Microelectronics System Design Research Group

School of Electrical and Electronic Engineering

An Algebra of Switching Networks

Andrey Mokhov

Technical Report Series

NCL-EEE-MSD-TR-2012-178

April 2012 (revised November 2014)

Contact: andrey.mokhov@ncl.ac.uk

Supported by EPSRC grants EP/K001698/1 and EP/K034448/1

NCL-EEE-MSD-TR-2012-178

Microelectronics System Design Research Group
School of Electrical and Electronic Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

An Algebra of Switching Networks

Andrey Mokhov

April 2012 (revised November 2014)

Is it right, that regardless of the existence of the already elaborated algebra of logic, the specific algebra of switching networks should be considered as a utopia?

Paul Ehrenfest, 1910

Abstract

A switch, mechanical or electrical, is a fundamental building element of digital systems. The theory of switching networks, or simply *circuits*, dates back to Shannon's thesis (1937), where he employed Boolean algebra for reasoning about the *functionality* of switching networks, and graph theory for describing and manipulating their *structure*. Following this classic approach, one can deduce functionality from a given structure via *analysis*, and create a structure implementing a specified functionality via *synthesis*. The use of two mathematical languages leads to a 'language barrier' – whenever a circuit description is changed in one language, it is necessary to translate the change into the other one to keep both descriptions synchronised. For example, having amended a circuit structure one cannot be certain that the circuit functionality has not been broken, and vice versa.

This work presents a unified *algebra of switching networks*. Its elements are circuits rather than just Boolean functions (as in Boolean algebra) or vertices/edges (as in graph theory). This unified approach allows one to express both the functionality and structure of switching networks in the same mathematical language, thus removing the language barrier. It also brings in new methods of circuit composition that are of high importance for modern system design and development, which heavily rely on the reuse of components and interfaces. In this paper we demonstrate how to use the algebra to formally transform circuits, reason about their properties, and even solve equations whose 'unknowns' are circuits.

1 Introduction

The dawn of computer engineering was marked by manual design at the level of basic switching elements, such as electromechanical relays. The elements were large and expensive by today's standards, thus each one had to be accounted for. Prior to the seminal work by Shannon [34], the design of relay networks was a trial and error process on graphs and required a great deal of ingenuity. Shannon demonstrated that Boolean algebra could be used to reason about the functionality of relay networks and described the first analysis and

synthesis methods [34][35] that liberated designers from routine exploration of possible network structures. Huffman [18], Hohn and Schissler [16], as well as many other researchers, contributed to the theory of switching networks over the next decades.

Relays were very unreliable due to mechanical switching, therefore the issue of network reliability was very important at that time. Switching networks were built to mitigate the uncertainty of individual relays by providing structural redundancy; it became clear that having only the functional specification of a network in terms of a system of Boolean equations was not enough and other, non-functional, requirements had to be considered. The advent of more reliable switching elements, such as vacuum tubes and transistors, changed the course of switching theory, however. The research activity shifted towards the automation of very-large-scale integrated (VLSI) circuits [6][7][23][33].

As semiconductor technology marched forward, the cost of manufacturing a single transistor rapidly dropped and became negligible. During that time, manual low-level design was abandoned, and engineers began thinking and designing systems in terms of higher-level *components* (first logic gates, then arithmetic units, and now even whole IP cores!) and their *configurations* rather than in terms of transistors and electrical circuits. Nowadays, switching networks are hidden under multiple layers of abstraction and are no longer synthesised directly; instead, they are structurally put together to allow as much component reuse as possible and to avoid extremely expensive re-design, verification and test.

1.1 Motivation

This work is motivated by the return of the uncertainty related challenges and new requirements to switching networks imposed by emerging technologies and design styles:

Uncertainty & energy. A consequence of the shrinking size of a transistor is the growing uncertainty of its characteristics, which leads to lower reliability and shorter life-span of large transistor networks [5]. Individual reliability as well as collective energy consumption of transistors are dominant problems that cannot be solved at the level of system components. Once again designers are forced to consider non-functional aspects of basic switching elements and their networks [2].

In addition to environmental and intrinsic uncertainties, designers of secure electronics often introduce *intended uncertainty* in the systems in order to randomise their timing and power characteristics thereby reducing their vulnerability to Differential Power Analysis attacks [32].

New technologies. Novel switching technologies bring new non-functional phenomena into consideration. For example, carbon nanotubes [37] can potentially replace conventional transistors for technologies beyond 10nm, but nanotube networks are very sensitive to noise and nanotube imperfections, thus requiring synthesis of so-called 'immune' networks. Another example is nano-electro-mechanical (NEM) relays [21][22] that can achieve zero power leakage and have other unique characteristics. However, in addition to electrical delays, NEM relays exhibit (much longer) mechanical delays and a lot of effort is currently being dedicated to the synthesis of custom switching networks, where all mechanical delays occur concurrently [36].

Multimodality. Modern microelectronic systems are expected to support a number of different operating modes in order to be adaptive to dynamically changing environmental conditions, such as temperature, supply voltage [29], and energy availability [43][42]. The design of such systems requires methods for composing sets of modes (functionalities) under certain structural constraints on the resulting switching network, e.g., its size. This is difficult to achieve using standard design methodologies providing poor support for functional

composition.

To sum up, there is a need for methodologies for formal reasoning about non-functional properties of switching networks (reliability, timing, and energy). Such methodologies must support both structural and functional composition and be scalable enough to cope with the size of today's switching networks.

1.2 State-of-the-art and beyond

Existing formalisms for modelling structural and functional aspects of switching networks can be divided into two groups.

Structural formalisms. Graphs are a very natural representation for switching networks. Hence, many hardware description languages (HDLs) were developed to describe circuit structure by hierarchical graphs. The two most popular HDLs are Verilog [9] and VHDL [20]; they are convenient for structural composition of circuits and provide support for abstraction, encapsulation and reuse. Unfortunately, they lack a precise formal semantics, thus most properties can only be checked through simulation. Balsa [1] and Tangram [38] provide a methodology for structural composition of *handshake components,* component-level optimisation, and compilation of the result into circuits. However, these languages separate specification of components and their actual circuit implementation, and one cannot use them to reason about switch-level circuit properties.

Functional formalisms. Many high-level formalisms are targeted at modelling functional (or behavioural) system properties: Petri Nets [11][31], various process algebras (CSP [14], π-calculus [24], DI algebra [19]), Causal Nets [39], Concurrent Kleene Algebra [15], and many others. They excel at modelling high-level processes, resource allocation conflicts, causality and concurrency; however, they are not well-suited for modelling low-level switching networks of real-life size. For example, a practically important task of checking two circuits for equivalence may be very hard or even undecidable if the networks are represented with Petri Nets [13]. For this reason, functionality of switching networks is still modelled using *Boolean connectivity matrices*, as first suggested by Shannon [34] back in 1937 and further elucidated by Hohn *et al*. [16] and Bryant [6]. This approach is inherently inefficient, because an $n \times n$ Boolean matrix is required to describe the functionality of a network with n nodes. Furthermore, a Boolean connectivity matrix can only describe a circuit's functionality; all its structural properties are lost.

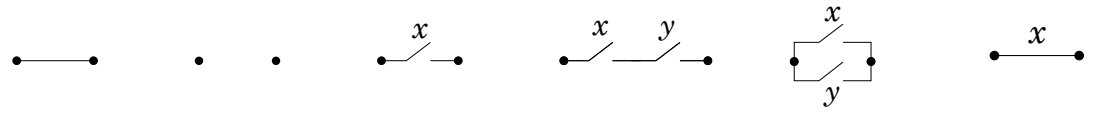
There is a clear separation between the above groups: the former is concerned with structural or *implementation* properties, while the latter is targeted at the *specification*. One can translate between the languages from different groups via *analysis* and *synthesis*, but these operations are non-trivial and time-consuming. A key objective of this work is to remove this language barrier.

In this paper we present a new *algebra of switching networks* that provides a mathematical instrument for reasoning about structural and functional properties of networks of interconnected Boolean switches. A network is represented by an algebraic expression, and the axioms of the algebra ensure that any permissible rewriting of the expression preserves the network functionality; the network structure, however, can be changed arbitrarily, thereby allowing exploration of the non-functional design space. The problem of analysis therefore corresponds to rewriting a given expression into a certain normal form, while the problem of synthesis translates to that of simplifying an expression subject to a set of non-functional constraints.

