Toward a Quantum Behavioral Interface Specification Language

Tiancheng Jin[1,a)] Jianjun Zhao[1,b)]

Abstract: ScaffML is a behavioral interface specification language tailored to Scaffold, a quantum programming language. It can specify pre- and post-conditions for Scaffold modules and allows assertions to be intermixed with Scaffold code, which helps in debugging and verification. This paper discusses the goals and overall approach of ScaffML and describes the basic features of the language with examples. ScaffML provides an easy-to-use specification language for quantum programmers, which can support static analysis, run-time checking, and formal verification of Scaffold programs.

Keywords: Specification languages, quantum programming, testing and verification, Scaffold

1. Introduction

1.1 Model-Oriented Specification Languages

Model-oriented specification languages combine the ideas from two seminal papers by Hoare [12, 13]. One paper [12] introduced to use of two predicates (pre- and postcondition) over program states to specify a computation. The first predicate (called *precondition*) specifies the requirements on the state before the computation, while the second predicate (called *postcondition*) specifies the desired final state. The other paper [13] introduced to use of an abstraction function that maps the implementation data structure (i.e., an array) to a mathematical value space (i.e., a set). The idea behind this is that one can use the abstract values to specify the *abstract data type* (ADT), which allows users of the ADT's operation to reason about calls without considering the details of the implementation. In terms of these ideas, model-oriented specification languages specify software modules (procedures, functions, and methods) using pre- and postconditions. The pre- and postconditions use the vocabulary specified in an *abstract model*, which mathematically specifies the abstract values.

The most widely used model-oriented specification languages are VDM [15] and Z [22]. Both come with a mathematical toolkit from which users can assemble abstract models for use in specifying procedures. The toolkit of VDM resembles that of a (functional) programming language; it provides certain basic types (integers, booleans, characters) and structured types such as records, Cartesian products, disjoint unions, and sets. The toolkit in Z is based on set theory; it has a relatively elaborate notation for various set constructions, as well as powerful techniques for combining specifications (the schema calculus).

Although generic model-oriented specification languages such as Z and VDM can specify programs written in different programming languages, they can not specify the exact interface of modules written in a specific programming language. This is because the details of the interface that need to be specified may differ from one programming language to another. To compensate, several behavioral interface specification languages (BISLs) have been designed, each tailored to a specific programming language. Examples include Larch/C++ for C++ [19], ACSL for C [6], and JML [20] and AAL [17] for Java. The advantage of tailoring each BISL to a specific programming language is that one can specify both the behavior and the exact interface to be programmed. Recent research results have shown that using BISLs to specify programs written in a specific programming language has great practical benefits both in static compile-time checking and run-time checking [11].

1.2 Formal Specification of Scaffold

Research for formal specification languages must adapt to the emergence of new language paradigms to specify programs written in these new languages by presenting new specification approaches that are relevant to these new languages. Quantum programming is the process of designing and building executable quantum computer programs to achieve a particular computing result and has been drawing increasing attention recently. A number of quantum programming approaches are available to write quantum programs, for instance, Scaffold [1], Qiskit [3], Q# [23], ProjectQ [24], and Quipper[9]. As research in quantum programming is reaching maturity with many active research products, researchers for formal specification languages in general and BISLs, in particular, should focus on this new paradigm.

The field of quantum programming has, so far, focused primarily on problem analysis, language design, and implementation. The specification and validation of quantum programs have

[1] Kyushu University, Fukuoka, Japan
[a)] jin.tiancheng.932@s.kyushu-u.ac.jp
[b)] zhao@ait.kyushu-u.ac.jp

received comparatively little attention. To formally verify quantum programs, we must have some means to specify the properties of quantum programs formally. However, although many generic formal specification languages and BISLs have been proposed for specifying programs written in classical procedural and object-oriented programming languages, no BISL exists, to our knowledge, that can be used to specify programs written in quantum programming languages such as Scaffold until now. Moreover, due to specific features such as quantum superposition, entanglement, and no-cloning in a quantum programming language, existing formal specification languages (and BISLs) for classical programming languages can not be applied to quantum programming languages straightforwardly. This motivates us to design a formal specification language suitable for specifying programs written in quantum programming languages.

1.3 Our Specification Approach to Scaffold

Instead of designing a generic specification language for quantum programming, we choose instead to design a *behavioral interface specification language* (BISL) tailored to Scaffold, a quantum programming language [1]. A BISL describes both the details of a module's interface with clients and its behavior from the client's point of view [20]. Using a BISL, we can formally specify both the behavior and the exact interface of Scaffold programs' modules, which is an essential step towards formally verifying these modules.

Our BISL for Scaffold is called ScaffML (Scaffold Modeling Language), which uses the same basic approaches as classical BISLs, such as ACSL [6] for C and JML [20] for Java, to specify Scaffold modules and interfaces. ScaffML provides annotations to specify Scaffold programs with pre- and postconditions and class invariants. These annotations enable both *dynamic analysis* in support of activities such as debugging and testing and *static analysis* in support of the formal verification of properties of Scaffold programs. Static analysis activities could verify that the code of a Scaffold module correctly implements its specification.

