Circuit integration through lattice hyperterms

Vardges Melkonian

Department of Mathematics, Ohio Universtiy, Athens, Ohio 45701, USA
vardges@math.ohiou.edu

Abstract

Reducing the size of a logic circuit through lattice identities is an important and well-studied discrete optimization problem. In this paper, we consider a related problem of integrating several circuits into a single hypercircuit using the recently-developed concept of lattice hyperterms. We give a combinatorial algorithm for integrating k-out-of-n symmetrical diagrams which play important role in reliability theory. Our results show that the integration can reduce the number of circuit gates by more than twice.

Keywords: logic circuit reduction; lattices; hyperterms; k-out-of-n diagrams; combinatorial optimization)

AMSC numbers: 90C27, 94C10

1. Introduction

One of the important problems in the theory of logic circuits (and generally structures described by block diagrams) is the circuit reduction. Redundant elements are eliminated and the circuit is brought to a reduced form which is equivalent to the original one. Algebraic structures like Boolean algebras and distributive lattices and their properties are the basis of solving this kind of reduction problems. The algebraic operations in lattices, $\vee$ and $\wedge$, are the mathematical equivalents of OR-gates and AND-gates in logic circuits. The reduction of the circuits can therefore be realized using lattice identities. This kind of applications of algebraic structures in logic circuits are well-known ([10, 17]) and continue to be an active area of research ([2, 3, 9, 14, 15]). But as stated in [10], "Logic synthesis is key for the quality of a netlist. The state of the art is rather poor, compared to other areas of chip design an improvement of logic synthesis by mathematical ideas is badly needed." And a goal of this paper is to apply a recently-developed algebraic concept and a combinatorial scheme for logic synthesis.

Recent developments in lattice theory, and particularly the concepts of lattice hyperterms and lattice hyperidentities ([7, 8, 20]) give a new perspective to the optimization of logic circuits. A hyperidentity is formally the same identity with functional variables replacing lattice operations. For example, instead of two commutative identities, (i) $x \vee y = y \vee x$, (ii) $x \wedge y = y \wedge x$, we can take a hyperidentity $F(x, y) = F(y, x)$, which becomes the first identity when F takes value $\vee$ and the second one when F takes value $\wedge$. The same principle could be applied to logic circuits: two separate circuits can be integrated in a single hypercircuit which will realize the functions of both circuits at appropriate modes. The mathematical equivalent of the mode choice is the substitution of a functional variable by a particular lattice operation.

If two or more simple circuits are efficiently combined in an equivalent hypercircuit then the total number of the elements used in the circuits can be reduced. The optimization problem is to find an equivalent hypercircuit which minimizes the total number of elements. In this paper we show how to find a minimal equivalent hypercircuit for symmetrical k-out-of-n circuits which play important role in reliability theory ([11, 19]). We show that the substitution of symmetrical k-out-of-n circuits by a single hypercircuit decreases the number of gates by more than twice. Our procedure includes algorithmic techniques from combinatorial optimization and integer programming.

A main goal of this paper is to attempt to connect the several research areas mentioned above: logic circuits, lattice hyperterms, reliability theory, combinatorial optimization. The idea is to establish a connection between these areas through the recently-developed concept of lattice hyperterms.

The paper is organized the following way. In Section 2 we review some basic concepts from algebra. Section 3 states the general problem of integrating circuits through lattice hyperterms. The solution of the problem for symmetrical k-out-of-n diagrams is given in Section 4. A short discussion of an important future direction, circuit integration through Boolean algebra hyperterms is given in Section 5. Other future directions are discussed in Section 6.

2. Basic Concepts from Lattice Theory

Below is a brief discussion of some concepts from the theory of lattices and hyperidentities that we need in this paper.

Definition 1 *An algebraic structure $(L, \vee, \wedge)$, consisting of a set L and two binary operations $\vee$ and $\wedge$ on L (called join and meet respectively) is called a* **lattice** *if the following identities hold for all elements a, b, c of L*

(i) idempotent laws: $a \vee a = a, \quad a \wedge a = a;$

(ii) commutative laws: $a \vee b = b \vee a, \quad a \wedge b = b \wedge a;$

(iii) associative laws: $a \vee (b \vee c) = (a \vee b) \vee c, \quad a \wedge (b \wedge c) = (a \wedge b) \wedge c;$

(iv) absorption laws: $a \vee (a \wedge b) = a, \quad a \wedge (a \vee b) = a.$

Definition 2 *A lattice $(L, \vee, \wedge)$ is called* **distributive** *if the distributive laws*

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c), \quad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

hold for all elements a, b, c of L.

The binary operations $\vee$ and $\wedge$ are the mathematical equivalents of OR-gate and AND-gate in logic circuits. Thus, the lattice identities can be used to reduce a circuit to an equivalent circuit which uses fewer gates. This type of reductions are well-known. In this paper we exploit a different kind of reduction based on lattice hyperidentities and lattice hyperterms. A hyperidentity has the same form as an identity; the only difference is that the actual operations are replaced by functional variables. Note that any pair of lattice identities (i)-(iv) have the same form, and could be obtained from each other by interchanging the places of $\vee$ and $\wedge$. Thus, any pair of those identities can be generalized to a higher level identity where the operations are replaced by functional variables. For example, a general form for associative laws is $F(a, F(b, c)) = F(F(a, b), c)$. The two associative laws of (iii) can be obtained by substituting $F \to \vee$ and $F \to \wedge$ respectively.

Definition 3 *A* **lattice hyperidentity** *is an equality consisting of binary functional variables $F_1, ..., F_n$ and simple (element) variables $x_1, ..., x_k$ that becomes a valid lattice identity for any substitution of $F_1, ..., F_n$ by actual binary operations $\vee$ and $\wedge$.*

Examples of valid lattice hyperidentities are $F(a, a) = a$ (idempotent law), $F(a, b) = F(b, a)$ (commutative law), $F(a, F(b, c)) = F(F(a, b), c)$ (associative law), $F(a, G(b, c)) = G(F(a, b), F(a, c))$ (distributive law).

Each side of the equality in lattice hyperidentities is a lattice hyperterm. Lattice hyperidentities can be used to reduce lattice hyperterms to equivalent hyperterms the same way as lattice identities can reduce lattice terms to equivalent terms. Lattice hyperterms are the basis of our circuit reduction problem stated in the next section.