We start by giving a background on switching networks and Boolean algebra in Section 2. The algebra of switching networks is then defined axiomatically in Section 3. Algebraic analysis and synthesis of switching networks are illustrated by examples in Sections 4 and 5.

2 Switching networks

This section introduces *switching networks*, our graphical notation, and principles of Shannon's Boolean analysis [34].



(a) $f_a = 1$ (b) $f_b = 0$ (c) $f_c = x$ (d) $f_d = x \wedge y$ (e) $f_e = x \vee y$ (f) Simplified (c)

Figure 1: Switching networks and their connectivity functions

Structurally, switching networks [34] consist of *nodes* and *switches* that establish (resp., break) an electrical connection between two nodes when being ON (resp., OFF). Fig. 1 shows several networks and their *connectivity functions*[1] that evaluate to 1 when the nodes are connected. The first two networks correspond to the basic cases when a switch is always ON (the connectivity function $f_a = 1$) and always OFF ($f_b = 0$). The next case has the simplest non-trivial connectivity function $f_c = x$, where signal x controls the switch. A series connection of two switches yields $f_d = x \wedge y$, where the Boolean operator $x \wedge y$ represents the logical proposition 'x and y': the nodes are connected only when both switches are ON. The last example is a parallel connection of two switches that yields $f_e = x \vee y$, where the Boolean operator $x \vee y$ represents the logical proposition 'x or y'. We will further represent switches by simple edges annotated with connectivity functions, see Fig. 1(f).

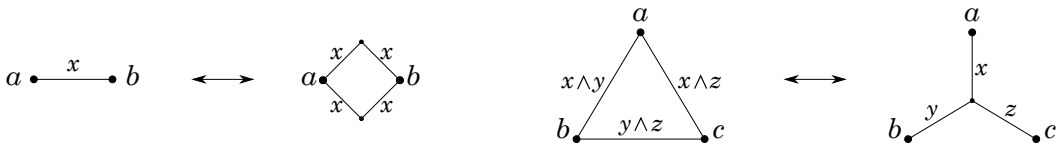
Using this natural Boolean interpretation of switching networks, one can write algebraic expressions corresponding to connectivity functions between any pair of nodes in a given network, thus performing its *Boolean analysis*. The inverse task of finding a switching network realising a giving set of connectivity functions is called *synthesis*. See [16] for an extensive review of these two fundamental concepts.

Separation of the structure (represented graphically by a switching network itself) and the functionality (captured by systems of Boolean equations) leads to an inability to formally reason about the functional correctness of structural transformations. For example, consider the two networks shown in Fig. 2(a). The leftmost network is just a switch with connectivity function $f_{ab} = x$. The rightmost network is an ingenious transformation of the former aimed at increasing its reliability. Assume that any switch can fail, i.e., become permanently ON or OFF, with a certain probability p. Then, in order for the transformed network to fail, at least two switches must fail – a much more rare event happening with a probability proportional to p^2. The transformed network has the same connectivity function $f'_{ab} = (x \wedge x) \vee (x \wedge x) = x$, thus the networks are *functionally equivalent*. However, the only way to prove the correctness of this simple structural transformation is to compute the connectivity functions and compare them. This is a time-consuming task that cannot be routinely performed for modern networks comprising billions of switches. Moreover, there is no efficient procedure for proving that the transformed network has higher reliability.

Figure 2(b) demonstrates another example of the same problem — the famous *delta-wye transformation* [34], which preserves the pairwise connectivity functions between the nodes a, b and c, but reduces the overall structural complexity of the network.

The aim of this paper is to develop a formal approach for reasoning about structural transformations that preserve the functionality of a switching network. The next section introduces the approach, while Section 4 demonstrates its application to the examples discussed above.

[1] The function of connectivity is also known under the names of *hindrance function* [35] and *transmission function* [18].



(a) Increasing network reliability (b) Delta-wye transformation

Figure 2: Examples of functionally equivalent transformations

3 Algebra

In this section we present a new algebra of switching networks that is built on the formalisms of *Conditional Partial Order Graphs* [25][30] and *Parameterised Graphs* [27] and inherits their key principle: it achieves a compact representation of multiple graphs by overlaying them in a graph annotated with Boolean conditions. Graphs with conditions were historically used to represent switching networks [6][34], but structural operations on such graphs were never axiomatised and the functional correctness of each structural transformation had to be proved in an ad hoc manner, not suitable for automation.

3.1 Undirected graphs

The structure of a switching network can be naturally represented by an *undirected graph* $G = (V, E)$, where *vertices* V correspond to nodes of the network, and *edges* $E \subseteq V \times V$ model the states of switches connecting the nodes. For a pair of nodes $u, v \in V$, an unordered pair of vertices $\{u, v\}$ belongs to the set of edges (denoted as $uv \in E$ for short) if there is an ON switch between them, i.e., the nodes are connected. We assume that a node is connected to itself, hence $vv \in E$ for all $v \in V$. If there is an OFF switch between nodes $u, v \in V$ or no switch at all then the corresponding edge is missing, that is, $uv \notin E$. We denote the *empty graph* $(\emptyset, \emptyset)$ by ε, and the *singleton graphs* $(\{v\}, \{\{v, v\}\})$ simply by v, for any $v \in \mathscr{V}$, where $\mathscr{V}$ is a fixed universe of vertices.

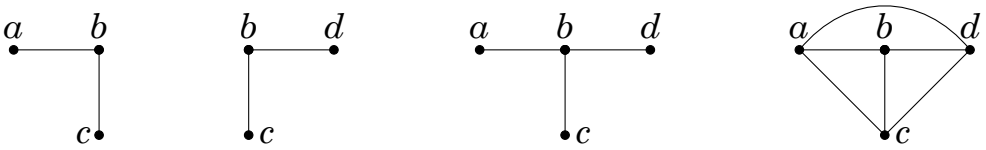
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. Here V_1 and V_2, as well as E_1 and E_2, are not necessarily disjoint. The following two operations on graphs are defined:

$$\text{Overlay:}\quad G_1 + G_2 \stackrel{\text{df}}{=} (V_1 \cup V_2, E_1 \cup E_2)$$

$$\text{Connection:}\quad G_1 \multimap G_2 \stackrel{\text{df}}{=} (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$$

An example in Fig. 3 illustrates these operations: $G_1 + G_2$ is a graph obtained by *overlaying* graphs G_1 and G_2, i.e., it contains the union of their nodes and edges; $G_1 \multimap G_2$ contains the union plus the edges connecting every node in G_1 to every node in G_2. Connection $\multimap$ has higher precedence than overlay $+$. By combining these operations one can construct *expressions* using the empty graph ε and the singleton graphs v as the basic building elements. Any graph $G = (V, E)$ can be constructed in this way, for example, by overlaying all of its edges: $G = \sum_{uv \in E} u \multimap v$. This is not the only way, nor is it the best one with respect to the size of the expression. A natural question arises: given two expressions, how can we decide if they represent graphs that are equivalent in some sense?

Semantically, two switching networks are *functionally equivalent* if they have the same set of nodes, and the connectivity function for each pair of nodes in the first network equals that of the corresponding pair in



(a) G_1 (b) G_2 (c) $G_1 + G_2$ (d) $G_1 — G_2$

Figure 3: Operations on graphs (self-loops are not shown)

the second network. A more concise definition is that two networks represented by graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are functionally equivalent if their *transitive closures* $G_1^* = (V_1, E_1^*)$ and $G_2^* = (V_2, E_2^*)$ coincide. The networks in Fig. 3(c,d) are equivalent as their nodes are connected either directly or via intermediate nodes. We call graphs with such equivalence relations *transitive* as in [27].

The equivalence relation is given by the following set of axioms:

- $+$ is commutative and associative:
  - $p + q = q + p$,
  - $(p + q) + r = p + (q + r)$.
- $—$ is commutative: $p — q = q — p$.
- The empty graph ε is an identity of $—$: $p — \varepsilon = p$.
- $—$ distributes over $+$: $p — (q + r) = p — q + p — r$.
- Decomposition: $(p — q) — r = p — q + p — r + q — r$.
- Closure: if $q \neq \varepsilon$ then $(p — q) — r = p — q + q — r$.