The key to the verification process is to develop a transformation tool that automatically transforms Scaffold programs with ScaffML specifications into corresponding verification conditions (VC) through VC generators, which are finally received by some interactive provers or automatic provers such as Coq [8] or CVC3 [4]. By doing this, we can formally check and verify Scaffold programs.

In this paper, we discuss the goals of ScaffML and its overall specification approach. We also provide examples of how to use ScaffML to specify Scaffold modules.

The rest of the paper is organized as follows. Section 2 presents the design rationale for ScaffML. Section 3 briefly introduces ANSI/ISO C Specification Language. Section 5 uses examples to show how Scaffold modules are specified in ScaffML. Section 6 discusses related work, and conclusion and future work are given in Section 7.

2. Design Rationale

Our purpose in developing ScaffML is to study how quantum programs can be specified and verified formally. Designing ScaffML is, therefore, only a part of our proposed activities. We must also develop techniques and tools to support the formal specification and verification of Scaffold programs augmented with ScaffML specifications. We have therefore chosen to design ScaffML as a compatible extension to Scaffold to (1) facilitate its adoption by current Scaffold users and (2) facilitate the adoption of existing ACSL-based verification toolchain to check Scaffold programs.

ACSL is an especially appropriate base for the ScaffML design for two reasons. First, Scaffold is an extension to C for implementing quantum programming, and ACSL is a BISL specially designed for C. By structuring ScaffML as an extension to ACSL, we can focus our attention on the new issues associated with the use of quantum modules. Second, ACSL has an efficient toolchain for supporting static and dynamic checking of C programs. If we can automatically transform Scaffold programs with ScaffML specifications into corresponding verification conditions (VC) through VC generators, which are finally received by some interactive provers or automatic provers such as Coq and CVC3, we can use the ACSL toolchain to verify Scaffold programs.

To focus on the key ideas of ScaffML, in this paper, we do not consider the specification of Scaffold classical modules, which can be specified in the same way as ACSL [6] for C.

3. ANSI/ISO C Specification Language

ANSI/ISO C Specification Language (ACSL) [6] is a formal BISL tailored to C programming language [16]. ACSL allows precondition, postconditions, and assertion to be specified for C functions. The predicates in ACSL are written using regular C expressions extended with logical operators and universal and existential quantifiers. ACSL specifications are expressed as special commends in C function definitions, enclosed between `/*@` and `*/`.

ACSL supports specifying a C function at both statement and function levels. These two-level specifications together form the complete behavioral specifications for the function. A function-level specification for a C function is a set of requirements over the arguments of the function and/or a set of properties that are ensured at the end of the function. The formula that expresses the requirements is called a *precondition*, whereas the formula that expresses the properties ensured when the function returns is a *postcondition*. Together, these conditions form a specification (or contract) between the function and its callers: each caller must guarantee that the precondition holds before calling the function. In exchange, the function guarantees that the postcondition holds when it returns.

3.1 The Workflow of ScaffML

Figure 1 shows the workflow of the ScaffML working system. First, a Scaffold program with its ScaffML specification is translated into the intermediate code written in an intermediate language. Then, the code is fed into a verification condition (VC) generator to generate the necessary verification conditions, which are finally received by an interactive prover or an automatic prover like Coq or CVC3.

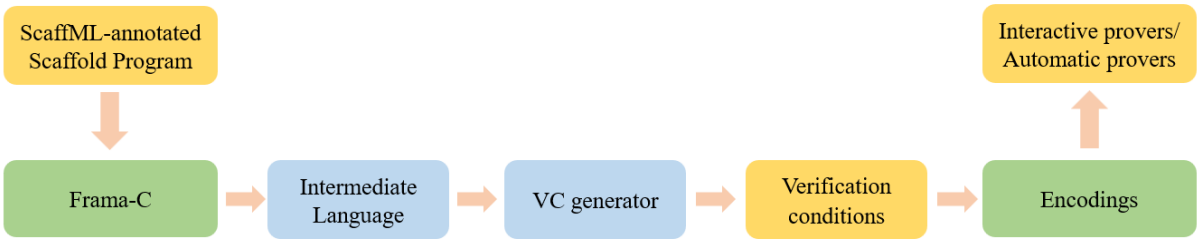


Fig. 1 The Workflow of ScaffML.

4. Specifying Scaffold Quantum Gates in ScaffML

The Scaffold provides a standard library of gates which consist of several types of most commonly used quantum gates. We list some of the quantum gates in Figure 2. As the first step, in this section, we try to specify several gates in ScaffML.

