On Synthesis of (k,K) Circuits.

Brief Contributions

On Synthesis of Easily Testable *(k, K)* Circuits

Srinath R. Naidu, *Student Member*, *IEEE*, and Vijay Chandru

Abstract—A (k, K) circuit is one which can be decomposed into nonintersecting blocks of gates where each block has no more than K external inputs, such that the graph formed by letting each block be a node and inserting edges between blocks if they share a signal line, is a partial k-tree. (k, K) circuits are special in that they have been shown to be testable in time polynomial in the number of gates in the circuit, and are useful if the constants k and K are small. We demonstrate a procedure to synthesise (k, K) circuits from a special class of Boolean expressions.

Index Terms—Testing, stuck-at fault, polynomial time, k-tree, treewidth, synthesis.

1 Introduction

THE problem of determining a test for a stuck-at fault in an arbitrary combinational circuit is known to be NP-complete [4]. Given this negative result, researchers have tried to come up with restricted classes of circuits for which the problem is known to be solvable in polynomial-time. One example of a class of circuits for which the stuck-at fault testing problem is solvable in polynomial-time is the class of (k, K) circuits introduced by [3]. They showed that for this class of circuits, the testing problem can be solved in $O(2^{kK}g^2)$ where k, K are constants and g is the number of gates in the circuit. Membership in this class of circuits is based on certain structural properties of circuits. Since the identification of this class of circuits, research has focused on determining if an arbitrary circuit can be classified as a (k, K) circuit for small values of k and K. Unfortunately, this problem has also been shown to be NP-complete by [9], thus appearing to severely limit the usefulness of the (k, K) class of circuits in real-life.

In this paper, we investigate a synthesis procedure for (k, K) circuits. More specifically, we determine how to take an arbitrary logic function obeying certain restrictions and produce an easily testable circuit belonging to the (k, K) class. The logic expressions implemented by our class of circuits are of the type $F_1 < op > F_2 < op > F_3 < op > ...F_n$, where each F_i represents arbitrary combinational logic on no more than K distinct primary inputs and op stands for another specific logic function. Our synthesis algorithm first checks if an intuitively defined term graph of the given logic expression is a special type of graph called a partial k-tree. If this is the case, then, it finds an embedding of the term graph into a full k-tree. Then, we work with this k-tree to establish an appropriate order of combining terms and come up with a $(2k-1, K+1)$ implementation of the logic expression.

- *S.R. Naidu is with the Department of Electrical Engineering, Eindhoven University of Technology, Den Dolech 2 5600 MB Eindhoven, The Netherlands. E-mail: s.r.naidu@ele.tue.nl.*
- *V. Chandru is with the Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.*

Manuscript received 2 Dec. 1999; revised 12 June 2001; accepted 22 Oct. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111024.

2 Preliminaries

A (k, K) combinational circuit is one which can be partitioned into nonintersecting blocks of gates such that each block has no more than K inputs and the graph formed by letting each block be a node and inserting an edge between two blocks if they share a signal line is a partial k-tree. We define a k-tree as follows: 1) A complete graph on k-vertices is a k-tree. 2) Let G be a k-tree. Select some k-clique subgraph of G. Let H be a graph obtained by adding a new vertex to G and making it adjacent to all the vertices in the selected k-clique. Then, H is a k-tree.

A partial k-tree is simply a subgraph of a k-tree. A vertex elimination operation on a graph applied on a vertex results in that vertex being removed and edges added to the resulting graph such that the neighbors of the removed vertex are all connected to each other. A k-elimination sequence is a sequence of all the vertices of the graph where at the time of the elimination each vertex has degree no greater than k. After the elimination sequence is applied, the original graph becomes the empty graph. A partial k-tree has a k-elimination scheme. The terms "partial k-tree" and "treewidth no greater than k" are equivalent [6] and we use them interchangeably in what follows.

3 Relation to Previous Work

The paper relates to previous work in polynomial-time testability in an interesting way. It has been shown in [8] that, if a parameter called cutwidth of a circuit is of $O(\log G)$, where G is the number of gates in the circuit, then the circuit is testable in $O(2^{O(C)} * G)$, where C is the cutwidth of the circuit. Since C is $O(\log G)$, the circuit is actually testable in time that is *quadratic* in G.

