CONVENTIONAL INTERPRETATIONS OF BOUNDARY LOGIC TOOLS

William Bricken
April 2002

The concepts of conventional logic can all be represented in the BL formalism. In so doing, conventional logic becomes simpler and more efficient. Similarly, the concepts of BL can be represented in conventional logic, but only by adding new tools and perspectives to conventional approaches.

Thus, BL improves and extends the power of conventional logic. Using BL concepts generally and implementing BL concepts in software data structures and algorithms and in semiconductor and other hardware designs (among other applications of BL) avoids the representational and computational complexities of conventional logic approaches. It is, however, possible to replicate the transformational rules and mechanisms of BL using a vocabulary of conventional logic. This might have the effect of creating new ideas for conventional logic that appear to be novel and unique, when they would be, in fact, derivative of BL innovations.

To make the relationship between BL and conventional logic clear, this document includes many comparisons of techniques in representation, in the form of theorems, in proofs, and in hardware architectures.

CONTENTS

INTRODUCTION
- Boolean Logic
  - NOR Basis
  - Huntington's Axioms

VOID-BASED BOUNDARY LOGIC
- Boundaries
- Parens
- Spaces
- Innovation
- Syntactic and Semantic Space
- Comparative Representation
  - The One-bit Full Adder

BOUNDARY ALGEBRA THEOREMS
- Axiomatic Basis
  - Occlusion
  - Involution
  - Simple Pervasion
- Proof
- Deep Pervasion
  - Broad Pervasion
  - Pervasion
  - Accumulative Pervasion
  - Broad Accumulative Pervasion
- Simple Theorems
  - Dominion
  - Idempotency
  - Broad Idempotency
  - Deep Occlusion
- Absorption Theorem
  - Simple Absorption
  - Deep Simple Absorption
  - Broad Absorption
  - Absorption
- Comparative Proofs
  - Idempotency
  - Occlusion
  - Two-Deep Pervasion

COMPARATIVE HARDWARE TECHNOLOGIES
- Conventional Architectures
  - Two-level Programmable Logic Arrays
  - NOR Arrays
  - Folded NORs
  - Look-up Tables
  - Cross-point Switches
- The Circuit Configuration Array

INTRODUCTION

The structures and transformations of Boundary Logic are expressed in terms of containers, containment, void-equivalence, and transparency. In textual form, containers can be expressed by delimiting tokens such as parenthesis, brackets, braces, and the like.

The BL representations, tools, techniques, and transformations can be expressed in other formal languages, particularly those of conventional logic. For example, a bounded space containing two items, (a b), could be read for logic as *NOR[a,b].* The delimiting parenthesis tokens, called *parens*, can be read equally well as spatial forms and as logic forms.

Many of the transformations of BL derive from the unique and novel concepts and applications of parens forms as configurations of containers. Most of the BL transformational rules do not exist in conventional logical approaches, however it would be possible to extend conventional logic to include these rules. One primary example is the BL use of the transparency of parens, stated as: if something is outside a parens nest, it is optionally inside, regardless of depth of nesting. In BL this rule might be recorded as

a {b a} = a {b}

where the curly braces stand in place of any arbitrary intervening containers or content. An instance of this rule is:

a (b (c a)) = a (b (c))

The transparency feature could be incorporated into a conventional logic system, but with some degree of modification of how logic usually works. For instance, the above example might be read as:

a OR (NOT (b OR (NOT (c OR a)))) = a OR (NOT (b OR (NOT c)))

Conventional logic has no rules which extend across many logical connectives. As well, there is no notion of "arbitrary intervening content". Thus the innovative transparency concept of BL would be awkward to express solely in logic notation. It could be phrased, however, rather awkwardly, as

"When a logical form is converted into only NOR connectives, and those NOR connectives are permitted to contain any number of arguments, then structures which are on the outside of this representation can be inserted or removed from the inside any arbitrary depth, provided that commutativity and associativity rules are freely and dynamically applied in order to construct structures which can be identified as identical."

The following discussion and examples show the relationship between conventional logic, particularly the *NOR* connective, and BL by presenting the same examples in both languages.

It is to be noted that logical forms have dual representations. A *logical dual* is a pair of forms which express the same logical intention but with certain values and operations exchanged. The two bases

{NOR, FALSE} and {NAND, TRUE}

are dual. Thus, a form expressed in terms of one pair can be read equivalently in terms of the other pair. For example:

(a NOR FALSE) = (a NAND TRUE)

Boolean Logic

The standard algebraic definition of Boolean logic contains redundancies in the function set. In particular, only one binary function is necessary to provide a complete functional basis for logic; all other binary functions can be defined in terms of this one. The identification of NOR as a sufficient basis is due to [Peirce]. In the following, the notation of Boolean algebra is used to express conventional logical forms. A table of notational correspondence follows:

Logic	*Boolean Algebra*	*Boundary Logic*
TRUE	1	( )
FALSE	0	<void>
NOT a	a'	(a)
a OR b	a+b	a b
a AND b	a*b	((a)(b))
a NOR b	(a+b)'	(a b)
a NAND b	(a*b)'	(a)(b)

NOR Basis

To derive the NOR basis of primary logic, first consider a common basis set such as that of Boolean algebra: {0,1,+,',0}. This set can be reduced to the smaller set {0,',+} by calling on the following definitions to eliminate * and 1:

a*b = (a'+b')'
1 = 0'

The reduced function set can be further simplified by defining each function in terms of binary NOR:

$$a' = NOR[a,a]$$

$$a+b = NOR[a,b]' = NOR[\, NOR[a,b], NOR[a,b] \,]$$

$$0 = NOR[\, NOR[a,a], NOR[a',a'] \,]$$

$$= NOR[\, NOR[a,a], NOR[\, NOR[a,a], NOR[a,a] \,] \,]$$

These definitions are due to [Sheffer]. They require two reduction axioms in support,

Involution (A')' = A

Idempotency A + A = A

Notice that as operators are eliminated to form the Sheffer basis set of {NOR}, the representation of the operations grows. This makes Sheffer's approach unusable.