4.1 Representation of Quantum Bit Coefficient

In classical programming languages, given an array `int a[i]`, we use `a[0]`, `a[1]`, ..., `a[i-1]` to express the elements of the array `a[i]`. In quantum programming languages such as Scaffold, we can define a quantum register using the statement `qreg q[i]`, and the elements of `q[i]` can be represented as `q[0]`, `q[1]`, ..., `q[i-1]`. For each qubit $|q\rangle = \alpha|0\rangle + \beta|1\rangle$, it can be also represented as an array $[\alpha, \beta]$, where the coefficients α and β are the squares of probability of $|0\rangle$ and $|1\rangle$. To distinguish the representations of the qubit coefficient and the number of qubits in a quantum register, in ScaffML, we use q[|0⟩] = α and q[|1⟩] = β to express the coefficient of each qubit. As a result, a qubit q can be expressed as $|q\rangle = \alpha|0\rangle + \beta|1\rangle = [\alpha, \beta]$ = [q[|0⟩], q[|1⟩]].

In fact, the intention behind $|0\rangle$ and $|1\rangle$ is to show the variable is a quantum bit. So the q[|0⟩] and q[|1⟩] can express the intrinsic of a qubit q. If a qubit is in a quantum register, for example, Figure 3 shows that we can express the coefficient α and β of q[0] as q[0][|0⟩] and q[0][|1⟩]; the[0] shows this qubit the first one in the quantum register q[], the [|0⟩] and [|1⟩] show the coefficients of $|0\rangle$ and $|1\rangle$ of q[0].

4.2 Pre-Defined Modules in ScaffML

To help programmers understand the equation meanings of pre- and postconditions, we defined several modules in ScaffML. Figure 4 shows some of these modules. The annotated equation is equivalent to the module above. For example, the sum of the coefficients' squares equals 1 for each qubit's. Equivalent to writing the equation for the sum of the coefficients' squares equals 1, we can use the defined module `qubitselfCheck()` to specify. The keyword **ensures** means the module or equation should match the preconditions and postconditions. The keyword **old** means the variable which holds the value before the Scaffold module.

4.3 Specifying a Pauli Gate

There are three types of Pauli gates: Pauli-X, Pauli-Y, and Pauli-Z. Here we take the Pauli-X gate as an example to show how to specify Pauli gates. Pauli-X is a single-qubit rotation through π radians around the x-axis. The X-gate is represented by the Pauli-X matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad (1)$$

For a qubit $|q\rangle = \alpha|0\rangle + \beta|1\rangle = [\alpha, \beta]$ = [q[|0⟩] , q[|1⟩]], the X$|q\rangle$ = $\beta|0\rangle + \alpha|1\rangle$ = $[\beta, \alpha]$ = [q[|1⟩], q[|0⟩]]. We can specify the Pauli-X gate, as shown in Figure 5. Here, **requires valid** specifies the name and size of the required array, respectively. And **assigns** specifies the length of the array should not be modified or changed. And then ensure the elements of the qubit are exchanged with each other. At last, referring to the last module in Figure 5, ensure the sum of the coefficients' squares equals 1.

4.4 Specifying a Hadamard Gate

The Hadamard gate is a single-qubit operation that maps the basis states $|0\rangle$ and $|1\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, respectively, thus creating an equal superposition of the two basis states. The Hadamard gate is represented by the Hadamard matrix:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Using ScaffML, we can specify the Hadamard gate, as shown in Figure 6. We should ensure that the result of the qubit after the Hadamard gate meets the condition of the Hadamard matrix.

4.5 Specifying a CNOT Gate

The Controlled NOT (CNOT) gate contains two qubits, where one of the qubits is called the control qubit while the other one is called the target qubit. The CNOT gate can realize the following operation: (1) Performs a Pauli-X gate on the target qubit when the control qubit is in state $|1\rangle$; (2) Performs the target qubit unchanged when the control qubit is in state $|0\rangle$. Figure 7 shows the Circuit Elements of the CNOT Gate. The CNOT Gate is represented by the CNOT matrix:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 7 shows the specification in ScaffML for the CNOT Gate. The command **behavior** means there are two situations. The first is that the control qubit $|a\rangle$ is 0. The **measZ()** measures

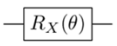
Name	Symbol	Usage	Description
NOT (Pauli X)		X(qreg input[1])	Bitwise NOT
CNOT		CNOT(qreg control[1], qreg target[1])	Controlled NOT: if (control) then NOT(target)
CCNOT (Toffoli)		Toffoli(qreg control1[1], qreg control2[1], qreg target[1])	if (control1 && control2) then NOT(target)
Hadamard		H(qreg input[1])	Hadamard gate
Rotation X		Rx(qreg input[1], angle)	Rotation around the X-axis
SWAP		SWAP(qreg input[1], qreg input[2])	Exchange two qubits
Fredkin (Controlled SWAP)		Fredkin(qreg control[1], qreg target1[1], qreg target2[1])	if (control) then SWAP(target1, target2)

Fig. 2 Some Examples of Quantum Gates.

