Int J Parallel Prog (2016) 44:1028–1053
DOI 10.1007/s10766-016-0407-8

Hierarchical Synthesis of Quantum and Reversible Architectures

Archimedes Pavlidis[1,2] · **Dimitris Gizopoulos**[1]

Received: 22 October 2015 / Accepted: 14 March 2016 / Published online: 25 March 2016

Abstract Reversible hardware finds application in emerging areas such as low power circuit design, quantum computing, optical computing, and DNA computing. Intensive research has recently focused on the synthesis of quantum and reversible architectures. Quantum architectures often take advantage of reversible circuit synthesis methods but in general they require dedicated synthesis approaches because they represent a more general computing paradigm. Most of these quantum and reversible synthesis approaches derive efficient or even optimal circuits with scalability being their major drawback: they can only handle small circuits (up to a few hundred inputs for the most promising ones). In this paper, we propose a graph-based hierarchical synthesis method for large reversible and quantum architectures which can be combined with any of the existing synthesis methods to deliver unlimited scalability in synthesizing arbitrary large and irregular architectures. The specification of any complex function is provided in the form of a sequential algorithm consisting of primitive pre-synthesized operations available in a library. The components of the library may have been designed by ad-hoc methods or synthesized by the known methods in the literature or even by the proposed synthesis procedure. The synthesized architecture is represented as a dependence graph whose nodes correspond to the available components of the library and their respective inverses so as no garbage remains at the output. The method can be recursively applied at multiple levels to build any complex reversible or quantum architecture.

✉ Dimitris Gizopoulos
dgizop@di.uoa.gr

Archimedes Pavlidis
adp@unipi.gr

[1] Department of Informatics and Telecommunications, University of Athens, Panepistimiopolis, Ilissia, 157 84 Athens, Greece

[2] Department of Informatics, University of Piraeus, 80, Karaoli & Dimitriou St., 18534 Piraeus, Greece

Keywords Quantum circuits · Reversible circuits · Synthesis

1 Introduction

Reversible computation occurs when the information is not erased during the course of the processing and the input can be retrieved from the knowledge of the output [5]. Reversible computation can be modeled as a *reversible circuit* comprising of primitive reversible gates [30]. Reversible circuit designs find application in emerging technologies and computation paradigms, such as quantum computation [21], low power design [30], optical computing [12], DNA computing [36], etc.

Number factoring is an example where a quantum algorithm, known as Shor's algorithm [28], finds the factors of an integer in polynomial time whereas the best classical algorithm needs exponential time.

Quantum systems simulations and optimization problems are other examples where quantum computation can become advantageous [11]. Quantum computers apply unitary transformations on qubits and thus these transformations are inherently reversible. For this reason the design of quantum circuits can exploit reversible circuit design methodologies [21].

Low power consumption is one of the limiting factors affecting the trend towards miniaturization of conventionally designed irreversible electronic circuits. Excluding technology aspects of the circuit implementation there is a fundamental lower limit (Landauer limit [14]) of power consumption and this is related to the loss of information when information processing is done in an irreversible manner. For each bit of information loss at least $kTln2$ Joules of energy are dissipated, where T is the environment temperature and k is the Boltzmann constant. Although today the power consumption due to this fundamental limit is still much lower than the power consumption due to technology, some estimations show that in the next decade the Landauer limit will become the dominant factor [7]. Therefore, general procedures for the design of arbitrary large reversible circuits will support forthcoming conventional technology scaling.

We present a graph-based hierarchical synthesis methodology for arbitrary large and irregular *quantum and reversible architectures*. An architecture is prescribed as a sequence of elementary operations that correspond to existing quantum or reversible components of a library. The library can be populated with new circuits synthesized by the same or other methods in a multilevel hierarchical synthesis setup. We have successfully applied our methodology to a recently proposed complex realization of Shor's factorization algorithm and results confirm the flexibility and scalability of the proposed synthesis methodology. Our methodology can be applied to any important quantum or reversible algorithm and target physical technology.

2 Background and Related Work

2.1 Reversible Circuits and Gates

A reversible circuit is a logic combinatorial circuit implementing a function $f : B^n \rightarrow B^n$, where $B = \{0, 1\}$, n is the number of input and outputs and the function f is a bijection or equivalently a permutation. This means that there exists the inverse function $f^{-1} : B^n \rightarrow B^n$ such that for each x giving $f(x) = y$ then $f^{-1}(y) = x$. A reversible circuit is constructed by combining *logic reversible gates* selected from a reversible library. Usually, libraries used to build reversible circuits contain some or all of the following reversible gates: NOT, CNOT and Toffolii. Reversible circuits can be built with ad hoc techniques, e,g, "by hand", or through an automated synthesis method, when adequate specifications of the desired function are given.

Sometimes the function to be implemented in a reversible circuit does not have equal number of inputs and outputs or it is not bijective, thus it is *irreversible*. Such an irreversible function has the form $f : B^n \rightarrow B^m$ where $m \leq n$ and/or there exist $k >1$ input vectors $x_i \in B^n$, $i = 1 \ldots k$ mapped to the same output combination, that is $f(x_1) = \cdots = f(x_k)$. An irreversible function can be transformed to a reversible one by embedding it into another constructed reversible function g: $B^{n+c} \rightarrow B^{m+g}$ having c additional inputs and g additional outputs. The reversibility requirement of equal number of inputs and outputs leads to the relation $n + c = m + g$. The n inputs and m outputs are the primary ones, while the additional g outputs are the garbage ones. The possible addition of constant variables due to the requirement of the addition of garbage outputs leads to the increment of wires carrying these variables in the implemented circuits. These wires are the ancillae of the circuit and they must be reset back to a known constant state, usually the 0 state, in order to be reused later as a constant input to a larger circuit. The number of the wires in a circuit is a valuable resource, especially in the context of the quantum computation, and for this reason effort must be done to reduce the ancilla used. Garbage qubits are qubits which can't be re-used as ancillae in subsequent computations. Our synthesis algorithm eliminates garbage (except for the input arguments) without using excessive ancillae.

2.2 Quantum Circuits and Gates

Implementations of quantum gates in various technologies are restricted to a small set of elementary one-qubits or two-qubits only. Commonly used quantum gates are the one qubit X, Y, Z gates, the Hadamard gate H the S and T gates and the two qubits *CNOT* gate. An n-qubits quantum gate U can be decomposed [3,6,27,32] (or approximated with arbitrary accuracy) into a sequence of elementary gates acting on different qubits each instant of time.

While a reversible circuit operates on classical bits, e.g. on variables taking discrete values 0 or 1, a quantum circuit operates on qubits taking values in a continuous range (namely the surface of a sphere called Bloch's sphere). Moreover, the reversible logic gates are a subset of the quantum gates (the reversible gates can be described by matrices having elements the integers 0 or 1). Nevertheless, quantum circuit synthesis can exploit known reversible circuit techniques as many quantum algorithms use arithmetic and logic operations (they are Boolean). An example is Shor's algorithm whose main parts are: (a) a modular exponentiation computation and (b) a quantum Fourier transform (QFT). The former part can be described in integer arithmetic terms on the computational basis and, therefore its construction can take advantage of reversible synthesis techniques, something that cannot be applied to the latter part of the QFT. In the former case a reversible circuit implementing the function can be invoked and then a transformation to the quantum circuit can be applied using the available quantum gates.

Among the different physical layer technologies that are being investigated, the ones that will prevail for large-scale quantum circuit implementations will be those that satisfy certain requirements. One of the stringent requirements is the scalability of the technology with the number of qubits (e.g. memory size) required for a particular algorithm. Qubits increase requires the employment of fault tolerance mechanisms that will deal with the problem of decoherence. The ion-trap technology for the implementation of large quantum circuits has been studied in [2,13,19,20,29] and such proposals of quantum architectures show that large scale quantum computers can be feasible in the near future. The programming (e.g. sequence of quantum operations or equivalently connections between quantum gates) on such large scale architectures requires *synthesis* algorithms capable to handle large quantum circuits. Our work contributes to this direction and is applicable to any physical technology that will eventually be used in large-scale quantum computers.

2.3 Reversible and Quantum Synthesis

In general, the reversible synthesis methods can be divided in two families: (a) optimal or asymptotically optimal and (b) heuristic. The former methods result in a circuit that minimizes a particular cost factor which is usually the number of gates. Optimal methods are practical for a few bits only (e.g. 3 or 4 bits) as they demand exponentially grown memory and time as a function of inputs [8,23,26]. Heuristic synthesis methods behave better referring to the bits handling capacity at the cost of relaxing the optimality requirement. Transformation [17,18], search [10], cycle [25] and Binary Decision Diagrams (BDD) [33] based methods fall under the latter category. A thorough overview can be found in [24]. In general, most of the methods suffer from limited scalability: they do not handle large circuits of more than 100 bits due to restrictions of memory and runtime as they consume exponential resources in arbitrary examples cases.

