A Transformation Based Algorithm for Ternary Reversible Logic Synthesis using Universally Controlled Ternary Gates

Erik Curtis, Marek Perkowski+
Mentor Graphics Corp., Wilsonville, OR 97070, Erik_Curtis@mentor.com
+ Department of Electrical Engineering, Portland State University,
Portland, Oregon, 97207-0751, mperkows@ee.pdx.edu

ABSTRACT

In this paper a synthesis algorithm for reversible ternary logic cascades is presented. The algorithm can find a solution for any reversible ternary function with n *inputs and* n *outputs utilizing ternary inverter gates and the new UCTG gates which are a powerful generalization of ternary Toffoli gates and Generalized Ternary Gates [4]. The algorithm is an extension of the algorithm presented by Dueck, Maslov, and Miller in [3]. A unique feature of this algorithm is that it utilizes no extra wires to generate the outputs. A basic compaction algorithm is defined to improve the results of the basic algorithm. This paper also provides the groundwork for transforming any n*n Toffoli based binary synthesis algorithm into a ternary synthesis algorithm using the new UCTG gates.*

1. INTRODUCTION

Binary reversible circuits are beneficial in may ways. It was shown in [2] that it is only possible to dissipate zero power using reversible logic. All contemporary quantum computers use binary reversible logic [6], thus development of CAD tools for such logic will be a necessary part of making quantum computers practical. Recently, multiple-valued quantum gates have been proposed and ternary quantum gates have been built [11]. This stimulates the research in ternary reversible logic [1, 4, 7, 10, 12, 14, 16, 17, 18] with gates that are realizable in one of quantum computing technologies, such as ion trap or Nuclear Magnetic Resonance.

The presented algorithm is based on the transformation-based algorithm of Dueck, Maslov, and Miller [3], which first finds a solution and next improves it by local equivalence-based transformations. We follow this general approach. Their algorithm has been extended here by converting the rules already present in their basic algorithm to ternary rules; as well as adding additional rules specific to the use of ternary logic.

The paper is organized as follows. The required background and definitions are presented in section 2. The basic algorithm and extensions are explained in section 3. Section 3 also contains a basic explanation of the algorithm. The compaction algorithm is described in section 4. Some results are given in section 5. Finally, a description of future work is in section 6 and the conclusion in section 7.

2. BACKGROUND

Definition 1: *An m-input, m-output totally specified Ternary function f(X),* $X=\{x_0, x_1, \ldots, x_n\}$ *where* $n = 3^m-1$ *is reversible if it is a one-to-one mapping, i.e. each output assignment is a mapping of a unique input assignment.*

A reversible ternary function can be written as a vector of integers in the range 0 to 3^m-1, where each input and output vector is considered a base 3 integer value. The input vector of the function is listed in numerical order, so one can represent the function as a permutation, with the output vector being just a permutation of the input vector. In the example shown in table 3, the output vector is (5,6,1,7,2,8,0,3,4). Dueck, Maslov, and Miller specify this relationship with regards to binary logic in the background section of [3].

Definition 2: *A generalized ternary inverter is a gate with a single input and a single output. The gate is always reversible. There are 6 generalized inverters that implement all possible mappings of {0, 1, 2}*

There are two possible mappings in binary logic {0,1} and {1,0}, implemented by a wire and an inverter, respectively. There are a total of six possible mappings for a 1-qubit ternary function, with the special case of a wire. Table 1 also shows the names chosen for each type of inverter [4].

In	—	+1	+2	01	02	12
0	0	1	2	1	2	0
1	1	2	0	0	1	2
2	2	0	1	2	0	1

Table 1. Generalized Inverters

In [9] the concepts of the generalized Feynman, Toffoli, Fredkin and Kerntopf gates (gates controlled with arbitrary function of input variables) were introduced for the first time. They we next further generalized in binary logic for multi-output control functions and more bits in data-path (controlled) parts into what was next called Perkowski's gate [5]. In ternary logic, a universal set of two 1-qubit and

two 2-qubit reversible gates has been created by De Vos et al. [10]. Another universal gates were proposed for full quantum (not only its permutative subset – multiple-valued reversible logic) by Muthukrishnan and Stroud in [11]. Some subset of their gates, specialized to ternary logic, together with the De Vos gates were next generalized to what we called the Generalized Ternary Gates [4]. Now, in an attempt to create powerful ternary gates we further generalize the Generalized Ternary Gates and the binary Perkowski's gates to Universally Controlled Ternary Gates (UCTG).

