Synthesis of Multi-Level Boolean Networks

Invited chapter for "Boolean Methods and Models
in Mathematics, Computer Science and Engineering",
Yves Crama and Peter L. Hammer (eds.), pp. 675-722,
Encyclopedia of Mathematics and its Applications 134,
Cambridge University Press, 2010

Tiziano Villa* Robert K. Brayton† Alberto L. Sangiovanni-Vincentelli‡

1 Boolean Networks

1.1 Introduction

Two-level logic minimization has been a success story both in terms of theoretical understanding and availability of practical tools (like ESPRESSO) [2, 37, 30, 14]. However, two-level logic is not suitable to implement large Boolean functions, whereas multi-level implementations allow to trade-off area and delay. Multi-level logic synthesis has the objective to explore multi-level implementations guided by some function of the following metrics:

1. the **area** occupied by the logic gates and interconnect (eg., approximated by **literals**, which correspond to **transistors** in technology-independent optimization).
2. the **delay** of the longest path through the logic.
3. the **testability** of the circuit, measured in terms of the percentage of faults covered by a specified set of test vectors, for an appropriate fault model (eg., single stuck faults, multiple stuck faults, etc.).
4. the **power** consumed by the logic gates and wires.

Often good implementations must satisfy simultaneously upper or lower constraints placed on these parameters and look for good compromises among the cost functions.

It is common to classify optimization as technology-independent vs. technology-dependent, where the former represents a circuit by a network of abstract nodes, whereas the latter represents a

*Dipartimento d'Informatica, Universita' di Verona, Strada Le Grazie, 15, 37134 Verona, Italy. E-mail: tiziano.villa@univr.it.

†Department of EECS, University of California at Berkeley, Berkeley, CA 94720. E-mail: brayton@eecs.berkeley.edu.

‡Department of EECS, University of California at Berkeley, Berkeley, CA 94720. E-mail: alberto@eecs.berkeley.edu.

circuit by a network of the actual gates available in a given library or programmable architecture. A common paradigm is to try first technology-independent optimization and then map the optimized circuit into the final library (technology mapping). In some cases it has been found advantageous to iterate this process; then it is called technology semi-independent optimization. The fact of splitting multi-level optimization in two steps has been suggested by the complexity of the problem, but pays the penalty of selecting sub-optimal solutions. It must be said that, contrary to the two-level and three-level cases, there is no established theory of optimum multi-level implementations, so here we are in the domain of heuristics.

Example 1.1 *Given the specification*

$$\begin{aligned} w &= ab + \overline{a}\overline{b} \\ \textit{if } w \textit{ then } z &= cd + \overline{a}\overline{d}; u = cd + \overline{a}\overline{d} + e(f+b) \\ \textit{else } z &= e(f+b); u = (cd + \overline{a}\overline{d})e(f+b) \end{aligned}$$

A straighforward multi-level implementation is:

$$\begin{aligned} w &= ab + \overline{a}\overline{b} \\ z &= w(cd + \overline{a}\overline{d}) + \overline{w}e(f+b) \\ u &= w(cd + \overline{a}\overline{d} + e(f+b)) + \overline{w}(cd + \overline{a}\overline{d})e(f+b) \end{aligned}$$

A more succinct multi-level implementation is:

$$\begin{aligned} w &= ab + \overline{a}\overline{b} \\ t &= cd + \overline{a}\overline{d} \\ s &= e(f+b) \\ z &= wt + \overline{w}s \\ u &= w(t+s) + \overline{w}ts \end{aligned}$$

Interesting work on the efficient decomposition of Boolean functions into networks of digital gates had been carried on since the sixties [29, 19, 44, 22, 18]. But it was only in the eighties that a combination of more powerful theory and computers gave wings to the field, culminating in the development of modern logic synthesis packages, often originated in academia and then engineered into industrial tools by a number of Electronic Design Automation (EDA) companies. The first industrially employed logic synthesis tool was LSS [17, 15, 16], developed at IBM and based on local transformations. A pioneering package from academia was MIS [4, 5], developed at Berkeley in the mid eighties. Two students who played a pivotal role in the theory and implementation of MIS were R. Rudell [38, 6] and A. Wang [43, 6, 21], who then joined a successful start-up in EDA. In parallel a research group at the University of Colorado, Boulder led by G. Hachtel developed the package BOLD [1]. A few years later a Berkeley team developed SIS [39, 40], a more advanced and complete package (including also sequential synthesis), which became the de-facto standard for comparing algorithms in multi-level logic synthesis. The current champion is the package ABC [41] developed at UCB, by a team whose main architect is A. Mishchenko; ABC achieves high scalability by using FRAIGs, i.e., a representation based on functionally reduced two-input AND gates and inverters [35].

In this chapter we will provide a rigorous survey of the basics of modern multi-level logic synthesis, developed since the beginning of the eighties under the influence of seminal papers on factoring and division by R. Brayton and C. McMullen, at the IBM Yorktown Research Labs [3, 8, 7]. We will describe first the setting of the problem based on the abstraction of a Boolean network and on factored forms of Boolean functions. Then we will describe Boolean division, for completely specified and incompletely specified Boolean functions, and heuristic algorithms to perform it. Next we will describe algebraic division, introducing weak algebraic division, and we will discuss how to find common algebraic divisors by restricting the search to kernels and other subsets of divisors. Finally we will show how division can be the engine of the main operations to restructure a Boolean network, namely factoring, decomposition, substitution and extraction.

Textbook expositions of multi-level logic synthesis can be found in [33, 24, 20, 23].

1.2 Network Representation

Let us consider the network representation first. A model used in multi-level logic synthesis is a network of nodes that are single-output functions, where each node is abstracted as a sum-of-products or a factored form.

Definition 1.1 *A* **Boolean network** *is a three-tuple $\mathcal{N} = (V, E, \mathbf{f})$, consisting of a directed acyclic graph (DAG) $G = (V, E)$ and a collection of logic functions $\mathbf{f}$.*

The set of nodes is $V = V^I \cup V^{int} \cup V^O$, where V^I is the set of source nodes, V^{int} is the set of internal nodes, V^O is the set of sink nodes. There may be an arc from any node in V^I to any node in V^{int}, and from any node in V^{int} to any node in V^{int}, as long as the graph is acyclic. For every node in V^O there is a unique node in V^{int} from which there is an arc to it. The network is characterized as follows:

1. *A primary input $x_i, i = 1, \ldots, m$ is associated to each node $i \in V^I, i = 1, \ldots, m$.*
2. *An internal variable $y_i, i = 1, \ldots, n$ and a representation of a completely specified logic function $f_i, i = 1, \ldots, n$ are associated to each internal node $i \in V^{int}, i = 1, \ldots, n$. It holds that $y_i = f_i(y_{i_1}, \ldots, y_{i_n}, x_{i_1}, \ldots, x_{i_m}), i = 1, \ldots, n$, where $y_{i_1}, \ldots, y_{i_n}$ are the internal variables associated to internal nodes for which there is arc to node i, and where $x_{i_1}, \ldots, x_{i_m}$ are the primary inputs associated to source nodes from which there is arc to node i.*
3. *A primary output $z_i, i = 1, \ldots, p$ and a completely specified output function $f_i, i = n + 1, \ldots, n + p$ are associated to each node $i \in V^O, i = 1, \ldots, p$. It holds that $f_i = f_k$, where f_k is the function associated to the unique internal node k from which there is an arc to node i.*

The set of primary inputs is represented by the vector of variables $\mathbf{x} = (x_1, \ldots, x_m)$. The set of internal variables is represented by the vector of variables $\mathbf{y} = (y_1, \ldots, y_n)$. The set of primary outputs is represented by the vector of variables $\mathbf{z} = (z_1, \ldots, z_p)$. The set of functions is represented by the vector of functions $\mathbf{f} = (f_1, \ldots, f_n, f_{n+1}, \ldots, f_{n+p})$

There is an external don't care set associated to the primary outputs and represented by the vector of completely specified functions $\mathbf{d} = (d_1(\mathbf{x}), \ldots, d_p(\mathbf{x}))$, where $d_i(\mathbf{x}) = 1$ if and only the primary output z_i under input $\mathbf{x}$ is unspecified, i.e., it may be either 0 or 1.

Notice that if $\mathbf{x}$ is a don't care for the primary output z_i, it means that we are free to synthesize a network that computes either value, choosing the one that makes the implementation more cost-effective.

Definition 1.2 *A* **literal** *is a variable or its complement, e.g., x or $\overline{x}$. A* **cube** *is a set C of literals,, e.g., in set notation $\{x, y, \overline{z}\}$; a cube represents the conjunction of its literals, e.g., $xy\overline{z}$. An* **expression** *or* **cover** *F is a set of cubes, e.g., in set notation $\{\{\overline{x}\}, \{x, y, \overline{z}\}\}$; an expression represents the disjunction of its cubes, also called Sum-Of-Product(SOP), e.g., $\overline{x} + xy\overline{z}$.*

Definition 1.3 *The* **support** *of an expression F, $sup(F)$, is the set of variables that appear in F, i.e.,*

$$sup(F) = \{x \mid \exists \text{ cube } C \text{ such that } x \in C \text{ or } \overline{x} \in C \ \}.$$

For example $sup(xy + \overline{xz}) = \{x, y, z\}$.

Definition 1.4 *Two expressions F and G are* **orthogonal**, *$F \perp G$, if they have disjoint support, i.e., $sup(F) \cap sup(G) = \emptyset$.*

Definition 1.5 *A node j is a* **fanin** *node of a node i if function f_i depends on variable y_j explicitly, i.e., there is an arc from j to i. The set of all fanins of a node i is denoted by*

$$FI(i) = \begin{cases} \emptyset & i \text{ is a primary input} \\ \{j \mid j \in sup(i)\} & otherwise \end{cases}$$

Definition 1.6 *The* **transitive fanin** *of a node i, denoted by $TFI(i)$, is defined recursively as:*

$$TFI(i) = \begin{cases} \emptyset & i \text{ is a primary input} \\ FI(i) \cup \bigcup_{j \in FI(i)} TFI(j) & otherwise \end{cases}$$

Definition 1.7 *A node j is a* **fanout** *node of a node i if function f_j depends on variable y_i explicitly, i.e., there is an arc from i to j. The set of all fanouts of a node i is denoted by*

$$FO(i) = \begin{cases} \emptyset & i \text{ is a primary output} \\ \{j \mid i \in sup(j)\} & otherwise \end{cases}$$

Definition 1.8 *The* **transitive fanout** *of a node i, denoted by $TFO(i)$, is defined recursively as:*

$$TFO(i) = \begin{cases} \emptyset & i \text{ is a primary output} \\ FO(i) \cup \bigcup_{j \in FO(i)} TFO(j) & otherwise \end{cases}$$

An example of Boolean network is given in Fig. 2.

The given notion of Boolean network is abstract enough that it can be used both for technology-independent and technology-dependent representations. What makes the difference is the type of node representation. Nodes may be abstract functions of many sorts in a technology-independent representation, whereas they are valid gates from a library in a technology-dependent representation. In the former case nodes may be classified as follows:

- General node: each node is the representation of an **arbitrary** logic function, then a theory is easier to develop since there are no arbitrary restrictions dependent on the technology. This is the choice of SIS, with the restriction that, in SIS, nodes must be single-output with the following possibile choices of function representation:
  - Sum-of-products form

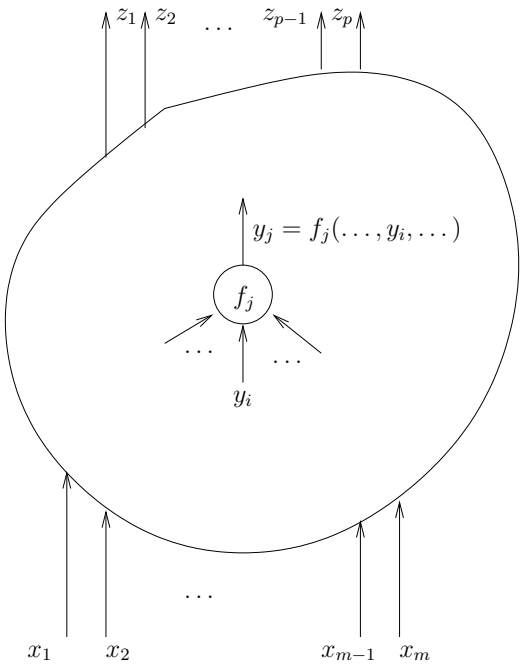

Figure 1: Structure of a Boolean network. External inputs are $x_1, \ldots, x_m$; external outputs are $y_1, \ldots, y_p$. A node f_j with output signal y_j computes the function $y_j = f_j(y_{j_1}, \ldots, y_{j_n}, x_{j_1}, \ldots, x_{j_m})$, where some local inputs may be external inputs.

  - Factored form
  - Binary decision diagram (BDD)

- Generic node: every node in the network is a simple generic node, like a 2-input NAND gate; some manipulations are much faster using this structure, but the network is finely decomposed in a particular way, and some natural structures may be lost. Recently, And-Invertor-Graphs (AIGs) have seen increasingly more use [28, 34].

- Discrete node. A node can be one of a small set of logic functions, such as AND, OR, NOT, DECODE, ADD or other complex blocks of logic manipulated as a single unit, allowing also multiple output nodes; it is used mostly in rule-based systems and a theory for such networks seems more difficult.

In the sequel we will develop technology-independent multi-level optimization based on general nodes represented as in SIS by Sum-Of-Products (SOPs) and factored forms. SOPs are easy to manipulate and there are many algorithms for their minimization, spawned by a rigorous theory of optimum two-level forms. However, their main disadvantage is that they do not represent reliably

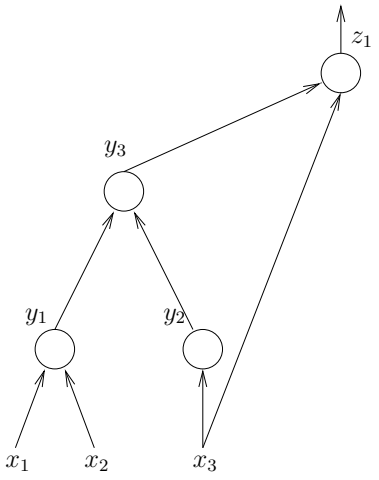

Figure 2: An example of Boolean network. The nodes compute the following functions: $y_1 = x_1x_2$, $y_2 = \overline{x}_3$, $y_3 = y_1y_2$, $z_1 = y_3 + x_3$.

the logic complexity of a function. Consider

$$\begin{array}{rcl} f & = & ad + ae + bd + be + cd + ce \\ \overline{f} & = & \overline{a}\overline{b}\overline{c} + \overline{d}\overline{e} \end{array}$$

These differ in their implementation only by one invertor, but their SOPs differ by many products and literals. So SOPs are not a good estimator of progress during logic minimization.

For ROBDDs we refer to another chapter in the same collection. Here it suffices to say that they represent a function and its complement with the same complexity, like factored forms; however they are not a good estimator for the complexity of implementation, because they are effectively networks of muxes restricted to be controlled by primary input variables. BDDs are a good replacement for truth tables, because they are canonical for a given ordering, and operations on them are well defined and efficient, with true support (dependency) exposed explicitly. Finding the best ordering is a difficult problem; moreover, there are functions like multipliers for which there is no ordering that yields a small BDD.