Quantum synthesis differs from reversible synthesis in the specifications and the libraries used to synthesize the circuit. Boolean specifications in the computational basis are adequate when the target circuit is an arithmetic one or logical one due to the linearity and the superposition principle. Thus, reversible circuit methodologies can be used and then library transformation can be applied to convert from the reversible library to a quantum library. When the specifications are in the form of a unitary matrix U of dimension $2^n \times 2^n$ for a circuit consisting of n qubits then decomposition methods can be applied [3,6,27,32]. In such methods the unitary U is decomposed in a sequence of one-qubit and two qubit gates where the specific gates depend on the library. Gates number is exponential in n.

As discussed above the various existing quantum and reversible synthesis methods need exponential computing resources and thus they do not scale well for large circuits. A step towards synthesis of large circuits would be the combined use of hierarchical methods where the circuit, being reversible or quantum, is built level by level using already synthesized functions of smaller blocks. These blocks can be ad-hoc designed if they correspond to well known circuits involving regularity (e.g. adders or other arithmetic circuits), automatically synthesized by a synthesis method chosen by the user, or synthesized by the proposed method if a bottom up design is needed to handle a complex specification. To our knowledge only a few hierarchical synthesis method have been presented in the literature [1,9,16,34,35]. Our hierarchical synthesis method applies to both reversible and quantum circuits and eliminates intermediate bits/qubits without excessive ancilla usage. It also supports unlimited circuit size handling capability on an existing library of components. It applies to the quantum case whenever a classically defined "oracle" arithmetic function must be embodied in the quantum algorithm.

3 Methodology Basics

In this section, we present a set of interoperating algorithmic routines which synthesize a complex reversible or quantum circuit using abstract functional blocks. Figure 1 outlines the basic steps of the methodology. The abstract blocks are part of a library and are assumed to be already synthesized (using our method or other lower level methods). The specifications of the target architecture come in the form of sequence of arithmetic/logical instructions. The circuit is synthesized progressively in three steps. In the *forward synthesis* part a directed acyclic graph representing the required computations by the specifications is built by interconnecting various library blocks and possibly by adding ancillae (whenever temporary variables are used), without

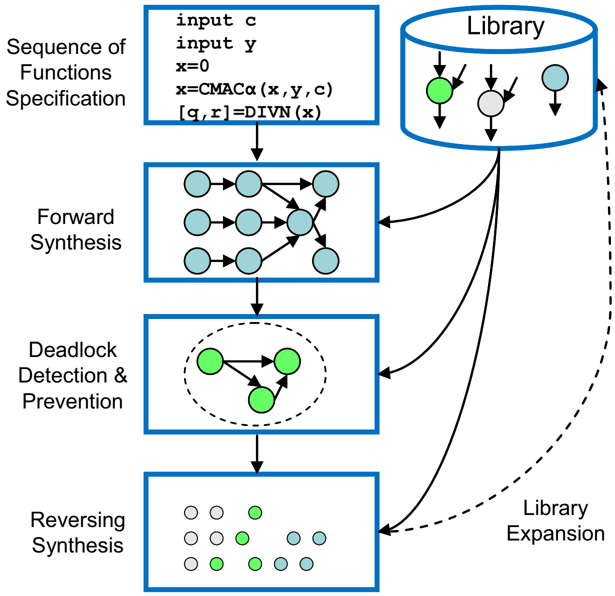


Fig. 1 High level description of the proposed synthesis methodology

taking into account the resetting of the ancillae. Subsequently, possible *deadlocks* which prevent the next step are detected and eliminated. The last step, *reversing*, is the expansion of the graph so as to reset the ancilla states.

3.1 Initial Specifications and Library

We consider sequences of arithmetic and logical operations describing the reversible circuit or the quantum oracle of the general form:

$$x = f_i(x, b) \tag{1}$$

Function f_i affects only one of its two input variables x and b. Index i is an identifier used to distinguish among the various available functions in the library. Variables x and b are integers of n_x and n_b bits, respectively. We call variable x the *affected variable* and b the *control variable*. There are special cases of elementary functions that fall under the description of Eq. (1). In the simplest case there is no control variable ($n_b = 0$) in the computation of a primitive assignment such as $x = \text{NOT}(x)$. In other cases the bits of the variables are partitioned into sets, where each set has its own index as shown in Eq. (2).

$$\left[x_1^{(out)}, \ldots, x_k^{(out)}\right] = f_i\left(\left[x_1^{(in)}, \ldots, x_l^{(in)}\right], [b_1, \ldots, b_m]\right) \tag{2}$$

In this case, output variable x (denoted as $x^{(out)}$) is partitioned in k subsets of bits, each one indexed as variable $x_i^{(out)}$, $i = 1, \ldots, k$ and consists of n_{x_i} bits. Similar description applies to variables $x^{(in)}$ and b. As an example consider the function of dividing a $2n$ bits affected input variable by a constant integer resulting in an n bits quotient and an n bits remainder whenever the quotient is less than 2^n. In this case $k = 2$ and $l = 1$ and the bits representing the dividend become the quotient and remainder bits of the output.

We assume that a library of quantum or reversible subcircuits implementing the classical functions in the form of Eqs. (1) or (2) like the one proposed in [15] is available. This library can also be viewed as the instruction set of a *quantum arithmetic logical unit* (QALU) and the result of our synthesis procedure can be viewed as the sequence of executions of the quantum instructions to various quantum registers of the QALU. An example quantum library of arithmetic and logical functions is given in Table 1. Various quantum circuit representations of these functions can be found in the literature or can be synthesized with known methods. The library can be arbitrarily extended by including more complex functions or even new functions synthesized by the algorithm described in this paper or other methods.

Each function shown in Table 1 transforms the quantum state (*input state*) of a qubits collection to another quantum state (*output state*). This transformation depends on the state of some other qubits which remain unaltered. We call the qubits that get transformed *affected* and the qubits that remain unaltered but influence the affected

Table 1 Example functions of a quantum library

Quantum function	Affected qubits			Control qubits	
	Input state	*Output state*	*Size*	*State*	*Size*
INP_F	–	0 or x	n	–	0
NOT	x	$\sim x$	n	–	0
CNOT	x	$x \oplus b$	n	b	n
COPY	$x = 0$	$0 \oplus b = b$	n	b	n
ADDC$_a$	x	$x + a \pmod{2^n}$	n	–	0
CADDC$_a$	x	$x + ca \pmod{2^n}$	n	c	1
ADD	x	$x + b \pmod{2^n}$	n	b	n
CADD	x	$x + cb \pmod{2^n}$	n	b, c	n, 1
MAC$_a$	x	$x + ab \pmod{2^{2n}}$	$2n$	b	n
CMAC$_a$	x	$x + cab \pmod{2^{2n}}$	$2n$	b, c	n, 1
DIV$_a$	x	$x/a, x \mod \alpha$	$2n$	–	0
OUT_F	x	x	n	–	0

qubits *control* qubits, in correspondence to the affected and control variables of function f in Eq. (1), respectively. In Table 1, for each quantum function shown in the first column the following columns show the number of affected qubits (size), their initial state and the output (transformed) state. The last columns show the state and the number of the control qubits. Since the state of the control qubits remains unaltered, Table 1 does not distinguish between input and output states for these qubits. The inverses of the functions of Table 1 (which are inherently reversible) are not shown, but are also included in the library.

3.2 Quantum Dependence Graph

The synthesized quantum circuit is represented as a directional acyclic graph (*Quantum Dependence Graph* or *QDG*) consisting of nodes corresponding to the quantum subcircuits (blocks) of the library functions and of arcs corresponding to the individual qubits or groups of qubits (qubit buses) connecting these blocks.

Figure 2 clarifies with an example the notation and the labels used in a QDG. Figure 2a is a quantum addition circuit block in standard notation implementing the function ADD (x,y). Qubits of input y remain unaffected at the output as the block is reversible and thus they correspond to control qubits. Figure 2b is the same block in a more compact form with the qubits organized in buses and connected in different ports. Figure 2c represents the same block as a QDG node and its incoming and outgoing arcs. Attached to the node are the `type` label which is equal to value ADD and the id label (it depends on the relative position in a particular QDG). The bottom left arc corresponds to the qubits carrying the x state and has three labels attached: The width of the arc (`width`) which is equal to the number of the qubits n, the port destination label `dest` which is =1 as there is only one affected input port for this block and the port source information (`source`) which depends on other nodes of the QDG.