Definition 3: *The Universally Controlled Ternary Gate (UCTG) is a n*n gate where the first n-1 wires are unchanged and wire n is transformed by one of the 6 generalized inverters based on an arbitrary function f of wires 1, 2, …, n-1.*

The UCTG is the ternary analogue to the n*n Toffoli gate in binary logic. The 'n' wire has a choice of three paths corresponding to f=0, f=1, and f=2. Observe that in GTG gates the control was always a single wire, and now function f is generalized to any function (not required to be reversible) of wires 1, 2, …, n-1. This is a very powerful generalization of both Perkowski's gates and GTGs. Observe that although UCTGs generalize Toffoli gate to ternary logic, it is a much more powerful generalization than in [4] where Galois addition replaces EXOR and Galois multiplication replaces Boolean AND in standard binary Toffoli gates. Now we have arbitrary controlling function and one of 215 controlled (data path) functions, so the number of all UCTGs is very high.

UCTGs can be extended to any radix, but in this paper we are concerned only with ternary logic. Such gates can be build from quantum realizable ternary gates, as shown in [12]. We believe, as expanded in [7, 13], that it is reasonable to synthesize quantum circuits from more complex gates because of the synthesis efficiency, rather than from the least granularity quantum operators. However, further experimental results are needed and comparisons among different methods [8, 12, 14, 15, 16, 17, 18] and the methods proposed here should be done.

3. THE ALGORITHM

An intrinsic property of any reversible gate is that a reversible gate composed with a reversible function creates a reversible function. The goal of the synthesis algorithm is to take a ternary reversible function and apply a series of reversible gates from output, creating a series of intermediate n-qubit functions until the output function has been transformed into the input function by creating a function which is an n-qubit identity.

3.1 The Basic Algorithm

The basic algorithm is a greedy one-pass over the entire function that transforms the output vector to the input vector one bit at a time, until the identity function is found. The algorithm consists of a special case loop for vector 0 and a general case loop for all other vectors in the function. See figure 1 for the listing of the pseudo-code of the basic algorithm.

The special case loop for $f(0)$ is executed first. In this case, there are no control lines associated with the transforms. If $f(0) = 0$, then vector 0 is in the correct location and no transformation is needed; therefore, the algorithm continues on to step 2. In the case where $f(0) \neq 0$, the algorithm then loops over every bit j in vector 0. Wherever j is equal to 1, an uncontrolled transform '+1' is applied to bit j in every vector. Wherever j is equal to 2, an uncontrolled transform '+2' is applied to bit j in every vector. At the end of step 1, $f(0) = 0$. At no time should this vector change from 0 to another value, now that this value has been locked in.

In the second step, the algorithm looks at every vector i, where $1 \leq i \leq 3^m - 1$, and every bit j in vector i (a minterm) in turn. A ternary vector p is created where p is the ternary representation of i. For each i and j, if the bit of the current function $f^+(i,j)$ is not equal to the expected bit $p[j]$, then apply the proper transform to all bits in position j, with respect to a set of control vectors $c1$ and $c2$. The control vectors $c1$ and $c2$ are used to keep the algorithm from transforming the previously locked bits. The control vectors are created by putting every bit equal to 2 in $f^+(i)$ except bit $f^+(i,j)$ into $c2$, and every bit equal to 1 in $f^+(i)$ except bit $f^+(i,j)$ into $c1$. Table 2 shows the logic table for transform selection.

The transforms in the second step are not always '+1' and '+2'. The transforms '+1' and '+2' transform all three variables; therefore, once a 0 bit has been locked in, the other transforms ('01', '02', and '12') must be used instead. The rule is that a '+1' or '+2' transform can only be used when the expected bit, p[j], equals 0. In the pseudo-code, function *find_transform* implements the function to create the proper transform based on the above rule, the current bit, and the expected bit.