Korach and Solel showed in [5] that, if a graph G has log-bounded cutwidth, then it must have constant treewidth. The polynomial-time testing algorithm of [7], [9] has been shown to be of $O(2^{kK}G)$, for (k, K) circuits, which is *linear* in G. The connection graph that we use in what follows is identical to the problem graph of [8]. Therefore, for the many circuits of log-bounded cutwidth that occur, in practice, the algorithm of [7] is more efficient thereby showing that for these circuits, treewidth is more useful than cutwidth.

4 Synthesis by Algebraic Factorization

Algebraic factorization is used in multilevel logic synthesis to reduce the number of times literals appear in the expression thus reducing the number of fanout variables. See Fig. 1a for an implementation of the function $ab + cdb + ad + ac$. The connection graph of Fig. 1b is constructed by letting each gate be a node and inserting edges between nodes if they share a signal line. This graph is a partial 4-tree as $C_8, C_7, C_6, C_5, C_4, C_3, C_2, C_1$ is a 4-elimination scheme. In Fig. 1c, we show an implementation corresponding to $a(b + d + c) + cdb$. The connection graph for this circuit, shown in Fig. 1d is only a partial 2-tree. Although factorization makes a difference, in this case, we show next that even if an expression is factored into a form where each variable appears exactly twice, it might still not be implementable as a (k, K) circuit for constant k and K. First, we note that the connection graph corresponding to a circuit implementing a logic expression where each variable occurs no more than twice, has degree 3. To see this, consider any gate in the circuit implementing such an expression. Assume that all gates are two-input gates. If the gate is a primary input gate, it has, at most, two primary inputs each of which may fan out to, at most, one other gate. This means the node in the connection graph corresponding to such a gate

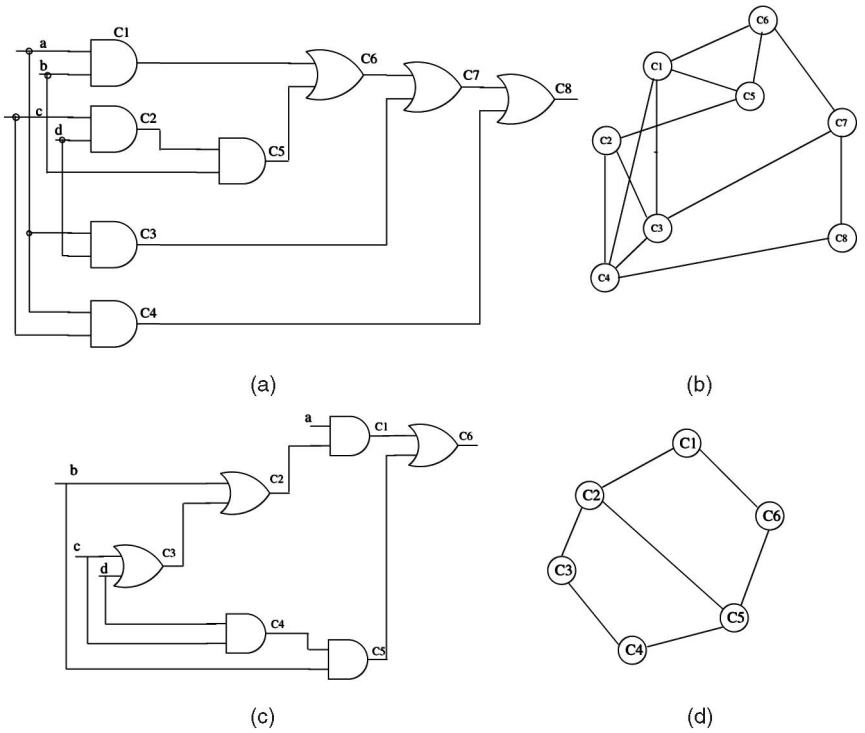

Fig. 1. Factoring reduces treewidth of the implementation.

must be of degree 3 because G shares signal values with, at most, three other gates. The same holds for an internal gate. For connection graphs of degree 3, we can state the following:

Lemma 1. *There exists a graph G of $O(m)$ vertices such that no vertex has degree greater than 3, whose treewidth is at least $\Omega(\sqrt{m})$.*