As it will be seen soon in the formal definition, factored forms are recursively defined as sums or products of factored forms down to the terminal cases of single literals. An example of factored form is: $(ab+\overline{b}c)(c+\overline{d}(e+a\overline{c}))+(d+e)(fg)$. Factored forms represent a function and its complement with the same complexity:

$$\begin{array}{rcl} f & = & ad + ae + bd + be + cd + ce \\ \overline{f} & = & \overline{a}\overline{b}\overline{c} + \overline{d}\overline{e} \\ f & = & (a + b + c)(d + e) \end{array}$$

They are good estimators of complexity of logic implementation and do not blow up easily; in many design styles the implementation of a function corresponds directly to its factored form (e.g.,

complex-gate CMOS, where factored form literal count correlates to transistor count that correlates to area, but area depends on wiring too). A disadvantage is that there are not as many algorithms available for manipulation, hence usually they must be converted into SOPs before operating on them.

1.3 Factored Forms

Definition 1.9 *An* **algebraic expression** *F is a sum of products representation of a logic function which is minimal with respect to single cube containment, i.e., such that every proper subset of F represents another function than F.*

Example 1.2 *$ab + cd$ is an algebraic expression, but $a + ab$ and $ab + abc + cd$ are not algebraic expressions (e.g., $a + ab$ is ruled out because factoring would yield $a(1 + b)$, where the 1 indicates single-cube containment).*

Definition 1.10 *The* **product** *of two expressions F and G is a set defined by*

$$FG = \{cd \mid c \in F \text{ and } d \in G \text{ and } cd \neq \emptyset\}.$$

Example 1.3 $(a + b)(ce + d + \overline{a}) = ace + ad + bce + bd + \overline{a}b$.
The result is not necessarily an algebraic expression, e.g., $(a + b)(a + c) = aa + ac + ab + bc$.

Definition 1.11 *FG is an* **algebraic product** *if F and G are algebraic expressions and have disjoint support (that is, they have no input variables in common), otherwise it is a* **Boolean product**.

Example 1.4 *$(a + b)(c + d) = ac + ad + bc + bd$ is an algebraic product, but $(a + b)(a + c) = aa + ac + ab + bc$ is a Boolean product.*

Lemma 1.1 *The algebraic product of two expressions F and G is an algebraic expression.*

Proof. By contradiction. Suppose there exists $c_i d_j \subseteq c_k d_l$ for some i, j, k, l, with either $i \neq k$ or $j \neq l$. Since $sup(F) \cap sup(G) = \emptyset$, then $c_i \subseteq c_k$ and $d_j \subseteq d_l$, against the hypothesis that F and G are algebraic expressions. □

A factored form is a parenthesized expression.

Definition 1.12 *A* **factored form** *can be defined recursively by the following rules:*

A factored form is either a product or a sum where:

- *A* product *is either a single literal or a product of factored forms.*
- *A* sum *is either a single literal or a sum of factored forms.*

In effect, a factored form is a product of sums of products of ... or a sum of products of sums of ... factored forms.

Any logic function can be represented by a factored form and any factored form is a representation of some logic function.

Example 1.5

- *Examples of factored forms are:*

$$x$$
$$\overline{y}$$
$$ab\overline{c}$$
$$a + \overline{b}c$$
$$((\overline{a} + b)cd + e)(a + \overline{b}) + \overline{e}$$

- *The following is not a factored form, because complemention is not allowed, since complements are allowed only on literals:*

$$\overline{(a + b)}c$$

- *The following are three equivalent factored forms (factored forms are not unique):*

$$ab + c(a + b)$$
$$bc + a(b + c)$$
$$ac + b(a + c)$$

Definition 1.13 *The* **factorization value** *of an algebraic factorization $F = G_1G_2 + R$ is defined to be*

$$\begin{aligned} fact\_val(F, G_2) &= lits(F) - (lits(G_1) + lits(G_2) + lits(R)) \\ &= (\mid G_1 \mid -1)lits(G_2) + (\mid G_2 \mid -1)lits(G_1) \end{aligned}$$

assuming G_1, G_2, and R are algebraic expressions. Here, for any algebraic expression P, $lits(P)$ = number of literals in SOP form of P, and $\mid P \mid$ is the number of products of the SOP P.

The factorization value is the number of literals saved by doing one level of factoring. It is important that the expressions are algebraic.

Example 1.6 *The algebraic expression*

$$F = ae + af + ag + bce + bcf + bcg + bde + bdf + bdg$$

can be expressed in the factored form

$$F = (a + b(c + d))(e + f + g)$$

Note that only 7, rather than 24 literals are required.

If $G_1 = (a+bc+bd)$, and $G_2 = (e+f+g)$, then $R = \emptyset$, and $fact\_val(F, G_2) = (2)(3)+(2)(5) = 16$. Note that the given factored form saved 17, rather than 16 literals. The extra literal that was saved comes from recursively applying the formula to the factored form of G_1.

The following facts are true:

- Factored forms are more compact representations of logic functions than the traditional sum-of-products forms. For example, the following factored form has 10 literals

$$(a+b)(c+d(e+f(j+i+h+g)))$$

but when represented as a sum-of-products form it requires 42 literals

$$ac+ade+adfg+adfh+adfi+adfj+bc+bde+bdfg+bdfh+bdfi+bdfj$$

- Every sum-of-products form can be viewed as a factored form.

When measured in terms of number of inputs, there are functions whose size is exponential in sum-of-products forms, but polynomial in factored forms.

Example 1.7 *Consider the Achilles' heel function represented as product-of-sums:*

$$\prod_{i=1}^{i=n/2}(x_{2i-1}+x_{2i}).$$

There are n literals in the factored form and $(n/2)\times 2^{n/2}$ literals in the sum-of-products form. Factored forms are useful in estimating area and delay in a multi-level logic synthesis and optimization system.

In most design styles (for example, complex-gate CMOS design) the implementation of a function corresponds directly to its factored form.

Factored forms can also be graphically represented as labeled trees, called factoring trees, in which each internal node including the root has a label of either "+" (or "$\vee$") for disjunction, or "$*$" for conjunction, and each leaf has a label of either a variable or its complement.

Example 1.8 *Fig. 3 shows the computation tree of the factored form $((\overline{a}+b)cd+e)(a+\overline{b})+\overline{e}$.*

Definition 1.14 *The* **size** *of a factored form F (denoted $\rho(F)$) is the number of literals in the factored form. A factored form is* **optimum** *if no other equivalent factored form has fewer literals.*

Example 1.9 *$\rho((a+b)c\overline{a})=4$ and $\rho((a+b+cd)(\overline{a}+\overline{b}))=6$.*

Definition 1.15 *A factored form F is* **positive unate** *in x if x appears in F but $\overline{x}$ does not.*
A factored form F is **negative unate** *in x if $\overline{x}$ appears in F and x does not.*
A factored form F is **unate** *in x if it is either positive unate or negative unate in x.*
A factored form F is **binate** *in x if it is not unate in x.*

Example 1.10 *$(a+\overline{b})c+\overline{a}$ is positive unate in c, negative unate in b and binate in a.*

Definition 1.16 *The* **cofactor** *of a factored form F with respect to a literal x_1 (or $\overline{x}_1$) is the factored form $F_{x_1}=F|_{x_1=1}(x)$ (or $F_{\overline{x}_1}=F|_{\overline{x}_1=1}(x)$) obtained by*

1. *replacing all occurrences of x_1 with 1 and all occurrences of $\overline{x}_1$ with 0, and*

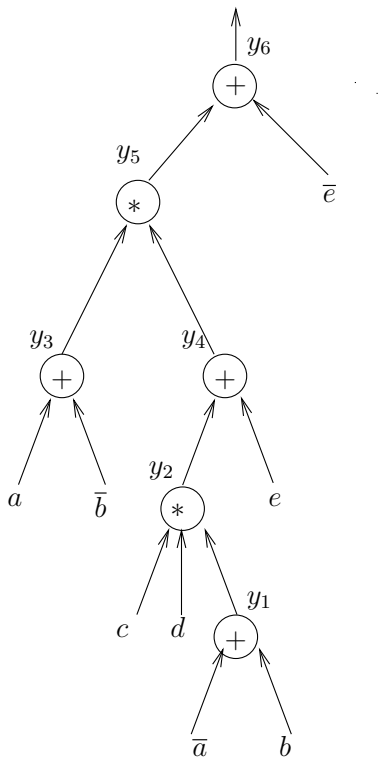

Figure 3: Factoring tree from Example 1.8. It represents the function $y_6 = ((\overline{a}+b)cd+e)(a+\overline{b})+\overline{e}$.

2. simplifying the factored form using the following identities of Boolean algebra:

$$\begin{aligned} 1x &= x \\ 1+x &= 1 \\ 0x &= 0 \\ 0+x &= x \end{aligned}$$

The cofactor of a factored form F with respect to a cube c is a factored form, denoted by F_c, obtained by successively cofactoring F with respect to each literal in c.

After constant propagation (all constants are removed), part of the factored form may appear as $G + G$. In general, G is another factored form. In fact, the two Gs may have different factored forms. Identifying these equivalent factored forms to apply the simplification $G + G = G$ is a non-trivial task.

Example 1.11 *Let $F = (x + \overline{y} + z)(\overline{x}u + \overline{zy}(v + \overline{u}))$ and $c = v\overline{z}$. Then*

$$\begin{aligned} F_{\overline{z}} &= (x + \overline{y})(\overline{x}u + \overline{y}(v + \overline{u})) \\ F_c &= (x + \overline{y})(\overline{x}u + \overline{y}) \end{aligned}$$

Note that cofactoring does not preserve algebraic expressions: $F = abc + bcd$, $F_a = bc + bcd$.

Sum-of-products forms are used as the internal representation of logic functions in most multi-level logic optimization systems. The advantage is that good algorithms for manipulating SOPs are available. Disadvantages are twofold:

1. The quality of solutions of SOP algorithms is unpredictable: they may accidentally generate a function whose sum-of-products form is too large.

2. Factoring algorithms have to be used constantly to provide an estimate for the size of the Boolean network, so that time spent when factoring may become significant; therefore there is a need of quick but still good factoring methods.

A possible solution to overcome the disadvantages is to avoid sum-of-products forms by using factored forms as internal representation. However this is not practical, unless we know how to perform logic operations on the factored forms directly without converting them to sum-of-products forms. Extensions to factored forms of the most common logic operations have been partially provided, but more research is needed.

1.4 Incompletely Specified Boolean Functions

Definition 1.17 *An incompletely specified function $\mathcal{F} = (f, d, r) : B^n \rightarrow \{0, 1, \star\}$ ($\star$ stands for* don't care *value) is a triple of completely specified Boolean functions f, d and r, respectively the onset, don't care and offset functions; i.e, $f(x) = 1$ if and only if $\mathcal{F}(x) = 1$, $d(x) = 1$ if and only if $\mathcal{F}(x) = *$, $r(x) = 1$ if and only if $\mathcal{F}(x) = 0$, where f, d, r are a partition of B^n, i.e., they are disjoint and together they cover all of B^n. When the don't care function d is empty, we have a completely specified Boolean function.*

An equivalent view is to say that an incompletely specified function $\mathcal{F} = (f, d, r)$ is a collection of completely specified functions g such that $f \subseteq g \subseteq f \vee d$. A cover for such a g (i.e., a SOP expression representing g) is said to be a **cover** of $\mathcal{F} = (f, d, r)$, and sometimes - with slight abuse of notation - g itself is said to be a cover of $\mathcal{F} = (f, d, r)$.

For minimization of incompletely specified Boolean functions there is a huge literature [37] and there are practical software packages [2]. Often in the text we will refer to applying minimization to functions associated to nodes of a Boolean network, e.g., the operation *minimize* (a.k.a. *simplification* or *simplify*), or *minimize with don't cares*, the latter when we underline that there is a non-empty don't care function d. Since each node has - among others - a representation of the associated function as a SOP, by minimization we mean applying any technique that will reduce the number of product terms and literals of the expression representing the function. According to the context and the practical requirements, we may be interested only to a light minimization, for instance at least ensuring that the final cover has no single-cube containment, or that it is an irredundant cover (no cube can be removed without uncovering at least a point of the onset), or to an absolute minimum. In practice often we are interested to an irredundant cover (minimal cover, which is a local optimum) as it can be obtained by the program ESPRESSO when run in the heuristic mode. In pseudo-code descriptions we may denote a call to espresso with a pseudo-instruction like *espresso(F,D,R)*, where F, D and R are respectively covers of the onset, don't care set and offset of the function to be minimized.

1.5 Manipulation of Boolean Networks

The basic techniques to restructure Boolean networks can be summarized as:

- Structural operations (they change the topological structure of the network), divided as:
  - Algebraic
  - Boolean
- Node simplification (they change the functions of nodes)
  - Computation of don't cares due 1) to the dependencies that the network topology imposes to the fanin signals (satisfiability or controllability don't cares), and 2) to the limited sensitivity of the outputs to vectors of inputs and internal nodes (observability don't cares)
  - Node minimization by exploiting the computed don't cares
- Phase assignment of the internal and output signals.

Given an initial network, the restructuring problem is to find the **best** network. In this chapter, for lack of space, we will discuss almost only structural operations.

Example 1.12 *Given the functions f_1 and f_2, we apply to them the operations* **minimize** *(node minimization performed by an algorithm like in* ESPRESSO *[2]),* **factor** *and* **decompose***, whose precise meaning we will discuss later.*

$$\begin{aligned} f_1 &= abcd + abce + a\overline{b}\overline{c}\overline{d} + a\overline{b}\overline{c}\overline{d} + a\overline{b}\overline{c}\overline{d} + \overline{a}c + cdf \\ &\quad + ab\overline{c}\overline{d}\overline{e} + a\overline{b}\overline{c}d\overline{f} \\ f_2 &= bdg + \overline{b}df\,g + \overline{b}\overline{d}g + b\overline{d}eg \end{aligned}$$

minimize:

$$\begin{aligned} f_1 &= bcd + bce + \overline{b}\overline{d} + \overline{b}f + \overline{a}c + ab\overline{c}\overline{d}\overline{e} + a\overline{b}\overline{c}d\overline{f} \\ f_2 &= bdg + df\,g + \overline{b}\overline{d}g + \overline{d}eg \end{aligned}$$

factor:

$$\begin{aligned} f_1 &= c(b(d+e) + \overline{b}(\overline{d}+f) + \overline{a}) + a\overline{c}(b\overline{d}\overline{e} + \overline{b}d\overline{f}) \\ f_2 &= g(d(b+f) + \overline{d}(\overline{b}+e)) \end{aligned}$$

decompose:

$$\begin{aligned} f_1 &= c(\overline{a}+x) + a\overline{cx} \\ f_2 &= gx \\ x &= d(b+f) + \overline{d}(\overline{b}+e) \end{aligned}$$

The basic structural operations which can be used to change the internal topology of a Boolean network are now briefly introduced on the basis of examples only. Operative definitions can be found in the manual of the multi-level synthesis package SIS [39]. Detailed definitions and descriptions will be given in the follow-up of the chapter, and especially in Sec. 5.