Figure 2 shows a ternary cascade synthesized by the basic algorithm. In Table 3, located right to it, we show steps of transforming the output function (column 0) to the identity function (column F). Its left-most columns are input variables A and B. Observe that to help the reader, the output of function is on the left and the inputs on the right, so analyzing the algorithm steps the reader can go from left to

right in both Figure 2 and Table 3 (recall that the gates are created from outputs to inputs). Step 1 in the algorithm applies a '+2' to wire B and then a '+1' to wire A. At this point, vector 0 is in the proper location, as is shown in column 2 of table 3. The next transform happens at wire A of vector 01, where '+1' is applied when function in current wire B is 1. The transform is still a '+1' because the expected value at wire A is 0. Next transform is when the current value of wire B is 2 (column 3). Columns 3 and 4 follow the same template as the transform in column 2. Column 5 is a transform where the expected value is 1, therefore according to table 2 the selected transform is '12'. Column 5 shows that the controlling wire is value 0, therefore the transform is uncontrolled. The circuit found by the basic algorithm from outputs to inputs is shown in figure 2. The controlled transforms in this case are shown as generalized inverters with control wires attached to them and denoted by symbols of controlling values. A detailed analysis of Figure 2 together with Table 3 should help the reader not only to understand our algorithm but its strengths and disadvantages and finally improvement ideas. Section 4 presents how these controlled transforms are converted into UCTG's.

Input	Expected p[j]		
	0	1	2
0	NA	01	02
1	+1	NA	12
2	+2	12	NA

Table 2. Transform Table.

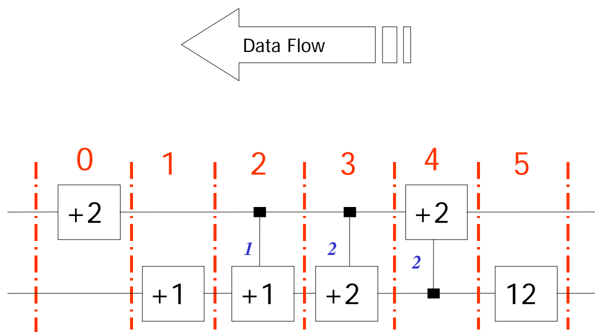

Figure 2. Basic Algorithm output

Figure 1. Basic Algorithm

```
Step 1:
Let m = # of inputs
If f(0) ≠ 0, then
For j in {0..m-1} {
   if(f(0,j) = 1 ) apply transform '+1'.
   if(f(0,j) = 2 ) apply transform '+2'.
}

Step 2:
Let f+ be the current function; f++ is next
function.
For i in {0..3^m-1} {
  let p be a ternary vector of i
  For j in {0..m-1} {
    if(f+(i,j) != p[j] ) {
      let c2 be a list of all bits in p = 2 except j
      let c1 be a list of all bits in p = 1 except j
      t = find_transform(f+(i,j), p[j])
      Transform( j, t, c1, c2)
       }
   }
}
function Transform( position, transform, c1, c2) {
  for i in {0 .. 3^m-1} {
    if( meets_constraints( f+(i), c1, c2 ) ) {
      f++( i, position) = apply( transform,  f
+(i,position) )
     }
   }
   f+ = f++
}
```

A	B	0		1		2		3		4		5		F	
0	0	1	2	1	0	0	0	0	0	0	0	0	0	0	0
0	1	2	0	2	1	1	1	0	1	0	1	0	1	0	1
0	2	0	1	0	2	2	2	2	2	0	2	0	2	0	2
1	0	2	1	2	2	1	2	1	2	2	2	2	0	1	0
1	1	0	2	0	0	2	0	2	0	2	0	2	1	1	1
1	2	1	0	1	1	0	1	2	1	2	1	2	2	1	2
2	0	2	2	2	0	1	0	1	0	1	0	1	0	2	0
2	1	0	0	0	1	2	1	1	1	1	1	1	1	2	1
2	2	1	1	1	2	0	2	0	2	1	2	1	2	2	2

Table 3. Input and intermediate variables for basic algorithm

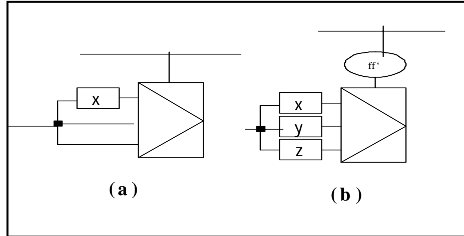

Fig. 3. UCTG realization of basic algorithm. (a) before transformation, (b) after.