Proof. Consider a complete graph H on $\Omega(\sqrt{m})$ vertices. We perform the following vertex splitting operation on vertices of H. Each vertex of degree greater than 3 is split into two vertices of smaller degree such that the two vertices together are joined to all the neighbors of the original vertex and to each other. We perform this splitting operation until all the vertices are of degree no greater than 3. Note that we cannot split a vertex of degree no greater than 3 vertex into two vertices, both of which are of strictly smaller degree. The splitting operation we have described is the reverse of the well-known edge contraction operation of graph minor theory. A graph H is a minor of G if H is isomorphic to a graph formed by contracting an edge of G [10]. From the theory, it is well-known that, if H is a minor of G, then $treewidth(H) \leq treewidth(G)$. The postsplitting graph, say G is of degree 3 and can have no more than $O(m)$ vertices. Since H is a minor of G, we know that $treewidth(G) \geq treewidth(H)$. Therefore, $treewidth(G)$ is at least $\Omega(\sqrt{m})$. □

This result is not surprising because of the well-known fact that the Hamilton circuit problem remains NP-complete even when restricted to cubic graphs and it is known that there exists a polynomial-time algorithm that determines if a graph of bounded treewidth has a Hamilton circuit [1].

5 Synthesis Procedure

Suppose we are given an expression of the form $F_1 < op > F_2 < op > \ldots F_n$, where the F_is are Boolean functions, each having no more than K inputs and op is another Boolean function, such as XOR, AND, etc., and our task is to find an implementation of this expression as a (k, K) circuit for some constants k and K. Each F_i can contain any kind of combinational logic whatsoever, but $< op >$ must be a commutative and associative logic operator since we can then have the freedom of linking the blocks of logic corresponding to the F_is in any order. We first create a term graph of the given expression. The term graph consists of a node for each F_i and and an edge between two nodes if the support sets of the functions they represent share a variable. For example, consider the expression $a'b'c + cbd'g + eac'kf + ih'g' + f'g'h + je'd + k'j'$. The term graph for this expression is shown in Fig. 2a. We decide to link up the nodes in this graph in the order shown in Fig. 2b to form the circuit implementing this expression. The graph of Fig. 2b is a partial 4-tree. Consider the decomposition of the circuit shown in Fig. 2c, where in each block we include a node of the term graph and at most one single OR node. Other decompositions are obviously possible but we choose this one because it follows naturally from the term graph which in turn can be easily constructed from the given logic expression. In Fig. 2d, we show the graph that results when each block in Fig. 2c is treated as a node and edges inserted between two blocks if they share a signal line. Notice that the graph minus the dotted edges is really just the original term graph. The graph of Fig. 2d is a partial 3-tree. The dotted edges represent the overhead of linking up nodes in a particular order, and can significantly change the treewidth of the resulting graph. Consider a partial 2-tree on n vertices. It is possible to assign a labeling $1, 2, \ldots, n$ to the vertices of the partial 2-tree, such that if one added edges as necessary so that for every $i, 1 < i < n$ there is an edge between i and $i + 1$, the resulting graph would have treewidth at least $\Omega(\sqrt{n})$. Therefore, it is important to find a labeling scheme that causes only a slight increase in treewidth. We describe a labeling scheme takes takes a graph of treewidth k and produces one of treewidth no greater than $2k - 1$.

Before we describe the labeling scheme, we introduce some notation. We refer to a k-clique of vertices by a label vector whose components are the labels associated with the vertices of the