Example 1.13 *The following steps show the application of basic operations to restructure Boolea networks:*

1. *Decomposition of a single function*

$$\begin{array}{rcl} f & = & abc + abd + \overline{ac}\,\overline{d} + \overline{b}\,\overline{c}\,\overline{d} \\ & \Downarrow & \\ f & = & xy + \overline{xy} \\ x & = & ab \\ y & = & c + d \end{array}$$

2. *Extraction, i.e., decomposition of multiple functions*

$$\begin{array}{rcl} f & = & (az + b\overline{z})cd + e \\ g & = & (az + b\overline{z})\overline{e} \\ h & = & cde \\ & \Downarrow & \\ f & = & xy + e \\ g & = & x\overline{e} \\ h & = & ye \\ x & = & az + b\overline{z} \\ y & = & cd \end{array}$$

3. *Factoring, i.e., series-parallel decomposition*

$$\begin{array}{rcl} f & = & ac + ad + bc + bd + e \\ & \Downarrow & \\ f & = & (a + b)(c + d) + e \end{array}$$

4. *Substitution, i.e., making the value of a function available into another function*

$$\begin{array}{rcl} g & = & a + b \\ f & = & a + bc \\ & \Downarrow & \\ f & = & g(a + c) \end{array}$$

5. *Collapsing (also called elimination), i.e., replacing a function by its expression*

$$\begin{array}{rcl} f & = & ga + \overline{g}b \\ g & = & c + d \\ & \Downarrow & \\ f & = & ac + ad + b\overline{c}\,\overline{d} \\ g & = & c + d \end{array}$$

In general, "elimination" is a term that we will use for partial collapsing.

Comparing factoring vs. decomposition, the former is restricted to a series-parallel graph restructuring of the given function, whereas the latter may produce a non-tree structure, so it is similar to BDD collapsing of common nodes and using negative pointers. But decomposition is not canonical, so there is not a perfect identification of common nodes.

We introduce next the value of a node to define a cost function that measures the effect of an operation to restructure a Boolean network. The value of a node is the difference in literal cost of the network without the node (node eliminated or no factoring) and with the node (node factored out). Given node j, let n_i be the number of times that literal y_j and/or $\overline{y}_j$ appears in the factored network and let l_j be the number of literals in the factored form of node j (we can treat y_j and $\overline{y}_j$ as the same since $\rho(f_j) = \rho(\overline{f}_j)$); moreover, let c be the cost of the rest of the nework. The literal cost of the network without the node (elimination, no factoring) is

$$l_j \sum_{i \in FANOUT(j)} n_i + c,$$

and the literal cost of the network with the node (no elimination, factoring) is

$$l_j + \sum_{i \in FANOUT(j)} n_i + c.$$

Their difference (cost change due to elimination) is

$$value(j) = (l_j \sum_{i \in FANOUT(j)} n_i + c) - (l_j + \sum_{i \in FANOUT(j)} n_i + c) = ((\sum_{i \in FANOUT(j)} n_i) - 1)(l_j - 1) - 1.$$

Example 1.14 *Fig. 4(a) shows a network fragment before elimination of literal cost* $5+7+5+c = 17+c$*, and Fig. 4(b) shows the same network fragment after elimination of literal cost* $9+15+c = 24+c$*, for a difference of* $24+c-(17+c) = 7$ *that is exactly what the formula* $value(node3) = (1+2-1)(5-1)-1$ *computes, given* $n_1 = 1, n_2 = 2, l_3 = 5$.

The value might be different if we were to eliminate, simplify, and then refactor. Fig. 5(a) shows a network fragment with values annotated for nodes l, m, n*, Fig. 5(b) shows the value of node* n *after* eliminate -1 *of nodes* l *and* m*. For instance, in (a)* $value(l) = -1$*, because the network without node* l *has value less by 1 of the network with node* l*; in (b) node* n *has value* 3 *because the value of the network with node* n *is* $18+c$ *and with node* n *is* $15+c$*, whose difference is* $18+c-(15+c) = 3$*. In general the operation* eliminate x *eliminates all the nodes whose removal does not increment the value of the network by more than* x*. The special case* eliminate -1 *eliminates the nodes whose removal decreases the value of the network by at least 1, and in particular it removes nodes that are used only once (as it can be verified by the formula accounting for the value). Order of elimination may be important.*

Note that "**Division**" plays a key role in all of these operations. Their essence can be abstracted as addressing the following two problems:

1. Find good common subfunctions.
2. Effect the division.

In the next section we will survey techniques to solve them.

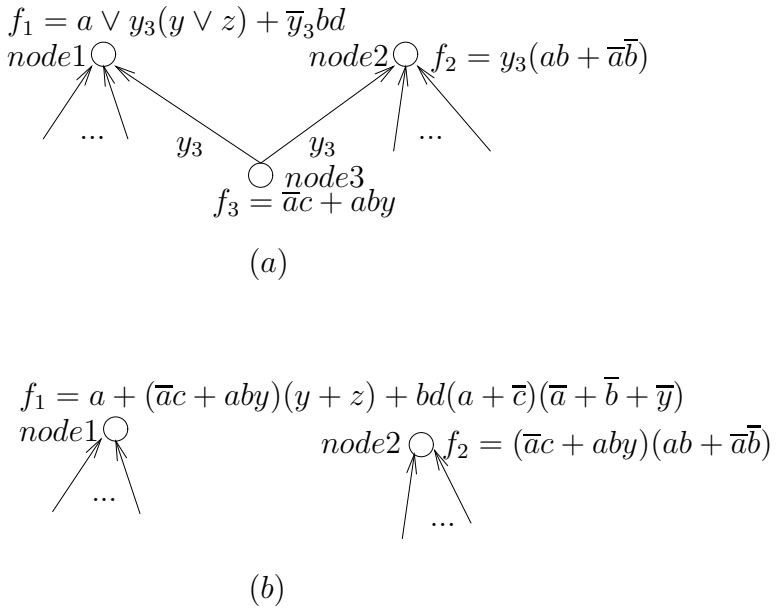

Figure 4: Illustration of Example 1.14. (a) Fragment of a Boolean network before elimination; (b) The same fragment after elimination; (c) A node j with l_j literals whose output fans out as wire y_j to nodes i_1, i_2, i_3.

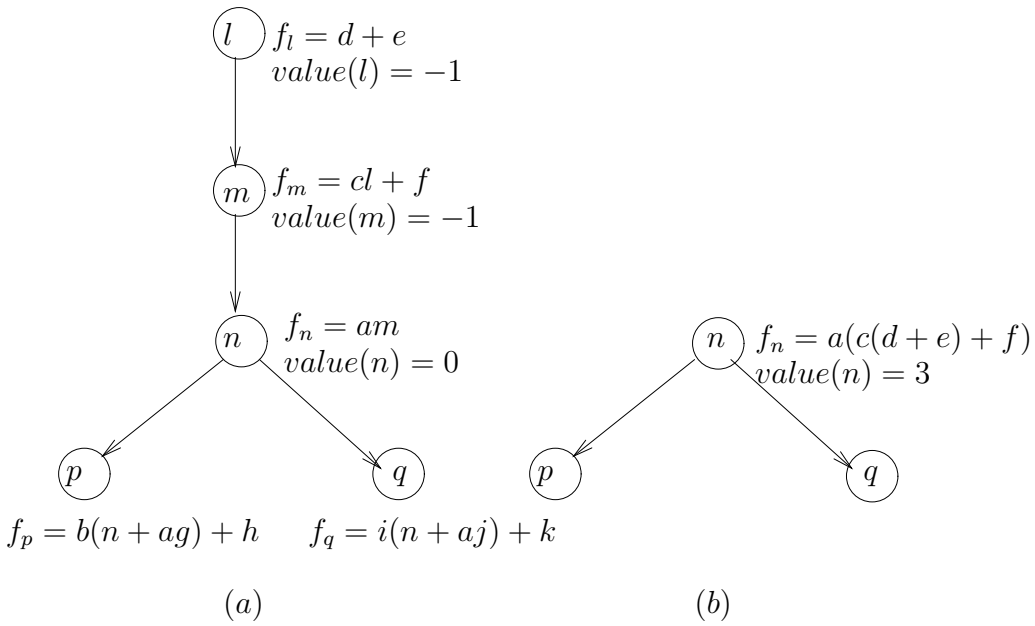

Figure 5: Illustration of Example 1.14. (a) Fragment of a Boolean network with some node values; (b) The same fragment after elimination of nodes with negative value.

2 Boolean Division

Division is central in the operations to restructure a Boolean network. To be able to perform it, we must investigate the following questions:

1. What is it (in the context of Boolean functions) ?
2. How to divide?
3. What to divide with?
4. How to apply it to restructure a network, e.g., in factoring, resubstitution, extraction ?

Definition 2.1 *A Boolean function g is a* **divisor** *of a Boolean function f if there exist Boolean functions h and r such that $f = gh + r$, with $gh \neq \emptyset$. In this case, h is called a* **quotient** *and r is called a* **remainder** *of the* **division** *of f by g. Note that h and r may not be unique.*

The function g is said to be a **factor** *of f if in addition $r = \emptyset$, i.e., if $f = gh$.*

We provide algorithms for division when the functions are represented by covers.

Definition 2.2 *Let F, G, H, R be covers respectively of the functions f, g, h, r in Defn. 2.1.*

If GH is restricted to be an algebraic product, G is an **algebraic divisor** *of F. Otherwise, G is a* **Boolean divisor** *of G.*

If GH is restricted to be an algebraic product and $R = \emptyset$, G is an **algebraic factor** *of F. Otherwise, G is a* **Boolean factor** *of F.*

If GH is restricted to be an algebraic product, H is the **algebraic quotient**, *denoted $F//G$. Otherwise, H is a (non-unique)* **Boolean quotient** *denoted $F \div G$.*

If H is an algebraic quotient, the operation $(F, G) \rightarrow (H, R)$ is called **algebraic division**, *denoted by $//$; otherwise it is called* **Boolean division**, *denoted by $\div$.*

We will reserve the notation F/G for the more useful "weak division" that is a type of algebraic division defined later.

Example 2.1 *Consider:*

$$\begin{aligned} F &= ad + ae + bcd + j \\ G_1 &= a + bc \\ G_2 &= a + b \end{aligned}$$

Examples of algebraic division are:

1. $F//(bc) = d$;
2. $F//a = d + e$; *also* $F//a = d$ *or* $F//a = e$, *i.e., algebraic division is not unique;*
3. $H_1 \equiv F//G_1 = d, R_1 = ae + j$.

Examples of Boolean division are:

1. $H_2 \equiv F \div G_2 = (a + c)d,\ R_2 = ae + j$, *i.e.,* $F = (a + b)(a + c)d + ae + j$;
2. $G_1 \div G_2 = a + c$, *i.e.,* $G_1 = (a + b)(a + c)$.

2.1 Boolean Division for Completely Specified Functions

Lemma 2.1 *A Boolean function g is a Boolean factor of a Boolean function f if and only if $f \subseteq g$ (i.e. $f\overline{g} = \emptyset$, i.e. $\overline{g} \subseteq \overline{f}$).*

Proof. $\Rightarrow$: g is a Boolean factor of f. Then $\exists h$ such that $f = gh$; Hence, $f \subseteq g$ (as well as h).

$\Leftarrow$: Assume $f \subseteq g$; since $f = fg + f\overline{g}$, then $f = fg = g(f + r)$ where r is any function $r \subseteq \overline{g}$. Thus $f = gh$, where $h = f + r$. □

Note that:

1. $h = f$ works fine for the proof.
2. Given f and g, h is not unique. To get a small h is the same as getting a small $f + r$. Since $rg = \emptyset$, this is the same as minimizing (simplifying) f with don't care (DC) $= \overline{g}$.

Example 2.2 *Let $f = (a+b)(a+c)+\overline{d}b$, it holds that $a+b \subseteq f$, then by Lemma 2.1 $g = a+b$ is a factor of f, i.e. $f = (a+b)h$. It is $f = fg + f\overline{g} = fg = ((a+b)(a+c)+\overline{d}b)(a+b)$. To get a small h we can minimize f and obtain $f = (a + bc + \overline{d}b)(a + b)$. The minimization of f is performed by representing $f = (a + b)(a + c) + \overline{d}b = a + ab + ac + bc + \overline{d}b$ by the minimal SOP representation $f = a + bc + \overline{d}b$.*

Actually, according to the proof, h can be any function of the form $f + r$, where r is orthogonal to g. Hence we can get a smaller h by minimizing f with $DC = \overline{g} = \overline{a}\overline{b}$, that gives $h = a+c+\overline{d}$ and so $f = (a + c + \overline{d})(a + b)$, where $g = a + b$ is the divisor and $h = f + r = a + c + \overline{d}$ is the quotient. The minimization of f with $DC = \overline{g} = \overline{a}\overline{b}$ is performed by applying two-level logic minimization to the incompletely specified function $\mathcal{F}$ whose onset is $a + bc + \overline{d}b$ and whose don't care set is $\overline{a}\overline{b}$, yielding the optimal SOP representation $h = a + c + \overline{d}$, where the minimization procedure used the don't care set $\overline{a}\overline{b}$ by adding to the onset the terms in the don't care set given by $r = \overline{a}\overline{b}\ \overline{d} + \overline{a}\overline{b}c \subseteq \overline{a}\overline{b}$.

Lemma 2.2 *g is a Boolean divisor of f if and only if $fg \neq \emptyset$.*

Proof.
$\Rightarrow$: $f = gh + r, gh \neq \emptyset \Rightarrow fg = gh + gr$. Since $gh \neq \emptyset, fg \neq \emptyset$.

$\Leftarrow$: $f = fg + f\overline{g} = g(f + k) + f\overline{g}$. Here $k \subseteq \overline{g}$. Then $f = gh + r$, with $h = f + k$, $r = f\overline{g}$. Since $fg = hg + rg = hg$ and $fg \neq \emptyset$ then $gh \neq \emptyset$. This means that g is a Boolean divisor of f, in view of Definition 2.1. □

Note that the requirement $fg \neq \emptyset$ is the less restrictive one. Hence f has **many** Boolean divisors. We are looking for a g such that $f = gh+r$, where g, h, r are **simple** functions. From $f = fg+f\overline{g} = gh + r$, given g and h, we could minimize $f\overline{g}$ with $DC = (fg) = gh$ to get a small r, and instead given g and r we could minimize $hg = fg$ with $DC = (f\overline{g}) = r$ to get a small h (or also with $DC = r + \overline{g}$ since $(r + \overline{g})g = rg$). The problem is how to minimize simultaneously all the unknown terms to obtain a small f, i.e., a representation of f with fewer literals.

2.2 Boolean Division for Incompletely Specified Functions

The following results extend Boolean division to incompletely specified functions $\mathcal{F} = (f, d, r)$.