Using the closure axiom in combination with decomposition, one can expand any graph to its *transitive closure*, or reduce it to its *transitive reduction* (by removing all transitive edges). This ensures that graph equality is defined modulo transitivity. Note that the condition $q \neq \varepsilon$ is necessary, as otherwise

$$a + b = a — \varepsilon + \varepsilon — b = (a — \varepsilon) — b = a — b,$$

which is clearly undesirable.

One can observe that if we drop the axioms of decomposition and closure, the remaining axioms are true in arithmetic, where $+$ and $—$ stand for addition and multiplication, respectively. Hence, undirected transitive graphs can be viewed as numbers equipped with two additional axioms.

One can easily check that all the axioms are satisfied at the semantic level of undirected graphs and switching networks, thus the algebra is *sound*. *Completeness* follows from the existence of the canonical form introduced in the next section. *Minimality* of the set of axioms can be proved by enumerating the fixed-size models of the algebra with the help of the ALG tool [3]: it turns out that removing any of the axioms leads to a different number of non-isomorphic models of a particular size, implying that all the axioms are necessary. Hence the following result holds.

Proposition 1 (Soundness, Minimality and Completeness)**.** *The set of axioms of the algebra of transitive undirected graphs is sound, minimal and complete.*

Several useful theorems can be derived from the axioms through equational reasoning.

Proposition 2. *The following theorems hold:*

- $—$ *is associative:* $(p—q)—r = p—(q—r)$
- ε *is an identity of* $+$*:* $p+\varepsilon = p$
- $+$ *is idempotent:* $p+p = p$
- *absorption:* $p+p—q = p—q$

Proof. We prove the first theorem and leave the others as an exercise for the reader (see the Appendix):

$$\begin{array}{rcl} (p—q)—r & = & \text{(decomposition)} \\ p—q+p—r+q—r & = & \text{(commutativity of } +, —) \\ q—r+q—p+r—p & = & \text{(decomposition)} \\ (q—r)—p & = & \text{(commutativity of } —) \\ p—(q—r). & & \end{array}$$

□

Remark 3. Since ε is a left and right identity of $+$ and $—$, there can be no other identities for these operations. Interestingly, unlike many other algebras, the two main operations have the same identity.

Now, equipped with the necessary mathematical toolkit, we can come back to the question of deciding equivalence of two given expressions L and R. One possible way would be to compute the transitive closures of the graphs specified by the expressions and directly compare them. However, this procedure has quadratic time and space complexity $O(|L|^2+|R|^2)$, where $|e|$ denotes the size of an expression e in terms of the count of singleton graphs that appear in it. The quadratic cost arises because a graph specified by an expression e may have $O(|e|^2)$ edges, e.g., expression $(a+b+c)—(d+e+f)$ has size $3+3=6$, but describes the graph with $3\times 3 = 9$ edges. Below we propose a more efficient solution.

Proposition 4. *Equality of expressions L and R can be decided in linear time and space* $O(|L|+|R|)$.

Proof. First, all occurrences of ε in both expressions are eliminated by the identity rules (unless one of the expressions is equal to ε, in which case we can immediately decide the equality). This preprocessing has linear complexity $O(|L|+|R|)$. Then we use the following key observation:

$$\begin{array}{rcl} (p—q)—r & = & \text{(associativity, commutativity of } —) \\ (p—r)—q & = & \text{(closure; here } r \neq \varepsilon) \\ p—r+r—q & = & \text{(commutativity of } —) \\ p—r+q—r & = & \text{(distributivity)} \\ (p+q)—r. & & \end{array}$$

That is, the closure axiom allows us to replace any nested occurrences of connection — with overlay +. By successively applying this rule to all the nested —'s we can eventually reduce each expression to an overlay of the following terms

$$(a_1+a_2+...)—(b_1+b_2+...)—(c_1+c_2+...)—...$$

where a_i, b_i, c_i, etc. are singleton graphs. Each such term represents a connected component within the specified graph. We further transform each term into the following chain of singletons

$$a_1—a_2+a_2—a_3+...+a_n—b_1+b_1—b_2+...$$

using the closure and distributivity axioms (here n stands for the number of singletons a_i in the first clause). Note that after the transformations the size of the expression is at most doubled. At this point one can use, for example, the depth-first search (DFS) [10] to identify all the connected components of the graph represented by the resulting expression. As all the brackets have been removed, there is no quadratic blow-up of the graph size – the number of its vertices and arcs is bounded by the size of the original expression $|V_L|+|E_L| = O(|L|)$ (similarly for R). Therefore, two DFS procedures combined will take linear time and space $O(|L|+|R|)$. Comparison of the obtained sets of components is trivially linear. □

We can conclude that the algebra provides a very compact and efficient way of representing and manipulating graphs: by operating on algebraic expressions we implicitly operate on graphs of potentially quadratic size. This allows us to avoid the disadvantages associated with connectivity matrices. An alert reader may notice that this is similar to representing a Boolean function by Boolean formulas instead of truth tables: the former are much more efficient in practice than the latter [41].

Example 5. Let us prove that the graphs in Figures 3(c,d) are equivalent. The two expressions are:

$$\begin{aligned} G_1 &= (a—b+b—c)+(b—c+b—d) \\ G_2 &= (a—b+b—c)—(b—c+b—d) \end{aligned}$$

By opening the brackets and dropping the repeated term $b—c$ using the rule of idempotence we can simplify G_1 into:

$$a—b+b—c+b—d.$$

G_2 can be simplified by following the steps described in the proof of Proposition 4:

$$\begin{aligned} (a—b+b—c)—(b—c+b—d) &= \text{(nested —'s)} \\ (a+b+b+c)—(b+c+b+d) &= \text{(idemp., comm.)} \\ (a+b+c)—(b+c+d) &= \text{(chaining)} \\ a—b+b—c+c—b+b—c+c—d &= \text{(idemp., comm.)} \\ a—b+b—c+c—d. & \end{aligned}$$

Now we can use DFS to check that both expressions specify graphs with the same connected component

$\{a,b,c,d\}$. Alternatively, we can complete the equational proof as follows:

$$\begin{array}{rcc} G_1 & = & \\ a—b+b—c+b—d & = & \text{(distributivity)} \\ a—b+b—(c+d) & = & \text{(nested —)} \\ a—b+b—(c—d) & = & \text{(assoc., comm.)} \\ a—b+c—(b—d) & = & \text{(nested —)} \\ a—b+c—(b+d) & = & \text{(distrib., comm.)} \\ a—b+b—c+c—d & = & G_2. \end{array}$$

Performing all these algebraic manipulations by hand is a tedious and error-prone process. We are currently implementing a prototype domain-specific language (DSL) in Haskell [17] which automates algebraic specification, analysis and synthesis of switching networks.

3.2 Families of graphs

Let us come back to Fig. 1. Clearly, we can represent the first and the second networks by expressions $a—b$ and $a+b$, resp., where a and b correspond to the two nodes. But how can we represent the other networks?

An undirected graph can only describe a particular static arrangement of switches. To capture the ability of a switch to change its state, we need a way to represent *families of graphs*. The network shown in Fig. 1(c) can be considered a family of two graphs: depending on the value of the controlling signal x the resulting network connectivity is equivalent to either $a—b$ or $a+b$. To describe graph families we extend the algebra of undirected graphs with additional *condition* operations:

Condition: $[1]G \stackrel{\text{df}}{=} G$ and $[0]G \stackrel{\text{df}}{=} \varepsilon$

The unary *condition* operations can either preserve a graph (*true condition* $[1]$) or nullify it (*false condition* $[0]$). These operations are not particularly useful until one considers replacing a Boolean constant 0 or 1 with a Boolean variable or a predicate, say, x, resulting in an expression $[x]G$, whose value depends on the value of a *parameter* x. This subtle conceptual step (which is akin going from arithmetic to algebraic expressions) brings up a new algebra with interesting properties, capable of describing both the functionality and structure of a switching network.