In order to contain this representational explosion, we now generalize the binary NOR function to apply to any number of arguments by defining *variary* NOR for any arbitrary finite n:

Arity	***Arguments***	***Boolean Form***	***N-ary Form***
0ary NOR:	NOR[ ]	= **0**'	= (0+0+...+0)'
unary NOR:	NOR[a]	= a'	= (a+0+...+0)'
binary NOR:	NOR[a,b]	= (a+b)'	= (a+b+...+0)'
n-ary NOR:	NOR[a,b,...n]	= (a+b+...+n)'	= (a+b+...+n)'

variary NOR: { 0ary-NOR, unary-NOR,..., n-ary-NOR }

NOR with no variables is TRUE, or 1, since the disjunction of no arguments is FALSE. In functional notations, a function with no arguments defines a ground value. NOR of one argument is NOT, since disjunction of one argument is simply that argument.

Variary operators necessarily incorporate associativity into the notation, since variarity undermines the mechanism which distinguishes the argument pairing required by binary operators. The generalized NOR could be considered to be a set-function with one argument which is a set containing any finite number of Boolean expressions.

We can now more efficiently express the Boolean operators in the generalized NOR notation:

```
1   =  NOR[   ]
a'  =  NOR[ a ]
0   =  NOR[ NOR[   ]]
a+b =  NOR[ NOR[a,b]]
a*b =  NOR[ NOR[a], NOR[b]]
```

Variary NOR could also be expressed as binary NOR by the following recursive definition, which unfortunately generates a representational explosion similar to that of the Sheffer stroke:

```
NOR[   ]          =  NOR[0,0]
NOR[ a ]          =  NOR[a,a]
NOR[a,b]          =  NOR[a,b]
NOR[a,b,c]        =  NOR[ NOR[ NOR[a,b], NOR[a,b]], c]
NOR[a,b,...,n]    =  NOR[ NOR[ NOR[a,...n-1], NOR[a,...n-1]], n]
```

Finally we add one other generalization to NOR. Already variary NOR has built-in associativity. We build in commutativity by not distinguishing the ordering of arguments. This is notated by not including a comma to separate arguments.

```
1           =  NOR[   ]
a'          =  NOR[ a ]
0           =  NOR[ NOR[   ]]
a+b+...+z   =  NOR[ NOR[a b...z]]
a*b*...*z   =  NOR[ NOR[a] NOR[b]...NOR[z]]
```

Although using the absence of an argument to define a ground value is common in functional notations, it is not a part of conventional logic, since logical connectives are not usually interpreted as functions.

Huntington's Axioms

Huntington's axioms define the properties, or invariants, of Boolean algebra. They are:

	+ operation	**** operation***
Commutativity	a+b = b+a	a*b = b*a
Identity	a+**0** = a	a***1** = a
Complement	a+a' = **1**	a*a' = **0**
Distribution	a+(b*c) = (a+b)*(a+c)	a*(b+c) = (a*b)+(a*c)

Huntington's axioms, expressed in the generalized NOR notation, become:

Commutativity built-in

Identity

```
NOR[ NOR[a NOR[ NOR[ ]]]] = a
NOR[ NOR[a] NOR[ NOR[ ]]] = a
```

Complement

```
NOR[ NOR[a NOR[a]]]       = NOR[ ]
NOR[ NOR[a] NOR[ NOR[a]]] = NOR[ NOR[ ]]
```

Distribution

```
NOR[ NOR[a  NOR[ NOR[b] NOR[c]]]]  = NOR[ NOR[ NOR[ NOR[a b]]]  NOR[ NOR[ NOR[b c]]]]
NOR[ NOR[a]  NOR[ NOR[ NOR[b c]]]] = NOR[ NOR[ NOR[ NOR[a] NOR[b]] NOR[ NOR[a] NOR[c]]]]
```

The next section introduces the BL perspective and notation that simplifies the variary NOR approach.

VOID-BASED BOUNDARY LOGIC

A *boundary algebra* uses boundaries, or containers, to represent functions and constants. Containment is not uncommon in mathematical notation: the square-root sign contains its arguments, matrix brackets contain an array of arguments, set curly-braces contain the members of the set, and function notation uses brackets to contain the arguments of the function.

The difference in boundary notation is that the properties of containment are used explicitly to simplify representation. In particular, *boundaries can contain nothing*, permitting both a syntactic approach to and a semantic interpretation of the lack of representation by referring to the empty container bounding no forms. As well, boundaries can contain an arbitrary number of elements, they are not limited to binary functions as is an infix or a binary notation.

Although varieties of parentheses and brackets are used in linear typography to represent boundaries, it is important to realize that boundaries themselves are of arbitrary dimension, enclosing a space of arbitrary dimension. Thus we can generalize the mathematical concept of container to be *a distinction drawn in a space of any dimension*. As well, boundaries are a representation of a partial ordered lattice, this lattices, or directed acyclic graphs can equally well express the relationship of containment.