The field of hyperidentities is an active research area in modern algebra. Some recent papers discussing different properties, new directions in development of hyperidentities and applications are discussed in [5, 6, 7, 12, 13, 16]. A discussion about hyperidentities in distributive lattices is given in [12]. The application of hyperidentities for reducing the complexity of switching circuits is discussed in [7].

A goal of this paper is to discuss a new application area for hyperidentities. [7] discusses how to use hyperidentities to reduce a given circuit to an equivalent simpler circuit. While our problem is the integration of several circuits into one.

3. The Problem Statement

We begin by formally defining hyperterms.

Definition 4 *A* **lattice hyperterm** *(mixed hyperterm) is an expression consisting of binary functional variables* $F_1, ..., F_n$*, binary operations* $\vee, \wedge$ *and simple (element) variables* $x_1, ..., x_k$*.*

Note that we allow to have $\vee$ and $\wedge$ in a hyperterm. In that sense the name mixed hyperterm is more appropriate but we will use the shorter name hyperterm in our discussion. The reason why we allow both functional variables and actual operations in hyperterms will become clear from the solution of our problem.

A lattice hyperterm becomes several different lattice terms when its functional variables are substituted by the operational symbols $\vee, \wedge$. For example, hyperterm $F(x, G(y, z))$ produces four different lattice terms:

- $x \vee (y \vee z)$ when $F \to \vee$, $G \to \vee$;
- $x \vee (y \wedge z)$ when $F \to \vee$, $G \to \wedge$;
- $x \wedge (y \vee z)$ when $F \to \wedge$, $G \to \vee$;
- $x \wedge (y \wedge z)$ when $F \to \wedge$, $G \to \wedge$.

Suppose we have a situation when circuits corresponding to all those four lattice terms are needed. Then we should build 4 circuits with total number of OR-gates and AND-gates 8. On the other hand, the hyperterm has only 2 functional variables. If we could build a circuit corresponding to the hyperterm, that would significantly reduce the number of gates. To build such a circuit a gate equivalent to a functional variable should be used. We will call

such a gate a **hypergate**. A hypergate can fulfill the functions of both OR-gate and AND-gate if appropriate modes are chosen. A primitive approach of building a hypergate is given in appendix section 7.1. An effective engineering realization of a hypergate using modern electronics is an interesting open question. But the solution of that problem is beyond the goals of this paper. The goal of this paper is to discuss the problem of circuit reduction by using hyperterms. Below we give the formal statement of the problem.

The hyperterm integration problem. Given simple lattice terms $T_1, T_2, ..., T_k$ we want to find a lattice hyperterm H such that each of T_i's can be obtained from H by an appropriate substitution of functional variables by $\vee$ and $\wedge$. While many this kind of hyperterms might exist we want to find the one which has as few hypergates and simple gates as possible.

The solution to the general problem for any lattice terms is an open question. It would make sense to start from special cases that also have practical importance. The next section gives a procedure for solving the problem for symmetrical diagrams.

4. Integrating Symmetrical Diagrams

k-out-of-n symmetrical diagrams play an important role in reliability theory ([4, 11, 18, 19]). According to [19], "the k-out-of-n system model is a very popular model which finds an extensive number of applications in industrial systems many variations of k-out-of-n systems have been widely studied in the last years. Among these, consecutive k-out-of-n systems have played a particularly relevant role in reliability analysis and design of integrated circuits, pipeline systems, communications networks, biological systems, etc.". Some of the industrial and military applications are described in [11]. It particularly states that "among applications of the k-out-of-n system model, the design of electronic circuits such as very large scale integrated (VLSI) and the automatic repairs of faults in an on-line system would be the most conspicuous". Algebraic and combinatorial methods are common in analyzing k-out-of-n systems ([18, 19]). Our goal is to integrate electronic circuits consisting of k-out-of-n systems using hyperidentities and combinatorial techniques.

A k-out-of-n symmetrical diagram consists of n identical units and will function if at least k units of total n work. This feature results in the symmetrical form of the block diagram

as well as the corresponding lattice term. For example, the lattice term corresponding to 3-out-of-4 symmetrical diagram is $(x \wedge y \wedge z) \vee (x \wedge y \wedge u) \vee (x \wedge z \wedge u) \vee (y \wedge z \wedge u)$, and the term corresponding to 2-out-of-4 symmetrical diagram is $(x \wedge y) \vee (x \wedge z) \vee (x \wedge u) \vee (y \wedge z) \vee (y \wedge u) \vee (z \wedge u)$.

In some situations it might be necessary to change the reliability of the system by switching from one symmetrical diagram to another. For example, switching from 3-out-of-4 diagram to 2-out-of-4 diagram will increase the reliability of the system. Thus, it is a relevant problem to have a system that could easily switch from one symmetrical diagram to another, and the corresponding optimization problem is to have such a system with minimum possible number of elements. Designing a hyperdiagram that integrates the simple diagrams of the system would be a solution to the problem.

In this section we will consider the following optimization problem. Given the k-out-of-n symmetrical diagrams for a fixed n and all $k \in 1, ..., n$, build a hyperdiagram that will integrate all n symmetrical diagrams by using as few gates as possible.

Before giving the solution of the problem for general n, we will start from the special cases of 3 and 4 units to get an initial idea about the integration process.

4.1 The integration in the case of 3 units

In this case we have 3 simple lattice terms:

(i) 1-out-of-3: $x_1 \vee x_2 \vee x_3$

(ii) 2-out-of-3: $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

(iii) 3-out-of-3: $x_1 \wedge x_2 \wedge x_3$

The total number of elements in the three symmetrical terms is $2 + 5 + 2 = 9$.

A hyperterm integrating the three terms is $F_2(F_2(F_1(x_1, x_2), F_1(x_1, x_3)), F_1(x_2, x_3))$. To get a better visual idea, we can represent the hyperterm as a binary tree. For example, the tree corresponding to $F_2(F_2(F_1(x_1, x_2), F_1(x_1, x_3)), F_1(x_2, x_3))$ is shown in figure 1. In our further discussion we will use the tree representation of hyperterms. Below we check that the hypertree of figure 1 really integrates the three simple terms.

When $F_2 \to \vee$ and $F_1 \to \vee$, the hyperterm becomes (i);

When $F_2 \to \vee$ and $F_1 \to \wedge$, the hyperterm becomes (ii);

When $F_2 \to \wedge$ and $F_1 \to \wedge$, the hyperterm becomes (iii).