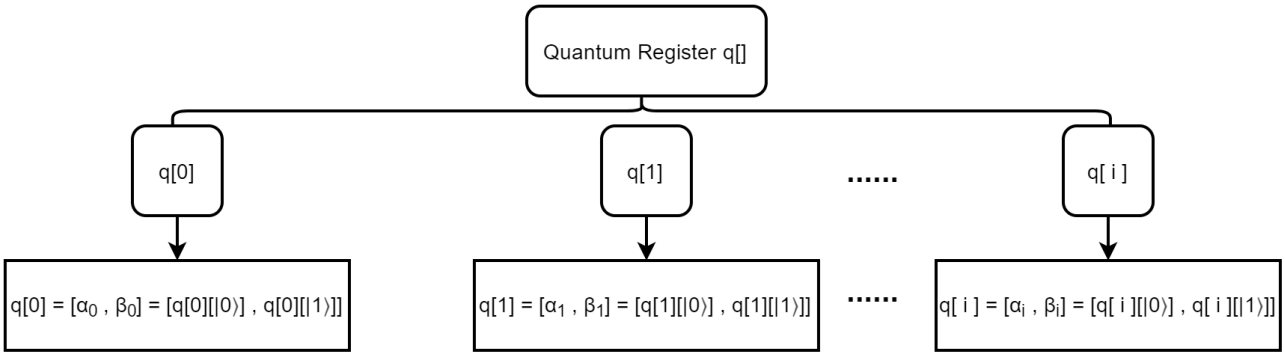


Fig. 3 The coefficient of a qubit in quantum register.

```
ensures equal_a: Unchanged{Here,Old}(a, 2);
//ensures a[|0>] == \old(a[|0>]);
//ensures a[|1>] == \old(a[|1>]);

ensures Hadamard_t: HadamardCheck{Here,Old}(t, 2);
//ensures t[|0>] == (\old(t[|0>]) + \old(t[|1>]))*sqrt(1/2);
//ensures t[|1>] == (\old(t[|0>]) - \old(t[|1>]))*sqrt(1/2);

ensures Phase_t: PhaseCheck{Here,Old}(t, 2);
//ensures t[|0>] == \old(t[|0>]);
//ensures t[|1>] == \old(t[|1>]) * e^(i*angle);

ensures qbitself_t: qbitselfCheck(t);
//ensures pow(t[|0>],2) + pow(t[|1>],2) == 1;
```

Fig. 4 Some predefined modules in ScaffML.

```
/*@
    requires valid: \valid(t[0]+(|0>..|1>));

    assigns t[0][|0>..|1>];

    ensures reverse: Reverse{Here,Old}(t[0], 2);
    ensures qbitself_t[0]: qbitselfCheck(t[0]);
*/
gate X(qreg t[1]);
```

Fig. 5 A simple program specifying a Pauli-X gate.

```
/*@
    requires valid: \valid(t[0]+(|0>..|1>));

    assigns t[0][|0>..|1>];

    ensures Hadamard_t[0]: HadamardCheck{Here,Old}(t[0], 2);
    ensures qbitself_t[0]: qbitselfCheck(t[0]);
*/
gate H(qreg t[1])
```

Fig. 6 A simple program specifying a Hadamard gate.

the qubit to compare with 0 or 1. In this situation, the target qubit $|b\rangle$ will not be changed. The second is that the control qubit $|a\rangle$ is 1. In this situation, the target qubit $|b\rangle$ will be reversed. In other words, the target qubit $|b\rangle$ passed a Pauli-X gate. The **complete behaviors** express the fact that for all ranges a and b that satisfy the preconditions of the contract, at least one of the specified named behaviors, in this case, **false** and **true**, applies. The **disjoint behaviors** show that for all ranges a and b that satisfy the preconditions of the contract at most, one of the specified named behaviors, in this case, **false** and **true**, applies.

4.6 Specifying a Rotation Gate

The rotation operators are generated by exponentiation of the Pauli matrices according to $exp(iAx) = cos(x)I + isin(x)A$. For example, R_x Gate is a single-qubit rotation through angle θ (radians) around the x-axis. The R_x Gate is represented by the R_x matrix:

$$R_x(\theta) = exp(-iX\theta/2) = \begin{bmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{bmatrix} \quad (2)$$

We can specify the R_x Gate as shown in Figure 8. We should ensure that the output of the result is consistent with the R_x matrix operation.

4.7 Specifying a Phase Shift Gate

The phase shift is a family of single-qubit gates that map the basis states $|0\rangle \mapsto |0\rangle$ and $|1\rangle \mapsto e^{i\varphi}|1\rangle$. The probability of measuring a $|0\rangle$ or $|1\rangle$ is unchanged after applying this gate, however, it modifies the phase of the quantum state. The Phase shift is represented by the Phase shift matrix:

$$P(\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix} \quad (3)$$

Using ScaffML, we can specify the Phase shift Gate as Figure 9 shows. The coefficient of $|0\rangle$ keeps itself. The coefficient of $|1\rangle$ rotated angle φ, so it multiplied $e^{i\varphi}$.