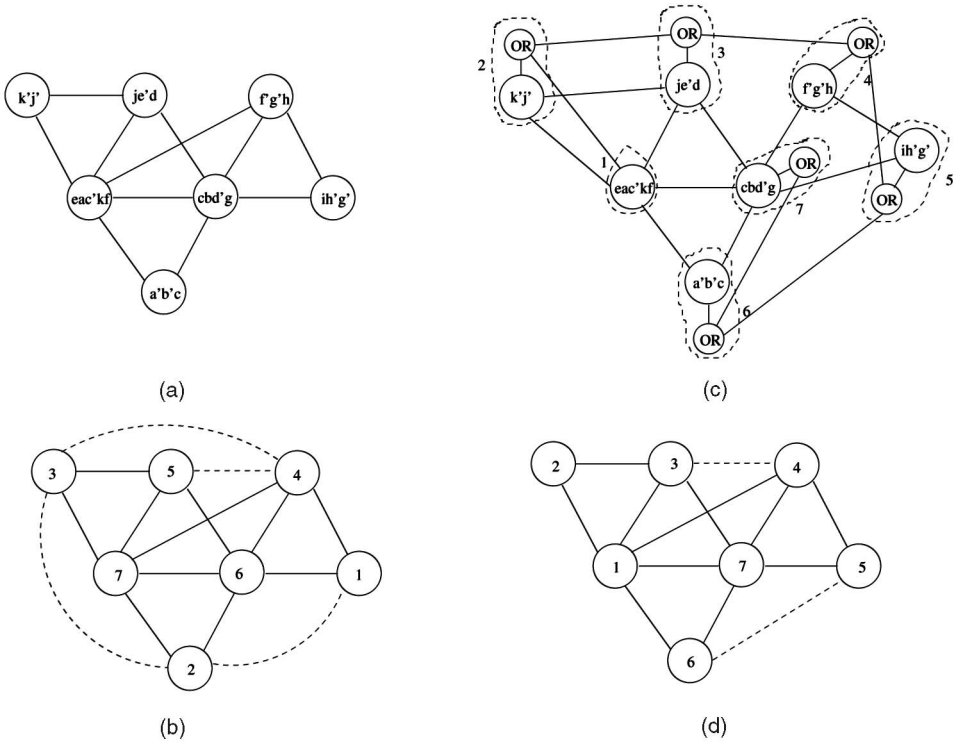

Fig. 2. Term graph and different ways of linking the terms.

k-clique in ascending order. For instance, $(x_1, x_2, \ldots, x_k)$ refers to a k-clique with labels $x_1, x_2, \ldots, x_k$ where $x_1 < x_2 < \ldots < x_k$. The label vector used to annotate a particular k-clique can change through the labeling process. The labeling scheme proceeds first by identifying some k-clique in the graph and labeling it with $(1, 2, \ldots k)$. We push this label vector onto a stack. The next vertex to get a label is an unlabeled vertex adjacent to all the k vertices of the k-clique at the top of the stack. If the label vector at the top of the stack is $(x_1, x_2, \ldots x_k)$, and we find a vertex adjacent to it, then all labels greater than or equal to x_k are first increased by one, and then the new vertex is assigned the label x_k. Assigning the new vertex with the label (x_k) creates k new cliques. Let X denote the set $\{x_1, x_2, \ldots x_k, x_k + 1\}$. We push onto the stack k new label vectors $X \setminus \{x_{k-1}\}, X \setminus \{x_{k-2}\}, \ldots, X \setminus \{x_1\}, X \setminus \{x_k + 1\}$ in that order onto the stack. As an example, let $k = 3$ and let the top of the stack be (1, 2, 3). Then, after the push operation this clique gets the label vector (1, 2, 4) while three new cliques (1, 3, 4), (2, 3, 4) and (1, 2, 3) are pushed onto the stack in that order. If there is no unlabeled vertex adjacent to the top of the stack, then pop the stack. The pushing and popping operations continue until all vertices are labeled.

Definition. *We shall denote a k-clique* $(x_1, x_2, \ldots, x_k)$ *to* contain *a k-clique* $(y_1, y_2, \ldots, y_k)$ *if* $x_1 \leq y_1, x_2 \leq y_2, x_3 \leq y_3, \ldots, x_{k-2} \leq y_{k-2}$ *and either (1)* $x_{k-1} < y_{k-1}$ and $x_k = y_k$ *or (2)* $x_{k-1} = y_{k-1}$ and $x_k > y_k$ *or (3)* $x_{k-1} < y_{k-1}$ and $x_k > y_k$.

It is easy to check that the k-clique at the top of the stack before a push operation contains the newly created k-cliques after the push operation. Next, we introduce the notion of a branch k-clique. A branch k-clique is one which, when it is on the top of the stack during the labeling procedure, is found to be adjacent to more than one unlabeled vertex. A branch k-clique $(x_1, x_2, \ldots x_k)$ is defined to be the *parent* of the branch k-clique $(y_1, y_2, \ldots, y_k)$ if there is no branch k-clique $(z_1, z_2, \ldots, z_k)$ such that $(x_1, x_2, \ldots x_k)$ contains $(z_1, z_2, \ldots, z_k)$ and $(z_1, z_2, \ldots, z_k)$ contains $(y_1, y_2, \ldots, y_k)$. All the ancestors of a given branch k-clique are present in the stack below the given branch k-clique during the time that it is on the stack. When a k-clique is popped off the stack, it is never pushed back onto the stack. We can create a branch k-clique graph creating a node for each branch k-clique and inserting an edge between two branch k-cliques if one of them is the parent of the other.