Definition 2.3 *A completely specified Boolean function g is a* **Boolean divisor** *of $\mathcal{F} = (f, d, r)$ if there exist completely specified Boolean functions h, e such that*

$$f \subseteq gh + e \subseteq f + d, \ i.e., \ f = gh + e \ mod \ d,$$

and

$$gh \not\subseteq d, \ i.e., \ gh \neq 0 \ mod \ d.$$

g is a **Boolean factor** *of $\mathcal{F} = (f, d, r)$ if there exists h such that*

$$f \subseteq gh \subseteq f + d, \ i.e., \ f = gh \ mod \ d.$$

Lemma 2.3 *$f \subseteq g$ if and only if g is a Boolean factor of $\mathcal{F} = (f, d, r)$.*

Proof.
$\Rightarrow$: Let $h = f + k$ where $kg \subseteq d$. Then $hg = (f + k)g = fg + kg \subseteq f + d$.
Since $f \subseteq g, f = fg$ and thus $f \subseteq fg + kg = (f + k)g = gh$. Thus

$$f \subseteq (f + k)g \subseteq f + d.$$

$\Leftarrow$: Suppose $\exists$ minterm m such that $f(m) = 1$ but $g(m) = 0$. Then $f(m) = 1$ but $g(m)h(m) = 0$ implying that $f \not\subseteq gh$. $\square$

Note that we would like to find the simplest h; since h has the form $f+k$, where $kg \subseteq d$ or $k \subseteq d+\overline{g}$ ($kg \subseteq d$, $kg + \overline{g} \subseteq d + \overline{g}$, $kg + k\overline{g} \subseteq d + \overline{g}$, $k \subseteq d + \overline{g}$), we can simplify $hf + k$ by minimizing f with $DC = d + \overline{g}$. So the steps would be:

1. Choose $g \supseteq f$ (with G cover of g).
2. Simplify $(f, d + \overline{g}, rg)$ to obtain H.
3. GH is a cover of $\mathcal{F}$.

Lemma 2.4 *$fg \neq \emptyset$ if and only if g is a Boolean divisor of $\mathcal{F} = (f, d, r)$.*

Proof.
$\Rightarrow$: Assume $fg \neq \emptyset$.
Define $\mathcal{F}_h = (fg, d+f\overline{g}, r)$. Now $g \supseteq fg$, so by Lemma 2.3 $\exists h$ such that $fg \subseteq gh \subseteq fg+d+f\overline{g} = f + d$.
Let e be a cover of $\mathcal{F}_e = (f\overline{g}, d + fg, r)$, i.e., $f\overline{g} \subseteq e \subseteq f\overline{g} + d + fg = f + d$.
Thus $f = fg + f\overline{g} \subseteq gh + e \subseteq f + d$.
From $fg \subseteq gh$ it follows $fg \subseteq fgh$ and $\emptyset \neq fg \subseteq ghf \rightarrow ghf \neq \emptyset$. Since $fd = \emptyset$ by definition and $ghf \neq \emptyset$ then $gh \not\subseteq d$, verifying the conditions for Boolean division.

$\Leftarrow$: Suppose $\exists h$ such that $gh \not\subseteq d$ and $f \subseteq gh + e \subseteq f + d$.
From $gh + e \subseteq f + d$ it follows $gh \subseteq f + d$ and, since $gh \not\subseteq d \rightarrow gh \neq \emptyset$, then $ghf \neq \emptyset$. Hence $fg \neq \emptyset$. $\square$

In summary, the steps to find H and E would be:

1. Choose g such that $fg \neq \emptyset$ (with G cover of g).

2. Simplify $(fg, d + f\overline{g}, r)$ to obtain H.

3. Simplify $(f\overline{g}, d + fg, r)$ to obtain E.

4. $GH + E$ is a cover of $\mathcal{F}$.

A correct variant of step 2. is:
2. Simplify $(fg, d+\overline{g}, r)$ to obtain H, because $fg \subseteq h \subseteq fg+d+\overline{g}$ implies $fg \subseteq gh \subseteq fg+dg \subseteq f+d$.
A correct variant of step 3. is:
3. Simplify $(f\overline{g} + f\overline{h}, d + fgh, r)$ to obtain H, because $f\overline{g} + f\overline{h} \subseteq e \subseteq f\overline{g} + f\overline{h} + d + fgh = f + d$.

Since there are many divisors g (only need $fg \neq \emptyset$), the following questions are the **unsolved problem of common Boolean divisors**:

1. Given (f, d, r), find a good g $(fg \neq \emptyset)$ such that both $(fg, d + f\overline{g}, r)$ and $(f\overline{g}, d + fg + r)$ simplify "nicely".

2. Given two functions (or more) (f_1, d_1, r_1) and (f_2, d_2, r_2) find a common g "good and simple" such that step 1. applies for both.

Lemma 2.5 *Suppose g is an algebraic divisor of F, a cover of $\mathcal{F} = (f, d, r)$. If $f \not\subseteq e$,* where e is the remainder in the algebraic division, i.e., *$F = gh + e$, then g is a Boolean divisor of $\mathcal{F}$.*

Proof. Assume $F = gh + e, gh \neq 0, f \not\subseteq e$. Since $f \subseteq gh + e$ and $f \not\subseteq e$, then $fgh \neq \emptyset$ implying that $fg \neq \emptyset$. Therefore, by Lemma 2.4, g is a Boolean divisor of $\mathcal{F}$. □

Lemma 2.6 *If g is an algebraic factor of F, a cover of $\mathcal{F} = (f, d, r)$, then g is a Boolean factor of $\mathcal{F}$.*

Proof. Assume $F = gh$. Since $f \subseteq F$, then

$$f \subseteq gh \Rightarrow f \subseteq g.$$

By Lemma 2.3, g is a Boolean factor of $\mathcal{F}$. □

Lemma 2.7 *Suppose g is an algebraic divisor of F, a prime cover of $\mathcal{F} = (f, d, r)$. Then g' is a Boolean divisor of $\tilde{\mathcal{F}} = (r, d, f)$.*

Proof. We need to show that $\overline{g}r \neq \emptyset$ by Lemma 2.4. Now $F = gh + e$, so $\overline{F} = \overline{e}\ \overline{g} + \overline{e}\overline{h}$. But $d + r \supseteq \overline{F} \supseteq r$, thus $\overline{F}r = r = \overline{e}\ \overline{g}r + \overline{e}\overline{h}r$.

Suppose by contradiction that $\overline{g}r = \emptyset$, then $r = \overline{e}\ \overline{h}r$, i.e., $\overline{e}\ \overline{h} \supset r$, and so $e + h \subseteq \overline{r} = f + d$.

But, from $F = gh + e \subseteq h + e \subseteq f + d$, then $e + h \subseteq f + d$ implies that the cubes of gh were not prime, reaching a contradiction. □

2.3 Performing Boolean Division

Given $\mathcal{F} = (f, d, r)$, and a divisor g, the problem is to find a cover for $\mathcal{F}$ in the form $GH + E$ where H, E are minimal in some sense, e.g., minimum factored form. A variant is to find a cover in the form $gH_1 + \overline{g}H_0 + E$, where again H_1, H_0, E are minimal.

There is a method for performing the Boolean division operation (i.e., finding H and E) based on ESPRESSO, even though a minimum SOP may not be the best objective. Informally the steps of the method are:

1. Create a **new** variable x to "represent" g.
2. Form the don't care set $\widetilde{d} = x\overline{g} + \overline{x}g$. (Since $x = g$ we don't care if $x \neq g$).
3. Minimize $(f\ \overline{(\widetilde{d})}, d + \widetilde{d}, r\ \overline{(\widetilde{d})})$ to get $\widetilde{f}$. Note that $(f\ \overline{(\widetilde{d})}, d + \widetilde{d}, r\ \overline{(\widetilde{d})})$ is a partition.
4. Return (H, E) where H are the terms of $\widetilde{f}$ containing x but with x removed, and E is the remainder of $\widetilde{f}$ (i.e., the terms not containing x).

We can say that the method returns $(H = \widetilde{f}/x, E)$, where f/x denotes "weak division", a maximal form of algebraic division introduced formally in Defn. 3.1.

Example 2.3 *Consider f with cover $F = a + bc$ and g with cover $G = a + b$. The Boolean division $F \div G$ can be performed as follows:*

- $\widetilde{d} = x\overline{a}\overline{b} + \overline{x}(a + b)$ *where* $x = g = (a + b)$
- *Minimize* $(a + bc)\overline{(\widetilde{d})} = (a + bc)(\overline{x}\overline{a}\overline{b} + x(a + b)) = xa + xbc$ *with* $DC = x\overline{a}\overline{b} + \overline{x}(a + b)$.
- *A minimum cover is $a + bc$ but it does not use x or $\overline{x}$!!*
- *Force x in the cover. It yields $F = xa + xc$ and, making it a prime cover, finally we get $F = a + xc = a + (a + b)c$.*

Heuristic: Try to find an answer **with x in it** *and which uses the* **least variables** *(or* **literals***).*

We provide two algorithms based on the previous method. Assume that F is a cover for $\mathcal{F} = (f, d, r)$, D is a cover for d and g is a divisor. The first algorithm, shown in Fig. 6, finds H, E such that $xH + E$ is a cover for (f, d, r), where x is a literal denoting the divisor g.

The second algorithm, shown in Fig. 7, finds H_1, H_0, E such that $xH_1 + \overline{x}H_0 + E$ is a cover for (f, d, r), where x is a literal denoting the divisor g and $\overline{x}$ is a literal denoting the divisor $\overline{g}$. The second algorithm is a slight variation of the first one: it uses $\overline{x}$ also while dividing, by skipping the step $F_2 =$ remove $\overline{x}$ from F_1.

The given algorithms use an operation MinLiteral_Support (or MinVariable_Support) that finds a prime cover with the smallest literal (or variable) support, i.e., a prime cover with the smallest number of literals (variables) that appear in at least a prime.

We remark that in any cover of primes of a completely specified function the minimum support is always the same and is the set of variables on which the function effectively depends, i.e., the essential variables of the function. However this is not true when there is a don't care set d, and

$$
\begin{aligned}
(H, E) &\leftarrow \text{Boolean\_Division}(F, D, g) \\
D_1 &= D + x\overline{g} + \overline{x}g \quad \text{(don't care)} \\
F_1 &= F\overline{D}_1 \quad \text{(on-set)} \\
R_1 &= \overline{(F_1 + D_1)} = \overline{F}_1\overline{D}_1 = \overline{F}\ \overline{D}_1 \quad \text{(off-set)} \\
F_2 &= \text{remove } \overline{x} \text{ from } F_1. \\
F_3 &= \text{MinLiteral\_Support}(F_2, R_1, x) \\
&\quad \text{(minimum literal support including } x) \\
F_4 &= \textsc{espresso}(F_3, D_1, R_1) \\
H &= F_4/x \quad \text{(quotient)} \\
E &= F_4 - \{xH\} \quad \text{(remainder)}
\end{aligned}
$$

Figure 6: An algorithm for Boolean division. Given covers F of (f, d, r) and D of d and a divisor g, it finds H, E such that $xH + E$ is a cover of (f, d, r), where x is a literal denoting the divisor g.

$$
\begin{aligned}
(H_1, H_0, e) &\leftarrow \text{Boolean\_Division}(F, D, g) \\
D_1 &= D + x\overline{g} + \overline{x}g \quad \text{(don't care)} \\
F_1 &= F\overline{D}_1 \quad \text{(on-set)} \\
R_1 &= \overline{(F_1 + D_1)} = \overline{F}_1\overline{D}_1 = \overline{F}\ \overline{D}_1 \quad \text{(off-set)} \\
F_3 &= \text{MinLiteral\_Support}(F_1, R_1, x, \overline{x}) \\
&\quad \text{(minimum literal support including } x, \overline{x}) \\
F_4 &= \textsc{espresso}(F_3, D_1, r_1) \\
H_1 &= F_4/x \\
H_0 &= F_4/\overline{x} \\
E &= F_4 - \{xH_1\} - \{\overline{x}H_0\}
\end{aligned}
$$

Figure 7: An algorithm for Boolean division. Given covers F of (f, d, r) and D of d and a divisor g, it finds H, E such that $xH_1 + \overline{x}H_0 + E$ is a cover of (f, d, r), where x is a literal denoting the divisor g and $\overline{x}$ is a literal denoting the divisor $\overline{g}$.

so the result does not apply to our context, where $d \neq \emptyset$ by construction. Given a cover F of $\mathcal{F} = (f, d, r)$, and the notations:

$$\begin{aligned} v\_sup(F) &= \{v | v \in c \text{ or } \overline{v} \in c \text{ for some } c \in F\} \\ \ell\_sup(F) &= \{\ell | \ell \in c \text{ for some } c \in F\}, \end{aligned}$$

the following lemma states that the minimum support sets *v_sup* and *ℓ_sup* are unique for all prime covers.

Lemma 2.8 *For a completely specified Boolean function, i.e., for $\mathcal{F} = (f, d, r)$ with $d = \emptyset$, then $v\_sup(F_1) = v\_sup(F_2)$ and $\ell\_sup(F_1) = \ell\_sup(F_2)$ for all prime covers F_1, F_2 of $\mathcal{F}$.*

Proof. Let F_1 and F_2 be two prime covers of $\mathcal{F}$, with F_1 independent of x, but let x be in F_2. Consider a prime cube $xc \in F_2$, where c is a cube covering the minterms $\{m_1, m_2, \ldots, m_k\}$ (defined over all variables except x). Since xm_i is an implicant of $\mathcal{F}$, it is covered by F_1, because $d = \emptyset$ and so all minterms in every implicant must be covered by any cover. However, F_1 is independent of x, and so xm_i covered by F_1 implies $\overline{x}m_i$ covered by F_1. Hence $\overline{x}m_i$ is also an implicant of $\mathcal{F}$. Hence m_i is an implicant of $\mathcal{F}$, and this holds for all $i = 1, \ldots, k$. Therefore c (that is the collection of all $m_i, i = 1, \ldots, k$) is an implicant of $\mathcal{F}$, and so xc can be raised to c, contradicting the fact that xc is a prime. □

The procedures to minimize the variable or literal support are based on the notion of blocking matrices, used in two-level logic minimization to expand cubes to primes [2, 37]. We remind [2] that the operation of expansion of an implicant is performed by removing one or more literals from its expression as a cube, which corresponds to increasing its size by a factor of 2 per deleted literal, and therefore to covering more minterms. It is legitimate to expand an implicant with respect to a literal, if the expanded cube does not cover minterms of the offset. Expanding a cover means expanding the single terms of the collection.

Given $\mathcal{F} = (f, d, r)$, let $F = \{c^1, c^2, \ldots, c^k\}$ be a cover of $\mathcal{F}$ and $R = \{r^1, r^2, \ldots, r^m\}$ be a cover of r. A blocking matrix B^i of a cube c^i of F keeps track of all variables that make cube c^i disjoint from each cube of R.

Definition 2.4 *The* **blocking matrix** *B^i of cube c^i of F is defined as*

$$(B^i)_{qj} = \begin{cases} 1 & \text{if } (c^i)_j \cap (r^q)_j = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

A row cover of a blocking matrix B^i of a cube c^i of F is a set of variables that make cube c^i disjoint from each cube of R.