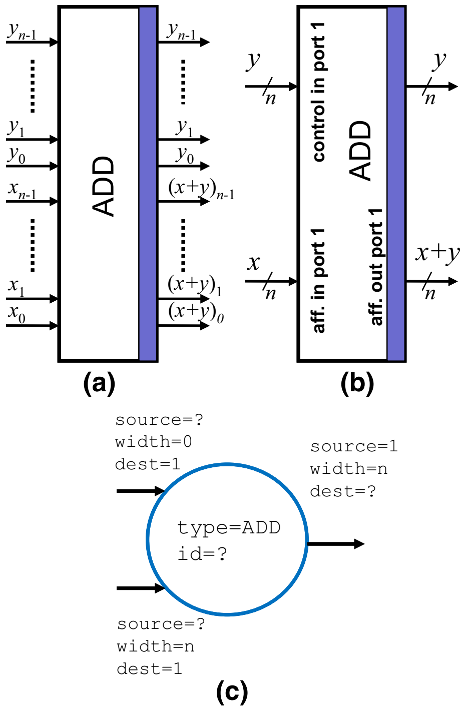


Fig. 2 Representation of a quantum functional block in the QDG notation. **a** Functional block showing all the qubits taking part in the operation along with their input and output states, **b** the same block with the qubits organized in buses connected to ports, and **c** the abstract notation of the same block as a node with arcs and their labels. The question marks mean that the respective label depends on the specific connections of the node relative to the other nodes of the QDG

Similar remarks hold for the rest of the labels attached to the other two arcs (control input arc and affected output arc).

The labels `id` and `type` can be represented by integers, for each type there is its negative type which corresponds to the block performing the inverse function. Also, the constant parameters of some of the blocks (e.g. parameter α of the function MACa) are assumed to be included in the `type` label. Later, a third label named `anc` (integer with values 1 or 0) will be used also.

The affected qubits (or affected qubit buses) transformed by a node are represented by arcs incoming to and outgoing from that node. Each affected arc, either incoming or outgoing, must also include the port source (arc tail connection) and port destination (arc head connection) information, because some nodes may have more than one input and/or output qubits buses and we need to distinguish the various possible ports of each node.

Similarly, the input control qubit buses of a quantum function are represented as incoming arcs to the corresponding node. Control arcs always have a width of 0 (no matter their real qubits width) so as to be distinguished from affected arcs. The tail of control arcs emerge from affected qubits output ports of an ancestor node. As control qubits are not altered by any node they entered, there is no need to show their exit by an outgoing arc. Similarly with the case of the affected qubits arcs, we need to include in each control arc the port source and port destination information.

Input nodes of the graph represent initial qubit states (quantum variables passed to the quantum algorithm represented by the QDG) or ancilla qubits initially set to a

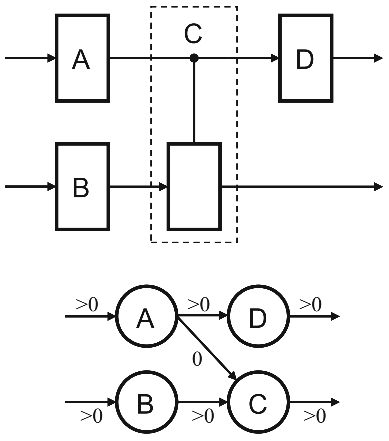


Fig. 3 Mapping between the standard notation (*top*) and the QDG notation (*bottom*). Affected arcs have width > 0, while control arcs have width $= 0$

zero state. For both cases, a node with the special type INP_F is used in the graph. Similarly, output nodes of the graph represent output qubits states (final results or ancilla states). The ancilla states correspond to garbage qubits states or ancilla qubits reset back to their initial zero state when the reversal procedure described later is applied. The output nodes are represented with a node of the special type OUT_F which acts as an identity node.

In some of the functions, the control qubits are grouped in different states variables. The same applies for the affected qubits in some of the functions. As an example, the controlled adder CADD has the two groups of control qubits b and c, of n qubits and one qubit, respectively. Also, the divider function DIV has $2n$ qubits wide input state, but the output state is grouped in two qubits buses of n qubits, namely the quotient and the remainder. In general, we allow such qubit grouping in the functions of the library because it facilitates the initial specifications.

The separation of the qubit buses into affected and control ones simplifies the internal representation of the circuit and the workings of the synthesis algorithm, especially the deadlocks detection developed later. The restriction that a node control input does not exit the same node does not contradict the standard notation of a quantum circuit; it is just a remapping of the notation shown in Fig. 3.

4 Forward QDG Synthesis

The first phase (see Fig. 1) of the QDG construction is dedicated exclusively to the forward computations, without taking into account the resetting of the possible ancillae qubits. This step is trivial and will be explained in short.

4.1 Representation of Classical Algorithm

Dependencies among the sequence of functions of the classical algorithm to be mapped as a quantum oracle exist when a variable in the list of affected output variables of a function is used as an input variable (affected or control) in a subsequent function. These dependencies will be reflected in the QDG through the use of an arc connecting two nodes. Initial values and input variables of the algorithm correspond to affected output ports of INP_F nodes, while the variables giving the final results (desired and garbage) of the algorithm correspond to affected input ports of OUT_F nodes. Intermediate variables (temporary) used in the algorithm for the calculations of the final results correspond mainly to ancilla qubits.

An arbitrary classical algorithm using elementary functions of the form of Eq. (2) can be equivalently described by arrays of integers and arrays of lists of size L, where L is the total number of functions comprising the algorithm. An integer array `type` describes the type of each function, arrays of lists `p`, `m` and `c` describe the lists of affected output, affected input and control input variables, respectively. Another array of integers, named `w` describes the number of bits used by each variable. Last, array `res` will discriminate which of the variables used in the algorithm are the desired final results and which are intermediate temporary variables. This last array definition is crucial for the final phase of the synthesis algorithm whose purpose is to reset intermediate garbage.

4.2 Forward Synthesis Algorithm

The main data structures used in the synthesis of the forward computations QDG are the graph structure itself, named `forwQDG`, and the arrays `type,p,m,c,w` and `res` describing the classical algorithm mentioned in the previous subsection.

The purpose of forward synthesis algorithm is to add nodes to the `forwQDG` (initially null), one for each function found in the classical algorithm and connect them with affected and control arcs based on the dependencies between the variables. In brief, the synthesis algorithm of the forward QDG consists of the steps shown in Table 2 and explained below. An example of a part of forward QDG built by such an algorithm is shown in Fig. 4.

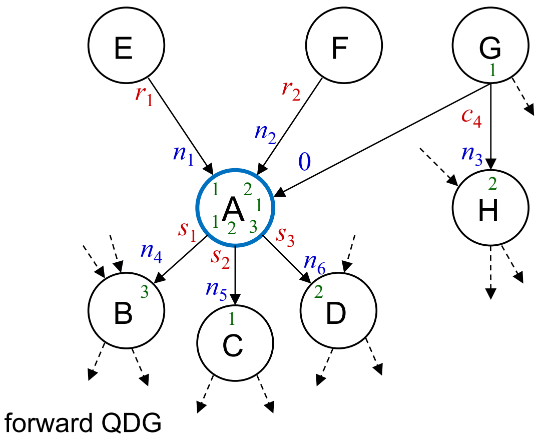
The synthesis algorithm executes the for loop of size L (lines 1, 5). For each integer `l`$(\mathtt{l} = 1 \ldots L)$ the following steps build gradually the forward QDG:

Line 2 Add a new node in the forward QDG. This node has a type label equal to the type of the function (`type[l]`). A new node id is assigned sequentially for each node added.

Table 2 Forward QDG synthesis algorithm

Operations	
1:	FOR each line `l` DO
2:	Add node
3:	Add incoming control arcs to node.
4:	Add incoming affected arcs to node
5:	END FOR
6:	Add terminal (`OUT_F`) nodes
7:	Record the terminal nodes carrying garbage in a list

Fig. 4 Part of an example forward QDG node. Attached at the tail of the *solid arcs* is the output state and at the head of the arcs is the width of the arc (0 for control arc). Inside the *circles* of the nodes the port numbers for each case of affected input, affected output and control input arcs are shown



Line 3 Scan the list of control input variables `c[l]` of this function. Then for each control variable in the list, find every function `k` that includes this variable in its output variable list `p[k]` and connect with an arc the two respective nodes of the QDG corresponding to these two functions, `l` and `k`. As the arc connecting the two nodes is related to a control input connection add a width label of value 0 on the arc. Add source and destination port labels reflecting the input and output ports that are connected by this arc. The position of the variable in the lists `p[k]` and `c[l]` is the source and destination port number, respectively.