After the labeling procedure is finished, we construct a belong-set S of vertices belonging to a k-clique $(x_1, x_2, \ldots, x_k)$ recursively as follows: 1) All vertices with labels greater than x_{k-1} and less than x_k which are adjacent to the k-clique $(x_1, x_2, \ldots, x_k)$, belong to set S and 2) Any vertex with label between x_{k-1} and x_k and adjacent to an edge whose endpoints both belong to $S \cup \{x_1, x_2, \ldots, x_k\}$, also belongs to set S. A vertex p is said to *properly* belong to a given branch k-clique $(x_1, x_2, \ldots, x_k)$ if it is in the belong set of $(x_1, x_2, \ldots, x_k)$ and is not in the belong set of any branch k-clique that is a descendant of $(x_1, x_2, \ldots, x_k)$. For a leaf branch k-clique, its belong-set and its proper belong-set are one and the same. We can provide another characterization of the belong-set S of vertices pertaining to a branch k-clique $(x_1, x_2, \ldots, x_k)$. Consider the phase of the stack during the labeling procedure when a given branch k-clique appears at the top of the stack until the branch k-clique is popped off the stack. Then, S is the union of the disjoint sets of vertices $P_1, P_2, \ldots, P_j$ corresponding to the j rise-falls of the stack for the given branch k-clique where a rise-fall corresponds to the stack growing up and falling off to the level of the branch k-clique in question. Each of the sets P_i is called a component. We can see that, if p is the smallest vertices in P_i and q the highest labeled vertex in P_i, then P_i contains all vertices with labels between p and q. As an example of our labeling algorithm, Fig. 2d is the graph that is produced when our labeling algorithm is given the term graph of Fig. 2a. Let H be the graph with dotted edges obtained by applying the labeling algorithm to the term graph. We now state several lemmas which will enable us to establish our main result, namely, that our labeling scheme gives us a $(2k - 1, K + 1)$ implementation of the given logic expression.

Lemma 2. *The branch k-clique graph is a forest of rooted trees.*

Lemma 3. *Let t the number of push-pop operations performed on the stack with n the maximum label assigned after t operations. If $(x_1, x_2, x_3, \ldots, q)$, $q \leq n$ is on the top of the stack then,*

1. *the labels m and $(m+1)$ are adjacent and the stack contains k-cliques with last two components $m, m+1$ for all m such that $q \leq m \leq n-1$.*
2. *the rightmost components of the k-cliques on the stack from top to bottom form a nondecreasing sequence and successive elements differ by at most 1.*
3. *Going from the top of the stack to the bottom, for all edges with the same rightmost component, the vectors corresponding to the first $k-1$ vertices in each k-clique form a strictly decreasing sequence. If X and Y are vectors, we say that $X < Y$ if each element of X is less than or equal to the corresponding element of Y and there is at least one element of X strictly less than the corresponding element of Y.*
4. *At some later point in time, the top of the stack carries a clique of the form $(y_1, y_2, y_3, \ldots, m, m+1)$ for all m such that $q \leq m \leq n-1$.*

Proof. All the points above can be proven by induction. □

Lemma 4. *Let there be a dotted edge between vertices labeled $(p-1)$ and p after the labeling procedure is completed. Then, $(p-1)$ and p belong to adjacent components of some branch k-clique $(y_1, y_2, \ldots, y_k)$. Moreover, $(p-1)$ is the vertex of largest label in the ith component of $(y_1, y_2, \ldots, y_k)$ and p is the vertex of smallest label in the $(i+1)$th component of $(y_1, y_2, \ldots, y_k)$.*