Definition 2.5 *A* **row cover** *of B (a 0-1 matrix) is a set of column indices $S = \{j_1, \ldots, j_v\}$ such that $\forall q, \exists j \in S$ such that $B_{qj} = 1$.*

Theorem 2.1 *Let S be a row cover of B^i and suppose $|S|$ is minimum. Construct cube*

$$(\widetilde{c}^i)_j = \begin{cases} (c^i)_j & \text{if } j \in S \\ \{0, 1\} = 2 & \text{otherwise} \end{cases} \tag{2}$$

Then $\widetilde{c}^i \supseteq c^i$ is the largest prime implicant of $\mathcal{F}$ containing c^i.

1. Construct blocking matrix B^i for each c^i
2. Form super-blocking matrix B
3. Find a minimum cover S of B, where $S = \{j_1, j_2, \ldots, j_v\}$
4. Build $\widetilde{F} \leftarrow \{\widetilde{c}^1, \widetilde{c}^2, \ldots, \widetilde{c}^k\}$, where $(\widetilde{c}^i)_j = \begin{cases} (c^i)_j & \text{if } j \in S \\ \{0,1\} = 2 & \text{otherwise} \end{cases}$

Figure 8: Steps of the algorithm MinVariable_Support to compute the minimum variable support.

Definition 2.6 *The* **super-blocking matrix** B *of cover* $F = \{c^1, c^2, \ldots, c^k\}$ *is defined as*

$$B = \begin{bmatrix} B^1 \\ B^2 \\ \vdots \\ B^k \end{bmatrix}$$

where B^i *is the blocking matrix of cube* c^i *of* F*.*

Theorem 2.2 *Let* S *be a minimum row cover of the super-blocking matrix* B*. The set* $\{x_j | j \in S\}$ *is a minimum set of variables which appear in any cover of* $\mathcal{F}$ *obtained by expanding* F*.*

Proof. Expand treats each $c^i \in F$ and builds B^i. Let $\hat{c}^i$ be any prime containing c^i. Then the variables in $\hat{c}^i$ "cover" B^i. Thus the union of the set of variables in $\hat{c}^i$ taken over all i covers B. Hence this set cannot be smaller than a minimum cover of B. □

Note that in general, there could exist another cover of $\mathcal{F}$ which has fewer variables, but one not obtained by expanding F.

In summary, given $\mathcal{F} = (f, d, r)$, $F = \{c^1, c^2, \ldots, c^k\}$ cover of $\mathcal{F}$, and $R = \{r^1, r^2, \ldots, r^m\}$ cover of r, the algorithm to find an expanded cover with the fewest variables is outlined in Fig. 8.

Given $\mathcal{F} = (f, d, r)$, $F = \{c^1, c^2, \ldots, c^k\}$ cover of $\mathcal{F}$, $R = \{r^1, r^2, \ldots, r^m\}$ cover of r, the minimum literal support is computed in a similar way, defining the literal blocking matrix as an extension of the standard (variable) blocking matrix.

Definition 2.7 *The* **literal blocking matrix** $\hat{B}^i$ *of cube* c^i *of* F *over* B^n *is defined as*

$$(\hat{B}^i)_{qj} = \begin{cases} 1 & \textit{if } v_j \in c^i \textit{ and } \overline{v}_j \in r^q \\ 0 & \textit{otherwise} \end{cases}$$

$$(\hat{B}^i)_{q,j+n} = \begin{cases} 1 & \textit{if } \overline{v}_j \in c^i \textit{ and } v_j \in r^q \\ 0 & \textit{otherwise} \end{cases}$$

Example 2.4 *Given* $c^i = a\overline{d}\overline{e}$, $r^q = \overline{a}ce$*, the literal blocking matrix* $\hat{B}^i_q$ *of* c^i *is*

$$\hat{B}^i_q = \begin{array}{cccccccccc} a & b & c & d & e & \overline{a} & \overline{b} & \overline{c} & \overline{d} & \overline{e} \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Then construct the literal super-blocking matrix $\hat{B}$ and get its row cover J.

Theorem 2.3 *Let J be a minimum row cover of the super-blocking matrix $\hat{B}$. The set $\{\ell_i | i \in J\} \cup \{\overline{\ell}_i | i + n \in J\}$ is a minimum set of literals which appear in any cover of $\mathcal{F}$ obtained by expanding F.*

Proof. Same reasoning as for minimum variable support. □

For a (non-trivial) cube, minimum literal support is the same as minimum variable support.

Example 2.5 *Given the on-set cube $c^i = a\overline{b}d$ and off-set $r = \overline{a}\,\overline{b}\overline{d} + ab\overline{d} + ac\overline{d} + bcd + \overline{c}\,\overline{d}$, the literal super-blocking matrix $\hat{B}^i$ is*

	a	b	c	d	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$
$\overline{a}\,\overline{b}\overline{d}$	1	0	0	1	0	0	0	0
$ab\overline{d}$	0	0	0	1	0	1	0	0
$ac\overline{d}$	0	0	0	1	0	0	0	0
bcd	0	0	0	0	0	1	0	0
$\overline{c}\,\overline{d}$	0	0	0	1	0	0	0	0

The minimum column cover is $\{d, \overline{b}\}$ and thus $\overline{b}d$ is the maximum prime covering the cube $a\overline{b}d$.

To see the different operations in action, the following example shows how to perform Boolean division by applying the steps of the algorithm in Fig. 6.

Example 2.6 *Given $F = a + bc$, by algebraic division we get $F/(a + b) = \emptyset$. By Lemma 2.1 and Lemma 2.2, $G = a + b$ is a Boolean factor and a Boolean divisor of the function represented by F. Moreover, it is easy to verify the Boolean division $F \div (a + b) = a + c$. Instead let us perform the Boolean division according to the algorithm in Fig. 6.*

- *Set $x = a + b$.*
- *Generate the don't care set $D_1 = \overline{x}(a + b) + \overline{a}\,\overline{b}x$.*
- *Generate the care on-set $F_1 = F\overline{D}_1 = (a + bc)(ax + bx + \overline{a}\,\overline{b}\overline{x}) = ax + bcx$. Let $\mathcal{C} = \{c^1 = ax, c^2 = bcx\}$.*
- *Generate the care off-set $R_1 = \overline{F}\ \overline{D}_1 = (\overline{a}\,\overline{b} + \overline{ac})(ax + bx + \overline{a}\,\overline{b}\overline{x}) = \overline{a}\,\overline{b}\overline{c}x + \overline{a}\,\overline{b}\overline{x}$. Let $R = \{r^1 = \overline{a}\,\overline{b}\overline{c}x, r^2 = \overline{a}\,\overline{b}\overline{x}\}$.*
- *Form the variable super-blocking matrix using column order (a, b, c, x). Notice that c^1 and c^2 are positive unate cubes.*

$$B = \begin{bmatrix} B^1 \\ B^2 \end{bmatrix} = \left[\begin{array}{cccc|c} a & b & c & x & \\ 1 & 0 & 0 & 0 & ax \\ 1 & 0 & 0 & 1 & \\ - & - & - & - & \\ 0 & 0 & 1 & 0 & bcx \\ 0 & 1 & 0 & 1 & \end{array}\right] \tag{3}$$

$$
\begin{array}{rcl}
(Q, R) & = & \text{Boolean\_Division}(F, G) \\
 & \Updownarrow & \\
(Q_1, R_1) & = & \text{Algebraic\_Division}(F, G) \\
D_1 & = & (x \oplus G) + Q_1 x \\
R_2 & = & \text{ESPRESSO}(R_1, D_1) \\
(Q_3, R_3) & = & \text{Algebraic\_Division}(R_2, x) \\
(Q, R) & = & (Q_3 + Q_1, R_3)
\end{array}
$$

Figure 9: An heuristic scheme for Boolean division that computes $(Q, R) = \text{Boolean\_Division}(F, G)$.

- *Find the minimum row cover* $S = \{a, c, x\}$.
- *Eliminate in* F_1 *all variables associated with* b. *So from* $F_1 = ax + bcx$ *we get* $F_3 = ax + cx = x(a + c)$ *(if we would notice* $F_3 = x(a + c)$, *then we could conclude* $F \div (a + b) = a + c$ *and we would be done).*
- *Simplifying* F_3 *(applying expand, irredundant cover), we get* $F_4 = a + cx$.
- *Thus the quotient is* $H = F_1/x = c$, *the remainder is* $E = a$ *and so we get the cover* $xH + E = xc + a$.
- *In summary, from* $F = a + bc$ *we got* $a + cx = a + c(a + b)$.

A question is: how to force x in the final cover ? ESPRESSO by default does not guarantee that it will keep it, one must restrict/modify its operation to preserve x in the final cover, for instance one could put MINVAR in the inner loop of ESPRESSO.

Another heuristic for Boolean division mentioned in [4, 5] interplays algebraic division and simplification as shown in the scheme of Fig. 9. For simplicity here we omitted the minimum literal support step. How to perform algebraic division is the topic of the next section.

Example 2.7 *Performing Boolean division by two steps of algebraic division (with some help from* ESPRESSO *too !).*

$$
\begin{array}{rcl}
F & = & ab\overline{c}\overline{d} + ab\overline{e}\overline{f} + cd\overline{a}\overline{b} + cd\overline{e}\overline{f} + ef\overline{a}\overline{b} + ef\overline{c}\overline{d} \\
G & = & ab + cd + ef \\
(Q_1, R_1) & = & \textit{Algebraic\_Division}(F, G) \\
(Q_1, R_1) & = & (\emptyset, F) \\
D & = & (ab + cd + ef) \oplus x \\
R_2 & = & \text{ESPRESSO}(F, D) \\
R_2 & = & \overline{a}\overline{b}x + \overline{c}\overline{d}x + \overline{e}\overline{f}x \\
(Q_3, R_3) & = & \textit{Algebraic\_Division}(R_2, x) \\
(Q_3, R_3) & = & (\overline{a}\overline{b} + \overline{c}\overline{d} + \overline{e}\overline{f}, \emptyset) \\
F & = & (ab + cd + ef)(\overline{a}\overline{b} + \overline{c}\overline{d} + \overline{e}\overline{f})
\end{array}
$$

3 Algebraic Division

The key motivations to introduce algebraic methods are:

1. They treat logic functions like a polynomial (often the irredundant prime representation is canonical, e.g., unate).
2. Fast methods for manipulation of polynomials are available (complexity from linear to quadratic).
3. There is a loss of optimality, but results are still quite good.
4. They can iterate and interleave with Boolean operations.

When introducing algebraic division, we noticed already that it does not guarantee the uniqueness of quotient and reminder.

Example 3.1 *Given* $F = (a+b+c)(d+e) + \overline{a}c$ *and* $G = d+e$*, we can get either*

$$\begin{aligned} F &= (d+e)(a+b) + c(\overline{a}+d+e) = GH_1 + R_1, \text{ or} \\ F &= (d+e)(a+b+c) + c\overline{a} = GH_2 + R_2 \end{aligned}$$

so H *and* R *are not unique.*

To achieve uniqueness we introduce weak division, a restricted definition of algebraic division, where the quotient is the largest (maximum number of terms) expression H such that $F = GH + R$.

Definition 3.1 *Given two algebraic expressions* F *and* G*, a division is called* **weak division** *if*

1. *it is an algebraic division, i.e., it generates expressions* H *and* R *such that* HG *is an algebraic product;*
2. R *has as few cubes as possible, or equivalently,* H *has as many cubes as possible;*
3. $HG + R$ *and* F *are the same expression (having the same set of cubes when multiplied out).*

The quotient H *resulting from weak division is denoted by* F/G*.*

Theorem 3.1 *Given the algebraic expressions* F *and* G*, the quotient* H *and the remainder* R *generated by weak division are unique.*

Example 3.2 *Given* $F = ac+ad+ae+bc+bd+be+\overline{a}b$*, we have* $F/a = c+d+e$*,* $F/b = c+d+e+\overline{a}$ *and* $F/(a+b) = c+d+e$*. Notice that* $F/(a+b) = F/a \cap F/b$*. Notice that* $F/(a+b) = (F/a)(F/b)$*.*

To compute the quotient of an algebraic division F/G, where $F = \{c_k\}$ and $G = \{a_i\}$, we define first the quotient when the divisor is a cube as

$$h_i = F/a_i = \{b_j | a_i b_j = c_k \in F\}.$$

Example 3.3 *Let* $F = abc + a\overline{b}\overline{c} + abd\overline{e} + abg + b\overline{c}$*, then* $F/(ab) = c + d\overline{e} + g$*.*

The following theorem argues that the quotient can be obtained by intersecting the quotients with respect to the single cubes of the divisor.

WEAK_DIVISION(F, G): {
$U = \{u_j\}$ - all cubes of F but where only literals in G have been kept
$V = \{v_j\}$ - all cubes of F but literals in G removed
/* note that $u_j v_j$ is the j-th cube of F */
$V^i = \{ v_j \in V : u_j = G_i \}$ /* one set for each cube of G */
$H = \cap V^i$ /* those cubes found in all V^i */
$R = F - GH$
return (H, R)
}

Figure 10: An algorithm to perform weak division.

Theorem 3.2 *Given $F = \{c_k\}$ and $G = \{a_i\}$, then*

$$F/G = \{d_j | d_j \in F/a_i \ \forall a_i \in G\} = \bigcap_i (F/a_i).$$

Proof. If $d_j \in F/a_i, \forall a_i \in G$, then

$$(F/G)G + R = (d_1 + \ldots + d_s)(a_1 + \ldots + a_{|G|}) + R$$

where R are the remaining terms $\{c_h\} = F \setminus \{d_j a_i | j = 1, \ldots, s, \ i = 1, \ldots, |G|\}$.

We show by contradiction that $\{d_j\}$ is the largest quotient; suppose not, then $\exists d \notin \{d_j\}$, and $d(a_1 + \ldots + a_{|G|}) \in \{c_k\}$. Then $d \in F/a_i, \forall a_i \in G$, and therefore $d \in F/G = \{d_j | d_j \in F/a_i \ \forall a_i \in G\}$, which is a contradiction. □

Th. 3.2 suggests an algorithm to perform weak division that is outlined in Fig. 10.

Example 3.4 *Given F and G*

$$\begin{aligned} F &= ace + ade + bc + bd + be + \overline{a}b + ab \\ G &= ae + b \end{aligned}$$

the algorithm for weak division in Fig. 10 computes the following expressions

$$\begin{aligned} U &= ae + ae + b + b + b + b + ab \\ V &= c + d + c + d + 1 + \overline{a} + 1 \\ V^{ae} &= c + d \\ V^{b} &= c + d + 1 + \overline{a} \\ H &= c + d = F/G \\ R &= be + \overline{a}b + ab \end{aligned}$$

Finally F can be factored as follows

$$F = (ae + b)(c + d) + be + \overline{a}b + ab$$

The time complexity of WEAK_DIVISION is $O(|F||G|)$.