4.8 Specifying a SWAP Gate

The SWAP gate is a two-qubit operation. Expressed in basis states, the SWAP gate swaps the state of the two qubits involved in the operation:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

We can specify the SWAP gate, as Figure 10 shows. The postcondition is the values of $|a\rangle$ and $|b\rangle$ are swapped. The command **EqualRanges** means the elements of the first array are equal to the elements of the second array one by one, **Here** and **old** is used to express the values of the array after or before the SWAP gate, the number **2** means each array has two elements.

```
/*@
    requires valid: \valid(control[0]+(|0>..|1>));
    requires valid: \valid(target[0]+(|0>..|1>));

    assigns control[0][|0>..|1>];
    assigns target[0][|0>..|1>];

    behavior false:
    assumes measZ(control[0]) == 0;
    ensures equal_control[0]: Unchanged{Here,Old}(control[0], 2);
    ensures equal_b: Unchanged{Here,Old}(target[0], 2);

    behavior true:
    assumes measZ(control[0]) == 1;
    ensures equal_control[0]: Unchanged{Here,Old}(control[0], 2);
    ensures reverse: Reverse{Here,Old}(target[0], 2);

    complete behaviors;
    disjoint behaviors;

    ensures qbitself_control[0]: qbitselfCheck(control[0]);
    ensures qbitself_target[0]: qbitselfCheck(target[0]);
*/
gate CNOT(qreg target[1], qbit control[1])
```

Fig. 7 A simple program with specifying CNOT gate.

```
/*@
    requires valid: \valid(t[0]+(|0>..|1>));

    assigns t[0][|0>..|1>];

    ensures Phase_Rx: PhaseCheck_Rx{Here,Old}(t[0], 2);
    //ensures t[0][|0>] == \old(t[0][|0>])*cos(angle/2) -
    \old(t[0][|1>])*isin(angle/2);
    //ensures t[0][|1>] == \old(t[0][|1>])*cos(angle/2) -
    \old(t[0][|0>])*isin(angle/2);
    ensures qbitself_t[0]: qbitselfCheck(t[0]);
*/
gate Rx(qbit t[1], float angle)
```

Fig. 8 A simple program with specifying R_x gate.

```
/*@
    requires valid: \valid(t[0]+(|0>..|1>));

    assigns t[0][|0>..|1>];

    ensures Phase_t[0]: PhaseCheck{Here,Old}(t[0], 2);
    //ensures t[0][|0>] == \old(t[0][|0>])
    //ensures t[0][|1>] == \old(t[0][|1>]) * e^(i*angle);
    ensures qbitself_t[0]: qbitselfCheck(t[0]);
*/
gate Phase(qreg t[0], float angle)
```

Fig. 9 A simple program with specifying Phase shift gate.

```
/*@
    requires valid: \valid(a[0]+(|0>..|1>));
    requires valid: \valid(b[0]+(|0>..|1>));

    assigns a[0][|0>..|1>];
    assigns b[0][|0>..|1>];

    ensures equal_a[0]: EqualRanges{Here,Old}(a[0], 2, b[0]);
    ensures equal_b[0]: EqualRanges{Old,Here}(a[0], 2, b[0]);
    ensures qbitself_b[0]: qbitselfCheck(a[0]);
    ensures qbitself_a[0]: qbitselfCheck(b[0]);
*/
gate SWAP( qreg a[1], qreg b[1] )
```

Fig. 10 A simple program with specifying SWAP gate.

5. Specifying Scaffold Programs in ScaffML

In Scaffold, a module is a modular unit implementation whose definition is similar to a C function. ScaffML can specify the properties of an individual module in a Scaffold program using *module specifications*, and the specifications of all the modules in the program form a specification of the whole program.

In ScaffML, the specification of a module is similar to that of a function in ACSL or a method in JML. The specification is annotated with three formulas, that is, a *precondition*, a *postcondition*, and a *frame condition* which is declared by a `requires`, `ensures`, and `modifies` clause respectively. The precondition, postcondition, and frame condition together form a *specification* of the module for checking the code of the module.

In this section, we use ScaffML to specify two of widely used modules in quantum algorithms: Bell state and Quantum Fourier Transform (QFT).

5.1 Specifying the Bell State

The Bell state is the simplest and purest type of entangled quantum state. As shown in Figure 11, the Bell state construction circuit can construct a pair of quantum bits of the Bell state,

they can be represented by $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. Measuring one of the quantum bits is equivalent to immediately giving the same measured value of the other quantum bit. In other words, in this system, the two bits have the same value, 0 or 1.

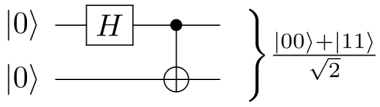


Fig. 11 Quantum circuit for the Bell state.