Line 4 Scan the list of affected input variables `m[l]` of this function. Then for each affected input variable find every function that includes this variable in its output variable list `p[k]` and connect with an arc the two QDG nodes which correspond to these two lines, `l` and `k`. Add labels on this arc reflecting the number of the qubits carried by this variable (`w[m[l]]`) and also the source and destination port similarly to *Line* 3 above. If no line with such an output variable is found then add a new node of type INP_F and make the required connection with the relevant labels (This means that the input variable is an input argument to the classical oracle to be synthesized).

After the execution of the loop, two more steps are necessary to prepare the reversing phase of the synthesis:

Line 5 For all unmatched affected output variables (this means that the variable is a final desired result or an ancilla output) add a new node of type OUTP_F and make the required arc connection assigning the relevant width and port labels.

Line 6 Record in a list (`GarbageTermList`), the terminal node ids of the forward QDG which carry garbage results, that is non desired final results. This discrimination is based on the array `res` mentioned in the previous subsection.

5 Reversible QDG Synthesis

The final phase of our synthesis algorithm transforms and expands the forward QDG so as to reset all ancilla qubits back to their initial constant states.

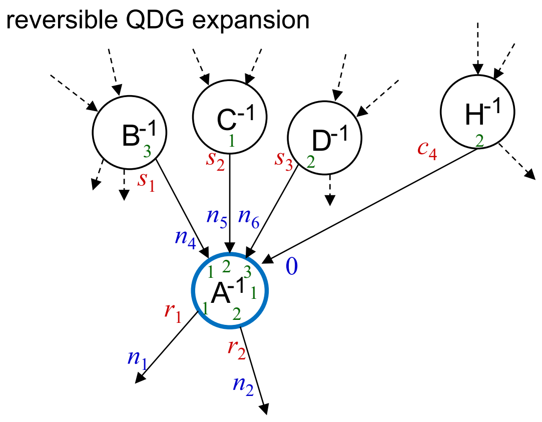


Fig. 5 Inversion of node A of the example forward QDG shown in Fig. 4. Legend of arc and node labels is similar as that of Fig. 4

5.1 Node Inversion

The un-computation of the state of ancilla qubits back to a constant state can be achieved by successively inverting the states of ancilla qubits that appear at the output of a terminal node through all the nodes up the respective input nodes affecting these ancilla qubits. These terminal nodes have been recorded in `GarbageTermList`.

To understand the requirements and the procedure to invert the state of the affected output qubits of a node in a QDG, we refer to Figs. 4 and 5 which represent a segment of an example forward QDG and its expansion (called `revQDG`), respectively. The purpose of the expanded QDG is to uncompute the garbage ancilla states.

In Fig. 5, nodes A^{-1}, B^{-1}, C^{-1}, D^{-1} and H^{-1} are the nodes required to invert the output states of node A of the forward QDG in Fig. 4. Inverting states s_1, s_2, s_3 of the output qubits of node A means to transform them into the states r_1 and r_2. Figure 4 shows that node A receives as affected inputs two arcs (with widths n_1 and n_2) from nodes E and F being in states r_1 and r_2, respectively. These are controlled by state c_4 (arc of width 0 emerging from node G) and transformed into the output states s_1, s_2, s_3. The inverse transformation is realized by another node of the reverse QDG, namely node A^{-1}, which is the inverse of node A (as we have previously mentioned, the quantum library contains the inverse of every function as well). So, if the affected input ports of node A^{-1} are fed with the states s_1, s_2, s_3 and its control input is fed with the state c_4 it is obvious that the required inversed states r_1 and r_2 become available at the affected output ports of node A^{-1}.

This inversion implies that states s_1, s_2, s_3 and c_4 must be available. In the example we have shown for the forward QDG states s_1, s_2, s_3 have already been processed by the successor nodes of A (nodes B, C, D) and the state c_4 has been processed by node H. This necessitates the inversion of nodes B, C, D and H before the inversion of node A.

The incoming arcs connections to node A^{-1} are as follows. Node A has two ports of affected input qubits, namely 1 and 2 of respective qubits width n_1 and n_2, and a unique control input port labeled as 1. Moreover, it has three affected output ports (1, 2 and 3) of widths n_4, n_5, and n_6, respectively. The ports of the affected input qubits of the inverted node A^{-1} are the ports of the affected output qubits of node A and vice versa. Control port 1 of A^{-1} corresponds to the same port (number 1) of node A. When connecting the incoming arcs of node A^{-1} the algorithm needs the extra information of which ports are engaged in these connections.

5.2 Global Considerations

The above per-node inversion procedure must be applied by taking global considerations into account. Some prerequisites and constraints are the following:

- *Selection of nodes which require inversion* Only some of the forward computation QDG nodes need to be inverted. The reversing algorithm must select and label the nodes of the forward computation that need inversion (using label `anc` with values 0 or 1 attached at each node). The reversing algorithm begins from the nodes listed in `GarbageTermList` and recursively marks all the internal nodes of the forward QDG that have a path connection to these leaf nodes as the nodes that require inversion. These paths must comprise exclusively of affected arcs, i.e. it marks only the ancestors of the output nodes that directly transform the final ancilla state. The nodes which need inversion will be called *ancilla nodes* and their `anc` label is set to value 1 (the rest of the nodes have a value 0 and will be called *non ancilla nodes*). This prerequisite to mark the ancilla nodes of the forward QDG will be taken into account later, at line 1 of the reversing algorithm shown in Table 6.
- *Sequence of inversion* The algorithm must check that each candidate node for inversion is ready and allowed for this operation. Only a subset of the ancilla nodes is able to be inverted at each instance. This is due to the existence of data dependencies between the various nodes. In the example of Fig. 4, node A can be inverted only if its children nodes are already inverted (in case they were ancilla nodes) because node A^{-1} requires the states s_1, s_2, s_3. These states are not available as the forward computation has already transformed these states by applying nodes B, C and D. The readiness condition just described is given in line 8 of the reversing algorithm in Table 6.
  Even if an ancilla node is ready for inversion, a postponement of this action may be necessary. This may happen if this blocks the inversion of other nodes. E.g. the candidate node for inversion has a control arc connection towards another ancilla node which is not yet inverted. The purpose of updating the `GarbageTermList` in line 11 of Table 6 is exactly this.
- *Tracking of intermediate states* The inversion algorithm needs to keep track of which forward QDG node output state corresponds to the new output states computed by each new inverted node. Referring again to Fig. 5, when node A^{-1} is added to invert node A, the necessary information of where to find the states s_1, s_2, s_3 must have been recorded. These states have been computed in a previous step of the algorithm when the inversion of nodes B, C and D took place, by adding nodes B^{-1}, C^{-1} and D^{-1} and their corresponding arcs, so these latter nodes can supply the required states s_1, s_2, s_3.

Table 3 Node inversion algorithm

Operations	
1:	ADD New Node of type A^{-1} in `revQDG`
2:	ADD affected arcs in `revQDG` from nodes of `RevInfo`(A) list towards node A^{-1}.
3:	FIND the ancestors $F_1, \ldots, F_m$ of node A that have affected arc connections with A.
4:	UPDATE the lists `RevInfo`(F_1), ..., `RevInfo`(F_m) with the new added node A^{-1}.
5:	FIND all the nodes $CA_1, \ldots, CA_n$, which control node A..
6:	ADD control arc connections in `revQDG` from nodes $CA_1, \ldots, CA_n$ to node A^{-1}.

An array of lists named `revinfo` is used for this purpose. A list is assigned to each node of the `forwQDG`. Initially, the lists corresponding to the terminal nodes of the `forwQDG` contain as records the same terminal nodes (meaning that the output states of the terminal nodes are available at the nodes themselves), while the `revinfo` lists of the internal nodes are empty (meaning that the states of the internal nodes of the `forwQDG` are no longer available as the forward computation has proceeded to the end). Additionally, the records of a list contain arcs information such as the source port, destination port and width. During the inversion of the nodes procedure, as the `revQDG` is expanded, states of internal nodes of the `forwQDQ` gradually become available (new terminal nodes in the `revQDG` appear) while states of other nodes are no longer available (they become internal nodes of the `revQDG`). Therefore, the revinfo lists must be updated anytime a node is inverted and this update becomes part of the node inversion algorithm shown in Table 3.