McGeer and Brayton investigated in [31] efficient implementations of the algebraic division algorithm and they were able to show that its complexity can be reduced to $O((|F|+|G|)\log(|F|+|G|))$. Moreover if F, G are already given as sorted cubes, i.e., in the order of their binary encoding (e.g., $a\overline{b}de$ can be encoded by the binary number 0110110101), they reported a **linear** time algorithm such that F/G and R are produced in sorted order. In fact, the operations of algebraic division, multiplication, addition, subtraction, and equality test were proved to be all linear and stable.

Definition 3.2 *A* **stable** *algorithm produces its output in sorted order if it receives its input in sorted order.*

If all algorithms are stable, then we can start with a Boolean network, do an initial sort on each node, and then use only stable operations.

The steps of the $\mathcal{O}(n\ log\ n)$ algorithm are outlined here:

1. Encode the cubes $a_i \in G$ as binary numbers $n_1, n_2, \ldots, n_{|G|}$.
2. Encode the cubes $c_i \in F$ restricted to the support of G, $c_{i|sup(G)} \equiv b_i$, as binary numbers $m_1, m_2, \ldots, m_{|F|}$.
3. Sort the cubes in the set $\{n_1, \ldots, n_{|G|}, m_1, \ldots, m_{|F|}\}$, in time $O((|F|+|G|)\log(|F|+|G|))$.
4. Define the set $I = \{i \mid \exists j\ m_i = n_j\}$, denoting the cubes of F divided by a cube of G.
5. Encode the set $\{d_i \equiv c_{i|sup(F)\setminus sup(G)} \mid i \in I\}$, as binary numbers $q_1, q_2, \ldots, q_{|I|}$.
6. Sort the cubes in the set $\{q_1, \ldots, q_{|I|}\}$, in time $O(|F|\log|F|)$.
7. Define the set $J = \{j \mid q_j$ appears $|G|$ times$\}$.
8. $F/G = \{c_{j|sup(F)\setminus sup(G)} \mid j \in J\}$.

Example 3.5 *Consider* $F = ac\overline{d} + a\overline{c}d + ae + bc\overline{d} + b\overline{c}d + be + ag + \overline{b}e$ *and* $G = a + b$. *The encoded cubes of* F *are (to ease readability, we report the binary encoding used in the algorithm with the common decoding convention* $01 \to 1$, $10 \to 0$, $11 \to 2$*):*

	ab	*cdeg*
(1)	12	1022
(2)	12	0122
(3)	12	2212
(4)	21	1022
(5)	21	0122
(6)	21	2212
(7)	12	2221
(8)	20	2212

Notice that $sup(G) = \{a, b\}$, $sup(F) \setminus sup(G) = \{c, d, e, g\}$. *Then it is* $I = \{1, 2, 3, 4, 5, 6, 7\}$ *and the sorted* $\{q_i\}$ *are:*

	cdeg
(1-4)	1022
(2-5)	0122
(3-6)	2212
(7)	2221

Thus $J = \{$*1-4,2-5,3-6*$\}$ *and* $F/G = c\overline{d} + \overline{c}d + e$.

Algebraic division filters were divised to speed up algebraic division. The function f_j is not an algebraic divisor of f_i if any of the following cases is true:

1. f_j contains a literal not in f_i.
2. f_j has more terms than f_i.
3. For any literal, its literal count in f_j exceeds that in f_i.
4. y_i is in the transitive fanin of f_j.

4 Algebraic Divisors and Kernels

So far, we learned how to divide a given expression F by another expression G. But how do we find G? The problem is that there are too many Boolean divisors, so a practical strategy is to restrict the exploration to algebraic divisors, i.e., we restrict the problem to: given a set of functions $\{F_i\}$, find common weak (algebraic) divisors.

4.1 Kernels and Kernel Intersections

Definition 4.1 *An expression* F *is* **cube-free** *if no cube divides the expression evenly, i.e., if there are no expression* G *and cube* c *such that* $F = Gc$.

Examples are: $ab \vee c$ is cube-free; $ab \vee ac$ and abc are not cube-free. Notice that a cube-free expression must have more than one cube.

Definition 4.2 *The* **primary divisors** *of an expression* F *are the elements of the set of expressions*

$$\mathcal{D}(F) = \{F/c \mid c \text{ is a cube } \}.$$

Definition 4.3 *The* **kernels** *of an expression* F *are the elements of the set of expressions*

$$\mathcal{K}(F) = \{G \mid G \in \mathcal{D}(F) \text{ and } G \text{ is cube-free } \}.$$

In other words, the kernels of an expression F are the cube-free primary divisors of F.

Definition 4.4 *A cube* c *used to obtain the kernel* $K = F/c$ *is called a* **co-kernel** *of* K, *and* $\mathcal{C}(F)$ *is used to denote the set of co-kernels of* F.

Example 4.1 *Given the following expression Q, the table below lists the kernels and co-kernels.*

$$\begin{aligned} Q &= adf + aef + bdf + bef + cdf + cef + g \\ &= (a+b+c)(d+e)f + g \end{aligned}$$

kernel	*co-kernel*
$a+b+c$	$df,\ ef$
$d+e$	$af,\ bf,\ cf$
$(a+b+c)(d+e)$	f
$(a+b+c)(d+e)f+g$	1

Brayton and McMullen established in [3] the fundamental Th. 4.1 that motivates the role of kernels. It states that if two expressions F and G are such that $\mathcal{K}(F)$ and $\mathcal{K}(G)$ have at most one term in common, then F and G have **no common algebraic divisors with more than one term**.

Lemma 4.1 *Every divisor G of expression F is contained in a primary divisor, i.e., if G divides F, then $\exists P \in \mathcal{D}(F)$ such that $G \subseteq P \in \mathcal{D}(F)$.*

Proof. From [20]. Let c be a cube of F/G. Then $G \subseteq F/(F/G)$ and $F/(F/G) \subseteq F/c \in \mathcal{D}(F)$. □

Example 4.2 *Let $F = ac+ad+bc+bd+ec+ed+cd$. Consider the algebraic divisor $G = a+b$ that is not a primary divisor and let us build a primary divisor that contains G according to the lemma. From $F/G = c+d$, take cube $c \in F/G = c+d$. Then $G = a+b \subseteq F/(F/G) \equiv F/(c+d) = a+b+e$ and $F/(F/G) = a+b+e \subseteq F/c = a+b+e+d \in \mathcal{D}(F)$. So given the algebraic divisor $G = a+b$ we found a primary divisor $P = a+b+e+d$ such that $G \subseteq P$, i.e., all cubes in G are also cubes in P.*

Theorem 4.1 *Expressions F and G have a common multiple-cube divisor if and only if $\exists K_F \in \mathcal{K}(F)$, $\exists K_G \in \mathcal{K}(G)$ such that $|K_F \cap K_G| > 1$, i.e., $K_F \cap K_G$ is an expression with at least two terms (it is a not a single cube).*

Proof. From [20].

If part: If there are a kernel $K_F \in \mathcal{K}(F)$ and a kernel $K_G \in \mathcal{K}(G)$ whose algebraic intersection D is an algebraic expression with at least two cubes, then D is by definition a common divisor of F and G with at least two cubes.

Only if part: Let D be an algebraic divisor with at least two cubes. Then there is a cube-free expression E that divides D (either D is cube-free or we make it cube-free by dividing it by the largest cube divisor).

By lemma 4.1, $\exists P_F \in \mathcal{D}(F)$ and $\exists P_G \in \mathcal{D}(G)$ such that $E \subseteq P_F \in \mathcal{D}(F)$ and $E \subseteq P_G \in \mathcal{D}(G)$.

Since E is cube-free also P_F and P_G are cube-free. This means that P_F and P_G are cube-free primary divisors, i.e., they are kernels of their respective functions: $P_F \in \mathcal{K}(F)$ and $P_G \in \mathcal{K}(G)$.

From $E \subseteq P_F$ and $E \subseteq P_G$, it follows $E \subseteq P_F \cap P_G$ and, since E is non-trivial (it has at least two cubes), also $P_F \cap P_G$ is non-trivial. □

In summary, if we compute the kernels of all functions and there are no non-trivial intersections, then the only common algebraic divisors left are single cube divisors (these are not the only common divisors of F and G, because there could be common Boolean divisors).

Example 4.3 *$F = a(bc + \overline{b}d) + e$, $G = a(bc + \overline{b}e) + d$;*
$\mathcal{K}(F) = \{bc + \overline{b}d, a(bc + \overline{b}d) + e\}$, $\mathcal{K}(G) = \{bc + \overline{b}e, a(bc + \overline{b}e) + d\}$;
$\{K_F \cap K_G : K_F \in \mathcal{K}(F),\ K_G \in \mathcal{K}(G),\ K_F \cap K_G \neq \emptyset\} = \{bc, abc\}$ are cubes, so F and G have no common non-trivial algebraic divisors.

$F = abc + cd + e$, $G = ab + \overline{c}d + e$;
$\mathcal{K}(F) = \{ab + cd + e\}$, $\mathcal{K}(G) = \{ab + \overline{c}d + e\}$;
$\{K_F \cap K_G : K_F \in \mathcal{K}(F),\ K_G \in \mathcal{K}(G),\ K_F \cap K_G \neq \emptyset\} = \{ab + e\}$ is a common non-trivial algebraic divisor of F and G.

Some subsets of kernels of interest in computations are defined introducing the notion of level of a kernel.

Definition 4.5

$$\mathcal{K}^n(F) = \begin{cases} \{k \in \mathcal{K}(F) \mid \mathcal{K}(k) = \{k\}\} & n = 0 \\ \{k \in \mathcal{K}(F) \mid \forall k_1 \neq k \in \mathcal{K}(k) \Rightarrow k_1 \in \mathcal{K}^{n-1}(F)\} & n > 0 \end{cases}$$

If $k \in \mathcal{K}^0(F)$, then k is a **level-0 kernel** *of F.*
If $k \in \mathcal{K}^n(F)$, but $k \notin \mathcal{K}^{n-1}(F)$, then k is a **level-n kernel** *of F.*

Intuitively, a kernel of level 0 has no kernels except itself. Similarly, a kernel of level n has at least one kernel of level $n-1$ but no kernels (except itself) of level n or greater. Thus

$$\mathcal{K}^0(F) \subset \mathcal{K}^1(F) \subset \mathcal{K}^2(F) \subset \cdots \subset \mathcal{K}^n(F) \subset \mathcal{K}(F).$$

Example 4.4

$$\begin{aligned} F &= (a + b(c + d))(e + g) \\ K_1 &= a + b(c + d) \in \mathcal{K}^1,\ \notin \mathcal{K}^0 \\ K_2 &= c + d \in \mathcal{K}^0 \end{aligned}$$

$$\begin{aligned} F &= a(b\overline{d} + c(\overline{b} + d)) + b\overline{c}d \\ \mathcal{K}^0(F) &= \{\overline{b} + d, a\overline{d} + \overline{c}d, b\overline{c} + ac\} \\ \mathcal{K}^1(F) &= \{b\overline{d} + c(d + \overline{b})\} \cup \mathcal{K}^0(F) \\ \mathcal{K}^2(F) &= \{a(b\overline{d} + c(\overline{b} + d)) + b\overline{c}d\} \cup \mathcal{K}^1(F) \\ \mathcal{K}(F) &= \mathcal{K}^2(F) \end{aligned}$$

Fig. 11 shows the pseudo-code of KERNEL, an algorithm to compute all kernels, when invoked as KERNEL$(0, F)$. Note that the literals appearing in F are denoted by $\ell_i, i = 1, \ldots, n$.

Note the following facts:

1. The test $(l_k \in c)$ is a **major** efficiency factor. It also guarantees that no co-kernel is tried more than once.

2. This algorithm has stood up to all attempts to find faster ones.

3. Can be used to generate all co-kernels.

```
KERNEL(j, G) {
    R = ∅
    if (G is cube-free) R = R ∪ {G}
    for (i = j + 1, ..., n) {
        if (ℓ_i appears only in one term) continue
        c = largest cube dividing G/ℓ_i evenly
        if (ℓ_k ∈ c, for some k ≤ i) continue
        else {
            R = R ∪ KERNEL(i, G/(ℓ_i c))
        }
    }
    return R
}
```

Figure 11: An algorithm to compute the kernels of F when invoked as KERNEL$(0, F)$.

4. A simple modification of the kerneling algorithm allows to generate only the kernels of a certain level d. In case of level 0, one observes that if no kernels are found in the *for* loop then the argument is a kernel of level 0.

Example 4.5 *Fig. 12 shows a fragment of the kerneling computation tree for* $F = a((bc+fg)(d+e)+de(b+cf)))+beg$. *Kernels and co-kernels are listed below.*

co-kernel	**kernel**
a	$bcd + bce + bde + cdef + dfg + efg$
ab	$cd + ce + de$
abc	$d + e$
abd	$c + e$
abe	$c + d$
ac	$bd + be + def$
acd	$b + ef$
ace	$b + df$
ad	$bc + be + cef + fg$
ade	$b + cf$
adf	$ce + g$
ae	$bc + bd + cdf + fg$
aef	$cd + g$
af	$cde + dg + eg$
afg	$d + e$
b	$acd + ace + ade + eg$
be	$ac + ad + g$
e	$abc + abd + acdf + afg + bg$
eg	$af + b$
g	$adf + aef + be$
1	$abcd + abce + abde + acdef + adfg + aefg + beg$

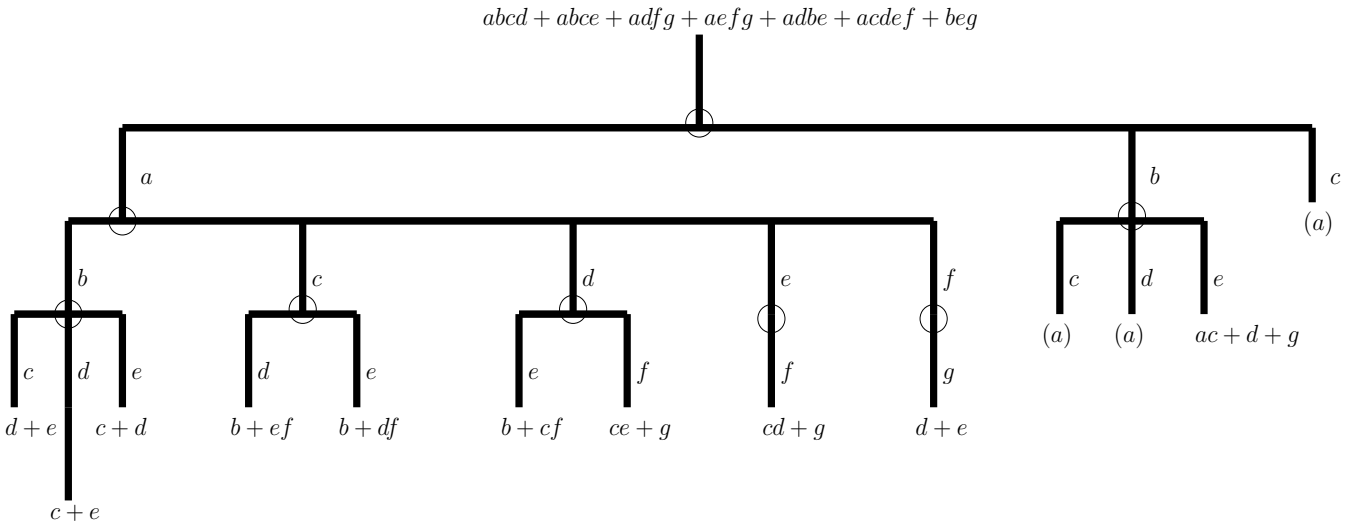

Figure 12: Kerneling illustrated from Example 4.5.