Boundaries

We have established that it is possible to express the functions and constants of conventional logic in single-function basis using variary NOR. We have used the token string "NOR[...]" to represent this function. With only one function in the basis, there is no loss of clarity to let the boundary itself represent that function, rather than the labeled boundary. To distinguish the NOR representation from a pure boundary notation, we use parentheses rather than brackets. We write NOR[...] as (...). The parenthesis forms are called *parens*. Huntington's axioms in this notation are:

Commutativity built-in

Identity

((a (( )))) = a
((a) (( ))) = a

Complement

((a (a))) = ( )
((a) ((a))) = (( ))

Distribution

((a ((b)(c)))) = ((((a b))) (((b c))))
((a) (((b c)))) = ((((a)(b)) ((a)(c))))

Parens

Parenthesis strings are familiar when used for grouping and for determining operator precedence. In BL, the semantics of parens is that of logical variary NOR. Formally,

Well-formed parens are recursively defined to be members of one of the following sets:

1. <void> is an element.
2. Variable tokens are elements.
3. If A and B are parens forms, then so are (A) and AB.

The juxtaposition of A and B in Rule 3 above is spatial, perhaps better written as {A B}. The innovative step, however, is Rule 1, which permits the absence of any representation to be a valid member of a set. It might be noted that in set theory, the empty set is a member of every set, however it is not written as a form in sets with existent members, such as {x, y, z}.

It might be said the empty set Phi is *implicit* in the representation of non-empty sets, and this is indeed the standard explanation for its explicit absence. The innovation of BL is to permit an interpretation of absence. As an analogy, Phi is not implicit, it is validly *in* a set as <void>, a non-represented member.

It is important to realize that variables and parentheses are all that is needed to fully represent BL and thus propositional logic, Boolean algebra, and multilevel combinatorial circuits.

Parens expressed in linear typography also represent trees, with either variables or empty space as leaf nodes. When identical leaf nodes of a parens tree are united, the parens tree becomes a directed leaf-acyclic graph (DAG). *Leaf-acyclic* means that only the leaves of the graph have fanout. In linear parens notation, fanin is represented by the forms contained within a single boundary and fanout is represented by multiple occurrence of the label representing the fanout form.

Spaces

We now build commutativity and associativity into the notation itself by using the spatial aspect of boundary containment.

Since boundaries contain an arbitrary space, we can define a space to be *without structure*, i.e. without ordering. This explicitly eliminates commutative axioms. Commutativity is not implicit in the notation, rather it is not a relevant concept.

Since boundary notation is variary in that any space can contain an arbitrary number of forms, we can define the space to be without grouping. This explicitly eliminates associative axioms. Again, associativity is not implicit in the notation, rather it is not a relevant concept.

The boundary notation for the operators of Boolean algebra are then:

```
0       = (( ))
1       =  ( )
a'      =  (a)
a+b     =  ((a b))
a*b     =  ((a)(b))
```

The Involution theorem

```
(A')' = A
```

is

```
((A)) = A
```

in boundary notation. It provides a further simplification.

```
0    =  ((   ))  =  <void>
a+b  =  ((a b))  =  a b
```

Since we have permitted <void>, the absence of a representation, within the definition of boundary forms, it is entirely legitimate to interpret a Boolean ground as <void>.

Huntington's axioms expressed in the single function parens notation of BL with Involution now take their final form:

Commutativity	not defined
Identity	a = a a = a
Complement	a (a) = ( ) ((a) a) =
Distribution	a ((b)(c)) = ((a b)(a c)) ((a)(b c)) = ((a)(b)) ((a)(c))

These forms are the axiomatic basis presented in [Spencer-Brown].

Of the four original sets of axioms which define Boolean algebra, one is eliminated and one is reduced to a simple statement of the identity of identical representations.

In the following sections, complement and distribution are shown to be theorems rather than axioms. Distributive transformations are compound rules which can be simplified to one syntactically and semantically simpler axiomatic concept. Complement too is a theorem of this simpler axiom.

Innovation

From a boundary mathematics perspective, we begin with operators which are defined to be pre-associative and pre-commutative. If necessary, we then impose notational restrictions on those functions which are non-commutative and non-associative. For example, both IF and IF-THEN-ELSE (ITE) are non-commutative, making definite distinctions between their arguments:

IF[a,b] = NOR[a',b]'	(((a) b)) = (a) b
ITE[a,b,c] = IF[a,b]*IF[a',c]	(((a) b)(((a)) c)) = (((a) b)(a c))

In the case of IF, the non-commutativity of the arguments is maintained by the boundary separating the antecedent from the consequent. Commutativity is maintained between any arguments which share the same space. Even though *a* and *b* do not commute, *(a)* and *b* do. Similarly, a configuration of boundaries is sufficient to distinguish the arguments of ITE without losing either the commutativity of space or the semantics of the ternary operator.

In generating the above equations, we encounter two unusual notational conventions:

1. When the form A in the equation ((A)) = A is an empty space, then the representation of the constant 0 is reduced to no marking at all. The equation reads:

(( )) =

Although it may seem strange to write nothing on the right-hand-side of the equation, this equation is valid in a system which permits the empty space to have an interpretation. In a void-based calculus, the void is a legitimate form.

2. Similarly, should A be permitted to match more than one single form, then the operator + is also reduced to no marking. That is, + is represented by its arguments sharing the same space. In the shallowest space, outside of all containers, the boundary of + is implicit. In our example,

((a b c)) = a b c

These two *syntactic and semantic* conventions define the central innovation of BL. Since empty boundaries indicate an empty space, that space itself can be interpreted semantically as a constant. When the space contains forms, it can be interpreted semantically as an operator.

Syntactic and Semantic Space

The *syntactic and semantic use of space* is a defining characteristic of void-based boundary systems. The implications for logic include:

1. Logic has an elegant and efficient, although unfamiliar, notation using parenthesis strings and variable names. The container is the only explicit operator and constant.

2. Forms without variables consist solely of nested and juxtaposed parens boundaries. Networks with known inputs are also represented solely by well-formed parens.

3. Since boundaries *connect* their two sides, they also represent connections between network nodes. Thus, the edges of a network are explicitly carried in its parens form.