- *Deadlocks resolution* There are cases where it is impossible to invert a given forward computations QDG without applying a certain transformation on it. These cases are again related to the data dependencies imposed between two nodes which prevent the inversion of another node. Treatment of these situations is described in detail in the following subsection.

5.3 Deadlocks Resolution

There are two possible cases that can lead to deadlock of the reversing algorithm. The two cases are depicted in Figs. 6 and 7. In both cases we assume that we have already marked all the nodes of the QDG as ancilla or non ancilla nodes.

The sequence of deadlock resolution algorithms is to first apply the algorithm for the second type and afterwards the algorithm for the first type, as it is possible the revocation of a second type deadlock to generate a first type deadlock, but not vice versa.

Both deadlock resolution algorithm use additional ancilla bits/qubits and copy the output port states of the nodes that cause the deadlocks to these new ancillae. This is a

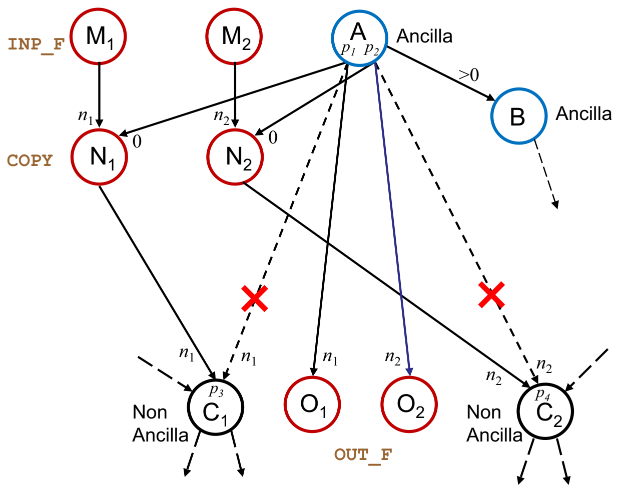


Fig. 6 First type of deadlock resolution. Nodes A and B are ancilla, nodes C_1 and C_2 are non-ancilla and M_1,M_2,N_1,N_2,O_1 and O_2 are the nodes added to prevent the deadlock. Next to each *arc* is shown its width. Ports are shown inside the *circles* of some nodes

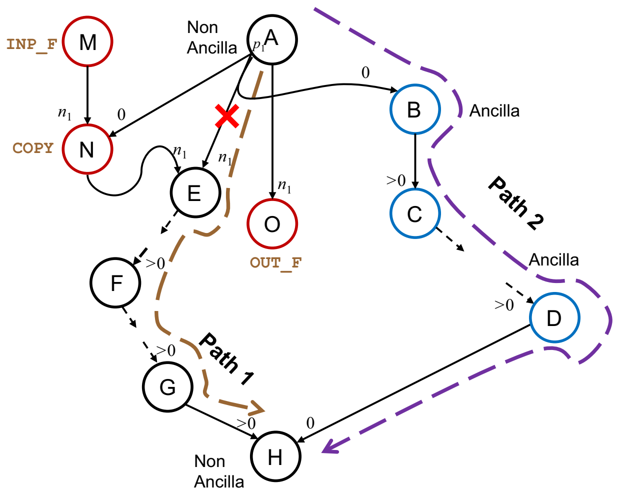


Fig. 7 Second type of deadlock. Nodes A, E, F, G, H are non -ancilla whereas nodes B, C, D are ancilla. Nodes M, N, O added to prevent the deadlock. Width of each *arc* is shown

bitwise copy operation which is simply performed with CNOT gates having as control the bit state to be copied and as target the new ancilla in the zero state.

Although a real copy operation is not permitted in the quantum context due to the “no-cloning” theorem, the purpose here is to reset an intermediate ancilla state back to zero. If someone analyzes the global operation of the circuit he can see that as long as the garbage states ends up in state 0 for every basis input state then the same holds for every superposition of them. That is the garbage states become 0 and disentangled from the other states. Consequently, the circuit operation is the desired one when analyzed globally (see Sect. 7 for more details).

In both deadlock cases I and II the drawback of adding more ancilla qubits can be counterbalanced by the reversing of other ancilla qubits which otherwise would be garbage qubits and could not be reused. This point will be further analyzed in Sect. 7.

5.3.1 Deadlock Type I

The handling of the first deadlock type is depicted in Fig. 6. Nodes A and B are marked as ancilla and are connected by an affected arc (width greater than zero). Nodes C_1 and C_2 are marked as non-ancilla and the arcs connecting node A with them have also a width greater than zero (n_1 and n_2) and emerge from different ports of node A, namely p_1 and p_2. It is the affected arc between nodes A and B the fact that gives the ancilla property to node A and consequently the necessity to invert it; its other children connections are towards the non ancilla nodes C_1 and C_2.

The deadlock condition then arises due to the fact that a node like A has affected arc connections to both ancilla and non ancilla children. It is impossible to invert the ancilla node A as this requires the prior inversion of nodes C_1 and C_2, but these nodes must not be inverted as they are non ancilla nodes and their output results must remain unaltered up to the end of the computations. This kind of deadlock can be prevented by "copying" the qubits emerging from ports p_1 and p_2 and thus releasing node A so as its output states can be inverted as desired. The detailed required actions to revoke such a deadlock case are described in Table 4 and Fig. 6.

5.3.2 Deadlock Type II

The second type of deadlock is illustrated in Fig. 7. Node A which is non-ancilla controls via an arc (width 0) emerging from port p_1 an ancilla node B. As the latter node is an ancilla node it must be inverted. This inversion need means either that the forward computations must not have proceeded beyond node A (towards nodes E, F, G and H) or that the output state of port $p1$ of A must be available somewhere else in the QDG.

The deadlock condition arises whenever a path like A $\rightarrow$ E $\rightarrow$ F $\rightarrow$ G $\rightarrow$ H (Path 1 in Fig. 7) consists exclusively of affected qubits arcs (width greater than zero), the nodes belonging to the path are non-ancilla and simultaneously exists a second path from B to H (Path 2 in Fig. 7) where the subpath B $\rightarrow$ D consists of affected arcs while the last arc D $\rightarrow$ H is supposed to be a control arc. The nodes belonging to path B-D are assumed to be ancilla nodes. Both paths must emerge from the same output port (shown in Fig. 7 as p_1) of node A.

If the last arc D $\rightarrow$ H wasn't a control arc then this case could be handled by the resolution of type I deadlock because in such case node D, being ancilla node, would have another outgoing arc of width greater than zero leading to another ancilla node. For the same reason, node A is supposed to be a non ancilla node, otherwise we would face an ancilla node having both ancilla and non ancilla children connected through affected arcs.

Table 4 Detection and resolution of deadlock I algorithm

```
Operations
1:  FOR each ancilla node A of forwQDG DO
2:      IF node A has affected arcs connections to both ancilla and
        non anciila children nodes THEN
3:          FOR each affected outgoing arc $i$ emerging from node A
            (port $p_i$) and leading to a non-ancilla node $C_i$ DO
4:              ADD a new node, $M_i$, of type INP_F initializing $n_i$
                qubits in zero state. The number $n_i$ is the width of the
                arc $A \rightarrow C_i$.
5:              ADD a new node, $N_i$, of type COPY whose purpose is
                to copy the port $p_i$ output state of node A.
6:              ADD an affectecd arc connection from node $M_i$ to
                node $N_i$ of width $n_i$ (the number of qubits to be
                copied). The port information attached to this arc is
                trivial (1 for both the tail and head) as an INP_F node
                has only one output port and a COPY node has only
                one affected input port.
7:              ADD a control arc connection (width 0) from node A
                to node $N_i$. The tail of this arc is port $p_i$ of node A and
                the head is port 1 as a COPY node has only one
                control input port.
8:              ADD an affected arc connection from node $N_i$ to node
                $C_i$. Arc's destination port is the destination port of arc
                $A \rightarrow C_i$ and its width is $n_i$. The source port of this arc is
                again 1 as only one output port exists on any COPY
                node.
9:              REMOVE the arc connecting node A with node $C_i$.
10:             ADD an OUT_F node.
11:             ADD arc connection from port $p_i$ of node A to node
                OUT_F.
12:             MARK node OUT_F as ancilla node.
13:         END FOR
14:     END IF
15: END FOR
```

This second deadlock condition can be justified for the following reasons. If the inversion of node B is done prior the advancement of the forward computations beyond node A (towards node H) then the output state of node D will not be longer available and the computation on node H could not be done. On the other hand as explained above, if the forward computation has advanced up to node H (so as node D can be inverted) then the inversion of node B cannot be done as the output state of node A is no longer available.