Co-kernels abc and afg generate both the same kernel $d + e$.

In [32] interesting properties of kernels with respect to prime factorization of logical expressions are investigated, where an expression is defined to be prime if it cannot be factored algebraically. We state some results.

Theorem 4.2 *(Unique factorization theorem for weak division) An expression* F *has a unique prime factorization* $F = \Pi_i F_i$*, where each* F_i *is a prime expression.*

Theorem 4.3 *If* P *is a kernel of* F *and is a prime expression, then* P *is a kernel of exactly one of the prime factors of* F.

Theorem 4.4 *If* K *is a level-0 kernel of* F*, then* K *is a kernel of exactly one of the prime factors of* F.

4.2 Fast Extraction of Divisors

Too much time may be spent for extracting divisors based on kernels: indeed some functions (e.g., symmetric functions) have too many kernels and kernels need to be recomputed often (e.g., after substitution). This motivates the restriction to a subset of divisors: 2-cube divisors, and 2-literal cube divisors, introduced by Rajski et Vasudevamurthy [36, 42].

Example 4.6

The expression $F = abd + \overline{a}\overline{b}d + \overline{a}cd$ *has the following 2-cube and 2-literal divisors:*

1. $ab + \overline{a}\overline{b}$, $\overline{b} + c$, $ab + \overline{a}c$ *(2-cube divisors);*

2. $\overline{a}d$ *(2-literal cube divisor).*

An algorithm for finding such restricted divisors has been provided in [36]. The extraction of 2-cube divisors is a polynomial operation, because there are $\mathcal{O}(n^2)$ 2-cube divisors in an algebraic expression with n cubes. Additional attractive computational features of the extraction procedure are:

1. Extraction of 2-cube divisors and 2-literal cube divisors is done concurrently.

2. Complement divisors are recognized concurrently.

3. The result can be expanded to multiple-cube divisors and single cube divisors of any size.

Example 4.7 *Consider* $F = abd + \overline{a}\overline{b}d + \overline{a}cd$*, the following complement divisors are recognized concurrently:*

1. $k = ab + \overline{a}\overline{b}, \overline{k} = a\overline{b} + \overline{a}b$ *(both 2-cube divisors);*

2. $k = ab + \overline{a}c, \overline{k} = \overline{ac} + a\overline{b}$ *(both 2-cube divisors);*

3. $k = \overline{a}b + ac, \overline{k} = \overline{a}\overline{b} + a\overline{c}$ *(both 2-cube divisors);*

4. $c = ab$ *(2-literal cube),* $\overline{c} = \overline{a} + \overline{b}$ *(2-cube divisor).*

The extraction procedure of 2-cube divisors has $\mathcal{O}(n^2)$ complexity and, given $F = \{c_i\}$, defines a set $\mathcal{D}_2(F) = \{d \mid d = \text{make\_cube\_free}(c_i + c_j), \forall c_i, c_j \in F\}$, which takes all pairs of cubes in F and makes them cube-free, by the operation *make_cube_free* that divides the terms of its argument by the largest cube common to them.

Example 4.8 *Given* $F = axe + ag + bcxe + bcg$*, then* $\mathcal{D}_2(F) = \{xe + g, a + bc, axe + bcg, ag + bcxe, xe + g\}$*, because make_cube_free*$(axe + ag) = xe + g$*, make_cube_free*$(axe + bcxe) = a + bc$*, make_cube_free*$(axe+bcg) = axe+bcg$*, make_cube_free*$(ag+bcxe) = ag+bcxe$*, make_cube_free*$(bcxe+bcg) = xe + g$*.*

Note that the functions are made algebraic expressions before generating double-cube divisors; also, not all 2-cube divisors are kernels.

Example 4.9 $\mathcal{K}(axe+ag+af) = \{xe+g+f\}$*, but* $\mathcal{D}_2(axe+ag+af) = \{xe+g, g+f, xe+f, \overline{g}\overline{f}\}$*.*

$\mathcal{K}(abd + \overline{a}\overline{b}d + \overline{a}cd) = \{ab + \overline{a}\overline{b} + \overline{a}c, \overline{b} + c\}$*, but* $\mathcal{D}_2(abd + \overline{a}\overline{b}d + \overline{a}cd) = \{ab + \overline{a}\overline{b}, \overline{b} + c, ab + \overline{a}c, \overline{a}d, \overline{a}b + a\overline{b}, b\overline{c}, \overline{ac} + a\overline{b}, a + \overline{d}\}$*.*

Some lemmas proved in [42] allow to establish what complements of elements in $\mathcal{D}_2(F)$ are also 2-cube divisors or 2-literal cube divisors to be added to $\mathcal{D}_2(F)$. Moreover, some lemmas established in [36] show the relations between single cube divisors, double-cube divisors and their complements, allowing the concurrent computation of all of them and their addition to $\mathcal{D}_2(F)$.

Given two algebraic expressions F and G, during decomposition we need to establish whether F has a complement cube divisor in G, and F has a common-cube divisor in G. A result from [36] states that given two algebraic expressions F and G, if certain structural relations are verified, then F has neither a complement double-cube divisor nor has a complement single-cube divisor in G.

A fundamental task during decomposition is to establish whether two or more expressions have any common algebraic divisors other than single cubes. Th. 4.1 stated that there are multiple-cube common divisors among different functions if and only if there are non-trivial intersections among their kernels, and justified the focus on kernels as a subset of algebraic divisors. The following theorem from [42] shows that the same argument holds also for 2-cube divisors, which are an even smaller subset of algebraic divisors.

Theorem 4.5 *Expressions F and G have a common multiple-cube divisor if and only if $\mathcal{D}_2(F) \cap \mathcal{D}_2(G) \neq 0$.*

Proof. If part: If $\mathcal{D}_2(F) \cap \mathcal{D}_2(G) \neq 0$ then $\exists D \in \mathcal{D}_2(F) \cap \mathcal{D}_2(G)$ that is a double-cube divisor of F and G. Then D is a multiple-cube divisor of F and G.

Only if part: Suppose $C = c_1 + c_2 + \ldots + c_m$ is a multiple-cube divisor of F and of G. Take any $E = c_i + c_j, c_i, c_j \in C$. If E is cube-free, then $E \in \mathcal{D}_2(F) \cap \mathcal{D}_2(G)$. If E is not cube-free, then let $\widetilde{E} = \text{make\_cube\_free}(c_i + c_j)$, given that $\widetilde{E}$ exists because F and G are algebraic expressions. Hence $\widetilde{E} \in \mathcal{D}_2(F) \cap \mathcal{D}_2(G)$. □

As a result of Th. 4.5, all multiple-cube divisor can be "discovered" from double-cube divisors.

Example 4.10 *Suppose that $C = ab + ac + d$ is a multiple divisor of F and G. If $E = ac + d$, E is cube-free and $E \in \mathcal{D}_2(F) \cap \mathcal{D}_2(G)$. If $E = ab + ac$, make_cube_free$(E) = \widetilde{E} = b + c$ and $\widetilde{E} \in \mathcal{D}_2(F) \cap \mathcal{D}_2(G)$.*

In summary, an algorithm for fast divisor extraction has the following steps:

1. Generate and store all 2-cube divisors and 2-literal cube divisors and recognize complement divisors.
2. Find the best (by value) 2-cube divisor or 2-literal cube divisor at each stage and extract it.
3. Update the set of 2-cube divisors after extraction
4. Iteratively extract divisors until no more improvement.

Experimentally fast extraction of divisors is much faster and of comparable quality with respect to general kernel extraction.

5 Optimization of Boolean Networks by Division

We discuss how algebraic and Boolean division are used in modern multi-level logic synthesis systems like SIS to define restructuring operations on a Boolean network. These network operations are: factoring, decomposition, substitution, extraction, elimination, which are variously combined into sequences of optimization steps in synthesis tools.

5.1 Factoring and Decomposition

Factoring is the operation to convert a logical expression usually available in SOP form into a factored form with a minimum number of literals. From the state-of-art of computational tools, exact techniques are too expensive and so one resorts to heuristic algorithms.

$FACTOR(F)$ {
    if (F has no factor) return F
    (e.g. if $|F| = 1$, or an OR of literals)
    $D = CHOOSE\_DIVISOR(F)$
    $(Q, R) = DIVIDE(F, D)$
    return $FACTOR(Q)FACTOR(D) + FACTOR(R)$
}

Figure 13: General structure of a factoring algorithm.

The abstract scheme of a factoring algorithm is shown in Fig. 13. By choosing a divisor and a division operation one can design a specific factoring algorithm. It must be said that this scheme is too simple-minded and that a better generic factoring algorithm (still uncommitted as type of divisor and division) improving hastily chosen divisors is shown in Fig. 14.

The next two examples Ex. 5.1 and 5.2 motivate the design of the generic algorithm in Fig. 14. The following notation with reference to the algorithm in Fig. 13 is used in Ex. 5.1 and 5.2: F is the original function, D is the divisor, Q is the quotient, P is a partial factored form, O is the final factored form by $FACTOR$. In the examples we assume algebraic operations only.

In the first example there is a problem due to the fact that the quotient is a single cube.

Example 5.1

$$\begin{aligned} F &= abc + abd + ae + af + g \\ D &= c + d \\ Q &= ab \\ P &= ab(c + d) + ae + af + g \\ O &= ab(c + d) + a(e + f) + g \end{aligned}$$

O is not optimal since not maximally factored. It can be further factored to

$$a(b(c + d) + e + f) + g.$$

This problem occurs when the quotient Q is a single cube, and some of the literals of Q also appear in the remainder R.

A solution of the problem highlighted in Ex. 5.1 is the following:

1. If the quotient Q is not a single cube, then done.
2. If the quotient Q is a single cube, then pick a literal ℓ_1 in the cube which occurs in the greatest number of cubes of F.
3. Divide F by ℓ_1 to obtain a new divisor D_1. Now, F has a new partial factored form

$$(\ell_1)(D_1) + (R_1)$$

   and literal ℓ_1 does not appear in R_1.

Note that the new divisor D_1 contains the original D as a divisor because ℓ_1 is a literal of Q. When recursively factoring D_1, D can be discovered again.

In the second example there is a problem due to the fact that the quotient is not cube-free.

Example 5.2

$$F = ace + ade + bce + bde + cf + df$$
$$D = a + b$$
$$Q = ce + de$$
$$P = (ce + de)(a + b) + (c + d)f$$
$$O = e(c + d)(a + b) + (c + d)f$$

Again, O is not maximally factored because $(c + d)$ is common to both products $e(c + d)(a + b)$ and $(c + d)f$. The final factored form should have been

$$(c + d)(e(a + b) + f).$$

A solution of the problem highlighted in Ex. 5.2 is the following:

1. Make Q cube-free to get Q_1.
2. Obtain a new divisor D_1 by dividing F by Q_1.
3. If D_1 is cube-free, the partial factored form is $F = (Q_1)(D_1) + R_1$, and we can recursively factor Q_1, D_1, and R_1.
4. If D_1 is not cube-free, let $D_1 = cD_2$ and $D_3 = Q_1D_2$. We have the partial factoring $F = cD_3 + R_1$ (i.e., just start with c as the divisor). Now recursively factor D_3 and R_1.

Various kinds of factoring are obtained by choosing different forms of $DIVISOR$ and $DIVIDE$, as listed next.

- $CHOOSE\_DIVISOR$ can be any of the following:
  - $LITERAL$ - choose a literal, or the best literal.
  - $QUICK\_DIVISOR$ - choose a level-0 kernel.
  - $BEST\_DIVISOR$ - choose the best kernel.
- $DIVIDE$ can be any of the following:
  - Algebraic Division.
  - Boolean Division.

Three specialized factoring algorithms to mention are:

- $QUICK\_FACTOR$, whose divisor is a level-0 kernel (produced by $QUICK\_DIVISOR$) and whose division is weak division.
- $GOOD\_FACTOR$, whose divisor is a level-0 kernel (produced by $BEST\_KERNEL$, which looks at all kernels and picks a kernel k that, when substituted into the original form F, maximally reduces the total number of SOP literals of F and k), and whose division is weak division.
- $BOOLEAN\_FACTOR$, whose divisor is the same as for $GOOD\_FACTOR$ and whose division is Boolean division.

```
GFACTOR(F, DIVISOR, DIVIDE) {
    D = DIVISOR(F)
    if (D = ∅) return F
    (Q, R) = DIVIDE(F, D)
    if |Q| = 1 return LF(F, Q, DIVISOR, DIVIDE)
    Q = make_cube_free(Q)
    (D, R) = DIVIDE(F, Q)
    second divide to improve divisor
    if (cube_free(D)) {
        Q = GFACTOR(Q, DIVISOR, DIVIDE)
        D = GFACTOR(D, DIVISOR, DIVIDE)
        R = GFACTOR(R, DIVISOR, DIVIDE)
        return (Q)(D) + R
    } else {
        C = common_cube(D)
        return LF(F, C, DIVISOR, DIVIDE)
    }
}


LF(F, C, DIVISOR, DIVIDE) {
    l = best_literal(F, C) /* most common literal */
    (Q, R) = DIVIDE(F, l)
    C = common_cube(Q) /* largest common cube */
    Q = cube_free(Q)
    Q = GFACTOR(Q, DIVISOR, DIVIDE)
    R = GFACTOR(R, DIVISOR, DIVIDE)
    return lC(Q) + R
}
```

Figure 14: Improved general structure of a factoring algorithm.

$DECOMPOSE(\mathcal{N}, f_i)$ {
    $k = CHOOSE\_KERNEL(f_i)$
    if ($k = \emptyset$) return
    $f_{m+\ell} = k$ /* create new node $m + \ell$ */
    $f_i = (f_i/k)y_{m+\ell} + (f_i/\overline{k})\overline{y}_{m+\ell} + r$
    /* $\mathcal{N}'$ is modified network */
    $\mathcal{N}'' = DECOMPOSE(\mathcal{N}', f_i)$
    $\mathcal{N} = DECOMPOSE(\mathcal{N}'', f_{m+\ell})$
}

Figure 15: General structure of a decomposition algorithm.

Example 5.3 *Given the SOP form $H = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg$, we show the results of some factoring algorithms previously mentioned:*

- *LITERAL_FACTOR:* $H = a(c + d + e + g) + b(c + d + e + f) + c(e + f) + d(f + g)$.
- *QUICK_FACTOR:* $H = g(a + d) + (a + b)(c + d + e) + c(e + f) + f(b + d)$.
- *GOOD_FACTOR:* $H = (c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce$.

An excellent discussion of factoring can be found in [24, 43].