For example, when F takes value $\vee$ and G takes value $\vee$,

$(x_1 \vee x_2) \vee (x_1 \vee x_3) \vee (x_2 \vee x_3) = x_1 \vee x_2 \vee x_3$ by lattice laws.

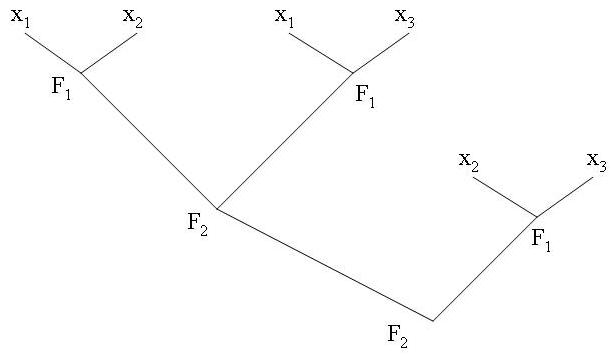

Figure 1: Hypertree with 3 units

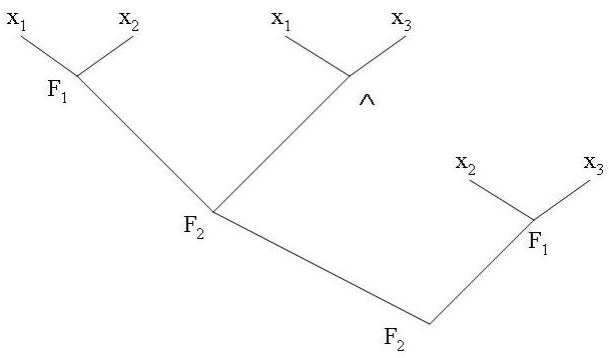

Figure 2: Modified hypertree with 3 units

Note that the hypertree has 5 hypergates compared to 9 gates in simple terms (i)-(iii).

If it is possible to use a lattice operation instead of a functional variable, then we obviously prefer the lattice operation. This is equivalent of using a simple gate instead of a hypergate. In our case, it can be easily verified that the hyperterm given in figure 2 also integrates (i)-(iii). Here one of the functional variables is replaced by $\wedge$. The new hyperterm has 4 hypergates and 1 simple gate.

4.2 The integration in the case of 4 units

In this case we have 4 simple lattice terms:

(i) 1-out-of-4: $x_1 \vee x_2 \vee x_3 \vee x_4$

(ii) 2-out-of-4: $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4)$

(iii) 3-out-of-4: $(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_4) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$

(iv) 4-out-of-4: $x_1 \wedge x_2 \wedge x_3 \wedge x_4$

The total number of gates in the four symmetrical terms is 3+11+11+3=28.

A hypertree integrating the four terms is given in figure 3. Below we check that the hypertree really integrates the four simple terms.

When $F_3 \to \vee$, $F_2 \to \vee$ and $F_1 \to \vee$, the hypertree becomes (i);

When $F_3 \to \vee$, $F_2 \to \vee$ and $F_1 \to \wedge$, the hypertree becomes (ii);

When $F_3 \to \vee$, $F_2 \to \wedge$ and $F_1 \to \wedge$, the hypertree becomes (iii);

When $F_3 \to \wedge$, $F_2 \to \wedge$ and $F_1 \to \wedge$, the hypertree becomes (iv).

The resulting hyperterm has 8 hypergates and 6 simple gates compared to 28 simple gates in (i)-(iv).

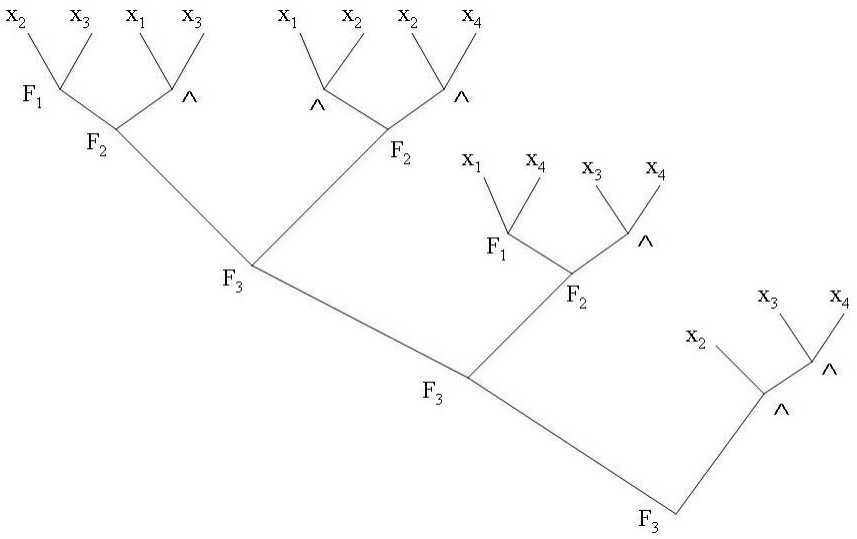

Figure 3: Hypertree with 4 units

4.3 General principles in the case of n units

Generalizing from the previous subsections we have the following principles for building the hypertree.

(i) The hypertree consists of the units $x_1, x_2, ..., x_n$ and operation symbols $F_1, ..., F_{n-1}, \wedge$.

(ii) If a subtree has $\wedge$ as a root operation then all other operations in the subtree are also $\wedge$.

(iii) If a subtree has F_k as a root operation then the possible operations in the subtree are $\wedge$ and F_i where $i \leq k$.

(iv) For any k, the corresponding k-out-of-n diagram is obtained from the hypertree by substituting

$F_i \rightarrow \wedge$, if $i < k$;

$F_i \rightarrow \vee$, if $i \geq k$.

To achieve (iv) the hypertree should satisfy the following condition. For any k-combination of elements $\{x_{i1}, x_{i2}, ..., x_{ik}\} \subseteq \{x_1, x_2, ..., x_n\}$, there should be a subtree such that

a) it consists exactly of units $x_{i1}, x_{i2}, ..., x_{ik}$;

b) it has operational symbols from $\wedge, F_1, ..., F_{k-1}$;

c) the operational symbol which is the parent of the root of the subtree is F_k.