An algorithm for detection of second type of deadlocks has been developed and is briefly described in Table 5. A detected deadlock can be revoked with similar actions as those of the first deadlock case, that is addition of nodes M, N, O and some

Table 5 Detection of deadlock II algorithm

```
Operations
1:  FOR each non-ancilla node A of the forwQDG DO
2:      Generate a list L0 with the non ancilla children of A
3:      Generate a list L1 with the ancilla children of A
4:      FOR each node C1 of forwQDG in L1 DO
5:          FOR each node C0 of forwQDG in L0 DO
6:              IF source(arc(A,C1))=source(arc(A,C0)) THEN
7:                  nonAncS=DFS1(C0); p=source(arc(A,C0)
8:                  FOR each S in nonAncS DO
9:                      path2=FindPath(C1,S)
10:                     IF path2≠NULL AND all arcs of path2 are
                        affected except the last one THEN
11:                         Deadlock 2 found at port p of node A
12:                     END IF
13:                 END FOR
14:             END IF
15:         END FOR
16:     END FOR
17: END FOR
```

rearrangement of arcs as depicted in Fig. 7. Detailed resolution operations are not exposed in Table 5 due to the similarity with first case.

The detection of the second type is as follows: Every non ancilla node A is checked for engagement in a possible deadlock (lines 1, 17). For each such node a list, L0, consisting of its non ancilla children is built (line 2) and another list, L1, consisting of its ancilla children is also built (line 3). The purpose of the double loop defined in lines 4, 16 and 5, 15 is to check if two arcs emerge from the same port p of node A towards an ancilla and a non ancilla node (lines 6, 14). This is a prerequisite for the deadlock of the second kind and this condition corresponds to the arcs A→E and A→B in Fig. 7. If such a condition is fulfilled then another list, nonAncS, that contains non ancilla nodes is built (line 7). This list contains only the non ancilla successor nodes of node C0 and a modified Depth First Search procedure can be applied for this retrieval. This modified search procedure traverses only the affected arcs (the ones with their width greater than zero). Now, every path from node A to each node S of the list nonAncS corresponds a path similar to Path1 in Fig. 7. The final check is to find if a second path exists from node C1 to any of the nodes recorded in list nonAncS (lines 8–13). This path must be composed of affected arcs only except the last one (line 10) and this could correspond to Path 2 in Fig. 7. If this final condition is true (line 11) then a procedure similar to that one exposed in Table 4 is applied to port p of node A which causes the second kind of deadlock.

5.3.3 Uniqueness of the Two Deadlock Conditions

The previous two deadlock types are the only ones that can arise. This can be justified if we examine all the possible connection cases of an ancilla node, e.g. B, which must be inverted. The necessary conditions to invert node B are: (1) its incoming control states be available at the instance of inversion and (2) its outgoing output states be also available, as explained previously.

The first condition means to investigate the possible cases of ancestor nodes of B that have control arcs connected to it. There are two cases: (1a) an ancestor node A is a non ancilla node and (1b) an ancestor node A is an ancilla node. Case 1a is covered by the type II deadlock. Case 1b means that at least one of the successors of A, e.g. C, with affected arc connection from node A is an ancilla node. If such a connection emerges from the same port as the arc A $\rightarrow$ B then node B can be inverted only if node C can be inverted so as this case is reduced to recursively check if node C is engaged in any deadlock. On the other hand if such a connection emerges from another port of node A then we can see that we fall back in a type I deadlock.

The second condition can be separated in the following subcases: (2a) All the successors of node B are ancilla nodes. This means that this condition can be reduced to assure recursively that the successors are not engaged in any deadlock. (2b) At least one of the successors is a non ancilla node and this case is handled again by the type I deadlock.

Therefore, all the necessary conditions to invert an ancilla node are covered by preventing just the two deadlock cases described previously.

5.4 Reversing Algorithm

The actions described previously to reset the garbage states are collected together in Table 6. The initialization actions already described are the marking of ancilla nodes of the forward QDG (line 1, Sect. 5.2), the application of the two deadlock resolution algorithms (lines 2, 3, Sects. 5.3.1 and 5.3.2) and the initialization of the `revinfo` lists (Sect. 5.2).

The `forwQDG` already modified by the two deadlock resolution procedures is then copied (line 5) to a second QDG called `revQDG`. New nodes will be progressively added to `revQDG`, while at the same time ancilla nodes will be deleted from `forwQDG`. The final synthesis product will be the `revQDG` graph.

The circular list, `GarbageTermList`, is used to store the ids of garbage terminal nodes of `forwQDG`. This list is initialized during the construction of the forward QDG (Sect. 4.2) and contains all the terminal nodes of the graph that carry non desired results, that is it initially contains the terminal nodes carrying garbage. This list is updated during the reversion algorithm by removing ancilla node ids just inverted and by adding ancilla node ids which became terminal nodes after the removal of inverted nodes in the `forwQDG`. The algorithm scans in a circular manner this list until it becomes empty (lines 6, 7, 13).

Each node id found (`curNode`) in the circular list is checked if it is ready for inversion in the `forwQDG` (lines 8, 12). This has been explained in *Sequence of*

Table 6 Reversing algorithm

Operations

```
1:  MARK ancilla nodes of forwQDG
2:  CALL Deadlock 2 Detection and Resolution procedure
3:  CALL Deadlock 1 Detection and Resolution procedure
4:  INITIALIZE revinfo[] lists
5:  COPY forwQDG to revQDG
6:  WHILE GarbageTermList Not Empty DO
7:      FIND next curNode in GarbageTermList
8:      IF there are no children of curNode in the forwQDG with
        affected arcs THEN
9:          CALL the node inversion procedure of Table 3 for
            curNode on revQDG
10:         REMOVE curNode and its arcs from forwQDG
11:         UPDATE GarbageTermList
12:     END IF
13  END WHILE
```

Inversion bullet in Sect. 5.2. In such a case, a new node with inverse type and its relevant arcs are added in the `revQDG` (line 9). These steps have been described in detail in Sect. 5.1.

We then remove this `curNode` from the `forwQDG` graph (line 10). This removal is necessary as it may release some other ancilla nodes of the `forwQDG` with their ids contained in the `GarbageTermList` to become ready for inversion in the next execution of the loop. Thus, this removal will be taken into account in the updating of the `GarbageTermList` in line 11. The procedure of updating this list is (i) to remove the `curNode` and (ii) to add all the ancilla parent nodes of `curNode` on the condition hey have no children (this is equivalent that they indeed have become terminal nodes after the removal of the `curNode` from the `forwQDG`).

A last post-processing step that rearranges some of the control arcs is not shown in Table 6. This rearrangement changes the tail connections of some control arcs so as to emerge from the new added nodes instead of the original ones and it is based on the `revinfo` lists.

6 Synthesis Example

This section presents a simple but complete arithmetic circuit synthesized by the proposed algorithm to clarify the previously described procedures. The circuit is a *controlled modular multiplier* which is an integral part of the modular exponentiation computation for Shor's algorithm. Various proposals exist for the implementation of this circuit, most of them based on the construction of a modular adder [4,31]. The example presented is based on a recent efficient design of Shor's algorithm [22]

Table 7 Specifications of a controlled modular multiplier

	Algorithm	Initial conditions			
Line `l`	Function	`type[l]`	`p[l]`	`m[l]`	`c[l]`
1	input c	INP_F	1	–	–
2	input y	INP_F	2	–	–
3	$x = 0$	INP_F	3	–	–
4	$x = \mathrm{CMAC}_{\alpha}(x, y, c)$	CMAC_{α}	4	3	1, 2
5	$[q, r] = \mathrm{DIV}_{\mathrm{N}}(x)$	$\mathrm{DIV}_{\mathrm{N}}$	5, 6	4	–

where the building blocks are a multiplier/accumulator by constant and a divider by constant. The example serves only to present the scalability of the proposed method to automatically build hierarchically large circuits given its specification in a classical algorithm and not to evaluate the resulting circuit in terms of quantum gates, circuit depth or qubits count, as these aspects depend on the components used in the library and consequently on the low level synthesis methods employed to generate them.