Decomposition is similar to factoring, from which it differs because each divisor is added as a new node in the network and the associated variable is substituted into the function being decomposed. Fig. 15 shows the scheme of a decomposition algorithm, based on choosing kernels as candidate factors. Similar to factoring, we can define

- QUICK_DECOMP: pick a level 0 kernel.
- GOOD_DECOMP: pick the best kernel.

In general factorization yields a tree network, because it uses factors only once, whereas decomposition yields a generic Boolean network, because it uses factors more than once.

5.2 Substitution

Substitution (or **Resubstitution**) checks whether an existing function f_i at node i in the network is the divisor of another function f_j or $\overline{f}_j$ at node j. If f_j is a divisor of f_i, then f_i is transformed into

$$f_i = qy_j + r.$$

Similarly for $\overline{f}_j$. Division may be either algebraic or Boolean, the former being more common. In practice, this is tried for each node pair of the network; if there are n nodes in the network, then one tries about $2n^2$ divisions (substitution is tried for positive and negative phase). Fig. 16 shows f_j being substituted in node f_i and Fig. 17 provides the operational scheme.

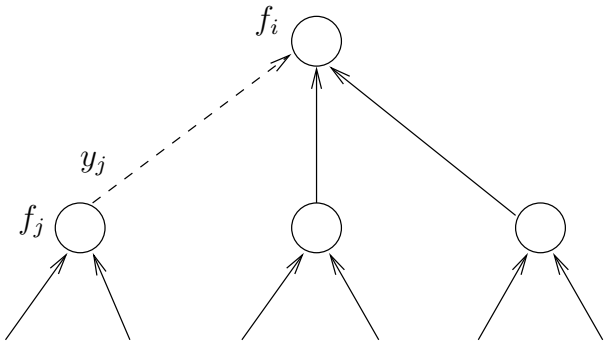

Figure 16: Illustration of substitution, where y_j (computed by node f_j) becomes a fanin of node f_i.

```
SUBSTITUTE(𝒩, f_i) {
    for each node f_j ≠ f_i {
        (h, r) = DIVIDE(f_i, f_j)
        if h ≠ 0 {
            f_i = y_i h + r
            f_new = h /* create new node f_new */
        }
    }
}
```

Figure 17: General structure of a substitution algorithm.

Example 5.4 *Let us consider Boolean substitution of x into F.*

$$
\begin{array}{lcl}
x & = & ab + cd + e \\
F & = & abf + \overline{a}cd + cdf + \overline{a}de + ef \\
D_1 & = & \overline{x}(ab + cd + e) + x\overline{(ab + cd + e)} \\
F_1 & = & xef + x\overline{a}cd + x\overline{a}de + xabf + xcdf \\
R_1 & = & ab\overline{f}x + ae\overline{f}x + \overline{d}e\overline{f}x + \overline{a}\ \overline{c}\ \overline{e}\ \overline{x} + \\
 & & \overline{b}\ \overline{c}\ \overline{e}\ \overline{x} + \overline{a}\ \overline{d}\ \overline{e}\ \overline{x} + \overline{b}\ \overline{d}\ \overline{e}\ \overline{x} + acd\overline{f}x \\
J & = & \{a, d, f, x\}\ \textit{(minimum literal support)} \\
F_3 & = & xf + x\overline{a}d + x\overline{a}d + xaf + xdf \\
F_4 & = & xf + x\overline{a}d \\
H_1 & = & f + \overline{a}d \\
F & = & x(f + x\overline{a}d) = (ab + cd + e)(f + \overline{a}d)
\end{array}
$$

For efficiency, we use filters to prevent trying a division. The cover G is not an algebraic divisor of F if any of the following is true:

1. G contains a literal not in F.

```
EXTRACT(𝒩) {
    repeat {
        for each node f_i {
            𝒩 = DECOMPOSE(𝒩, f_i)
        }
        for each node f_i {
            𝒩 = SUBSTITUTE(𝒩, f_i)
        }
        𝒩 = ELIMINATE(𝒩) /* eliminate nodes with small value */
    } until (cost function does not improve)
}
```

Figure 18: General structure of an extraction algorithm.

2. G has more terms than F.

3. For any literal, its count in G exceeds that in F.

4. F is in the transitive fanin of G.

5.3 Extraction

Extraction identifies common subexpressions that become new nodes in the Boolean network. Common subexpressions are multiple-cube common divisors obtained by computing the kernels of the functions in the network. The best kernel intersection may be chosen as the new factor; the cost function to evaluate the value of a kernel intersection may be the number of literals in the factored form for the network. Since exhaustive kerneling may be expensive, a good compromise is to look only for double cube divisors. So the steps of extraction are as follows (see also Fig. 18):

1. Find **all** kernels of **all** functions

2. Choose one with best "value".

3. Create a new node with this as function.

4. Algebraically substitute the new node everywhere.

5. Repeat 1,2,3,4 until the best value is less or equal than a given threshold.

We can combine decomposition and substitution to provide an effective extraction algorithm. Fig. 19 shows the extraction of a factor common out of three nodes, and the creation of a new node realizing the function of the selected factor.

Example 5.5 *An example of extraction follows.*

$$\begin{aligned} F_1 &= ab(c(d+e)+f+g)+h \\ F_2 &= ai(c(d+e)+f+j)+k \end{aligned}$$

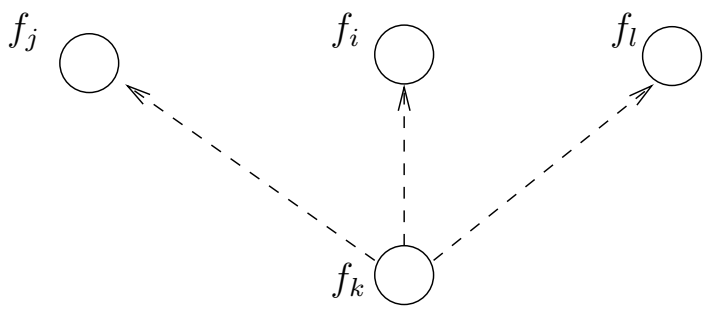

Figure 19: Illustration of extraction of node f_k out of nodes f_j, f_i and f_l.

kernel extraction: $\mathcal{K}^0(F_1) = \mathcal{K}^0(F_2) = \{d + e\}$

$$\begin{aligned} l &= d + e \\ F_1 &= ab(cl + f + g) + h \\ F_2 &= ai(cl + f + j) + k \end{aligned}$$

kernel extraction: $\mathcal{K}^0(F_1) = \{cl + f + g\}$, $\mathcal{K}^0(F_2) = \{cl + f + j\}$, $\mathcal{K}^0(F_1) \cap \mathcal{K}^0(F_2) = cl + f$

$$\begin{aligned} m &= cl + f \\ F_1 &= ab(m + g) + h \\ F_2 &= ai(m + j) + k \end{aligned}$$

kernel extraction: *no kernel intersections at this point.*
cube extraction: $\{am\}$

$$\begin{aligned} n &= am \\ F_1 &= b(n + ag) + h \\ F_2 &= i(n + aj) + k \end{aligned}$$

6 Conclusions

In this chapter we covered the foundations of algebraic and Boolean division in multi-level logic synthesis. Variants of division have been proposed in the literature, e.g., algebraic division where the distributive law is augmented with annihilation ($a\overline{a} = 0$) and idempotency ($aa = a$), used in the system M32 [27], and an analogous proposal called coalgebraic division [25]; a form of constrained Boolean division, called extended algebraic division used in the system GLSS [26, 12], but many more could be listed. A way to perform Boolean division by redundancy addition and removal has been presented in [13].

To keep the chapter short we did not delve into the theory of optimal and near-optimal factored forms [43, 9, 10, 11].

Moreover, we skipped altogether rectangle covering, which refers to an elegant and useful reduction of common-cube extraction and kernel intersection extraction to the problem of covering a matrix with rectangles [6]. Rectangles in a matrix provide an alternate way of interpreting the kernels of a logic function, e.g., given $F = ab\overline{d} + acd + bcd$, then its related cube-literal matrix

	a	b	c	d	$\overline{d}$
$ab\overline{d}$	1	1	0	0	1
acd	1	0	1	1	0
bcd	0	1	1	1	0

has the property that an arbitrary cube is a co-kernel of F if and only if it is the cube corresponding to a prime rectangle with at least two rows of the cube-literal matrix of F. The co-rectangle of the prime rectangle identifies the kernel. A prime rectangle is a rectangle of 1s not contained in any other rectangle. In this case, the prime rectangle $(\{1,2\},\{1\})$ corresponds to co-kernel a and kernel $b\overline{d} + cd$, the prime rectangle $(\{1,3\},\{2\})$ corresponds to co-kernel b and kernel $a\overline{d} + cd$, and the prime rectangle $(\{2,3\},\{3,4\})$ corresponds to co-kernel cd and kernel $a + b$ (the trivial kernel F is not accounted by this interpretation).

Finally, it must be mentioned that a major effort in multi-level logic synthesis has been devoted to node simplification, by exploiting the don't care set that expresses the restricted controllability and observability of a node in the network. Node minimization uses global network information, but the quality of the final result is affected by the starting point, obtained by preliminary network restructuring based on algebraic and Boolean division. To do justice to this subject would take another chapter [4, 1, 5].

Acknowledgments

The authors thank Prof. Yves Crama for very careful readings and discussions on the drafts of this chapter that helped a lot in improving the text.

References

[1] D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. The Boulder optimal logic design system. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 62–65, November 1987.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] R. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *The Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, April 1982.

[4] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, pages 1062–1081, November 1987.

[5] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *Proceedings of the IEEE*, vol. 78(no. 2):264–300, February 1990.

[6] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangular covering problem. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 66–69, November 1987.

[7] Robert K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–198, 1987.

[8] Robert K. Brayton and Curt McMullen. Synthesis and optimization of multi-stage logic. In *ICCD*, pages 23–28, October 1984.

[9] Giuseppe Caruso. Near optimal factorization of Boolean functions. *IEEE Transactions on Computer-Aided Design*, 10(8):1072–1078, Aug 1991.

[10] Giuseppe Caruso. Boolean factoring of logic functions. *Proceedings of the 36th Midwest Symposium on Circuits and Systems*, Vol. 2:1312–1315, Aug 1993.

[11] Giuseppe Caruso. An improved algorithm for Boolean factoring. *ISCAS '94, IEEE International Symposium on Circuits and Systems*, Vol. 1:241–244, May 30 -June 2 1994.

[12] Giuseppe Caruso. An algorithm for exact extended algebraic division. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(2):462–471, February 2003.

[13] Shih-Chieh Chang and D.I. Cheng. Efficient Boolean division and substitution using redundancy addition and removing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(8):1096–1106, Aug 1999.

[14] O. Coudert. Two-level logic minimization: an overview. *Integration*, 17-2:97–140, October 1994.

[15] John A. Darringer, Daniel Brand, John V. Gerbi, William H. Joyner Jr., and Louise Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):537–545, 1984.

[16] John A. Darringer, Daniel Brand, John V. Gerbi, William H. Joyner Jr., and Louise Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 44(1):157–166, 2000.

[17] John A. Darringer, William H. Joyner Jr., C. Leonard Berman, and Louise Trevillyan. Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272–280, 1981.

[18] E.S. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Transactions on Computers*, vol. C-18(12):1098–1109, December 1969.

[19] Ronald C. de Vries. Comment on Lawler's multilevel Boolean minimization. *Communications the ACM*, vol. 13(no. 4):265–266, April 1970.

[20] S. Devadas, A. Ghosh, and K. Keutzer. *Logic synthesis*. McGraw-Hill, 1994.

[21] S. Devadas, A. Wang, R. Newton, and A. Sangiovanni-Vincentelli. Boolean decomposition in multilevel logic optimization. *IEEE Journal of solid-state circuits*, pages 399–408, April 1989.

[22] D. Dietmeyer and Y. Su. Logic design automation of fan-in limited NAND networks. *IEEE Transactions on Computers*, vol. C-18(11):11–22, January 1969.

[23] M. Fujita, Y. Matsunaga, and M. Ciesielski. Multi-level logic optimization. In R. Brayton, S. Hassoun, and T. Sasao, editors, *Logic Synthesis and Verification*, pages 29–63. Kluwer, 2001.

[24] G. Hachtel and F. Somenzi. *Logic synthesis and verification algorithms*. Kluwer Academic, 1996.

[25] W.-J. Hsu and W.-Z. Shen. Coalgebraic division for multilevel logic synthesis. In *DAC '92: Proceedings of the 29th ACM/IEEE Conference on Design Automation*, pages 438–442, 1992.

[26] Bo-Gwan Kim and D.L. Dietmeyer. Multilevel logic synthesis with extended arrays. *IEEE Transactions on Computer-Aided Design*, 11(2):142–157, Feb 1992.

[27] Victor N. Kravets and Karem A. Sakallah. M32: a constructive multilevel logic synthesis system. In *DAC '98: Proceedings of the 35th Conference on Design Automation*, pages 336–341, June 1998.

[28] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, 21(12):1377–1394, December 2002.

[29] Eugene L. Lawler. An approach to multilevel Boolean minimization. *Journal of the ACM*, vol. 11(no. 3):283–295, July 1964.

[30] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: a new exact minimizer for logic functions. *IEEE Transactions on VLSI Systems*, 1(4):432–440, December 1993.

[31] Patrick C. McGeer and Robert K. Brayton. Efficient, stable algebraic operations on logic expressions. In *Proceedings of International Conference on VLSI*, August 1987.

[32] Patrick C. McGeer and Robert K. Brayton. Efficient prime factorization of logic expressions. In *The Proceedings of the Design Automation Conference*, pages 221–225, June 1989.

[33] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[34] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting - A fresh look at combinational logic synthesis. In *The Proceedings of the Design Automation Conference*, pages 532–536, July 2006.

[35] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, ERL Tech. Rep., EECS Dept., UCB, Berkeley, CA, March 2005.

[36] Janusz Rajski and Jagadeesh Vasudevamurthy. Testability preserving transformations in multilevel logic synthesis. In *Proceedings of the International Test Conference*, pages 265–273, September 1990.

[37] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6:727–750, September 1987.

[38] Richard Rudell. *Logic synthesis for VLSI design*. PhD thesis, University of California, Berkeley, April 1989. Tech. Report No. UCB/ERL M89/49.

[39] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, Tech. Rep. No. UCB/ERL M92/41, Berkeley, CA, May 1992.

[40] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *The Proceedings of the International Conference on Computer Design*, pages 328–333, October 1992.

[41] Berkeley Logic Synthesis and Verification Group. ABC: a System for Sequential Synthesis and Verification. *http://www-cad.eecs.berkeley.edu/∼alanmi/abc*, Release 61225, 2009.

[42] Jagadeesh Vasudevamurthy and Janusz Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *ICCAD*, pages 510–513, 1990.

[43] Albert Wang. *Algorithms for Multi-Level Logic Optimization*. PhD thesis, University of California, Berkeley, April 1989. Tech. Report No. UCB/ERL M89/50.

[44] G. Whitcomb. Exact factoring of incompletely specified functions. EE290LS Class Project Report, EECS Dept., UCB, May 1988.