Proof. Since $(p-1)$ and p are not adjacent labels at the end of the labeling procedure, there must be some point during the labeling procedure when no unlabeled vertex is found adjacent to any k-clique label vector with $(p-1)$ and p as the last two components. The next unlabeled vertex must be found adjacent to a k-clique $(y_1, y_2, \ldots, v, p)$ with $v < p-1$. If this were not the case, then by Lemma 3.2 all k-cliques with rightmost endpoint p would be popped off the stack and thereafter the rightmost endpoint of k-cliques remaining on the stack would be $(p+1)$ or higher. This would freeze the positions of the labels $(p-1)$ and p and they would remain adjacent at the end of the labeling procedure. Therefore, it is the assignment of a label p to the unlabeled vertex adjacent to $(y_1, y_2, \ldots, v, p)$ that $(p-1)$ and p first become nonadjacent. Further processing never causes them to be adjacent again. □

Lemma 5. *Let P_i be some component with respect to the branch k-clique $(x_1, x_2, \ldots, x_k)$. Let the vertex with the smallest label in P_i be p and let the vertex with largest label in P_i be q. Then, p is adjacent to the $(k-1)$-clique $(x_1, x_2, \ldots, x_{k-1})$ and q is adjacent to x_k.*

Proof. We will prove the statement by induction. When the first vertex of the component P_i is added, the statement of the lemma is clearly true. Let the statement be true after n vertices of P_i have been processed. Let the branch k-clique for which P_i is a component have the label $(x_1, x_2, \ldots, x_{k-1}, v)$ with $v < x_k$. If p' and q' are the smallest and largest labels, respectively, then p' is adjacent to $(x_1, x_2, \ldots, x_{k-1})$ and q' to v by the induction hypothesis. There are three possible labels that the $(n+1)$th vertex can obtain. If the $(n+1)$th vertex acquires the label p', then it must have been added adjacent to some k-clique with p' as the rightmost endpoint. But, the only labels smaller than p' are $x_1, x_2, \ldots x_{k-1}$ which means that it must have been added adjacent to $(x_1, x_2, \ldots, x_{k-1}, p')$. If the $(n+1)$th vertex acquires some label u such that $p' < u \leq q'$, then the vertex previously labeled q' gets the label $q'+1$. It still remains the highest labeled vertex, and the statement of the lemma continues to hold. Finally, if the $(n+1)$th vertex gets the label v, then it must have been added adjacent to some k-clique with v as its rightmost endpoint, say, $(w_1, w_2, \ldots, w_{k-1}, v)$. The new highest label in the component becomes v and it is adjacent to the highest labeled vertex in the branch k-clique which has the label $v+1$. After the vertices of P_i are removed from the stack, the labels of the vertices in P_i are frozen. Thus, the vertex of the smallest label in P_i remains adjacent to the first $k-1$ vertices of the branch k-clique and the vertex of greatest label remains adjacent to the rightmost endpoint of the branch k-clique. □

Lemma 6. *Let $(x_1, x_2, \ldots, x_k)$ be a leaf branch edge with belong-set S. Let p and q be the smallest and largest labels in S, respectively. Then, the edges $(p-1, p)$ and $(q, q+1)$ cannot both be dotted edges.*

Proof. Assume that there exists a dotted edge between $(p-1)$ and p. Then, by Lemma 4, $(p-1)$ and p belong to adjacent components of some branch k-clique $(y_1, y_2, \ldots, y_k)$. Further, $(y_1, y_2, \ldots, y_k)$ must be an ancestor of $(x_1, x_2, \ldots, x_k)$. Since p is the lowest labeled vertex in some component of $(y_1, y_2, \ldots, y_k)$ as well as in some component of $(x_1, x_2, \ldots, x_k)$, by Lemma 5 it must be adjacent to both $(k-1)$ cliques $\{x_1, x_2, \ldots, x_{k-1}\}$ and $\{y_1, y_2, \ldots y_{k-1}\}$. We note that $y_1 \leq x_1, y_2 \leq x_2, \ldots, y_{k-1} \leq x_{k-1}$. Further, p can be adjacent via solid edges only to vertices in the belong-set of $(x_1, x_2, \ldots, x_k)$ and the vertices $\{x_1, x_2, \ldots, x_k\}$. Thus, p can only be adjacent to vertices with labels between x_{k-1} and x_k, in addition, to the vertices in the $(k-1)$-clique $(x_1, x_2, \ldots, x_{k-1})$. This means $(y_1, y_2, y_3, \ldots, y_{k-1}) = (x_1, x_2, \ldots, x_{k-1})$ and the branch k-clique $(y_1, y_2, y_3, \ldots y_k)$ is really $(x_1, x_2, \ldots, x_{k-1}, y_k)$.