A controlled modular multiplier calculates the function $x = cay \ mod \ N$ where a and N are constants of n bits. Control variable c is 0 or 1, x is $2n$ bits wide and y is n bits wide. If a multiplier accumulator by constant α (CMAC α) and a divider by constant N (DIV_N) are available in a synthesis library like that of Table 1 then the computation of the above function can be done as in Table 7.

The second column of Table 7 is the sequence of the elementary functions while the last four columns are the initial conditions passed in the synthesis algorithm in the form of array of lists as described in Sect. 3.2. Additional to these initial conditions and not depicted in the table are the array of the variables width and the array defining the final results. These are initialized as follows : `w` $= [n, n, 2n, 2n, n, n]$ and `res`=[0,0,0,0,0,1]; variables assigned the values 3 and 4 have a width of 2n qubits while the desired result is the remainder r which is numbered as the 6th variable.

The synthesized QDG is depicted in Fig. 8. It consists of the forward part at the left and the reverse part at the right. Three INP_F nodes are used in the forward part to represent the initial states c,y and $x = 0$ with width 1, n and $2n$, respectively. Nodes OUT_F with id 6 and 7 are the output nodes added by the forward synthesis algorithm and carry the garbage quotient state q and the desired state of the remainder $r = c\alpha y \ mod \ N$.

The purpose of the reverse part of the synthesis is to reset the garbage state q back to a constant zero state. This means reversing node with id 6 and all its ancestors nodes (1, 2, 3, 4 and 5). All these nodes have been marked as ancilla nodes by the forward synthesis algorithm, but nodes 1, 2 and 3 did not need reversion since they are INP_F nodes. A checking for possible deadlocks must be made before initiating the reverse algorithm. As can be seen in the figure, a deadlock of type I is found at node 5 as it is an ancilla node connected through affected arcs to ancilla node 6 and non-ancilla node 7. The actions of the deadlock algorithm is to add nodes with ids 8, 9 and 10, add some arcs as described in Sect. 5.3 and remove the arc $5 \rightarrow 7$. After these actions take place nodes 1, 2, 3, 4, 5, 6 and 10 become ancilla. Reversion algorithm results in the right part of Fig. 8. Nodes with ids 11, 12, 13 and 14 are the inverses of 10, 6, 5 and 4,

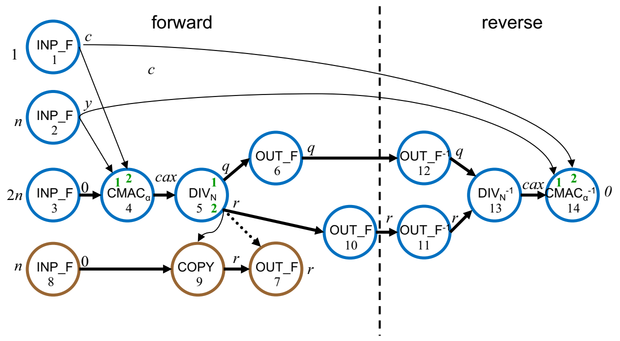


Fig. 8 Quantum or reversible architecture result in the form of QDG (forward and reverse) for the controlled modular multiplier. Inside each node the function type and the id are shown. Next to each arc the state it carries is shown. *Thick* and *thin arcs* are affected and control, respectively. *Ports numbering* is shown inside the node, when necessary

respectively. Subsequently, the complete QDG calculates the desired function and the resulted circuit is identical to controlled modular multiplier accumulator proposed in Fig. 19 of [22].

7 Features and Comparisons

In general, a direct comparison of a hierarchical synthesis method to flat (low-level) synthesis methods in terms of quantum cost, qubits count and circuit depth is not meaningful because a hierarchical synthesis is based on a previously synthesized library and the results depend on the particular low-level synthesis methods used to build the library. As for the execution time of the proposed hierarchical synthesis algorithm, it is obvious that the hierarchical technique of dividing a large circuit to smaller parts, gives a scalability advantage over a low-level strategy to handle a large and complicated circuit by a flat method.

A rough complexity analysis for the synthesis algorithm in terms of the number of lines L of the specifications follows. It is assumed that the number of input/output ports of a node is constant and much smaller than L (a reasonable assumption as shown in Table 1). The number of nodes of the forward QDG is L while that of the final reverse QDG is double. With the above assumptions we can estimate that the arcs number is $O(L)$. We concentrate on the reversing part as the forward part is easily shown to have a complexity $O(L)$ by investigating Table 2. The main part of the reversing algorithm consists of operations in lines 7–11 with constant complexity nested in a while loop scanning the ancilla nodes. Thus it has a $O(L)$ complexity. It can be seen that the most computation intensive part is the deadlock II resolution procedure. The algorithm of Table 5 consists of two nested loop each of complexity $O(L)$ (line 1 and 8); the other loops have a constant complexity due the above assumptions. Combining the nesting of the FindPaths procedure which has a linear complexity too, we conclude that the complexity of deadlock II detection is $O(L^3)$. This is the dominant complexity for the whole synthesis algorithm

Below is presented another kind of analysis that is related to the garbage generation and indirectly related to the ancilla requirement. A top level view of a reversible or quantum circuit U with its respective input and outputs signals is shown in the left part of Fig. 9. Input bits/qubits are discriminated in argument input x and ancilla input initially in a constant state, usually zero. The ancilla qubits are used internally to assist the computation. On the other hand, output bits/qubits are discriminated in the desired output $f(x)$, that is the target of the computation, ancilla output which is usually part of the ancilla input being reset back to its initially state and the garbage output $g(x)$ which depends on the input argument x and thus is not constant as the ancilla output. The garbage output contains intermediate results of the computation. When the function embedded in U is not invertible (for example, the addition of two non-constant integers is not invertible) the elimination of the input argument is impossible [5].

The garbage output is an undesired effect of the computation which is dependent on the input argument. This means that in the case we refer to a classical reversible circuit it cannot be simply "erased" as this would contradict the notion of a reversible circuit. On the other hand, in the quantum circuit case the garbage output is entangled with the desired output and a possible quantum measurement to "erase" it would affect the useful desired output. The repeated use of a circuit such as that of U then would require an accumulation of ancilla wires usage. On the other hand, if all the ancilla input wires emerge as ancilla output wires then they could be reused on successively usage of the circuit block. Consequently, it is important to eliminate the ancilla usage as much as possible because this means lower cost in terms of wires which is an important factor especially in the quantum circuit design domain.

A well known technique (Bennett's trick [5]) to eliminate the garbage, excluding the input argument, is depicted in Fig. 9. The output wires of U are copied onto new ancilla wires and then the inverse circuit U^{-1} is applied to the outputs (desired, garbage and ancilla) of U. The final result of this processing eliminates any garbage $g(x)$ as shown below and leaves only the input argument x:

$$(x, 0, 0) \overset{U}{\rightarrow} (f(x), g(x), 0, 0) \overset{copy}{\rightarrow} (f(x), g(x), 0, f(x)) \overset{U^{-1}}{\rightarrow} (x, 0, f(x)) \quad (3)$$

In the above formula, x belongs to an orthonormal set (e.g. computational basis) and the same applies for the states $f(x)$ and $g(x)$ as f and g are unitary transformations

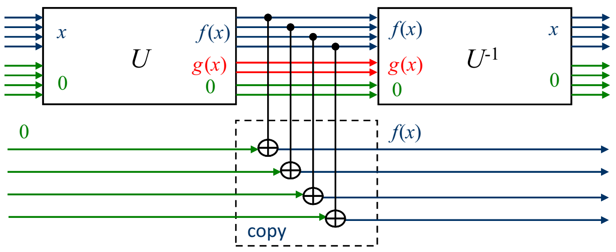


Fig. 9 Input and output wires definitions of a reversible/quantum circuit U [input argument x, ancilla input and output 0, desired output $f(x)$ and garbage output $g(x)$] and garbage elimination (except the input argument) using Bennett's trick of copying the output and applying the inverse U^{-1}

in the quantum case. Although the no-cloning theorem does not allow a general copy operation for arbitrary states, it allows such an operation for orthonormal states, in our case $f(x)$. Consequently, as we can eliminate with this method the garbage state $g(\mathrm{x})$ for every input state x in the computational basis, the same hold for an arbitrary superposition of input.

Bennett's method to eliminate intermediate garbage doubles the cost in the number of gates and depth as the inverse U^{-1}circuit must be added. Another feature is that as many ancilla wires as the desired output wires are required to hold the copy this output. This is an important disadvantage especially in the cases where the number of garbage wires to be eliminated are smaller the number of the desired output wires. This can happen when a circuit is broken down in multiple levels of hierarchy for reason already explained.