4. 0 and + are *pervasive* throughout a form. All subforms the same space are joined by +. Since the constant 0 is confounded with space itself, 0 implicitly exists wherever space exists in a form, which is everywhere.

5. Forms which evaluate to 0, i.e. to the non-represented void, can be freely added anywhere within a parens expression without effecting its value. This is a primary technique of *boundary deduction*.

6. The function + and the constant 0 are confounded in their mutual void representation. Generally, the distinction between form and function is confounded in parens notation. That is, a parens form can be read either as a data structure or as a program. Parens can be evaluated both by pattern matching and by function evaluation. Spaces and containers are both objects and operators.

7. Each boundary form represents an infinite class of logic network forms. For example, the form ( ) can be read as the constant 1, as the unary complement operator applied to an empty interior since 1 = 0', and as the binary IF operator which distinguishes the interior from the exterior, since 1 = IF[0,0]. Since + is also implicit in space, other readings of ( ) include 1+0, 0+1, (0+0)', 0+0+0'+0, IF[0,0+0+0], and of course, NOR[0], NOR[0,0] and NOR[0,0,0,0]. The topological arrangement of a DAG is explicitly carried in its parens representation.

8. Since space pervades a boundary form, transformations can be defined which apply explicitly to space, rendering boundaries transparent and depth of nesting irrelevant. This is how BL accomplishes multilevel logic synthesis.

Comparative Representation

We provide a comparative example of the representation of a common circuit, the one-bit full adder, in both conventional NOR gate form and the parens boundary form. Although the parens boundaries in the parens form can be directly read as NOR gates, their expressive power is much greater than a simple rewriting, since the parens form can equally well be read as many alternative combinations of gates, such as a circuit composed of AND and NOT gates.

The One-bit Full Adder

The one-bit full adder takes three single bit inputs, {a, b, p}, consisting of the two bits to be added together, *a* and *b*, and the carry bit *p* from

previous additions. The output consists of two bits {s, c}, the sum and the carry bit from the present addition. Intermediate connections {i1, i2, i3} in the network are labeled to allow structure sharing.

In NOR notation, nine gates are needed:

```
i1 = NOR[a,b]
i2 = NOR[ NOR[a,i1],NOR[b,i1] ]
i3 = NOR[p,i2]

s  = NOR[ NOR[p,i3],NOR[i2,i3] ]
c  = NOR[i1,i3]
```

Note that i2 and s can equally well be *read* as AND gates taking two OR gate inputs. In parens notation, nine parens are needed:

```
i1 = (a b)
i2 = ((a i1)(b i1))
i3 = (p i2)

s  = ((p i3)(i2 i3))
c  = (i1 i3)
```

The parens forms can be directly substituted for their labels, constructing a representation without intermediate connections. Substituting i1 into i2:

```
i2 = ( (a  i1  ) (b  i1  ) )
   = ( (a (a b)) (b (a b)) )
```

And continuing to substitute for i2 and i3:

```
i3 = ( p          i2          )
   = ( p ((a (a b))(b (a b))) )

s  = ( (p           i3         ) (        i2                    i3         ) )
   = ( (p (p ((a (a b))(b (a b))))) (((a (a b))(b (a b))) (p ((a (a b))(b (a b))))) )

c  = (  i1               i3          )
   = ( (a b) (p ((a (a b))(b (a b)))) )
```

The NOR gate formulation of this elementary circuit introduces tremendous redundancy, similar to that introduced by the use of other conventional single operator formalizations, such as the Sheffer stroke. We provide simple rules which reduce the representation to a minimal form. In this example, the literal transcription from NOR gates to parens can be reduced in the following manner:

```
i1 = (a b)

i2 =       ((a i1)(b i1))
   =  i1   ((a   )(b   ))                          Distribution
   = (a b) ((a   )(b   ))

i3 = (p (a b) ((a)(b)))

s  =     ((p i3)(i2 i3))
   =  i3 ((p   )(i2   ))                           Distribution
   = (p (a b) ((a)(b))) ((p) ((a b)((a)(b))))

c  = ( (a b) (p (a b) ((a)(b))) )
   = ( (a b) (p       ((a)(b))) )
```

By pattern-matching, intermediate structures can be reintroduced for structure sharing, reconstructing the functionality of the full adder using different logical gates via a different reading of the parens forms:

```
s  =  (p j3) ((p)(j3))              OR[ NOR[p,j3], AND[p,j3]]
c  =  (j1 (p j2))                   NOR[j1, NOR[p,j2]]

j1 =  (a b)                         NOR[a,b]
j2 =  ((a)(b))                      AND[a,b]
j3 =  j1 j2                         OR[j1,j2]
```

BOUNDARY ALGEBRA THEOREMS

We have developed a single boundary operator notation which maintains the full expressability of logic while not requiring either associativity or commutativity. The following axioms of BL provide a theory of transformation between boundary forms, and thus a calculus for multilevel logic synthesis.

There are many axiomatic bases for a particular algebra, Huntington's above is one, Spencer-Brown's is another. Below, we establish an isomorphism between Huntington's axioms and our axiomatization of BL. We have shown how to transcribe logic into BL, we now show that each of Huntington's axioms is a theorem in the transcribed system.

Axiomatic Basis

The name of each axiom of BL is provided, as well as names for each direction of transformation in the calculus:

OCCLUSION	(A ( )) = <void>	REVEAL <=> OCCLUDE
INVOLUTION	((A)) = A	ENFOLD <=> CLARIFY
SIMPLE PERVASION	A (A B) = A (B)	INSERT <=> EXTRACT

Variables correspond to arbitrary forms (represented by capital letters), not only to ground variables (represented by small letters). The transformations can be applied to networks as well as to signals. As a function calculus, capital letters can also represent arbitrary Boolean functions. Under that interpretation, the same letter has the same functionality, but may have a significantly different representation across multiple references.