Let there exist a dotted edge between q and $(q+1)$. As before, q and $(q+1)$ must belong to adjacent components of some branch k-clique $(z_1, z_2, \ldots z_k)$. $(z_1, z_2, \ldots, z_k)$ contains $(x_1, x_2, \ldots, x_k)$ and by Lemma 5, q must be adjacent to both z_k and x_k since it is the largest label in components belonging to $(x_1, x_2, \ldots, x_k)$ as well as $(z_1, z_2, \ldots, z_k)$. In this case, $z_k \geq x_k$. But, q can only be adjacent to vertices in the belong-set of $(x_1, x_2, \ldots x_k)$ as well as vertices in $\{x_1, x_2, \ldots, x_k\}$. Thus, $z_k = x_k$. Therefore, the ancestor branch k-clique $(z_1, z_2, \ldots, z_k)$ is really $(z_1, z_2 z_3, \ldots, z_{k-1}, x_k)$. Therefore, if there are dotted edges $(p-1, p)$ and $(q, q+1)$, the leaf branch k-clique $(x_1, x_2, \ldots, x_k)$ has as ancestors the branch k-cliques $(x_1, x_2, \ldots, x_{k-1}, y_k)$ and $(z_1, z_2, \ldots, z_{k-1}, x_k)$, where $y_k > x_k$, $z_{k-1} < x_{k-1}$ and for all $i, 1 \leq i \leq k-2, z_i \leq x_i$. The branch k-clique graph is a forest of trees and one of these k-cliques must be the ancestor of the other. But, this is impossible as neither k-clique can contain the other. □

Theorem. *H is a partial $(2k-1)$-tree.*

Proof. We will provide a $(2k-1)$-elimination scheme for the graph H which will prove that H is a partial $(2k-1)$-tree. We will proceed bottom-up on the branch k-clique tree, removing leaf branches first. Removing a leaf-branch means that we convert the given leaf branch k-clique into a non-branch k-clique. Let $(x_1, x_2, \ldots, x_k)$ be a leaf branch k-clique with components $P_1, P_2, \ldots, P_j$. Let the lowest labeled and highest labeled vertex in component P_i for all i be p_i and q_i, respectively, and the vertex in each component that is adjacent to all the vertices of the leaf branch k-clique be m_i. Due to lemma 6, there are three cases to consider:

1. *p_1 and q_j do not have dotted edges incident on them:* The induced subgraph on the vertices of the first component, $\{x_1, x_2, \ldots, x_k, p_1, \ldots, m_1, \ldots, q_1\}$, is a k-tree. We follow an elimination scheme for these vertices in H that is a k-elimination scheme in the induced subgraph. However in H, the elimination scheme becomes a $(k+1)$-elimination scheme owing to the possibility of the existence of a dotted edge between some k-leaf and p_2, the lowest labeled vertex in the component P_2. When all the vertices in P_1 are eliminated, the result is the

possible introduction of a dotted edge between p_2 and x_k. We can find another $(k+1)$-elimination scheme using arguments as above to eliminate the vertices $p_2, \ldots, m_2 - 1$ of the second component, if they exist (such vertices may not exist if $p_2 = m_2$). Then, we use another $(k+1)$-elimination scheme to eliminate the vertices $m_2, \ldots, q_2$. We can repeat these scheme on the first $j-1$ components of the leaf branch k-clique, and then remove the vertices $p_j, \ldots, m_j - 1$ of the last component. Doing so, ultimately converts $(x_1, x_2, \ldots, x_k)$ into a nonbranch k-clique.