Following these principles, the hypertree can be built recursively: once we have subtrees corresponding for every k+1-combination then those subtrees will be populated with subtrees

corresponding to k-combinations. This can be done in many different ways. But our goal is to use as few operational symbols as possible. First of all we want to have exactly one subtree for each k-combination. Then the k-combinations should be distributed among the k+1-combinations such that the number of operational symbols is minimized. Below we explain how it is done and estimate the number of operational symbols.

To demonstrate our procedure, the recursive steps for $n = 4$ are shown in figures 4 and 5. Figure 4 demonstrates how to populate the original 4-combination tree with 3-combination subtrees; figure 5 shows how to populate 3-combination subtrees with 2-combination subtrees. Note that after populating 2-combination subtrees with 1-combinations we get the final tree as shown in figure 3.

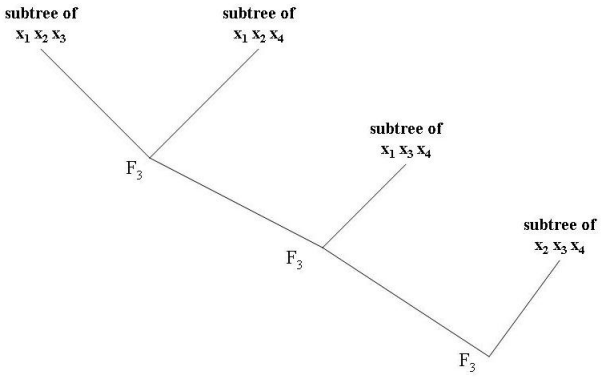

Figure 4: Populating the tree with 3-combinations

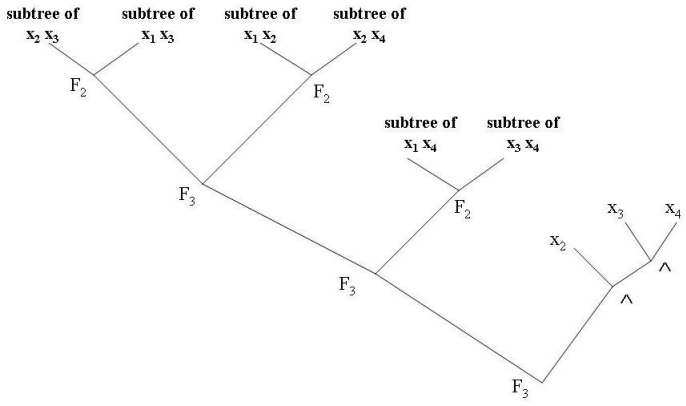

Figure 5: Populating 3-combination subtrees with 2-combinations

To populate a k+1-combination subtree by exactly m k-combination subtrees we need $m - 1$ operational symbols F_k. For example, in figure 5 the 3-combination $\{x_1, x_2, x_3\}$ is populated by 2-combinations $\{x_2, x_3\}$ and $\{x_1, x_3\}$, and we need 1 operational symbol F_2 for that. The only exception is when a k+1-combination subtree is populated by exactly 1 k-combination subtree; in this case we need 1 operational symbol F_k. Also note that if a k+1-combination subtree is not populated by any k-combination then the subtree is simply the $\wedge$-product of the $k + 1$ units. For example, in figure 5 the 3-combination $\{x_2, x_3, x_4\}$ is not populated by any 2-combinations, and thus the corresponding subtree is $x_2 \wedge x_3 \wedge x_4$. Based on these observations and keeping in mind our goal of minimizing the number of operational symbols, we have the following directions for populating the k+1-combination subtrees by k-combination subtrees:

1) Maximize the number of k+1-combination subtrees populated by k-combinations;

2) If a k+1-combination subtree is populated by k-combinations then the number of those k-combinations should be at least 2.

This is a discrete optimization problem. In the next subsection we give a solution method for it.

4.4 Populating k+1-combination subtrees by k-combinations

The problem how to allocate k-combinations to k+1-combinations has the following equivalent combinatorial problem. Suppose we have a set of students X and a set of projects Y. There is a compatibility graph between X and Y, specifying which student can do which projects. We want to assign students to work on the projects by satisfying 3 main principles: (i) every student should be assigned to exactly one project; (ii) if a project is chosen to be pursued then at least 2 students should be assigned to that project; (iii) do as many projects as possible while satisfying conditions (i) and (ii) (assuming that there is a feasible solution at all).

Our problem is a special case of this general problem. In our case, X is the set of all k-combinations, Y is the set of all k+1-combinations, a k-combination is compatible with a k+1-combination if the k-combination is a subset of the k+1-combination.

We give two methods for populating the k+1-combination subtrees by k-combinations.

Method 1: Integer programming solution

The optimization problem can be solved by integer programming. We define two sets of binary integer variables. For any k-combination i and any k+1-combination j such that i is a subset of j we define a variable x_{ij}; it takes value 1 if the k+1-combination j is populated by the k-combination i, and takes value 0 otherwise. For any k+1-combination j we define a variable y_j; it takes value 1 if the k+1-combination j is populated by any k-combination, and takes value 0 otherwise. Let K and L be the sets of all k-combinations and k+1-combinations

correspondingly. Then the integer program is:

$$\begin{aligned}
\max \quad & \textstyle\sum_{j\in L} y_j && && (1)\\
\text{s.t.} \quad & \textstyle\sum_{j\in L} x_{ij} = 1, && \text{for each } i \in K, && (2)\\
& \textstyle\sum_{i\in K} x_{ij} \geq 2y_j, && \text{for each } j \in L, && (3)\\
& x_{ij} \leq y_j, && \text{for each } i \in K, j \in L && (4)\\
& x_{ij}, y_j \quad \textit{binary}, && \text{for each } i \in K, j \in L. && (5)
\end{aligned}$$

The objective function (1) maximizes the number of k+1-combination subtrees populated by k-combinations. Constraint (2) provides that each k-combination populates exactly one k+1-combination subtree. Constraint (3) makes sure that if a k+1-combination subtree is populated by k-combinations then the number of those k-combinations should be at least 2. Constraint (4) provides that a k-combination i populates a k+1-combination j only if $y_j = 1$.