Each BL axiom is a theorem of standard logic. For comparison, in the axioms of BL in the notation of logic are:

OCCLUSION	(A + 1)' = 0
INVOLUTION	(A')' = A
PERVASION	A + (A + B)' = A + B'

In standard logic, the axioms would read as:

OCCLUSION	NOT (FALSE IMPLIES A) IFF FALSE
INVOLUTION	(NOT (NOT A)) IFF A
PERVASION	((A OR B) IMPLIES A) IFF (B IMPLIES A)

And in the notation of generalized variary NOR:

OCCLUSION	NOR[A NOR[ ]] = NOR[NOR[ ]]
INVOLUTION	NOR[NOR[A]] = A
PERVASION	NOR[NOR[A NOR[A B]]] = NOR[NOR[A NOR[B]]]

In comparison, the representation of an axiomatic basis for logic in the notation of BL is

OCCLUSION	(A ( ))	= <void>
INVOLUTION	((A))	= A
PERVASION	A (A B)	= A (B)

Occlusion is called a Bound Law in logic, Involution has the same name, and Simple Pervasion has no analog in normal lists of logic theorems. These particular BL axioms have the property that computational reduction proceeds in an unambiguous direction, since the right-hand-side of each is strictly smaller than the left-hand-side. Note that transforming the left into the right-hand-side of each axiom occurs by *void-substitution*, an explicit form is replaced by non-representation. Conventionally, this is called *deletion* or *erasure*.

Proof

Theorems in BL are proved by algebraic manipulation. Substitution and replacement can occur in any space in which the pattern of an axiom occurs. Thus Occlusion and Involution can be freely applied at any depth within a parens form. When used as a term rewrite system or as a graph reduction system, the axioms can be applied convergently in any order.

The principle which makes BL efficient and elegant is that form variables (i.e. capital letters) bind to void. To illustrate the proof process, the simple theorem of Dominion is established next. The technique is to convert the left-hand-side of the theorem into the right-hand-side. Examples of the proof technique are in the following sections.

Deep Pervasion

The spatial aspect of BL requires an extension of the standard representation of form variables, the capital letters above. We usually think of a variable standing in place of only one form. However, BL *form variables* can stand in place of both the absence of a form (void substitution) and a collection of forms (set substitution). To use Simple Pervasion as an example:

A (A B) = A (B)		Pervasion
A (A) = A ( )		let B =

As well:

d (a b) c ((a b) f d g) = d (a b) c (f g)	let A = d (a b)

Simple Pervasion can be generalized to Extract redundant forms at arbitrary depths in a boundary form, providing non-local reduction. In networks, this *deep pervasion* reaches to the transitive fanin of each node to achieve simplification by removing redundant reconvergent fanout.

To generalize the typography of Pervasion and other theorems, we will need some new notation, linear tokens which indicate characteristics of network and containment structures.

We first extend Pervasion to *broader* forms, those forms with several subforms within the same space. The two-level Sum of Products form (SOP) is the extreme of broad Boolean forms. For broad theorems only, we use the truncated ellipsis ".." to indicate any arbitrary number of forms occupying a space. We let an indefinite number of *boundary crossings,* each boundary having arbitrary contents, be indicated by curly brackets with a label, such as "{A }". The structure {A } represents any level of nesting with any arbitrary intervening forms. When possible, for comparison, each theorem is also stated in the generalized NOR notation.

BROAD PERVASION A B..(A B..N) = A B ..(N)

NOR[A B .. NOR[A B .. N]] = NOR[A B .. NOR[.. N]]

The proof of Broad Pervasion is simply an iteration of many steps of Simple Pervasion, one for each of the subforms in "A B ..".

In fact, the notation for Broad Pervasion is redundant, since the form variable *A* in Simple Pervasion is sufficient to incorporate the breadth of subforms at the shallowest level. A notation for broad forms is necessary for some of the following theorems.

PERVASION A {B A} = A {B}

The generalized NOR notation has no representation for arbitrarily deep nestings. This rule would have to be build out of repetitions of the simple Pervasion rule:

NOR[A NOR[A B]] = NOR[A NOR[B]]

The presence of two references to the form A, regardless of depth of nesting, can be reduced to one reference. Conversely, any form is also present in all spaces dominated by that form. One particularly useful way to characterize Pervasion is that boundaries are *transparent* to forms on the exterior.

We prove the deep version of Pervasion with an inductive argument. The base-case is Simple Pervasion:

A (A B) = A (B)

The inductive proof proceeds in a constructive direction by Inserting A into successively deeper spaces one at a time, and then in reverse by Extracting A one at a time from the deepest space which still contains it:

```
A (  M (  N (  P (Q A))))                Insert A
A (A M (A N (A P (Q A))))                Insert A
A (A M (A N (A P (Q  ))))                Extract A
A (A M (A N (  P (Q  ))))                Extract A, ...
```

Fundamentally, deep Pervasion defines the *semipermeability of boundaries*. This semipermeability may be more familiar when the boundary is read as logical IF. From this perspective, parens forms are nested implicational structures with the least influential premises at the deepest levels. Pervasion removes redundantly implied forms. Initial states (inputs) are nested in the deepest spaces and directly imply only forms in the next shallower space. Pervasion is thus similar to the global flow techniques which build implication graphs of subnetworks with transitive fanout [Trevillyan]. Insertion takes the place of the additional logic added by these techniques on the way to minimization.

There is yet a further generalization to the strength of deep transformations, identifying the accumulative structure of the deep transformations. We will indicate this generalization for Pervasion, but will not include it in the formulation of the theorems which follow.