3.2 The Bidirectional Algorithm

Reversible functions can be transformed on both inputs and outputs of the function, which means reversible cascade can be build from outputs to inputs (backward direction) or from inputs to outputs (forward direction). The choice of which transform to apply is determined on a vector basis and is kept the same through all bits in that vector [15]. An input transform is used on a vector if the number of bits needed to convert that vector from the current input to the input value that is paired with the output value we want in the current input is less than the number of bits needed to convert the current output vector to the current input vector. The bidirectional algorithm will not always produce the minimal function transform; it can only guarantee to produce the minimal number of transforms for the current vector (local optimization).

4. COMPACTION

The result of the basic generation algorithm is a list of transforms that may or may not have a list of control lines associated with them (see figure 2). This translates into the gate shown in figure 3a, which is very inefficient due to the fact that we are only using two of the three available paths through the gate (the transformed path, and one of the non-transformed path). The uncontrolled transforms can be converted directly into reversible ternary inverters which are already efficient. In addition, for simplification, two subsequent inverters can always be merged into a single inverter. Further, an inverter can be merged into the three inverters on the input of a UCTG. Therefore, our goal is to create as many as possible UCTG gates with three transforming paths (as shown in figure 3b). The optimization algorithm for compaction is given in figure 4.

A way to improve the results is to combine two controlled transforms into a single UCTG whenever possible. This is possible when the two controlled transforms are adjacent, transform the same wire, and have only 3 possible paths through them. In general, there are 4 possible paths through any controlled transform where each transform can be either taken or not taken. In certain cases 2 of the 4 possible paths are identical and can be collapsed.

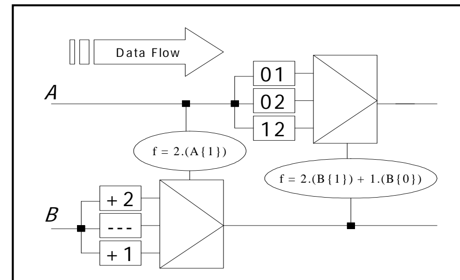

Figure. 5. Compressed result.

The first of the two cases is when the transforms are compatible. Compatible transforms are transforms that are the same or that combine to form a transform that is one of the already seen transforms (the case of '+1' and '+2' yields no transform, which is the case of neither transform taken, therefore it meets this requirement). Table 4 shows the compatibility table for the 5 inverter types. In this case, the UCTG is formed by placing one of the transforms on input 2 and the other transform or the merged transform on input 1 while input 0 has no transform.

Figure 4. Compression Algorithm

Step 1: For every uncontrolled gate, find the closest controlled gate on the same wire. Move the uncontrolled gate next to the controlled gate. Every gate that the uncontrolled gate passes over that contains a control line attached to the same wire as the uncontrolled gate, has that control transformed by the uncontrolled gates transform, taking into account the direction of movement of the uncontrolled gate.

Step 2: Find controlled gates that are on the same wire and are next to each other. If the transforms of these gates are compatible, then the gates can be merged by merging the control lines and converting the transforms into the new merged transforms.

Step 3: Find control gates that are on the same wire and are next to each other. If the control lines of these gates are exclusive, then the gates can be merged by setting one control line set to be output 1 and the other control line set to be output 2.

Step 4: Merge the uncontrolled gates. Find the uncontrolled gates and merge them into the control gates they are next to, if it is possible.

The second case is when the control lines are exclusive. Exclusive control lines mean that there can never be the case that both transforms are taken, therefore one of the four possible paths is removed. In this case, the UCTG is formed by placing one of the transforms on input 2, the second transform on input 1, and no transform on input 0.

Uncontrolled transforms next to controlled transforms on the same wire can be merged together. The uncontrolled transform inverter is simply placed on all inputs of the UCTG and merged with other inverters on the same input. Unlike controlled transforms, the uncontrolled transforms do not have to start out next to a transform on the same wire. The uncontrolled transforms can be pushed next to the nearest controlled transform on the same wire. If the uncontrolled transform slides past another transform with a control line on the same wire as the uncontrolled transform, then the control line should be modified according to the value of the uncontrolled transform.