Method 2: Recursive allocation

In this subsection we give a recursive procedure for building the hypertrees. Suppose for n units we know how to allocate k-combinations to k+1-combinations for any k. Using that we want to do the same thing for $n+1$ units. Namely, we will show how to allocate k+1-combinations to k+2-combinations for $n+1$ units. Let S_k^n denote the set of all k-combinations for n units. Let $S_k^n(n+1)$ denote the set of all k-combinations for n units with an extra element $n+1$ added to each of them.

Note that $S_{k+1}^{n+1} = S_{k+1}^n \cup S_k^n(n+1)$. $S_k^n(n+1)$ includes all those k+1-combinations of S_{k+1}^{n+1} that contain $n+1$; on the other hand, S_{k+1}^n includes all those k+1-combinations of S_{k+1}^{n+1} that do not contain $n+1$. Thus, S_{k+1}^n and $S_k^n(n+1)$ do not overlap and form a partition for S_{k+1}^{n+1}. Similarly, S_{k+2}^n and $S_{k+1}^n(n+1)$ form a partition for S_{k+2}^{n+1}. We allocate k+1-combinations to k+2-combinations based on those partitions. Thus, two cases are considered. Take any $x \in S_{k+1}^{n+1}$.

Case 1: Suppose $x \in S_{k+1}^n$. By recursion, for n units x was allocated to some k+2-combination $y \in S_{k+2}^n \subset S_{k+2}^{n+1}$. Do the same allocation for $n+1$ units.

Case 2: Suppose $x \in S_k^n(n+1)$. Then $x = p \cup \{n+1\}$ for some $p \in S_k^n$. By recursion, for n units p was allocated to some $q \in S_{k+1}^n$. Then for $n+1$ units allocate $x = p \cup \{n+1\}$ to $y = q \cup \{n+1\} \in S_{k+1}^n(n+1)$.

Summarizing, we have the following recursive procedure for allocating the combinations of S_{k+1}^{n+1} to the combinations of S_{k+2}^{n+1}.

1) Partition each of them using combinations for n units: $S^{n+1}_{k+1} = S^n_{k+1} \cup S^n_k(n+1)$, $S^{n+1}_{k+2} = S^n_{k+2} \cup S^n_{k+1}(n+1)$.

2) Allocate k+1-combinations of S^n_{k+1} to k+2-combinations of S^n_{k+2} as in case 1.

3) Allocate k+1-combinations of $S^n_k(n+1)$ to k+2-combinations of $S^n_{k+1}(n+1)$ as in case 2.

This kind of allocation achieves the following conditions stated in the previous subsection.

• Every k+1-combination is allocated to a k+2-combination.

• If a k+2-combination subtree is populated by k+1-combinations then the number of those k+1-combinations is at least 2 (by recursion).

Maximizing the number of k+2-combination subtrees populated by k+1-combinations is also achieved with one exception. The maximization is not achieved in the special case when $C(n,k)/C(n,k+1) < 2$ while $C(n,k+1)/C(n,k+2) > 2$. When $C(n,k+1)/C(n,k+2) > 2$ some k+2-combinations in S^n_{k+2} get more than two k+1-combinations allocated to them; while when $C(n,k)/C(n,k+1) < 2$ some k+2-combinations in $S^n_{k+1}(n+1)$ do not get anything allocated to them. Thus, to maximize the number of k+2-combinations that get populated we might need reallocation. Note that there is no such a problem and the maximization is achieved when either both $C(n,k)/C(n,k+1) \leq 2$ and $C(n,k+1)/C(n,k+2) \leq 2$ or both $C(n,k)/C(n,k+1) \geq 2$ and $C(n,k+1)/C(n,k+2) \geq 2$.

It is an open problem how to do the reallocation in the special case discussed above. Without the reallocation, the recursive allocation procedure will give slightly worse results (in terms of number of gates) than the integer programming procedure. But for practical purposes we can combine the two procedures to get an algorithm that has the recursive allocation as its basis and calls the integer programming subroutine only for the special cases discussed above.

We implemented the procedures and ran computations for up to $n = 15$. The integer program was written and solved on AMPL which is a modeling language for Mathematical Programming. The complete integer program in AMPL is given in appendix section 7.2. The hypertree for $n = 5$ is given in appendix section 7.3.

4.5 Number of gates

In this section we count the number of gates in the hypertree built in the previous section and show that this number is a significant improvement on the number of gates in simple symmetrical diagrams.

The number of gates depends on the average number of k-combinations that each k+1-combination subtree is populated with. Thus we will distinguish cases based on the value of $C(n,k)/C(n,k+1)$.

Case 1: $C(n,k)/C(n,k+1) > 2$. In this case each k+1-combination subtree is populated with 2 or more k-combinations. Then if a k+1-combination subtree is populated by m k-combinations we need $m-1$ operational symbols F_k. This makes the total number of F_k-hypergates needed $C(n,k) - C(n,k+1)$. No simple gates are needed at this stage.

Case 2: $C(n,k)/C(n,k+1) \leq 2$. In this case following our principle we need to maximize the number of those k+1-combination subtrees which are populated by at least two k-combinations. Thus, each k+1-combination subtree is populated by 2 or 0 k-combinations. In the rare case when $C(n,k)$ is odd, one k+1-combination subtree will be populated by either exactly 1 or exactly 3 k-combination(s). Then the number of F_k-hypergates needed is $\lceil C(n,k)/2 \rceil$. In those k+1-combination subtrees where no k-combination is populated we just have the $\wedge$-product of the $k+1$ units. Thus the number of simple gates needed is $k*(C(n,k+1) - \lceil C(n,k)/2 \rceil)$. Note that instead of $\lceil C(n,k)/2 \rceil$ we could take just $C(n,k)/2$ since it makes only insignificant difference for big n and k.

Let p be the smallest number for which we still have Case 1. Then the total number of hypergates in Case 1 is:

$$\begin{aligned} & \sum_{k=p}^{n-1} (C(n,k) - C(n,k+1)) \\ = \; & (C(n,p) - C(n,p+1)) + (C(n,p+1) - C(n,p+2)) + (C(n,n-1) - C(n,n)) \\ = \; & C(n,p) - 1 \end{aligned}$$

Thus, the total number of hypergates in the hypertree is $C(n,p) - 1 + \sum_{k=1}^{p-1} C(n,k)/2$ and the total number of simple gates is $\sum_{k=1}^{p-1} [k * (C(n,k+1) - C(n,k)/2)]$.