ACCUMULATIVE PERVASION

```
A {M B {N A {P B}}}}  =  A {M B {N   {P  }}}}
```

Here all references which dominate their respective spaces, regardless of depth of first appearance, pervade deeper references. Accumulation suggests an algorithm which accumulates new forms as it descends, and removes any matches to the accumulated set as it passes them. In full generality, this process is not restricted to a single nested form, but also includes subforms with breadth.

BROAD ACCUMULATIVE PERVASION

```
A {M B {N A {P B C}}} (C {Q A {R B C}} = A {M B {N A {P   C}}} (C {Q   {R B  }}
```

Symbolic notation is poorly suited for handling the implications of void-equivalence, since it emphasizes single or multiple discrete occurrences of a reference. In a boundary form, Pervasion in its most general sense means that any form which explicitly occurs is also optionally present throughout the space it occurs within, to any depth of nesting. Boundaries are transparent to a Pervasive form.

Simple Theorems

The following three theorems illustrate proof in the extended notation, for *broad* and to *deep* forms. The second theorem, Idempotency, provides a mechanism for structure sharing in BL forms, and extends Deep Pervasion to the space containing a form.

DOMINION A ( ) = ( )

NOR[NOR[A NOR[ ]]] = NOR[ ]

Proof:

```
  A ( )                                  Left-hand-side
((A ( )))                                Involution
(       )                                Occlusion, rhs.
```

IDEMPOTENCY A A = A REPLICATE <=> COALESCE

NOR[NOR[A A]] = A

Proof:

```
A   A   =  A
A ((A)) =  A                             Enfold A
A (( )) =  A                             Deep Extract A
A       =  A                             Clarify, Identity.
```

BROAD IDEMPOTENCY A .. A = A

NOR[NOR[A .. A]] = A

Proof:

```
A   .. A   =  A
A ((.. A)) =  A                          Enfold .. A
A ((    )) =  A                          Deep Extract .. A
A          =  A                          Clarify, Identity.
```

The intent of the ".." mark is to iterate the same pattern many times. Thus, it does not stand for "anything which follows"; rather it stands for "an indefinite reference to forms equivalent to *A*".

The next theorem is the general case of Occlusion:

DEEP OCCLUSION A {M (N (A))} = A {M }

Proof:

A {M (N (A))}	lhs
A {M (N ( ))}	Deep Extract A
A {M }	Void Occlude N, rhs.

Absorption Theorem

Pervasion and Absorption are closely related, each represents a perspective from each side of the distinction. Pervasion matches forms within the same pervasive space, while Absorption matches subforms of bounded forms within the same space. That is, the matches in Pervasion share the same space, while the matches in Absorption do not, although the bounds which contain the matches in Absorption do share the same space.

SIMPLE ABSORPTION (A) (A B) = (A) EXPAND <=> SUBSUME

NOR[NOR[NOR[A] NOR[A B]]]] = NOR[A]

Proof:

(A) ((A) A B) = (A)	Insert (A)
(A) (( ) A B) = (A)	Extract A
(A) = (A)	Occlude A B, Identity.

DEEP SIMPLE ABSORPTION (A) {M (A N)} = (A) {M }

A comparison of Deep Pervasion and Deep Absorption illustrates their similarities.

DEEP PERVASION A {M (A N)} = A {M (N)}

Both deep theorems reference the form variable A twice on the left-hand-side. In the case of Pervasion, the erased A takes on the value *void*, i.e. it is erased. In the case of Absorption, the boundary containing the erased A is also erased. This can be seen to be equivalent to the erased A taking on the value *mark*.

We now generalize the Simple Absorption theorems above. In the proof of Simple Absorption, one form was inserted into another. We will call the form

that was inserted, the *put-form*. The form which was inserted into is the *into-form*. The target of simplification is always the into-form.

We first extend Absorption to broad forms, occurrences of multiple subforms of the same type at the same level:

BROAD ABSORPTION (A (C D)..) (A B (C)..) = (A (C D)..)

NOR[NOR[NOR[A NOR[C D]..] NOR[A B NOR[C]..]]] = NOR[A NOR[C D]..]

Index the depth of a form by assigning the outermost space zero. Each boundary crossing inwards increases the depth by one, each crossing outwards decreases the depth by one.

The form variable A indicates that the put-form contains an arbitrary number of subforms at the shallowest level which match those in the into-form. The variable B indicates that the into-form can contain additional subforms that do not match contents of the put-form. The constraint is that all put subforms must be matched. Next there are any number of bounded subforms within the put-form. At depth two, each bounded form (e.g. *(C D)*), must itself be Subsumed by a form in the into-form (e.g. *(C)*), a case of recursive embedding of the simple theorem. The ".." indicates an iteration of forms with the same structure of *(C D)*, i.e. forms with structure *(E F)*. Under these conditions, the into-form is completely redundant.

We indicate the Inserted form using *carets ^...^*, tokens with only the meta-syntactic intent of emphasizing the inserted put-form. Single forms contained in a boundary are called *bounds*. In other sections, we also use brackets, [ ], as a meta-syntactic convention to highlight certain parens structures. Functionally, the brackets are identical to parens.

The proof of Broad Absorption now makes the constraints clear:

```
(A (C D)..) (              A B (C)..)        lhs
(A (C D)..) (^(A (C D)..)^ A B (C)..)        Insert put
(A (C D)..) (^(  (C D)..)^ A B (C)..)        Extract A
(A (C D)..) (^    C D .. ^ A B (C)..)        Clarify
(A (C D)..) (^    C D .. ^ A B ( )..)        Extract C
(A (C D)..)                                  Occlude C D A B, rhs.
```

Above, the ".." indicates an indefinite number of bounds of the form *(C D)..* can be Subsumed by their respective counterparts, *(C)...*

Broad Absorption can now be generalized to a fully general broad and deep theorem:

ABSORPTION

```
(A B (C D)) {M B {N (C) {P (A N)}}} = (A B (C D)) {M B {N (C) {P      }}}
```

The *structure of Absorption* is one of many put subforms, all at the same relative polarity as their into counterparts. The existence of so many tokens does not limit the power of the theorem, since any of them can be void.