The final algorithm is shown in figure 4. The example function shown in figure 2 can be compressed into 2 transforms, as shown in figure 5. In general, compression can work on any controlled gates, controlled with arbitrary functions.

The improvements presented in this paper take into account only single wires. Multiple wire compactions would be yield a much better result, but are more difficult to find algorithms for. In the case where a controlled transform is bounded on both sides by controlled transforms on other wires, no compaction is done. In testing this yields a modest reduction in size, with benefits remaining largely static as the wire count increases.

	---	+1	+2	01	02	12
---	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
+1	*yes*	*yes*	*yes*	**NO**	**NO**	**NO**
+2	*yes*	*yes*	*yes*	**NO**	**NO**	**NO**
01	*yes*	**NO**	**NO**	*yes*	**NO**	**NO**
02	*yes*	**NO**	**NO**	**NO**	*yes*	**NO**
12	*yes*	**NO**	**NO**	**NO**	**NO**	*yes*

Table 4. Compatibility Chart

5. RESULTS

There are no standardized benchmarks for ternary functions, therefore we created our own benchmark functions. These benchmark functions can be found at the first author's website [8]. Table 5 shows the results of synthesis of functions with 2, 3, 5, 6, and 7 variables [8]. The first circuit, *bench2-1,* is the same circuit that is used in the explanation of the algorithm in section 3. All blue fields in the table are the best raw score, and all red fields are the best merged score for each benchmark.

From the benchmark it is easy to tell that the method of choosing the best transform does not always

generate the best results. Remember that the best transform is only locally best for that particular vector, not for the design overall. In the large benchmarks (vector length 5, 6, and 7) the best transform method produces the best results. In *bench7-1*, the reversible method saves almost 1500 gates over the input or output transform methods. For the smaller test cases, it is most likely an artifact of the specific benchmark that defines one method to be much better than the other one.

The merging algorithm doesn't compress the results very well. This is most likely due to the restriction that in order to merge, there must be two transforms on the same line next to each other. This is not likely to happen on larger and larger test cases because the vector length increases. Note that for a function with only one "care" output wire (where the other output wires are chosen to satisfy the requirements of the reversibility, but aren't used internally), the results can be much better if the dummy outputs have been chosen wisely.

More advanced merging algorithms that can selectively slide controlled gates back and forth will be able to shrink the transform results to a greater extent. Such algorithms are feasible since they generalize to ternary an approach found very useful for binary reversible logic. They may be, however, more difficult to apply.

The runtime of this algorithm has been not shown, because the test program is not optimized for speed, and it was run on a heavily loaded machine. In a sample run, testcase *bench5-3* took approximately 10s wall clock, and testcase bench8-2 took approximately 1s wall clock. Using a program tuned for speed should decrease these numbers significantly, however the trend upwards is a matter of the complexity of the benchmark. The main loop must loop over m^4 (m^3 vectors, m bits per vector) vectors for a function of length m. The merging loop runs in linear time over the size of the raw transform list, but the transform list grows at a greater than linear rate versus m.

6. FUTURE WORK

There are several ideas for future work to improve this algorithm. This is a work in progress.

6.1 Controlled Transform Sliding

In section 4, the idea was introduced of shuffling the order of uncontrolled transforms in order to allow them to be compressed. This can be applied on a limited scale to controlled transforms as well. A controlled transform cannot be shuffled past another transform that has a control attached to the same wire as the transform in question. In large wire counts, there is a good possibility that only a subset of the entire wire set is used for controlling any transform. If two transforms on the same wire can be brought together given this constraint, then more of the controlled transforms could be merged. This improvement would be greatly enhanced by an algorithm that could minimize the number of selected control lines (see section 6.3).