2. *p_1 has a dotted edge but not q_j:* We first follow a k-elimination scheme for the vertices $m_j + 1, \ldots, q_j$, that is, a k-elimination scheme for these vertices in the induced graph on the vertices $\{x_1, x_2, \ldots, x_k, m_j, \ldots, q_j\}$ which is a k-tree. Similarly, we follow a $(k+1)$-elimination scheme on the vertices $p_j, \ldots, m_j - 1$ following a k-elimination scheme for these vertices in the induced subgraph on the vertices $\{x_1, x_2, \ldots, x_k, p_j, \ldots, m_j\}$. The last vertex to be removed is m_j. It is adjacent to the leaf branch k-clique $(x_1, x_2, \ldots, x_k)$ and has a dotted edge connection to q_{j-1}. Since q_{j-1} is the highest labeled vertex in P_{j-1} it is, by lemma 4, adjacent to x_k. Therefore, eliminating m_j will create at most $k-1$ new dotted edges leading from q_{j-1} to the vertices $\{x_1, x_2, \ldots, x_{k-1}\}$. Thus, we will now have a $(2k-1)$-elimination on the vertices $m_{j-1} + 1, \ldots, q_{j-1}$. Going forward, we can eliminate all the vertices,

$$p_{j-1}, \ldots, m_{j-1}, p_{j-2}, \ldots, m_{j-2}, \ldots q_{j-2}, p_{j-3}, \ldots,$$
$$m_{j-3}, \ldots, q_{j-3}, \ldots, p_2, \ldots m_2, \ldots q_2,$$

and a part of the first component consisting of the vertices $m_1 + 1, \ldots, q_1$ via a $(2k-1)$-elimination scheme. This converts $(x_1, x_2, \ldots, x_k)$ into a nonbranch k-clique.

3. *p_1 has no dotted edge but q_j does:* This case is similar to Case 1 and we can follow the same strategy as in Case 1.

After converting all branch k-cliques into nonbranch k-cliques, we are left with a single k-tree without any dotted edges whose vertices can be eliminated by a k-elimination scheme. □

There still remains the question of checking if a given graph is embeddable into a k-tree, given that the input to our labeling algorithm is a k-tree. Checking if a given graph is a partial 2-tree is easy as one only needs to find a degree 2-elimination scheme for the graph. For the general case, there is an order $O(n^{k+2})$ algorithm in [2].

6 Conclusions

In this paper we examined the question of synthesising (k, K) circuits. These circuits are known to be easily testable provided k and K are small. We require the input expressions to be of the form $F_1 < op > F_2 < op > \ldots F_n$, where *op* refers to some operator that is the same throughout the expression and the F_is are Boolean functions on no more than K distinct inputs.

References

[1] S. Arnborg and A. Proskurowski, "Linear Time Algorithms for NP-Hard Problems Restricted to Partial k-Trees," *Discrete Applied Math.*, vol. 23, pp. 11-24, 1989.

[2] S. Arnborg, D.G. Corneil, and A. Proskurowski, "Complexity of Finding Embeddings in a k-Tree" *SIAM J. Algebraic and Discrete Methods*, vol. 8, no. 2, pp. 277-284, Apr. 1987.

[3] S.T. Chakradhar, V.D. Agrawal, and M.L. Bushnell, *Neural Models and Algorithms for Digital Testing:* Dordrecht, The Netherlands, Kluwer Academic Publishers, 1991.

[4] H. Fujiwara and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits," *IEEE Trans. Computers*, vol. 31, no. 6, pp. 553-560, June 1982.

[5] E. Korach and N. Solel, "Treewidth, Pathwidth and Cutwidth," *Discrete Applied Math.*, vol. 43, pp. 97-101, 1993.

[6] J. van Leeuwen, "Graph Algorithms," *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity Theory*, pp. 527-631. Amsterdam: North Holland Publishing, 1990.

[7] S.R. Naidu, "Polynomial-Time Testable Combinational Circuits," MSc (Engg) thesis, Dept. of Computer Science and Automation, Indian Inst. of Science, July 1998.

[8] M.R. Prasad, P. Chong, and K. Keutzer, "Why Is ATPG Easy?" *Proc. 36th Design Automation Conf.*, pp. 22-28, 1999.

[9] N.S.V. Rao and S. Toida, "On Polynomial-Time Testable Combinational Circuits" *IEEE Trans. Computers*, vol. 43, no. 11, pp. 1298-1309, Nov. 1994.

[10] N.D. Robertson and P.D. Seymour, "Graph Minors III. Planar Tree-Width," *J. Combinatorial Theory, Series B 36*, pp. 49-64, 1984.