Figure 12 gives the specification of the Bell state. At first, we ensure the input qubits a and b are both in the state of $1|0\rangle + 0|1\rangle$. After preparing a Bell pair, there are two situations. The first is the Bell pair system is in the state of $|00\rangle$, in this situation, a and b are measured as 0, and the CNOT gate has not worked. The second is the Bell pair system is in the state of $|11\rangle$, in this situation, a and b are measured as 1, and the CNOT gate has worked.

```
/*@
    requires valid: \valid(a[0]+(|0>..|1>));
    requires valid: \valid(b[0]+(|0>..|1>));

    assigns a[0][|0>..|1>];
    assigns b[0][|0>..|1>];

    ensures \old(a[0][|0>]) == 1;
    ensures \old(a[0][|1>]) == 0;
    ensures \old(b[0][|0>]) == 1;
    ensures \old(b[0][|1>]) == 0;
    ensures a[0][|0>] == sqrt(1/2);
    ensures a[0][|1>] == sqrt(1/2);
    ensures b[0][|0>] == sqrt(1/2);
    ensures b[0][|1>] == sqrt(1/2);

    behavior CNOTfalse:
        assumes measZ(a[0]) == 0;
        ensures measZ(b[0]) == 0;

    behavior CNOTtrue:
        assumes measZ(a[0]) == 1;
        ensures measZ(b[0]) == 1;

    complete behaviors;
    disjoint behaviors;

    ensures qbitself_a[0]: qbitselfCheck(a[0]);
    ensures qbitself_b[0]: qbitselfCheck(b[0]);

*/
module PrepareBellPair(qreg a[1], qreg b[1]) {
    H(a[0]);
    CNOT(a[0],b[0]);
}
```

Fig. 12 Specifying the Bell state (an entangled state).

5.2 Specifying the QFT

The Quantum Fourier Transform (QFT) is a discrete Fourier transform that decomposes the original equation into a simpler product of multiple Unitary Matrix. Using this decomposition, Figure 13 shows the discrete Fourier transform can be used as a quantum circuit, which contains multiple Hadamard gates and controlled phase shifting gates.

Figure 14 gives the specification of QFT. We analyze the output of module QFT and ensure the states of every qubit. In order to form the output of module QFT, some of the qubits should be in the state of $1|0\rangle + 0|1\rangle$ and the other qubits should be in the state of $0|0\rangle + 1|1\rangle$.

6. Related Work

There has been significant work in the field of generic specification languages in general and BISLs in particular. Widely used generic specification languages include Z [22], VDM [15], B [2], and Larch [10]. Several BISLs based on Larch have been designed, each tailored to a specific programming language. Examples include LCL (for C) [10], LM3 (for Modula-3) [10], and Larch/C++ [19]. In addition to the Larch family, Meyer's work on the programming language Eiffel has advanced the cause of applying formal methods to object-oriented programs [21]. In Eiffel, unlike a Larch-style interface specification language, one can use Boolean expressions to specify pre- and postconditions for operations on ADTs written in Eiffel; that is, program expressions can be used in pre- and postconditions. In addition, in Eiffel, one can use class invariants to specify the global properties of instances of the class. On the other hand, several projects have been carried out to support the Design By Contract (DBC) principle, originally introduced by Meyer in Eiffel [21]. Examples include iContract [18] and Jass [5].

Recently, the emergence of Java as a popular object-oriented programming language has led to several BISLs designed for Java. Examples include JML [20], ESC/Java [7], and AAL [17]. JML allows assertions to be specified for Java classes and interfaces and provide very expressive power to specify Java modules (classes and interfaces). ESC/Java is a static checking tool for Java. It can statically check for various errors in a Java program without executing the program. The annotation language in ESC/Java is a subset of JML for annotating Java code in various ways. AAL is an annotation language designed for annotating and checking Java programs. Like JML, AAL supports runtime assertion checking. AAL also supports full static checking for Java programs similar to ESC/Java. AAL translates annotated Java programs into Alloy [14], a simple first-order logic with relational operators, and uses Alloy's SAT solver-based automatic analysis technique to check Java programs.

Although the specification languages mentioned above can specify programs written in various classical programming languages, they are not designed to specify programs written in quantum languages such as Scaffold. In summary, ScaffML is the first BISL tailored to Scaffold that can be used to specify quantum programs.

7. Conclusion

In this paper, we presented ScaffML, a behavior interface specification language tailored to Scaffold, and discussed the goals of ScaffML and the overall specification approach. ScaffML is an extension to ACSL, a BISL for C, for specifying Scaffold programs. ScaffML uses the same way as ACSL to specify Scaffold classical modules and extends ACSL with new notations to specify Scaffold quantum modules.

On the one hand, ScaffML provides a way to specify Scaffold programs with assertions (pre- and postconditions and module invariants), supporting runtime checking such as debugging and testing scaffold programs. On the other hand, ScaffML offers the possibility of fully automatic compile-time analysis for Scaffold