The proof of fully general Absorption follows the same induction as the proof of Deep Pervasion. As the put-form descends into the into-form, whenever it passes a subform which either Extracts or Subsumes a portion of it, that portion is erased. When all subforms in the put-form have been erased, leaving an empty parens, ( ), the immediate into-form context which contains the empty parens is Absorbed and Clarified.

Comparative Proofs

The notation and axioms of BL are more powerful than those of both conventional logic and Boolean algebra. Although all three calculi are equivalent, a comparison of the available tools illustrates their computational differences. We prove Idempotency, Occlusion, and a case of Pervasion using logical techniques.

Idempotency

A proof of Idempotency using the notation of logic and Huntington's axioms follows:

```
x+x = (x+x)*1                      a = a*1        let a=x+x
    = (x+x)*(x+x')                 1 = a+a'       let a=x
    = x+(x*x')           (a+b)*(a+c) = a+(b*c)    let a=x, b=x, c=x'
    = x+0                       a*a' = 0          let a=x
    = x                          a+0 = a          let a=x, QED.
```

Occlusion

To be comparable, a proof of Occlusion in logic should use only given axioms plus established theorems. Although Idempotency was easy to prove from the axioms of logic, Occlusion is exceptionally difficult, requiring insight and several new theorems. A map of the proof follows, each step requires proof of the corresponding theorem:

```
a = a+(a'+b)'                          Occlusion
  = a+(a''*b')                         DeMorgan
  = a+(a*b')                           Involution
  = a                                  Absorption, Identity.
```

The Robbins problem [Robbins] asks: what is the smallest number of axioms needed to define logic? It remained unsolved for years, but became recently the first significant unresolved proof by a computer. The proof of Occlusion is a tour through the symbolic issues of the Robbins Problem, since to conduct the proof, most significant small theorems in BA have to be established.

We first must develop tools to handle the complement of a compound expression, (x+y)'. DeMorgan's Laws are thus essential. To prove DeMorgan, we need the two Bound Laws and Complement Uniqueness. Once DeMorgan is applied, we need Involution, and then Absorption to complete the proof of Occlusion.

```
x*0  =  0                                  Bound Law to prove
x*0  =  x*0 + 0                                  Identity
     =  x*0 + x*x'                               Complement
     =  x*(0+x')                                 Distribution
     =  x*(x'+0)                                 Commutativity
     =  x*x'                                     Identity
     =  0                                        Complement, QED.
```

The second Bound law, x+1 = 1, has an identical dual proof as above. This proof can simply call upon the *duality principle* of logic that forms remain equivalent when both 0 is exchanged for 1 and * is exchanged for +. We continue with Complement Uniqueness:

```
x*y  =  0                                  Assume
x+y  =  1                                  Assume
y  =  x'                                   Unique Complement to prove

y  =  y*  1                                      Identity
   =  y*(x+x')                                   Complement
   =  (y*x)+(y*x')                               Distribution
   =  (x*y)+(x'*y)                               Commutativity twice
   =    0  +(x'*y)                               Assumption
   =  (x'*y)+ 0                                  Commutativity
   =  (x'*y)+(x*x')                              Complement
   =  (x'*y)+(x'*x)                              Commutativity
   =  x'*(y+x)                                   Distribution
   =  x'*(x+y)                                   Commutativity
   =  x'* 1                                      Assumption
   =  x'                                         Identity, QED.

a'*b'  =  (a+b)'                           DeMorgan to prove
```

```
(a+b)*(a'*b')  =  0                          Lemma
   =  (a'*b')*(a+b)                                lhs, Commutativity
   =  ((a'*b')*a)+((a'*b')*b)                      Distribution
   =  (a*(a'*b'))+((a'*b')*b)                      Commutativity
   =  ((a*a')*b')+(a'*(b'*b))                      Associativity twice
   =  ((a*a')*b')+(a'*(b*b'))                      Commutativity
   =  (  0   *b')+(a'*  0   )                      Complement twice
   =  (b'*0)+(a'*0)                                Commutativity
   =     0  +  0                                   Bound Law twice
   =  0                                            Identity, rhs.
```

A second Lemma, (a+b)+(a'*b') = 1, has an identical proof by the duality principle, as above. Completing the DeMorgan proof:

```
(a+b)*(a'*b')  =  0                          Lemma
(a+b)+(a'*b')  =  1                          Lemma
(a'*b')  =  (a+b)'                           Unique Complement,
                                               x = a+b, y = a'*b', QED.
```

The second DeMorgan Law, (a'+b') = (a*b)', has an identical dual proof as above. We can now continue with the support theorems of Involution and Absorption:

```
 x''  =  x                                   Involution to prove
a'''  =  a'                                        Let x = a'
a'''  =  (a''+  0   )'                             Identity
      =  (a''+(a*a'))'                             Complement
      =  ((a''+a)*(a''+a'))'                       Distribution
      =  ((a+a'')*(a'+a''))'                       Commutativity twice
      =  ((a+a'')*   1    )'                      Complement
      =  ((a+a'')* (a+a') )'                       Complement
      =  (a+(a''*a'))'                             Distribution
      =  (a+(a'*a''))'                             Commutativity
      =  (a+   0    )'                             Complement
      =   a'                                       Identity
 x''  =   x                                        Let a' = x, QED.

x+(x*y)  =  x                                Absorption to prove
x+(x*y)  =  (x*1)+(x*y)                            Identity
         =    x  *(1+y)                            Distribution
         =    x  *(y+1)                            Commutativity
         =    x  *  1                              Bound Law
         =    x                                    Identity, QED.
```