In contrast, the proposed method selectively copies only the wires that are engaged in the two kinds of deadlocks described in Sect. 5.3. Essentially, it applies Bennett's trick locally on wires that cause deadlocks to the inversion procedure, instead to apply it globally on the total number of desired output wires. A study of the conditions leading to deadlocks shows that the total number of ancilla wires needed for the deadlocks resolution is always less than or equal compared to original globally applied Bennett's trick. Even if there is no gain in the ancilla usage, it is obvious that there is gain in terms of the circuit size and its depth, as there is no need for the application of the whole inverted circuit U^{-1} but only addition of locally inverted nodes.

The proposed synthesis algorithm has been implemented (initially in MatLab) and successfully applied on various examples, including complex and irregular circuits such as the divider by constant used in a recent implementation of Shor's factorization algorithm [22]. In this specific example no gain has been observed in terms of the qubits size ($6n$ qubits required for an n bits constant divider) due the presence of a deadlock. But compared to the standard Bennett's trick the quantum cost and depth is reduced by about 25 %.

Studying other hierarchical methods appeared in the literature as part of integrated platforms we can see that the drawback of RevKit [34,35] is the excessive generation of garbage bits for intermediate results which are not reset back to constant value [7]. The approaches of [1] (CTQG part of ScaffCC) don't reduce ancilla significantly and the connections between the modules must be done by the user in a description language (structural synthesis approach as opposed to our behavioral synthesis approach). On the other hand, Chisel-Q [16] and Quipper [9] exploit the globally applied Bennett's method on each block of the hierarchy and thus in general it requires more ancilla qubits.

8 Conclusions

We have presented a generic hierarchical method for the synthesis of arbitrary large and irregular arithmetic and logical quantum and reversible architectures. The architecture is specified as a sequence of elementary operations that correspond to existing quantum or reversible components of a library. The library can be populated with circuits synthesized by the proposed method, or by any other method, permitting multilevel hierarchical synthesis of any depth. Parts of the library could be the synthesis output results of tools like [34,35] for the reversible case or tools like [1,9,15,16] for the quantum case by invoking these tools as back-end and passing them the parameters of the required parts (function type, input and output size). Another option could be the integration of the proposed method in the above mentioned tools.

Hierarchical synthesis methods for quantum and reversible architectures offer several advantages compared to flat gate level methods in the following aspects: (a) easier description of complex circuits, (b) efficient handling of arbitrary large size circuits and (c) short synthesis run-time even for significantly large circuits. Our hierarchical synthesis method, when combined with other low-level synthesis methods delivers architectures in very short time, and compared to previous hierarchical synthesis approaches it has the important advantage that it does not pollute the synthesized architecture with many ancilla wires.

References

1. Abhari, A.J., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: ScaffCC: a framework for compilation and analysis of quantum computing programs. In: Proceedings of the ACM 11th Conference on Computing Frontiers, Art. 1 (2014)
2. Balensiefer, S., Kreger-Stickles, L., Oskin, M.: An Evaluation Framework and Instruction Set Architecture for Ion-Trap based Quantum Micro-architectures, 32nd ISCA, pp. 186–196 (2005)
3. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D.P., Margolus, N., Shor, P., Sleator, T., Smolin, J., Weinfurter, H.: Elementary gates for quantum computation. Phys. Rev. A **52**, 3457–3467 (1995)
4. Beckman, D., Chari, A.N., Devabhaktuni, S., Preskill, J.: Efficient networks for quantum factoring. Phys. Rev. A **54**(2), 1034–1063 (1996)
5. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**, 525–532 (1973)
6. Cybenko, G.: Reducing quantum computations to elementary unitary operations. Comput. Sci. Eng. **3**(2), 27–32 (2001)
7. Drechsler, R., Wille, R.: Reversible circuits: recent accomplishments and future challenges for an emerging technology. In: 16th International Symposium VDAT, pp. 383–392 (2012)
8. Golubitsky, O., Maslov, D.: A study of optimal 4-bit reversible Toffoli circuits and their synthesis. IEEE Trans. Comput. **61**(9), 1341–1353 (2012)
9. Green, A.S., Lumsdaine, P.L., Ross, N.J.: Quipper: A Scalable Quantum Programming Language, ACM PLDI 13 (2013)
10. Gupta, P., Agrawal, A., Jha, N.K.: An algorithm for synthesis of reversible logic circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(11), 2317–2330 (2006)
11. Jones, N.C., Whitfield, J.D., McMahon, P.L., Yung, M.-H., Van Meter, R., Aspuru-Guzik, A., Yamamoto, Y.: Faster quantum chemistry simulation on fault-tolerant quantum computers. New J. Phys. **14**, 115023 (2012)
12. Kotiyal, S., Thapliyal, H., Ranganathan, N.: Mach—Zehnder Interferometer Based Design of all Optical Reversible Binary Adder, DATE (2012)
13. Kreger-Stickles, L., Oskin, M.: Microcoded architectures for Ion-tap quantum computers. In: 35th ISCA, pp. 165–176 (2008)
14. Landauer, R.: Irreversiblity and heat generation in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
15. Lin, C-C., Chakrabarti, A., Jha, N.K.: QLib: Quantum Module Library, ACM JETC, vol. 11(1), Art. 7 (2014)
16. Liu, X., Kubiatowicz, J.: Chisel-Q: Designing quantum circuits with a scala embedded language. In: 31st ICCD, pp. 427–434 (2013)
17. Miller, D.M., Maslov, D., Dueck, G.W.: A Transformation Based Algorithm for Reversible Logic Synthesis, DAC, pp. 318–323 (2003)
18. Maslov, D., Dueck, G.W., Miller, D.M.: Techniques for the synthesis of reversible Toffoli networks. ACM Trans. Des. Autom. of Electron. Syst. **12**(4), 42:1–42:28 (2007)

19. Metodi, T.S., Chong, F.T.: Quantum Computing for Computer Architects. Morgan and Claypool Publishers, San Rafael (2006)
20. Metodi, T.S., Thaker, D.D., Cross, A.W., Chong, F.T., Chuang, I.L.: A quantum logic array microarchitecture: scalable quantum data movement and computation. In: 38th ISCA (2005)
21. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2011)
22. Pavlidis, A., Gizopoulos, D.: Fast quantum modular exponentiation architecture for Shor's factoring algorithm. Quantum Inf. Comput. **14**(7&8), 0649–0682 (2014)
23. Prasad, A.K., Shende, V.V., Markov, I.L., Hayes, J.P., Patel, K.N.: Data structures and algorithms for simplifying reversible circuits. ACM JETC **2**(4), 277–293 (2006)
24. Saeedi, M., Markov, I.L.: Synthesis and optimization of reversible circuits—a survey. ACM J. Comput. Surveys **45**(2), 21 (2013)
25. Saeedi, M., Zamani, M.S., Sedighi, M., Sasanian, Z.: Reversible circuit synthesis using a cycle-based approach. ACM JETC. **6**(4), Art. 13 (2010)
26. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **22**(6), 710–722 (2003)
27. Shende, V.V., Bullock, S.S., Markov, I.L.: Synthesis of quantum logic circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **25**(6), 1000–1010 (2006)
28. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
29. Thaker, D.D., Metodi, T.S., Cross, A.W., Chuang, I.L., Chong, F.T.: Quantum memory hierarchies: efficient designs to match available parallelism in quantum computing. In: 33rd ISCA, pp. 378–390 (2006)
30. Toffoli, T.: Reversible computing, MIT/LCS/TM-151 (1980)
31. Van Meter, R., Itoh, K.M.: Fast quantum modular exponentiation. Phys. Rev. A **71**, 052320 (2005)
32. Vartiainen, J.J., Mottonen, M., Salomaa, M.S.: Efficient decomposition of quantum gates. Phys. Rev. Lett. **92**(17), 177902 (2004)
33. Wille, R., Drechsler, R.: BDD-based synthesis of reversible logic for large functions. In: ACM/IEEE DAC, pp. 270–275 (2009)
34. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer, Berlin (2010)
35. Wille, R., Offermann, S., Drechsler, R.: SyReC: A Programming Language for Synthesis of Reversible Circuits, Forum on Specification and Design Languages (FDL), pp. 184–189 (2010)
36. Wood, D.H., Chen, J.: Fredkin gate circuits via recombination enzymes. Congr. Evol. Comput. **II**, 1896–2000 (2004)