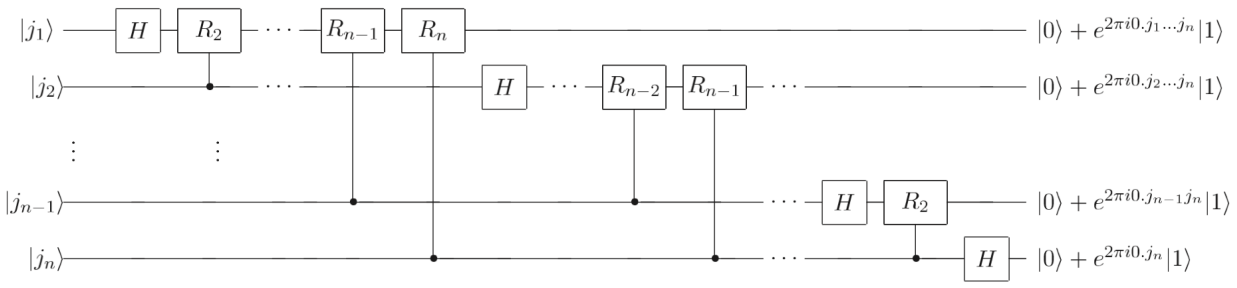


Fig. 13 Quantum circuit for the Quantum Fourier Transform (QFT) algorithm.

```
/*@
    requires valid: \valid(qbits[]+(|0>..|1>));

    assigns qbits[][|0>..|1>];

    module QFTCheck(qbits[], width, M_PI){
    int r = M_PI/pi;
    for ( int s = width-1; s >= 0; s-- ){
        if ( power(2,s) <= M_PI/pi )
        {
            //ensures qbit[s][|1>] == 1;
            //ensures qbit[s][|0>] == 0;
        r = r - power(2,s);
        }
        else
        {
            //ensures qbit[s][|0>] == 1;
            //ensures qbit[s][|1>] == 0;
        }
    }
    }
    ensures QFTCheck_qbits[]: QFTCheck(qbits[], width, M_PI);
    ensures qbitself_qbits[]: qbitselfCheck(qbits[]);
*/
module QFT (qreg qbits[width])
```

Fig. 14 Specifying the QFT.

programs, such as checking the code of a Scaffold module against its specification.

We also gave examples of ScaffML to show how to specify the deserved set of quantum gates in Scaffold, the entanglement state (Bell state), and the QFT quantum algorithm.

The work presented in this paper is preliminary; much work remains to be done to make ScaffML practical. We list our future work as follows:

- We would like to investigate our specification framework further and refine our proposed specification constructs for Scaffold.
- For formally verifying Scaffold programs, we would like to develop a transformation tool that automatically converts Scaffold programs with ScaffML specifications into corresponding verification conditions (VC) through VC generators, which are finally received by some interactive provers or automatic provers such as Coq or CVC3.
- We would like to develop formal semantics for ScaffML to support static analysis, checking, and testing.
- We plan to conduct some case studies using ScaffML to specify some Scaffold programs for implementing complex quantum algorithms such as Shor's and Grover's algorithms.

References

[1] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, Department of Computer Science, Princeton University, 2012.

[2] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge university press, 2005.

[3] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. Qiskit: An Open-source

Framework for Quantum Computing. jan 2019.

[4] Clark Barrett and Cesare Tinelli. Cvc3. *International Conference on Computer Aided Verification*, pages 298–302, 2007.

[5] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass—java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specification. 2008.

[7] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[8] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[9] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

[10] John V Guttag and James J Horning. *Larch: languages and tools for formal specification*. Springer Science & Business Media, 1993.

[11] John Hatcliff, Gary T Leavens, K Rustan M Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*, 44(3):1–58, 2012.

[12] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[13] Charles Antony Richard Hoare. Proof of correctness of data representations. In *Programming methodology*, pages 269–281. Springer, 1978.

[14] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[15] Cliff B Jones. *Systematic software development using VDM*, volume 2. Englewood Cliffs: Prentice Hall, 1990.

[16] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Prentice Hall, 1988.

[17] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, 2002.

[18] Reto Kramer. icontract-the java design by contract tool. In *Proceedings. Technology of Object-Oriented Languages. TOOLS 26*, pages 295–307. IEEE, 1998.

[19] Gary T Leavens. An overview of larch/c++: Behavioral specifications for c++ modules. *Object-Oriented Behavioral Specifications*, pages 121–142, 1996.

[20] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[21] Bertrand Meyer. *Object-oriented software construction*. Interactive Software Engineering (ISE) Inc., 2010.

[22] J Michael Spivey and JR Abrial. *The Z notation*. Hemel Hempstead: Prentice Hall, 1992.

[23] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[24] ProjectQ Team. Projectq. *Accessed on: April, 2020*, 2017.

Appendix

A.1 Scaffold

Scaffold [1] is a quantum programming language developed by a team from Princeton University and others. Using Scaffold, the computational operations and data structures involved in a quantum algorithm can be programmed and finally compiled into a machine-executable form. The team also developed Scaffold's compiler, ScaffCC, which optionally compiles Scaffold source code into instructions that can eventually be executed on QX, a quantum simulator developed by QuTech Labs.

The Scaffold is an extension of the C language. It adds new data types, such as `qbit` and `cbit`, and defines quantum operations, including Pauli-X gates, Hadamard gates, and other quantum logic gates. A program written with Scaffold comprises the *classical part* and the *quantum part*. The former includes classical data types and control structures; the latter includes quantum data types and operations.

A complete Scaffold program usually consists of multiple modules, and since quantum circuits are always "reversible," these modules must satisfy the following requirements to be executed on a quantum device.

- Either consists only of unitary quantum operations.
- Or be able to be compiled into unitary quantum operation instructions.