The *algebra of switching networks*[2] is $\langle \mathscr{N}, +, —, [0], [1] \rangle$, where $\mathscr{N}$ is a set of switching networks. To define the equivalence relation we import the axioms from the algebra of undirected graphs and add the condition operations axioms:

- condition: $[1]p = p$ and $[0]p = \varepsilon$

One can prove the following theorems (see Appendix) by case analysis on the values of Boolean parameters x and y:

- conditional ε: $[x]\varepsilon = \varepsilon$

[2]Following [27], a more precise term is the *algebra of parameterised transitive undirected graphs,* however, we find it overly verbose.

- conditional overlay: $[x](p+q) = [x]p + [x]q$
- conditional connection: $[x](p \multimap q) = [x]p \multimap [x]q$
- AND-condition: $[x \wedge y]p = [x][y]p$
- OR-condition: $[x \vee y]p = [x]p + [y]p$

Finally, for the sake of convenience a ternary operation called a *switch* is introduced as a combination of the three basic operations:

$$p \overset{x}{\multimap} q \overset{\text{df}}{=} p + q + [x](p \multimap q)$$

As the name suggests, the operation corresponds to a switch connecting two networks p and q, which is ON when $x = 1$ and OFF when $x = 0$. We can consider $p \overset{x}{\multimap} q$ as a family of two graphs:

$$p \overset{x}{\multimap} q = \begin{cases} p + q + [0](p \multimap q) = p + q & \text{if } x = 0 \\ p + q + [1](p \multimap q) = p \multimap q & \text{if } x = 1 \end{cases}$$

The above holds due to the condition axioms and the absorption rule $p + p \multimap q = p \multimap q$ (see Prop. 2).

Example 6. The following simple algebraic expressions represent the switching networks shown in Fig. 1: $N_a = a \multimap b$, $N_b = a + b$, $N_c = a \overset{x}{\multimap} b$, $N_d = a \overset{x \wedge y}{\multimap} b$, and $N_e = a \overset{x \vee y}{\multimap} b$.

4 Analysis

Any algebraic expression representing a switching network can be rewritten in the *canonical form* as stated by the following proposition. Two expressions are equivalent if their canonical forms coincide. This also means that one can demonstrate the equivalence of two expressions by rewriting one of them into the other.

Proposition 7 (Canonical form)**.** *Any expression can be rewritten in the following canonical form*[3]*:*

$$\left(\sum_{v \in V} [f_v] v \right) + \left(\sum_{\substack{u,v \in V \\ u < v}} [f_{uv}](u \multimap v) \right),$$

where:

1. *V is a subset of singleton graphs that appear in the original expression;*
2. *for all $v \in V$, f_v are canonical forms of Boolean expressions and are distinct from 0;*
3. *for all $u, v \in V$, $u < v$, f_{uv} are canonical forms of Boolean expressions such that $f_{uv} \Rightarrow f_u \wedge f_v$ (this requirement ensures that a switch cannot appear without its nodes); we assume that nodes are totally ordered by $<$ and $f_{uv} = f_{vu}$ for simplicity;*
4. *for all $u, v, w \in V$, $f_{uv} \wedge f_{vw} \Rightarrow f_{uw}$ (the transitivity requirement, i.e., if nodes u and v are connected and so are nodes v and w then nodes u and w must also be connected).*

[3] We assume that nodes are ordered by $<$.

Proof. We refer the reader to [27] where a very similar result is proved (in particular, see Prop. 5.1). Note a subtle difference that in this paper a node is always considered to be connected to itself by a self-loop, hence the use of $<$ instead of $\leq$ in the canonical form. □

The process of constructing the canonical form of an expression matches the process of Boolean *analysis* of the corresponding network, in particular, the obtained matrix (f_{uv}) is called the *switching matrix* in classic Boolean analysis [6][16]. Therefore, we claim that the algebra of switching networks allows one to perform analysis of a network's functionality in the same language that was used to describe its structure.

Before proceeding with an example of analysis, we add a new instrument into our mathematical toolkit.

4.1 Node contraction and partial equivalence

In many cases a relaxed notion of equivalence if useful; for example, to prove that the networks in Fig. 2 are equivalent it is necessary to remove auxiliary nodes from one of the networks [8]. This subsection describes a procedure, called *node contraction*, that performs such a removal and thereby establishes a relaxed notion of equivalence, called *partial equivalence*.

Consider an expression s and a node t that may or may not appear in the expression. *Node contraction* produces a new expression s' which is free from t but preserves the connectivity functions for all pairs of vertices $u \neq t$ and $v \neq t$. This is formally denoted as: $s' = s\backslash t$.

Proposition 8 (Node contraction)**.** *Let s be an expression with the following canonical representation:*

$$s = \left(\sum_{v \in V} [f_v] v \right) + \left(\sum_{\substack{u,v \in V \\ u<v}} [f_{uv}] (u — v) \right),$$

and t be a node not necessarily appearing in the expression. Then the node contraction $s\backslash t$ has the following canonical form:

$$s\backslash t = \left(\sum_{v \in V \setminus \{t\}} [f_v] v \right) + \left(\sum_{\substack{u,v \in V \setminus \{t\} \\ u<v}} [f_{uv}] (u — v) \right).$$

In other words, all the terms corresponding to node t are dropped in $s\backslash t$.

Proof. Let $u \neq t$ and $v \neq t$ be two nodes appearing in the expression s. Due to the transitivity requirement of the canonical form, the connectivity function f_{uv} captures all the paths, including those passing through node t. Since f_{uv} is preserved in the expression $s\backslash t$, we can conclude that the connectivity between nodes u and v is not affected by the node contraction. □

Remark 9. One can see that node contraction is a *confluent* transformation, i.e., the order of contractions does not matter:

$$s\backslash t_1\backslash t_2 = s\backslash t_2\backslash t_1.$$

This allows one to generalise node contraction to sets of nodes. For example, $s\backslash\{t_1, t_2\} = s\backslash t_1\backslash t_2$.

Node contraction can be computed from the canonical form, but the following properties often provide convenient shortcuts.

Proposition 10. *Let T be a non-empty set of nodes, and $t \neq \varepsilon$ be an expression containing only nodes from the set T. Let also p be an expression that is free from occurrences of nodes from the set T. Then the following equalities hold:*

1. $t\backslash T = \varepsilon$
2. $p\backslash T = p$
3. $(p+t)\backslash T = p$
4. $(p \multimap t)\backslash T = p \multimap p$
5. $(p \overset{x}{\multimap} t)\backslash T = p \overset{x}{\multimap} p$

Proof. (1) and (2) trivially follow from Prop. 8, since in the first case $V = T$, hence the whole expression is contracted, and in the second case $T \cap V = \emptyset$, hence p is not changed. (3) holds because t is disconnected from the rest of the network, therefore it cannot provide any additional connectivity. To prove (4) we rewrite its left part as follows:

$$\begin{array}{rcl} (p \multimap t)\backslash T & = & \text{(idempotence, commutativity)} \\ (p \multimap t + t \multimap p)\backslash T & = & \text{(closure, } t \neq \varepsilon\text{)} \\ ((p \multimap t) \multimap p)\backslash T & = & \text{(commutativity, associativity)} \\ ((p \multimap p) \multimap t)\backslash T & & \end{array}$$

Nodes in the subexpression $p \multimap p$ form a fully connected graph regardless of the auxiliary nodes T. Therefore, t can now be contracted by simply dropping it from the resulting expression as it will not contribute any additional connectivity in the canonical form of $p \multimap p$.

Finally, (5) can be proved by case analysis on the value of Boolean parameter x: when $x = 0$ the result follows from (3) and from equality $p \overset{0}{\multimap} p = p$; when $x = 1$ the result follows from (4) and from equality $p \overset{1}{\multimap} p = p \multimap p$. □

Given two networks p and q with sets of nodes V_p and V_q, resp., we say that p and q are *partially equivalent* if and only if:

$$p\backslash(V_p \backslash V_q) = q\backslash(V_q \backslash V_p).$$

In other words, networks are partially equivalent if they are equivalent after contracting all but common nodes $V_p \cap V_q$.

