables, millions of clauses, and tens of millions of literals. Hence, we expect reasonable results on a normal classical computer is obtainable. For quantum circuit verification, state-of-the-art techniques [22] use improved decision diagrams and can handle middle-size problems fast. However, automatic verification of large circuits is limited by constraints imposed by NP-complete and $\mathsf{P}^{\mathsf{PP}}$ problems.

A key design strategy that significantly enhances the scalability of the NQCS Toolbox is to take advantage of the hierarchical design of quantum circuits. More specifically, quantum circuits can be partitioned into repetitively-used quantum modules. This means that mapping one instance of these modules is sufficient. Accordingly, HF-QASM has been chosen as the main input to the back-end tool. Table 8 compares sizes of compiled codes in QASM and HF-QASM formats for various quantum benchmarks. As can be seen, for complex algorithms such as *Triangle Finding Problem*, QASM format is completely inefficient because it requires more than 60GB of disk space. On the other hand, HF-QASM can be successfully adopted. For more information, please refer to [9].

Table 8. Comparison of QASM and HF-QASM file sizes for different quantum benchmarks. *module count* denotes the number of modules used in the benchmark.

| **Benchmark** | **Ref.** | **Module Count** | **Problem Size** | **File Size** | | **Size Ratio** |
|---|---|---|---|---|---|---|
| | | | | **QASM** | **HF-QASM** | |
| Grover's Algorithm | [2] | 6 | n=100[†] | 548KB | 52KB | 11 |
| | | 6 | n=300[†] | 3.2MB | 160KB | 20 |
| | | 6 | n=500[†] | 7.0MB | 268KB | 27 |
| Binary Welded Tree | [30] | 3 | $n = 43$, $3 \leq s \leq 19$[‡] | 5.52MB – 34.9MB | 84KB – 88KB | 67 – 406 |
| Ground State Estimation | [3] | 170 | M=6[§] | 481MB | 200KB | 2,462 |
| Triangle Finding Problem | [4] | 332 | n=5[*] | 63GB | 504KB | 129,418 |
| | | 10,202 | n=10[*] | Failed[¶] | 28MB | N/A[¶] |

[†] A database of $2^n$ elements is being searched.
[‡] $n$ is the height of the tree and $s$ is a time parameter within which the solution is found.
[§] $M$ is the molecular weight of a molecule.
[*] $n$ is the number of nodes in the graph.
[¶] The QASM file size exceeded 75GB and Scaffold compiler was crashed while generating the output file.

## 8. Further Reading

For further background on quantum information and computation, interested readers are encouraged to consult the main textbooks including the book by Nielsen and Chuang [6]. For an introduction of reversible circuits and related synthesis and optimization methods refer to the recent survey by Saeedi and Markov [28]. For simulation and verification of quantum circuits please consult [22].

## List of Acronyms

**CTQG** Classical code To Quantum Gate

**CAD** Computer Aided Design

**CULB** Computing Universal Logic Block

**ECC** Error Correction Code

**EFFBD** Enhanced Functional Flow Block Diagram

**FT** Fault-Tolerant

**ISA** Instruction Set Architecture

**MCL** Machine Control Language

**MCT** Multiple-Control Toffoli

**MULB** Memory Universal Logic Block

**NQCS** Next-generation Quantum Computing Systems

**O2I** Operation-to-Instruction

**PMD** Physical Machine Description

**QASM** Quantum Assembly Language

**H-QASM** Hierarchical Quantum Assembly Language

**HF-QASM** Hierarchical Fault-tolerant Quantum Assembly Language

**QC** Quantum Control

**QEC** Quantum Error Correction

**QGS** Quantum Gate Set

**QIDG** Quantum Instruction Dependency Graph

**QPD** Quantum Physical Designer

**QPL** Quantum Programming Language

**QUFD** Quantum Universal logic block Factory Designer

**RAW** Read After Write

**RC** Resource Calculation

**RMDDS** Reed-Muller Decision Diagram Synthesis

**SAT** Satisfiability

**ULB** Universal Logic Block

**WAR** Write After Read

**WAW** Write After Write

**XML** Extensible Markup Language

## Acknowledgments

The authors would like to thank Prof. Todd Brun, Dr. Hadi Goudarzi, Ali JavadiAbhari, Dr. Chia-Chun Lin, Dr. Mehdi Saeedi, and Dr. Martin Suchura for their helpful discussions and inputs.

This research was supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center contract number D11PC20165.

## References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Theory of Computing*, 1996, pp. 212–219.

[3] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik, "Simulation of electronic structure hamiltonians using quantum computers," *Molecular Physics*, vol. 109, no. 5, pp. 735–750, 2011.

[4] F. Magniez, M. Santha, and M. Szegedy, "Quantum algorithms for the triangle problem," *SIAM Journal on Computing*, vol. 37, no. 2, pp. 413–424, 2007.