To get an idea how much improvement we have compared to the simple symmetrical diagrams we also need to count the number of simple gates in those kind of diagrams. A k-out-of-n symmetrical diagram has $C(n,k) - 1$ OR-gates and $C(n,k) * (k-1)$ AND-gates. Thus, the total number of gates in all symmetrical diagrams is:

$$\sum_{k=1}^{n} [(C(n,k) - 1 + C(n,k) * (k-1)] = \sum_{k=1}^{n} [k * C(n,k) - 1] = n * 2^{n-1} - n$$

Table 1 compares the number of gates in hypertrees and simple trees for up to $n = 20$. The improvement factor is more than 2. Our computations show that the factor converges to

2.1 when n is increased to 40. The trees for larger n do not have much practical importance, so the comparisons are presented only for $n \leq 20$.

Table 1: Number of gates in hypertrees vs. simple trees

	hypertrees		simple trees	improvement
n	simple gates	hypergates	simple gates	factor
3	2	3	9	1.8
4	6	8	28	2
5	18	16	75	2.21
6	49	32	186	2.3
7	138	53	441	2.31
8	318	116	1016	2.34
9	737	240	2295	2.35
10	1679	492	5110	2.35
11	4007	916	11253	2.29
12	8866	1908	24564	2.28
13	19455	3916	53235	2.28
14	42440	7967	114674	2.27
15	94699	15422	245745	2.23
16	203618	31524	524272	2.23
17	436365	63944	1114095	2.23
18	932408	129064	2359278	2.22
19	2014060	253828	4980717	2.2
20	4266423	513456	10485740	2.19

Lower bounds on the number of gates

It is important to have lower bounds on the number of gates to see how much improvement is possible in minimizing the size of the hypertree. Here are some considerations in that regard.

Based on the principle (iii) of building the hypertree (Section 4.3) if a subtree has F_k as a root operation then the possible operations in the subtree are $\wedge$ and F_i where $i \leq k$. This principle gives a certain form to the hypertree which in a sense is equivalent to the canonical sum-of-products form of simple Boolean functions. Below we will call it a hypertree of canonical form or simply a *canonical hypertree*. The logic behind building the hypertree in Section 4.3 is to minimize the number of gates for this kind of canonical hypertrees (it was summarized in the discrete optimization problem stated at the end of Section 4.3). Thus, our conjecture is that the number of functional gates is optimal (or close to optimal) for the canonical hypertrees.

A trivial lower bound for the canonical hypertrees could be obtained by taking the number of gates in the canonical simple circuit of $n/2$-out-of-n diagram. For given n, the number of

gates in k-out-of-n diagram for any k is a lower bound for the number of gates in a canonical hypertree. And that lower bound is maximized when we take $k = n/2$ in which case the lower bound is $n/2 * C(n, n/2) - 1$. But this is not a tight lower bound as can be seen from Table 2. As stated above our conjecture is that the number of gates obtained by our construction should be much closer to the minimum possible number of gates.

Table 2: Lower bound vs. number of gates

n	lower bound	gates in hypertree
8	279	434
9	566	977
10	1259	2171
11	2540	4923
12	5543	10774
13	11153	23371
14	24023	50407
15	48262	110121
16	102959	235142
17	206634	500309
18	437579	1061472
19	877590	2267888
20	1847559	4779879

It is possible to further reduce the number of functional gates if we build a hypertree which is not in a canonical form. For example, the hypertree in Figure 1,

$F_2(F_2(F_1(x_1, x_2), F_1(x_1, x_3)), F_1(x_2, x_3))$

with 5 hypergates can be changed to an equivalent hypertree by applying the distributive law:

$F_2(F_1(x_1, F_2(x_2, x_3)), F_1(x_2, x_3))$

This new hypertree has 4 hypergates. Thus, non-canonical hypertrees can further reduce the number of functional gates. It is an open question how to design a systematic reduction algorithm to obtain a non-canonical hypertree of minimal size for any n and k.

It should be noted that finding lower bounds on the circuit size is not an easy problem ([1, 9]). It is particularly stated in [1]: "There has been a great deal of pessimism about the likelihood of anyone making significant new progress in circuit complexity. Much of this pessimism can be traced to the fact that there has been very little progress on separating circuit complexity classes in the two decades that have passed since the work of Razborov and Smolensky. An additional factor is that Razborov and Rudich identified some significant obstacles that must be overcome before circuit lower bounds can be proved, in their work on Natural Proofs ." Of course, the lower bounds mentioned above are about the general

case of Boolean circuits while we consider a special type of circuits. But on the other hand, our problem (i) integrates several circuits versus simplifying just one circuit, (ii) uses both hypergates and simple gates. This makes the minimization problem even more complex.

5. Integration through Boolean Algebra Hyperterms

Most logic circuits have not only AND-gates and OR-gates but also NOT-gates. This corresponds to unary negation operation which makes the lattice a Boolean algebra. Integration can also be done for circuits corresponding to Boolean algebra terms. Boolean algebra hyperterms still have only binary functional variables which can take values $\vee$ and $\wedge$. There is no need for unary functional variables because Boolean algebras have only one unary operation.

Here is one example showing how the integration can be done in this case and how effective it can be. Suppose we want to integrate two circuits corresponding to simple Boolean terms $x \vee (\bar{y} \wedge z) \vee y$ and $(x \wedge y) \vee \bar{y} \vee z$. The integrating hyperterm $\vee(G(x,y), H(\bar{y},z))$ is represented by the hypertree given in figure 6. When $G \to \vee, H \to \wedge$, the hyperterm becomes $x \vee (\bar{y} \wedge z) \vee y$; when $G \to \wedge, H \to \vee$, the hyperterm becomes $(x \wedge y) \vee \bar{y} \vee z$. The hyperterm has 2 hypergates and 2 simple gates compared to 8 simple gates in the two simple terms (including the NOT-gates). Note that we didn't need a hypergate for the root operation of the hypertree. If we

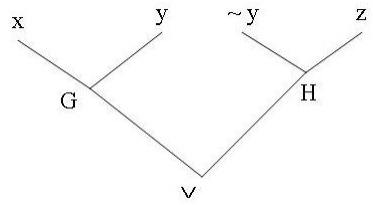

Figure 6: Boolean hypertree

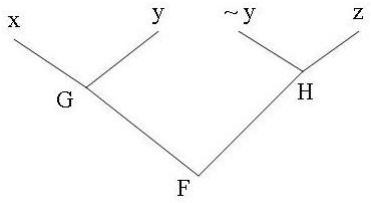

Figure 7: Modified Boolean hypertree

replace that operation with a functional variable F as it is done in figure 7 then the new hypertree could integrate even more simple circuits. Namely,