4.2 Examples

First, consider the switching networks shown in Fig. 2(a). The rightmost network is a transformation of the leftmost one aimed at increasing reliability of a single switch by replacing it with four identical switches connected in a bridge structure. The networks can be specified algebraically by expressions $a \overset{x}{\multimap} b$ and $(a+b) \overset{x}{\multimap} (t_1 + t_2)$, resp., where $T = \{t_1, t_2\}$ is the set of auxiliary nodes.

The networks are clearly not equivalent, because they have different sets of nodes. Our intention is to check that the connectivity function between nodes a and b is the same in both networks, which can be achieved using the node contraction transformation:

$$
\begin{array}{rcc}
((a+b)\overset{x}{—}(t_1+t_2))\backslash T & = & \text{(Prop. 10)} \\
(a+b)\overset{x}{—}(a+b) & = & \text{(distributivity)} \\
a\overset{x}{—}a+a\overset{x}{—}b+b\overset{x}{—}a+b\overset{x}{—}b & = & \text{(closure, commutativity)} \\
(a\overset{x}{—}a)\overset{x}{—}b+(b\overset{x}{—}b)\overset{x}{—}a & = & \text{(any node } v\in V \text{is connected to itself: } v\overset{x}{—}v=v\text{)} \\
a\overset{x}{—}b+b\overset{x}{—}a & = & \text{(idempotence, commutativity)} \\
a\overset{x}{—}b & &
\end{array}
$$

Therefore the networks are partially equivalent on the common set of nodes $\{a,b\}$.

Now consider the networks shown in Fig. 2(b), which represent the 'delta' Δ and the 'wye' Y in the well-known delta-wye transformation [34]. The rightmost network Y contains an auxiliary node, which will be denoted as t; the leftmost network Δ has no auxiliary nodes. Algebraic representations of the networks are given below:

$$
\begin{array}{rcc}
\Delta & = & a\overset{x\wedge y}{—}b+a\overset{x\wedge z}{—}c+b\overset{y\wedge z}{—}c \\
Y & = & a\overset{x}{—}t+b\overset{y}{—}t+c\overset{z}{—}t
\end{array}
$$

The canonical forms of the expressions are:

$$
\begin{array}{rcc}
\Delta & = & a+b+c+[x\wedge y](a—b)+[x\wedge z](a—c)+[y\wedge z](b—c) \\
Y & = & a+b+c+t+[x](a—t)+[y](b—t)+[z](c—t)+[x\wedge y](a—b)+[x\wedge z](a—c)+[y\wedge z](b—c)
\end{array}
$$

As one can now see $\Delta = Y\backslash t$, hence the networks are equivalent after contracting the auxiliary node t.

Note that the original expressions $\Delta = a\overset{x\wedge y}{—}b+a\overset{x\wedge z}{—}c+b\overset{y\wedge z}{—}c =$ and $Y = a\overset{x}{—}t+b\overset{y}{—}t+c\overset{z}{—}t$ perfectly capture the structure of the networks despite having the same functionality, and by using the algebraic transformations one can transform one of them into another for the purpose of optimising a non-functional criteria, such as, for example, the overall complexity of the network in terms of the number of switches, thereby performing a form of *synthesis*. See Section 5 for further discussions on this topic.

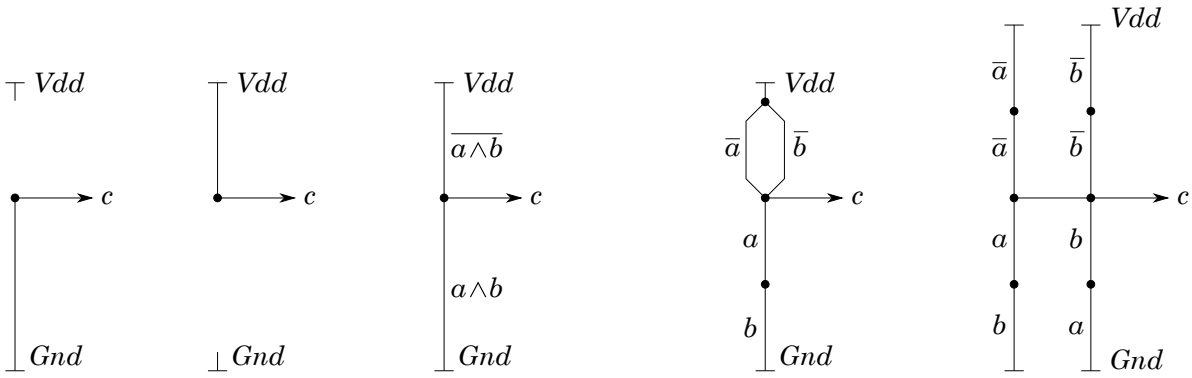
4.3 Verification of non-functional properties

In the previous subsection we have verified that the switching networks in Fig. 2(a) are partially equivalent. This, however, is a rather weak result. Indeed, one may demand a stronger verification outcome: can we prove that the transformation achieves the intended property of increased network reliability?

The answer is positive. First, we need to express the required property in algebraic terms. In this example, by 'increased reliability' we mean that the network maintains the original functionality even if one of the switches fails. Since the transformed network has left-to-right and top-to-bottom symmetry, it is sufficient to consider only the case when the top-left switch has failed:

$$N_{failure} = a\overset{y}{—}t_1+t_1\overset{x}{—}b+a\overset{x}{—}t_2+t_2\overset{x}{—}b.$$

The failed switch is modelled as a switch that is controlled by an unknown signal y, which may be a constant (0 if the switch has become permanently open, and 1 if the switch has become permanently closed) or a variable that is changing unpredictably and does not follow the proper control signal x (perhaps, due to a strong interference with other parts of the circuit).



(a) Logic 0 (b) Logic 1 (c) Composition (d) Standard NAND (e) Symmetric NAND

Figure 4: NAND gate synthesis stages

Our goal therefore is to show that regardless of the value of y network $N_{failure}$ is partially equivalent to the original network $N_{original} = a \xrightarrow{x} b$. We can do that in three steps. First, subexpression $a \xrightarrow{y} t_1 + t_1 \xrightarrow{x} b$ can be rewritten as $a \xrightarrow{y} t_1 + t_1 \xrightarrow{x} b + a \xrightarrow{x \wedge y} b$ using the closure axiom. Similarly, subexpression $a \xrightarrow{x} t_2 + t_2 \xrightarrow{x} b$ can be rewritten as $a \xrightarrow{x} t_2 + t_2 \xrightarrow{x} b + a \xrightarrow{x} b$. Now one can observe that $a \xrightarrow{x \wedge y} b + a \xrightarrow{x} b$ conveniently simplifies to $a \xrightarrow{x} b$ due to the Boolean absorption law $x \wedge y \vee x = x$. Therefore, after contracting nodes t_1 and t_2 the resulting network becomes equivalent to $N_{original}$ as required.

Having verified that the transformed network is capable of withstanding the failure of one switch, one may ask what happens in case of more than a single failure. The algebraic approach can provide the answer. Let us explicitly model all four possible failures:

$$N_{uncertain} = a \xrightarrow{y_1} t_1 + t_1 \xrightarrow{y_2} b + a \xrightarrow{y_3} t_2 + t_2 \xrightarrow{y_4} b.$$

We can now pose the question: what condition must be satisfied so that we have the equality $N_{uncertain} \backslash \{t_1, t_2\} = N_{original}$? By computing $N_{uncertain} \backslash \{t_1, t_2\}$ symbolically one can see that the sought condition is $y_1 \wedge y_2 \vee y_3 \wedge y_4 = x$. An interesting consequence is that the transformed network can in some situations withstand failure of three out of four switches! For example, if $y_1 = x$ (proper operation), $y_2 = 1$ (permanently closed), $y_3 = 0$ (permanently open) and y_4 is uncertain (interference) then the network's functionality is still maintained: $x \wedge 1 \vee 0 \wedge y_4 = x$.

To conclude, in this section we demonstrated that the algebra provides a unified mathematical language for analysis of both functional and non-functional properties (such as reliability) of switching networks.

5 Synthesis

In this section we show how to synthesise switching networks with required properties using the algebra.