To compile some classical modules into unitary quantum instructions, Scaffold includes the CTQG module, which can compile some classical circuits into an instruction set consisting of only NOT, controlled-NOT (CNOT), and Toffoli gates. For example, when calculating the addition `a+b`, if `N`-bit binary numbers can represent both `a` and `b`, this classical instruction can be composed by `6N-3` CNOT gates and `2N-2` Toffoli gates without auxiliary quantum bits. Therefore, the operations such as classical addition computed by the user in Scaffold are eventually compiled and executed as a set of quantum operation instructions.

In the following, we briefly introduce what syntactic details Scaffold adds to the classical programming language.

A.1.1 Quantum Data Types

The most basic quantum data type in Scaffold is the quantum register `qreg`, which can be declared by the statement `qreg qs[n]`, where `n` denotes the number of quantum bits in the register; even if `n=1`, it is still treated as a quantum register, not as a separate quantum bit. Alternatively, multiple quantum registers can be declared as a single quantum structure `qstruct`:

```
qstruct struct1 {
qreg first[10];
qreg second[10];
};
```

In this case, quantum structure `struct1` contains two quantum registers: `first` and `second`. Through the following two statements, one can access the first qubit of `second` in the quantum structure `struct1`.

```
struct1 qst;
qst.second[0];
```

A.1.2 Quantum Gates

The quantum gate operation in Scaffold can be divided into two categories according to the implementation; one is the built-in quantum gates in the standard library; the other is the quantum gates defined by the gate prototype function. The use of the first type of gate function only requires the introduction of the header file `gates.h`.

According to the prototype function, the second type of gate operation requires its own definition.

```
gate gatename
(type_1 parameter_1, ..., type_n parameter_n);
```

The above statement defines the gate operation named `gatename`, which consists of `n` arguments, the data type of these `n` arguments can be quantum registers or classical unsigned integers, characters, floating point numbers, and double precision floating point numbers, both of which must be passed to the function by reference (if the classical data type with the `const` keyword can be passed by value). At this point, in the module that defines the gate operation `gatename`, the gate operation can be called according to the following statement:

```
gatename(parameter_1, ..., parameter_n);
```

```
//module prototypes. They are defined elsewhere
module U (qreg input[4], int n);
module V (qreg input[4]);
module W (qreg input[4], float p);

//Quantum control primitive
module control_example(qreg input[4]) {
if (control_1[0]==1 && control_2[0]==1) {
    U(input); }
else if (control_1{0}==1 && control_2[0]==0) {
    V(input); }
else {
    W(input); }
}
```

Fig. A·1 Example quantum control primitive for controlled execution of 3 different modules, U, V, and W [1]

A.1.3 Loops and Control Structures

Like the C language, Scaffold supports `if`, `switch`, and `loop` statements, but the control styles of these statements can only contain classic information. Scaffold also provides a quantum control statement; the control predicate contains qubits. When computing the control predicate statement, the quantum state of the qubit is determined, and the code is executed according to the result. For example, Figure A·1 shows a simple Scaffold program taken from [1], which describes quantum control primitive for controlled execution of 3 different modules. Obviously, this program will determine which code to execute according to the quantum states of control qubits `control_1[0]` and `control_2[0]`. According to the principle of delay measurement, this code will be compiled into the circuit shown in Figure A·2. Note that the qubits in the control predicate cannot be used again in the program.

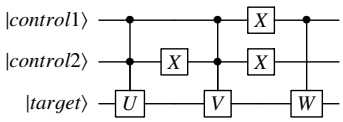


Fig. A·2 The quantum circuit corresponding to the program in Figure A·1.

A.1.4 Modules

The Scaffold supports modular design for readability, maintainability, and other features. A complete program usually consists of one or more modules, each of which is responsible for one task and can pass data of classical or quantum data types between modules. The syntax for defining a module is as follows.

```
return_type module module_name
(type_1 parameter_1, ..., type_n parameter_n);
```

where `return_type` can be null, integer, character, floating-point, double-precision floating-point, or structure, and the requirements for the argument list are the same as those for the gate prototype function. The defined module can be called by

```
module_name(parameter_1, ..., parameter_n);
```

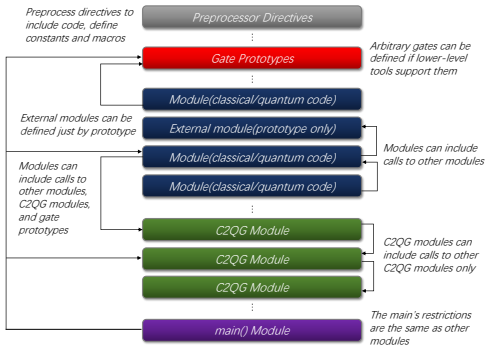


Fig. A·3 The whole structure of a Scaffold program [1].

Figure A·3 shows the whole structure of a Scaffold programs, taken from [1].

In this paper, we use ACSL [6] as our basis for designing ScaffML; we extend ACSL with a few new notations to specify Scaffold programs. To focus on the key ideas of ScaffML, in this paper, we do not consider how to specify Scaffold classical modules, which can be specified in the same way as ACSL for C.