- when $F \to \vee, G \to \vee, H \to \wedge$ the hyperterms becomes $(x \vee y) \vee (\bar{y} \wedge z) = x \vee (\bar{y} \wedge z) \vee y$;
- when $F \to \vee, G \to \wedge, H \to \vee$ the hyperterm becomes $(x \wedge y) \vee (\bar{y} \vee z) = (x \wedge y) \vee \bar{y} \vee z$;
- when $F \to \vee, G \to \wedge, H \to \wedge$ the hyperterm becomes $(x \wedge y) \vee (\bar{y} \wedge z)$;
- when $F \to \wedge, G \to \vee, H \to \vee$ the hyperterm becomes $(x \vee y) \wedge (\bar{y} \vee z) = (x \wedge \bar{y}) \vee (x \wedge z) \vee (y \wedge z)$;
- when $F \to \wedge, G \to \wedge, H \to \vee$ the hyperterm becomes $(x \wedge y) \wedge (\bar{y} \vee z) = x \wedge y \wedge z$;

- when $F \rightarrow \wedge, G \rightarrow \vee, H \rightarrow \wedge$ the hyperterm becomes $(x \vee y) \wedge (\bar{y} \wedge z) = x \wedge \bar{y} \wedge z$.

Thus, the hyperterm with 3 hypergates and 1 simple gate can integrate 6 simple terms with total number of gates 21.

Finally we note that most of the future directions for lattice hyperterms listed in the next section can be stated for Boolean hyperterms too.

6. Future Directions

Below we list several directions of future work.

- The solution to the general problem stated in section 3 is an open question. Complexity results also would be interesting. While the general problem is likely to be hard to solve, interesting special cases with effective hyperterm solutions could be a good starting point.

- Finding non-trivial tight lower bounds on the number of gates in a hypertree for k-out-of-n symmetrical diagrams is an open question.

- It is an open problem how to do the reallocation of k-combinations in the special case when $C(n,k)/C(n,k+1) < 2$ while $C(n,k+1)/C(n,k+2) > 2$ (in the Recursive Allocation Method).

- An effective construction of a hypergate using tools of modern electronics engineering is an open question.

- We integrated k-out-of-n symmetrical diagrams for all $k \in 1, ..., n$. While this was an interesting theoretical problem, in most practical applications we might not need the diagrams for all k. For example, for $n = 4$ we might need the diagrams only for $k = 2, 3$. It would be interesting to show how to extend our results to those cases.

- The students/projects problem stated at the beginning of subsection 4.4 is an optimization problem that to the best of our knowledge has not been considered before. It would be interesting to obtain solution methods and complexity results for it.

- In some situations we might want to integrate the lattice terms in more than one hyperterm if the number of gates is minimized that way. Thus, the general problem stated in section 3 could be modified to allow several hyperterms.

7. Appendix

7.1 A primitive approach of building a hypergate

A hypergate can fulfill the functions of both OR-gate and AND-gate if appropriate modes are chosen. In figure 8 we give an example of possible realization of a hypergate. The cross-shaped switch device in the middle of the circuit can be in two different modes. When it is in the position as in the left circuit x and y are in series connection. When the switch is in the position as in the right circuit x and y are in parallel connection.

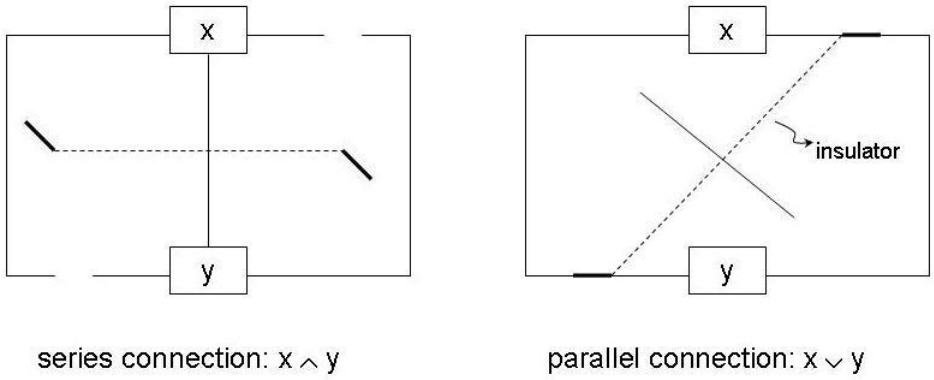

Figure 8: Hypergate

Of course, this is a primitive approach of building a hypergate. An effective engineering realization of a hypergate using modern electronics is an interesting open question.

7.2 The integer program for solving the subtree populating problem

```
param n integer > 0;

param k integer > 0, < n;

set S := 0 .. n - 1;
```

```
set SS := 0 .. 2**n - 1;

set POW {i in SS} := {j in S: (i div 2**j) mod 2 = 1};

set k_comb_ind := {i in SS: card(POW[i])==k};
set K_COMB {i in k_comb_ind } := {j in S: (i div 2**j) mod 2 = 1};

set k1_comb_ind := {i in SS: card(POW[i])==k+1};
set K1_COMB {i in k1_comb_ind } := {j in S: (i div 2**j) mod 2 = 1};

param compat {i in k_comb_ind, j in k1_comb_ind}:=
if (K_COMB[i] within K1_COMB[j]) then 1 else 0;

var assign {i in k_comb_ind, j in k1_comb_ind: compat[i,j]=1} binary;

var pursued {j in k1_comb_ind} binary;

maximize assigned_k1_combin: sum{j in k1_comb_ind} pursued[j];

s.t. each_k_combin_assigned{i in k_comb_ind}:
sum{j in k1_comb_ind: compat[i,j]=1} assign[i,j] = 1;

s.t. assigned_atleast_two_if_pursued{j in k1_comb_ind}:
2 * pursued[j] <= sum{i in k_comb_ind: compat[i,j]=1} assign[i,j];

s.t. cant_assigned_if_not_pursued{i in k_comb_ind, j in k1_comb_ind: compat[i,j]=1}:
assign[i,j] <= pursued[j];
```