5.1 Synthesis of a NAND gate in CMOS technology

We demonstrate the algebraic synthesis on a transistor-level implementation of a NAND gate that given two input signals a and b produces the output signal c connected to ground if condition $a \wedge b$ holds, and to Vdd otherwise.

Let nodes $\top$ and $\bot$ denote the Vdd and ground power lines, resp. We need to connect one of these nodes to the output as shown in Fig. 4(a,b). This can be abstractly expressed using the following system of equations:

$$\begin{cases} X = \top + c — \bot & \text{if } a \wedge b \\ X = \top — c + \bot & \text{if } \overline{a \wedge b} \end{cases}$$

That is, an unknown circuit X, which we would like to find, must connect c either to $\bot$ or to $\top$ depending on condition $a \wedge b$. Let us solve this system of equations.

A conditional statement '$A = B$ if F' can be algebraically expressed as $[F]A = [F]B$, hence we can rewrite the system in the following way:

$$\begin{cases} [a \wedge b]X = [a \wedge b](\top + c — \bot) \\ [\overline{a \wedge b}]X = [\overline{a \wedge b}](\top — c + \bot) \end{cases}$$

By congruence, two equations $A = B$ and $C = D$ imply the equation $A + C = B + D$, therefore the system of equations can be collapsed into a single equation:

$$[a \wedge b]X + [\overline{a \wedge b}]X = [a \wedge b](\top + c — \bot) + [\overline{a \wedge b}](\top — c + \bot).$$

The result can be simplified by collapsing the left hand side to X due to condition rules and Boolean algebra:

$$[a \wedge b]X + [\overline{a \wedge b}]X = [a \wedge b \vee \overline{a \wedge b}]X = [1]X = X$$

Simplification of the right hand side of the equation gives us the following result:

$$X = c\frac{a \wedge b}{\quad}\bot + c\frac{\overline{a \wedge b}}{\quad}\top.$$

We have found the unknown circuit X by simple algebraic manipulations, see the result in Fig. 4(c). Since in the CMOS technology each switch can be controlled only by one signal (positive literals correspond to n-type transistors, and negative ones correspond to p-type transistors), we have to refine the result by splitting the switches into simpler ones. This requires an addition of a new auxiliary node t:

$$X = (c\frac{a}{\quad}t + t\frac{b}{\quad}\bot + c\frac{\overline{a}}{\quad}\top + c\frac{\overline{b}}{\quad}\top)\backslash t.$$

See Fig. 4(d) for the final circuit, which matches the standard NAND gate implementation in the CMOS technology. One can automate the process of splitting complex switches into a network of n-type and p-type transistors in the form of a rewrite rule. By checking that each step in the rule maintains the network equivalence in accordance to the axioms of the algebra one can formally argue about the correctness of the rule in general (i.e., not only for this specific network), which would be impossible to achieve without the presented algebraic approach due to the aforementioned language barrier.

One can take the idea of using rewrite rules further and automate the synthesis of networks with various specific non-functional properties. For example, it has been shown that symmetric transistor networks exhibit much more predictable timing and power characteristics, which is especially valuable in the sub-threshold design [2][4]. One can express the symmetric transformation of transistor networks using the algebra and

apply it to our NAND gate example producing its symmetric 8-transistor version shown in Fig. 4(e). Many structural transformations that are routinely used by designers can be efficiently and conveniently automated as a collection of verified rewrite rules that can then be readily shared and reused by designers thereby increasing their productivity.

5.2 Parameterised circuits

In this subsection we will discuss algebraic synthesis of *parameterised circuits* [29], which in addition to conventional input/output interfaces (through which they exhibit the functionality to the environment) have a number of parametric inputs that are added to allow the selection of a particular implementation of the required functionality in run-time. Such circuits are also often referred to as *reconfigurable circuits* [40][28]. The parameters that select a particular implementation of the circuit may come from various sources: i) some of them can be statically pre-set during the product testing/binning stage on the basis of post-manufacturing information; ii) others can be changed in run-time by a power management controller or by a system utility at the software level; iii) it is also possible to use the parameters for the maintenance purposes, e.g., to reconfigure the system taking into account the information about faults or transistor ageing effects, thus allowing for a graceful system degradation.

A trivial way of synthesising such circuits is to directly include all the required implementations as separate blocks, connect their inputs via wire forks, and then choose the outputs of the currently selected implementation by a multiplexer. This trivial solution is easy to implement and has the benefit of reusing existing implementations. However, it is often the case that the implementations are similar to each other, which can be exploited by merging certain parts of different implementations in order to reduce the overall area and power consumption overheads.

Two examples of parameterised transistor networks are shown in Fig. 5. In particular, Fig. 5(b,c,d) show standard static CMOS implementations of an AND gate, a C-element [2] and an OR gate, while Fig. 5(a,e) show optimised parameterised networks implementing parameterised C/AND and C/OR elements [29]. The latter networks have the additional input p which can activate a particular network functionality. As one can see, the parameterised networks reuse common parts of the constituent functionalities (for example, the output inverter is reused).

One can synthesise these optimised switching networks using the algebraic approach. The key idea is to represent the required functionality as an overlay of simpler blocks. If algebraic expressions AND, OR, and C stand for the switching networks implementing AND gate, OR gate and C-element, resp., then the following expressions will represent the sought parameterised elements controlled by parameter p:

$$\begin{aligned} C/AND &= [\overline{p}]AND + [p]C \\ C/OR &= [\overline{p}]OR + [p]C \end{aligned}$$

By using the rules of the algebra one can factor out common terms occurring in these parameterised expressions thereby performing optimisation of the resulting switching networks. This is particularly relevant for new transistor technologies, such as double-gate transistors, that support fine-grain reconfiguration via *polarity control* [12].

Generalising the above approach one can represent parameterised networks controlled by more than one parameter. An example of such a network can be found in [40] which describes a *configurable transistor*

of the PAnDA architecture, which connects seven transistors $M_0 - M_6$ of varying physical characteristics in parallel so that they can be independently activated by reconfiguration bits $b_0 - b_6$ (see Fig. 6 of [40]) thereby achieving fine-grain adaptability and resilience to the effects of intrinsic variability, as well as fault tolerance. This transistor arrangement can be algebraically captured by the expression

$$\sum_{0 \le k \le 6} [b_k] M_k.$$

The ideas discussed in this subsection are further elaborated below in more high-level settings.

5.3 Structural composition

In this subsection we show how to structurally compose switching networks. We will use the notation supported by our tool for algebraic circuit specification, analysis and synthesis.

Circuits are first-class values in our Haskell-based DSL, therefore we can easily create functions that manipulate circuits. For example, a function that given three signals a, b, and c constructs a CMOS NAND gate can be defined[4] as follows:

```
NAND a b c = [  a ∧ b ](⊤ + c——⊥)
           + [¬(a ∧ b)](⊤——c + ⊥)
```

To demonstrate structural composition, let us also define the inverter gate function:

```
INV a b = [a](⊤ + b——⊥) + [¬a](⊤——b + ⊥)
```

Now one can create a circuit implementing the AND gate functionality in CMOS by calling the above functions and overlaying the results:

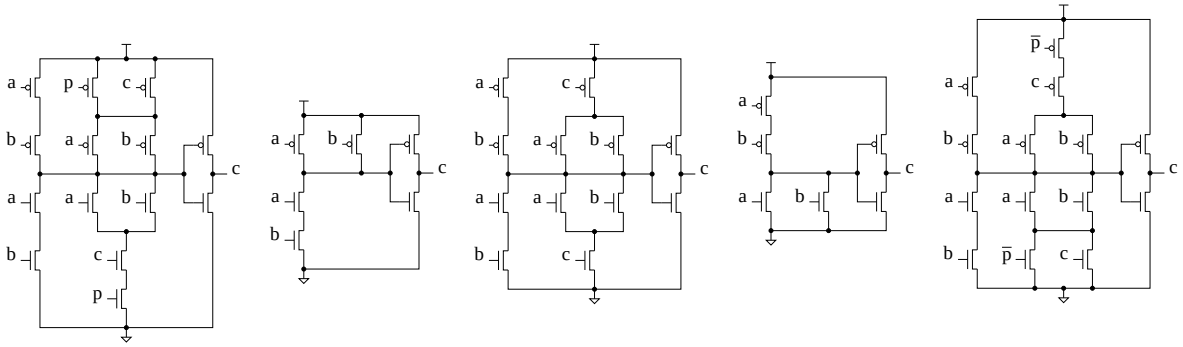
```
AND_gate = NAND a b t + INV t c
```

This particular implementation uses signals a and b as the inputs of the AND gate, signal c as its output, and signal t as an intermediate result. We can also compose functions (instead of circuits) to obtain a more reusable implementation:

```
AND a b c t = NAND a b t + INV t c
```

Now we can call the function `AND` whenever we need to instantiate a CMOS AND gate with a particular combination of input and output signals. Note that the implementation details are hidden and one can operate on high-level components like `AND_gate` without dealing with individual switches directly. This is similar to Verilog and VHDL hardware description languages, but the difference is that at any point of the design process we know exactly not only the current structure of the circuit, but also its functionality, because they are inseparable. In Verilog or VHDL, once a circuit is designed it is necessary to perform its analysis in order

[4] The actual syntax is slightly different due to technical reasons (e.g., unary operations are not well supported in Haskell). We decided to stick to the mathematical notation used in this paper not to confuse the reader.