[5] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien, "Quantum computers," *Nature*, vol. 464, no. 7285, pp. 45–53, 2010.

[6] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[7] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. Chong, and M. Martonosi, "ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs," in *International Conference on Computing Frontiers*, May 2014.

[8] H. Goudarzi, M. J. Dousti, A. Shafaei, and M. Pedram, "Design of a universal logic block for fault tolerant realization of any logic operation in an ion-trap quantum circuit," *Quantum Information Processing*, vol. 13, no. 5, pp. 1267–1299, Jan. 2014.

[9] M. J. Dousti, A. Shafaei, and M. Pedram, "Squash 2: A Hierarchical Scalable Quantum Mapper Considering Ancilla Sharing," submitted.

[10] A. JavadiAbhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, "Scaffold: Quantum Programming Language," Department of Computer Science, Princeton University, Tech. Rep. TR-934-12, June 2012.

[11] K. M. Svore, A. V. Aho, A. W. Cross, I. Chuang, and I. L. Markov, "A layered software architecture for quantum computing design tools," *Computer*, vol. 39, no. 1, pp. 74–83, Jan. 2006.

[12] N. E. Rainwater and S. J. Kapurch, "NASA systems engineering handbook," *National Aeronautics Space Administration, Washington DC, NASA SP-2007-6105 Rev 1*, Dec. 2007.

[13] C. Lin and N. K. Jha, "RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits," *Journal on Emerging Technologies in Computing Systems*, 2013.

[14] C. M. Dawson and M. A. Nielsen, "The Solovay-Kitaev algorithm," *Quantum Information & Computation*, vol. 6, no. 1, pp. 81–95, Jan. 2006.

[15] D. Kudrow, K. Bier, Z. Deng, D. Franklin, Y. Tomita, K. R. Brown, and F. T. Chong, "Quantum rotations: a case study in static and dynamic machine-code generation for quantum computers," in *Proceedings of the International Symposium on Computer Architecture*, 2013, pp. 166–176.

[16] C. Lin, A. Chakrabarti, and N. K. Jha, "Optimized quantum gate library for various physical machine descriptions," *IEEE Transactions on VLSI Systems*, 2013.

[17] M. Saeedi, R. Wille, and R. Drechsler, "Synthesis of quantum circuits for linear nearest neighbor architectures," *Quantum Information Processing*, vol. 10, no. 3, pp. 355–377, Jun. 2011.

[18] E. Chi, S. A. Lyon, and M. Martonosi, "Tailoring quantum architectures to implementation style: a quantum computer for mobile and persistent qubits," in *Proceedings of the International Symposium on Computer Architecture*, 2007, pp. 198–209.

[19] M. Suchara, A. Faruque, C.-Y. Lai, G. Paz, F. Chong, and J. Kubiatowicz, "QuRE: The Quantum Resource Estimator Toolbox," in *International Conference on Computer Design*, October 2013.

[20] R. Wille, D. Große, D. M. Miller, and R. Drechsler, "Equivalence checking of reversible circuits," *International Symposium on Multiple-Valued Logic*, pp. 324–330, 2009.

[21] S.-A. Wang, C.-Y. Lu, I.-M. Tsai, and S.-Y. Kuo, "An XQDD-based verification method for quantum circuits," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E91-A, no. 2, pp. 584–594, Feb. 2008.

[22] G. F. Viamontes, I. L. Markov, and J. P. Hayes, *Quantum Circuit Simulation*. Springer, 2009.

[23] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, p. 052328, Nov. 2004.

[24] M. G. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz, "A fault tolerant, area efficient architecture for Shor's factoring algorithm," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 383–394.

[25] M. J. Dousti, A. Shafaei, and M. Pedram, "Squash: A Scalable Quantum Mapper Considering Ancilla Sharing," in *Proceedings of Great Lakes Symposium on VLSI*, May 2014, pp. 117–122.

[26] M. Saeedi, M. S. Zamani, M. Sedighi, and Z. Sasanian, "Reversible circuit synthesis using a cycle-based approach," *Journal on Emerging Technologies in Computing Systems*, vol. 6, no. 4, pp. 13:1–13:26, Dec 2010.

[27] P. Gupta, A. Agrawal, and N. Jha, "An algorithm for synthesis of reversible logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2317 –2330, Nov. 2006.

[28] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits — a survey," *ACM Computing Surveys*, vol. 45, no. 2, pp. 21:1–21:34, Mar. 2013.

[29] V. V. Shende, I. L. Markov, and S. S. Bullock, "Minimal universal two-qubit quantum circuits," *Physical Review A*, vol. 69, pp. 062 321:1–062 321:8, 2004.

[30] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman, "Exponential algorithmic speedup by a quantum walk," in *Proceedings of the Theory of Computing*, 2003.