7.3 The hypertree for 5 units

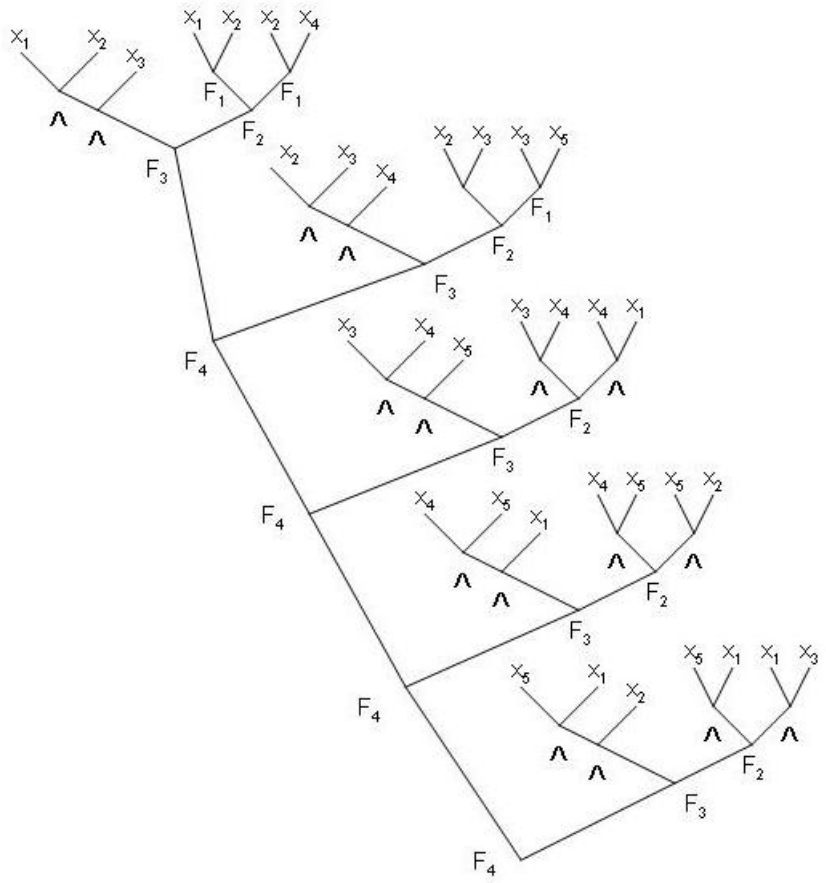

Figure 9: Hypertree with 5 units

References

[1] E. Allender, Circuit Complexity, Kolmogorov Complexity, and Prospects for Lower Bounds, In Proceedings of 10th International Workshop on Descriptional Complexity of Formal Systems, Charlottetown, Canada 2008.

[2] U. Brenner, M. Struzyna, J. Vygen, *BonnPlace: Placement of leading-edge chips by advanced combinatorial algorithms*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27 (2008), pp. 1607–1620.

[3] D. Buchfuhrer and C. Umans, *The complexity of Boolean formula minimization*, Automata, Languages and Programming, Lecture Notes in Computer Science, Volume 5125, 2008, pp. 24–35.

[4] Y. Chen and Q. Yang, *Reliability of two-stage weighted k-out-of-n systems with components in common*, IEEE transactions on reliability, 54(3) (2005), pp. 431–440.

[5] K. Denecke, J. Koppitz, S. Shtrakov, J. Meakin, *Multi-hypersubstitutions and colored solid varieties*, International Journal of Algebra and Computation, 16(4) (2006), pp. 797–815.

[6] K. Denecke, K. Saengsura, *Separation of clones of cooperations by cohyperidentities*, Discrete Mathematics, 309(4) (2009), pp. 772–783.

[7] K. Denecke and S. Wismath, *Hyperidentities and clones*, Algebra, Logic and Applications, Volume 14, Gordon and Breach Science Publishers, 2000.

[8] G. Gratzer, *General lattice theory*, Birkhuser, 2003.

[9] V. Kabanets and J. Cai, *Circuit minimization problem*, In Proceedings of 32nd Symposium on Theory of Computing, Portland, Oregon 2000, pp. 73–79

[10] B. Korte and J. Vygen, *Combinatorial problems in chip design*, In: Building Bridges Between Mathematics and Computer Science (M. Grötschel, G.O.H. Katona, eds.), Springer, Berlin 2008, pp. 333–368.

[11] W. Kuo and M. Zuo, *Optimal reliability modelling*, Wiley and Sons, New Jersey, 2003.

[12] Y. Movsisyan, *Binary Representations of Algebras with At Most Two Binary Operations: A Cayley Theorem for Distributive Lattices*, International Journal of Algebra and Computation, 19(1) 2009, pp. 97–106.

[13] Y. Movsisyan, A. Romananowska, J. Smith, *Superproducts, Hyperidentities, And Algebraic Structures Of Logic Programming*, Journal of Combinatorial Mathematics and Combinatorial Computing, Volume 58 (2006), pp. 101-112.

[14] A. Prasad, V. Shende, I. Markov, J. Hayes, K. Patel, *Data structures and algorithms for simplifying reversible circuits*, ACM Journal on Emerging Technologies in Computing Systems 2(4), (2006), pp. 277–293.

[15] O. Prokopyev and M. Pardalos, *Minimum ϵ-equivalent Circuit Size Problem*, Journal of Combinatorial Optimization, 8(4) (2004), pp. 495–502.

[16] W. Puninagool and S. Leeratanavalee, *Complexity of terms, superpositions, and generalized hypersubstitutions*, Computers and Mathematics with Applications, 59(2) (2010), pp. 1038–1045.

[17] T. Sasao, *Switching theory for logic synthesis*, Springer, 1999.

[18] E. Sjenz-de-Cabezon and H. Wynn, *Betti numbers and minimal free resolutions for multi-state system reliability bounds*, Journal of Symbolic Computation, Volume 44 (2009), pp. 1311-1325.

[19] E. Sjenz-de-Cabezon and H. Wynn, *Computational algebraic algorithms for the reliability of generalized k-out-of-n and related systems*, Mathematics and Computers in Simulation, 2010, article in press.

[20] W. Taylor, *Hyperidentities and hypervarieties*, Aequationes Mathematicae, 23(1) (1981), 30–49.