6.2 Direction Selection

The bidirectional algorithm was discussed in section 3.2. The direction was chosen to minimize the number of transforms per vector. It is easy to observe in the benchmarks that this doesn't always provide an optimal solution. An alternative solution is to try out both the forward and reverse transform for every bit that needs to be transformed and determine which transform yields the smaller complexity. Complexity in this case is defined similar to Dueck, Maslov, and Miller's definition in [3], where it is used to determine the best combination of control lines. Complexity is the sum total of all bits that are not equal to their expected bit in the current function. Choosing the transform that minimizes the complexity should require a smaller total number of transforms. This is not an absolute minimization criterium, since subsequent transforms may increase the complexity further and it may yield a poorer compression results. Additional heuristics should be also developed and tested.

6.3 Control Line Selection

The concept of Control Line Selection s an improvement to the algorithm described by Dueck, Maslov, and Miller in [3]. In their algorithm, the idea is to pick the allowed subset of control lines that maximally reduce the complexity of the circuit. This doesn't necessarily yield the smallest number of control lines, but instead attempts to minimize the number of Toffolli gates needed, similar to the reasoning in section 6.2.

We predict that choosing the minimum number of control lines is even more important for the presented algorithm. A smaller number of control lines will yield a smaller control function to implement. A smaller subset also can yield a greater chance of being able to merge controlled gates if the improvement in section 6.1 has also been implemented.

The control line selection problem is not as simple as in [3] due to the fact that there are a set of control lines that must be 1 and a set that must be 2. In this case we may be able to generalize a control line to a generalized don't care (i.e. 0 or 1, 1 or 2, 0 or 2), but

not remove it completely. The benefits to generalizing the control lines haven't been fully investigated, so it is not sure what combination would yield the best results. The exciting part of this optimization is that the chances to improve the compression through the combination of this and the method from section 6.1 increase with the size of the vector, unlike the current compression algorithm which seems to be fairly linear with respect to the transform size.

6.4 Transform Selection

The algorithm currently always chooses a '+1' or '+2' when the expected bit value is zero. This is not always the best transform selection. In some cases it may be better to apply a transform that only operates on two of the values ('01', '02', or '12'). Similar to section 6.2, choosing the transform that yields the smallest complexity is a good optimization. This optimization may backfire given that the '+1' and '+2' transforms are more compatible (see table 4), and cause the results to be worse after compression.

Test Name	Vector Length	Output Transforms		Input Transforms		Best Transforms	
		Raw	Merged	Raw	Merged	Raw	Merged
bench2-1	2	**9**	*4*	10	7	**9**	8
bench3-1	3	43	36	46	40	**41**	*35*
bench3-2	3	48	42	**42**	*35*	43	37
bench3-3	3	44	*36*	50	45	**41**	*36*
bench5-1	5	741	724	727	719	**627**	*602*
bench5-2	5	774	761	719	701	**634**	*616*
bench5-3	5	740	718	753	739	**618**	*599*
bench6-1	6	2787	2768	2700	2684	**2296**	*2269*
bench7-1	7	9664	9638	9652	9629	**8193**	*8153*

Table 5. Benchmark Results

7. CONCLUSION

A new algorithm to synthesize ternary logic into reversible ternary gates has been presented. This algorithm can solve a ternary function of arbitrary size as long as the function itself is a reversible function.

In contrast to the algorithm from [12] this algorithm cannot transform a non-reversible function to a reversible one during the synthesis, but its asset is that in contrast to the algorithm from [12], it does not require any additional wires (called also ancillae bits). The presented algorithm can be used to solve a non-reversible function by adding dummy lines to the input and output to make the entire function reversible. The algorithm can be used to create good solutions for these problems if proper assignments are made to the dummy lines, however finding the proper assignment is not a trivial exercise.

The future work outlined above should improve the final results of the algorithm. It is difficult to guess their benefits without implementing the circuits, but a cursory look at the combination of control line selection (for minimum control lines) and controlled transform sliding has the potential to save hundreds of gates on the larger benchmarks. Improved control line selection also should save many gates in the creation of the controlling function. The comparison of the final version of this algorithm and other ternary reversible algorithms from the literature [4, 12, 14] should use realistic cost functions (fitness functions) that would take into account the numbers of electromagnetic pulses or other quantum technology-specific parameters.