(a) C/AND-element (b) AND gate (c) C-element (d) OR gate (e) C/OR-element

Figure 5: Transistor-level implementations of specialised and parametrised gates

to be sure that the circuit functionality satisfies the specification. The algebraic approach to hardware design eliminates this step completely: one can start with an abstract specification of the functionality and then keep refining the implementation by following the rules of the algebra until the result satisfies all the structural constraints of a particular technology, as shown in Figure 4.

5.4 Functional composition

One particularly unique and useful feature of the algebra is *functional compositionality*. Consider two systems A and B, which may potentially be very complex and contain billions of switches. As elaborated in the previous subsection, we already know that their *structural composition* C can be expressed simply as overlay $C = A + B$: if the systems have common interface signals, they will be 'glued' together as shown in Fig. 3(c). While this type of composition is usually handled well by other methods, our approach also allows one to perform the *functional composition* of systems, which is a lot more difficult to handle efficiently.

For example, if one wants to describe a system C that delivers functionality A under a certain condition x, and functionality B under the opposite condition $\overline{x}$ then the solution can be algebraically expressed as $C = [x]A + [\overline{x}]B$. This has already been demonstrated in synthesis of a NAND gate and parameterised networks. Importantly, if the two functionalities are similar, one can simplify the resulting expression by factoring out common terms. For instance, $[x]A + [\overline{x}]A = [x \vee \overline{x}]A = A$, i.e. if the functionalities coincide then one can algebraically prove it and remove the condition x, as well as the redundant copy of A, altogether. Typically, if A and B are functionalities that a system delivers in two operating modes then they have a lot in common. It is crucial to detect such similarities in order to design efficient multimodal systems; to achieve that we can adopt existing methods for finding similarities in graphs and families of graphs, e.g., [26].

Finally, the axiomatic definition of switching network equivalence allows a designer to substitute a part of an expression A with an equivalent part B without any additional checks of the resulting system's global properties. As long as the local equivalence $A = B$ holds, it is guaranteed that the rest of the system is not affected by the substitution. Algebraic compositionality opens the way for new methodologies and techniques for system optimisation in various aspects, such as latency, power consumption, reliability, etc. by performing local provable transformations of an expression representing an entire system.

6 Conclusions

This paper discusses the glorious past of the theory of switching networks, whose roots can be traced back to the beginning of the 20^{th} century. Today, the theory forms the backbone of any computation system, however, it is hidden by multiple layers of abstraction and largely forgotten; little or no development is going on at present for it is believed that all the useful facts about switching networks have already been discovered.

This work is an attempt to revive the old theory by introducing a new mathematical construct – an algebra of switching networks – that unifies the notions of function and structure of a computation system that were always separated. The algebra is specified axiomatically, and the soundness, minimality and completeness of the resulting sets of axioms are proved. The transformations required for algebraic analysis and synthesis of switching networks are developed and demonstrated on a set of examples.

The future work includes the development of a scalable software support tool capable of handling switching networks consisting of billions of switches, as well as the application of the presented techniques in other areas, where modelling conditional connectivity is important, for example, in the analysis of protein-protein interactions in large-scale biological networks. The most promising direction for automation is via *automated theorem proving* software that is capable of proving mathematical statements within a given theory. Although theorem proving is undecidable in the general case, the axioms of the proposed theory of switching networks allow efficient equality checking as has been demonstrated in Section 4. Efficient verification of more complex non-functional properties, e.g., related to reliability or to energy constraints, requires further algorithmic research.

Acknowledgement

The author would like to thank Marc Riedel for the introduction to the world of Shannon's switching networks, Victor Khomenko for the inspiration to creating new algebras, and Alex Yakovlev for his continued contribution to the theory of conditional graphs. Special thanks go to the participants of the *Designing with Uncertainty – Opportunities & Challenges 2014* workshop for their valuable comments on the earlier version of this work. This work was partially supported by EPSRC research grants UNCOVER (EP/K001698/1) and PRIME (EP/K034448/1).

References

[1] A. Bardsley and D. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, 2000.

[2] H. K O Berge, A. Hasanbegovic, and S. Aunet. Muller C-elements based on minority-3 functions for ultra low voltage supplies. In *IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 195–200, April 2011.

[3] A. Bizjak and A. Bauer. ALG *User Manual*, Faculty of Mathematics and Physics, University of Ljubljana, 2011.

[4] J. E. Bjerkedok. Subthreshold Real-Time Counter. Master's thesis, Norwegian University of Science and Technology, Norway, 2013.

[5] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[6] R. E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on CAD of Integrated Circuits*, 6:618–633, 1987.

[7] R. E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-aided Design*, 6:634–649, 1987.

[8] W. Chen. Boolean matrices and switching nets. *Mathematics Magazine*, pages 1–8, January 1966.

[9] M. D. Ciletti. *Advanced Digital Design with the VERILOG HDL*. Prentice Hall PTR, 2002. ISBN: 0130891614.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[11] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.

[12] M. De Marchi, D. Sacchetto, S. Frache, J. Zhang, P. Gaillardon, Y. Leblebici, and G. De Micheli. Polarity control in double-gate, gate-all-around vertically stacked silicon nanowire fets. In *Electron Devices Meeting (IEDM), 2012 IEEE International*, pages 8–4. IEEE, 2012.

[13] J. Esparza. Decidability and complexity of Petri net problems – an introduction. *Lectures on Petri Nets I: Basic Models*, pages 374–428, 1998.

[14] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[15] T. Hoare, S. van Staden, B. Möller, G. Struth, J. Villard, H. Zhu, and P. O'Hearn. Developments in concurrent kleene algebra. pages 1–18, 2014.

[16] F. E. Hohn and L. R. Schissler. Boolean Matrices and Combinational Circuit Design. *Bell Systems Technical Journal*, (34):177–202, 1955.

[17] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.

[18] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954.

[19] M.B. Josephs and J.T. Udding. An overview of D-I algebra. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, volume 1, pages 329–338, 1993.

[20] R. Lipsett, C. A. Ussery, and C. F. Schaefer. *VHDL, Hardware Description and Design*. Kluwer Academic Publishers, 1993. 9780792390305.

[21] T.-J. K. Liu, D. Markovic, V. Stojanovic, and E. Alon. MEMS Switches for Low-Power Logic. *IEEE Spectrum*, April 2012.

[22] T.J.K. Liu, J. Jeon, R. Nathanael, H. Kam, V. Pott, and E. Alon. Prospects for MEM logic switch technology. In *Electron Devices Meeting (IEDM), 2010 IEEE International*, pages 18–3. IEEE, 2010.

[23] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[24] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[25] A. Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.

[26] A. Mokhov, A. Alekseyev, and A. Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *IET Computers and Digital Techniques*, 5(6):427–439, 2011.

[27] A. Mokhov and V. Khomenko. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing*, 13(4s), 2014.

[28] A. Mokhov, M. Rykunov, D. Sokolov, and A. Yakovlev. Design of processors with reconfigurable microarchitecture. *Journal of Low Power Electronics and Applications*, 4(1):26–43, 2014.