References

1. A. Al-Rabadi, L. Casperson, M. Perkowski and X. Song, "Multiple-Valued Quantum Logic,", *Proc. ULSI 2002,* Boston, May 15, 2002.
2. C. Bennett, "Logic Reversibility of Computation," *IBM J. Res. Dev.* 17:525-532, 1973.
3. G. Dueck, D. Maslov, and D. M. Miller, "A Transformation Based Algorithm for Reversible Logic Synthesis," Proc. *DAC.* 2003 Anaheim, CA, pp. 318-323

4. M. H. A. Khan, M. Perkowski and P. Kerntopf, "Multi-Output Galois Field Sum of Products Synthesis with New Quantum Cascades," *Proceedings of 33rd International Symposium on Multiple-Valued Logic,* 16-19 May 2003, Meiji University, Tokyo, Japan, pp. 146-153.
5. M. Lukac, M. Pivtoraiko, A. Mishchenko, and M. Perkowski, "Automated Synthesis of Generalized Reversible Cascades using Genetic Algorithms," *Proceedings of Symposium on Boolean Problems,* Freiberg, Germany, pp. 33 – 45, September 2002.
6. M. Nielsen and I. Chuang, "Quantum Computation and Quantum Information," *Cambridge University Press*, 2000.
7. M. Perkowski, A. Al-Rabadi, P. Kerntopf, "Multiple-Valued Quantum Logic Synthesis," *Proc. Conference on New Directions in VLSI Design,* Sendai, Japan, , pp. 41 - 47, December 2002.
8. A PERL program implementing the algorithm described in this paper is available at the first author's website: http://mysite.verizon.net/Erik.Curtis/ternary. In addition, the sample functions and results files are also available for download at the same site.
9. A. Khlopotine, M. Perkowski, P. Kerntopf, Reversible Logic Synthesis by Iterative Compositions, *Proc. IWLS* 2002, pp. 261 – 266.
10. A. De Vos, B. Raa, and L. Storme, "Generating the group of reversible logic gates", *Journal of Physics A: Mathematical and General*, Vol. 35, 2002, pp. 7063-7078.
11. A. Muthukrishnan, and C. R. Stroud, Jr., "Multivalued logic gates for quantum computation", *Physical Review A,* Vol. 62, No. 5, Nov. 2000, 052309/1-8.
12. N. Denler, B. Yen, M. Perkowski, and P. Kerntopf, "Minimization of Arbitrary Functions in a New Type of Reversible Cascade built from Quantum- Realizable "Generalized Multi-Valued Gates" ", submitted to IWLS.
13. M. Perkowski, M. Lukac, M. Pivtoraiko, P. Kerntopf, M. Folgheraiter, D. Lee, H. Kim, H. Kim, W. Hwangboo, J.-W. Kim, and Y.W. Choi, "A Hierarchical Approach to Computer Aided Design of Quantum Circuits," Proceedings of 6th International Symposium on Representations and Methodology of Future Computing Technology, March 2003, Trier, Germany, pp. 201 – 209.
14. M. H.A. Khan, M. Perkowski, Genetic Algorithm Based Synthesis of Multi-Output Ternary Functions Using Quantum Cascade of Generalized Ternary Gates, submitted to Congress on Evolutionary Computation, 2004.
15. M. Perkowski, A. Al-Rabadi, P. Kerntopf, A. Buller, M. Chrzanowska-Jeske, A. Mishchenko, M. Md. Mozammel Azad Khan, A. Coppola, S. Yanushkevich, V. Shmerko, and L. Jozwiak, "A General Decomposition for Reversible Logic". *Proc. RM'2001,* August 2001

B. Yen, N. Denler, M. Perkowski, Synthesis of Ternary Logic Using Generalized Ternary Gate Cascades in a Filtering Model Approach, submitted to ULSI 2004.

16. M. H.A. Khan, M. Perkowski, and G.Greenwood, Memetic Algorithms Based Synthesis of Multi-Output Ternary Functions Using Quantum Cascade of Generalized Ternary Gates, in preparation.
17. M.H.A. Khan, M.A. Perkowski, M.R. Khan, and P. Kerntopf, "Ternary GFSOP Minimization using Kronecker Decision Diagrams and Their Synthesis with Quantum Cascades", Submitted to *Journal of Multiple-Valued Logic and Soft Computing: Special Issue to Recognize T. Higuchi's Contribution to Multiple-Valued VLSI Computing*.