[29] A. Mokhov, D. Sokolov, and A. Yakovlev. Adapting Asynchronous Circuits to Operating Conditions by Logic Parameterisation. *Int'l Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–24, 2012.

[30] A. Mokhov and A. Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[31] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[32] M. Renaudin, G. F. Bouesse, Ph. Proust, J. P. Tual, L. Sourgen, and F. Germain. High security smartcards. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 228–233, 2004.

[33] J. E. Savage. *Models of Computation – Exploring the Power of Computing*. Addison-Wesley, 1998.

[34] C. E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Trans. of the American Institute of Electr. Engineers*, 57:713–723, 1938.

[35] C. E. Shannon. The Synthesis of Two-Terminal Switching Circuits. *Bell Systems Technical Journal*, 28:59–98, 1948.

[36] M. Spencer, F. Chen, C. C. Wang, R. Nathanael, H. Fariborzi, A. Gupta, H. Kam, V. Pott, J. Jeon, T.-J. K. Liu, D. Markovic, E. Alon, and V. Stojanovic. Demonstration of Integrated Micro-Electro-Mechanical Relay Circuits for VLSI Applications. *J. Solid-State Circuits*, 46(1):308–320, 2011.

[37] S.J. Tans, A.R.M. Verschueren, and C. Dekker. Room-temperature transistor based on a single carbon nanotube. *Nature*, 393(6680):49–52, 1998.

[38] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, 1991.

[39] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Causal Nets: A Modeling Language Tailored towards Process Discovery. In *International Conference on Concurrency Theory (CONCUR)*, pages 28–42, 2011.

[40] J. A. Walker, M. A. Trefzer, S. J. Bale, and A. M. Tyrrell. PAnDA: A Reconfigurable Architecture that Adapts to Physical Substrate Variations. *IEEE Transactions on Computers*, 62(8):1584–1596, 2013.

[41] Ingo Wegener. *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universitat, 1987.

[42] F. Xia, A. Mokhov, Y. Zhou, Y. Chen, I. Mitrani, D. Shang, D. Sokolov, and A. Yakovlev. Towards power-elastic systems through concurrency management. *Computers & Digital Techniques, IET*, 6(1):33–42, 2012.

[43] A. Yakovlev. Energy-modulated computing. In *Design Automation and Test in Europe (DATE) conference*, pages 1340–1345, 2011.

Appendix

Proof of the equalities not involving conditions from Subsection 3.1:

- ε is an identity of $+$: $p+\varepsilon = p$
- $+$ is idempotent: $p+p = p$
- absorption: $p+p—q = p—q$

Proof. First we prove an auxiliary equality, called *reduced decomposition* or *r-decomposition*: $p = p+p+\varepsilon$.

$$\begin{array}{rcl} p & = & (—\text{-identity}) \\ p—\varepsilon—\varepsilon & = & (\text{decomposition}) \\ p—\varepsilon+p—\varepsilon+\varepsilon—\varepsilon & = & (—\text{-identity}) \\ p+p+\varepsilon & & \end{array}$$

Now the equality $p+\varepsilon = p$ can be proved as follows:

$$\begin{array}{rcl} p & = & (\text{r-decomposition}) \\ p+p+\varepsilon & = & (\text{r-decomposition}) \\ p+p+(\varepsilon+\varepsilon+\varepsilon) & = & (+\text{-commutativity}) \\ (p+\varepsilon)+(p+\varepsilon)+\varepsilon & = & (\text{r-decomposition}) \\ p+\varepsilon & & \end{array}$$

The idempotence of + can be proved as follows:

$$\begin{array}{rcl} p & = & (\text{r-decomposition}) \\ p+p+\varepsilon & = & (+\text{-identity}) \\ p+p & & \end{array}$$

Absorption is proved as follows:

$$\begin{array}{rcl} p+(p—q) & = & (—\text{-identity}) \\ (p—\varepsilon)+(p—q) & = & (\text{distributivity}) \\ p—(\varepsilon+q) & = & (+\text{-identity}) \\ p—q & & \end{array}$$

□

Proof of the equalities involving conditions from Subsection 3.2:

- Conditional ε: $[x]\varepsilon = \varepsilon$
- Conditional overlay: $[x](p+q) = [x]p+[x]q$
- Conditional sequence: $[x](p—q) = [x]p—[x]q$
- AND-condition: $[x \wedge y]p = [x][y]p$

- OR-condition: $[x \vee y]p = [x]p + [y]p$

Proof. First, suppose the value of x is 0 (*). Then:

Conditional ε:

$$\begin{array}{rcl} [x]\varepsilon & = & (*) \\ [0]\varepsilon & = & (\text{false condition}) \\ \varepsilon & & \end{array}$$

Conditional overlay:

$$\begin{array}{rcl} [x](p+q) & = & (*) \\ [0](p+q) & = & (\text{false condition}) \\ \varepsilon & = & (+\text{-identity}) \\ \varepsilon+\varepsilon & = & (\text{false condition}) \\ [0]p+[0]q & = & (*) \\ [x]p+[x]q & & \end{array}$$

Conditional sequence:

$$\begin{array}{rcl} [x](p \mathbin{—} q) & = & (*) \\ [0](p \mathbin{—} q) & = & (\text{false condition}) \\ \varepsilon & = & (\mathbin{—}\text{-identity}) \\ \varepsilon \mathbin{—} \varepsilon & = & (\text{false condition}) \\ [0]p \mathbin{—} [0]q & = & (*) \\ [x]p \mathbin{—} [x]q & & \end{array}$$

AND-condition:

$$\begin{array}{rcl} [x \wedge y]p & = & (*) \\ [0 \wedge y]p & = & (\text{Boolean algebra}) \\ [0]p & = & (\text{false condition}) \\ \varepsilon & = & (\text{false condition}) \\ [0][y]p & = & (*) \\ [x][y]p & & \end{array}$$

OR-condition:

$$\begin{array}{rcl} [x \vee y]p & = & (*) \\ [0 \vee y]p & = & (\text{Boolean algebra}) \\ [y]p & = & (\text{false condition}) \\ \varepsilon+[y]p & = & (+\text{-identity}) \\ [0]p+[y]p & = & (*) \\ [x]p+[y]p & & \end{array}$$

Now, suppose the value of x is 1 (**). Then:

Conditional ε:

$$\begin{array}{rcl} [x]\varepsilon & = & (**) \\ [1]\varepsilon & = & (\text{true condition}) \\ \varepsilon & & \end{array}$$

Conditional overlay:

$$\begin{array}{rcl} [x](p+q) & = & (**) \\ [1](p+q) & = & (\text{true condition}) \\ p+q & = & (\text{true condition}) \\ [1]p+[1]q & = & (**) \\ [x]p+[x]q & & \end{array}$$

Conditional sequence:

$$\begin{array}{rcl} [x](p—q) & = & (**) \\ [1](p—q) & = & (\text{true condition}) \\ p—q & = & (\text{true condition}) \\ [1]p—[1]q & = & (**) \\ [x]p—[x]q & & \end{array}$$

AND-condition:

$$\begin{array}{rcl} [x \wedge y]p & = & (**) \\ [1 \wedge y]p & = & (\text{Boolean algebra}) \\ [y]p & = & (\text{true condition}) \\ [1][y]p & = & (**) \\ [x][y]p & & \end{array}$$

OR-condition:

$$\begin{array}{rcl} [x \vee y]p & = & (**) \\ [1 \vee y]p & = & (\text{Boolean algebra}) \\ [1]p & & \end{array}$$

The value of y under the current assignment of variables is either 0 or 1, so we consider the two possible cases:

if the value of y is 0 (#) then

$$\begin{array}{rcl} [1]p & = & (+\text{-identity}) \\ [1]p+\varepsilon & = & (\text{false condition}) \\ [1]p+[0]p & = & (**) \\ [x]p+[0]p & = & (\#) \\ [x]p+[y]p & & \end{array}$$

if the value of y is 1 (##) then

$$\begin{array}{rcl} [1]p & = & (+\text{-idempotence}) \\ [1]p+[1]p & = & (**) \\ [x]p+[1]p & = & (\#\#) \\ [x]p+[y]p & & \end{array}$$

In all the possible cases the equalities hold. □