The point of this arduous proof is not to deduce the foundations of logic but to demonstrate that the inherent mechanisms of logic are far more complex than those of BL. Even with the several logic theorems established above, this *relative complexity* characterizes the entire system, showing up to some

degree in all logic proofs. In particular, each theorem and lemma above makes use of a void-equivalent in some way, usually in the form of Complement x*x' = 0. More importantly, the introduction of the void-equivalent requires an insightful step, one that is not supported by the problem context it is introduced into. By providing a transparent structure for insertion of void-equivalents, and by providing an operational concept of depth, BL greatly simplifies the mechanisms of deduction

Two-Deep Pervasion

A proof of Deep Pervasion using Huntington's notation and axioms is cumbersome. To illustrate this, let us assume that all of the theorems of logic have been established, so that the only issue we face is depth of application. Let us also assume that we have built in associativity and commutativity and n-arity of + and *. Let us further prove only a special case of Deep Pervasion, in order to avoid having to define a concept of indefinite nesting.

Let us prove "Two-deep" Pervasion:

A (B (C (A D))) = A (B (C (D)))

One approach would be to flatten the problem using distribution.

a+(b+(c'*(a+d)))'	lhs
= a+(b+(c'*a)+(c'*d))'	Distribute
= a+(b'*(c'*a)'*(c'*d)')	Extended DeMorgan
= a+(b'*(c+a')*(c+d'))	DeMorgan
= (a+b')*(a+c+a')*(a+c+d')	Extended Distribute
= (a+b')*(c+1)*(a+c+d')	Complement
= (a+b')*(a+c+d')	Absorption
= a+(b'*(c+d'))	Distribute
= a+(b+(c+d')')'	DeMorgan
= a+(b+(c'*d))'	DeMorgan, rhs.

For comparison, a void-based "proof" might read: A form is arbitrarily in all spaces deeper than its first appearance because the space it is in pervades all deeper spaces.

BOUNDARY NOTATION

We have represented networks in the linear typography of parens.

Advantages

The advantage of boundary notation is in its clarity. This comes from

1. the convenience of having a void. Half of each duality is not present in the representation. Void-substitution is the primary transformation, so that forms erase rather than rearrange. But the profound gain is that the void facilitates every one of the following advantages. None work when the void is assigned any form of explicit representation.

2. not having to assign forms to each variable in each axiom. Variables can bind to void. The axioms of BL deal directly with boundary operators, manipulating functional structures as well as data structures.

3. the compactness of a single operator system. It is not necessary to translate between constants {0,1} and operators {+,*} at every step. That is, DeMorgan transformations not necessary, they specify identified in the notation.

4. the flexibility of complementation. Complement is simply bounding a form, rather than percolating the complement operator across levels of + and *.

5. the disassembly of Distribution. In logic, the distributive axiom is difficult to use, since it requires substantive pattern-matching, has three separate variables to assign, and does not have an obvious preferred direction of application.

6. the scope of deep operations. Deep transformations are particularly difficult in a binary notation which has no concept of Pervasion. All logic transformations must carry the weight of flattening every form in order to match the available transformations.

7. the transparency of boundaries to specific operations. Logic operators are never transparent, that is, ignorable. Since both FALSE and OR are non-existent in the BL notation, they pervade all spaces, permitting boundaries to be ignored in the case of constructing void-equivalent forms anywhere in the representation and in the case of Inserting existent forms into any deeper space.

COMPARATIVE HARDWARE TECHNOLOGIES

The BILD and the Comesh hardware architectures can be interpreted as relying on a spatial array of logical NOR gates. Several existing architectures also use spatial arrays of NOR gates, or at least spatial arrays of logical information. None of these have the same structure or functionality as the BILD and Comesh Circuit Configuration Array (CCA). We describe these conventional architectures and contrast them with the CCA approach.

Conventional Architectures

Two-level Programmable Logic Arrays

All logic expressions can be converted to a two-level form, in which the maximum nesting of logic gates, not counting inverters, is two. This form is often referred to as SOP, for *Sum of Products*. In the vocabulary of Boolean algebra, sums are logical OR, and products are logical AND. Thus, the SOP form consists of a level of AND gates feeding a level of OR gates. The dual is POS, for *Product of Sums*.

Two-level logic is used extensively in programmable logic arrays (PLAs). The difficulty with two-level forms is that they become exponentially large as the number of variables increases. For example, the nested expression

```
OR[ AND[a,b], AND[OR[c,d], OR[e,f]]]
```

expands to the following two level POS expression:

```
AND[ OR[a,c,d], OR[b,c,d], OR[a,e,f], OR[b,e,f]]
```

In parens form, the POS expansion is one of successive distribution.

```
DISTRIBUTION        A ((B)(C)) => ((A B)(A C))
```

In the distribution step, the form A is replicated during the course of making an expression shallower. The above example expressed in parens form is:

```
((a)(b)) ((c d)(e f))
((c d ((a)(b)))(e f ((a)(b))))          dist ((a)(b))
((a c d)(b c d)(a e f)(b e f))          dist c d, dist e f
```

The three distribution steps in the example increase the variable occurrences from six to twelve. In conventional two-level hardware architectures, this means that the fanout of each input variable doubles. Further nestings would continue to double the variable fanout, resulting in an exponential explosion

of physical wires. Consequently, the two-level architecture is not feasible for large circuits.

To address this problem, CPLDs consist of a collection of simple two-level PLAs, called macrocells, with associated interconnect wiring between the macrocells. This results in two serious difficulties:

- Logic must be partitioned into small chunks to fit into small, fixed-resource macrocells, and
- Fixed-resource interconnect must be provided for all desired logic flow between macrocells.

The CCA approach encounters neither of these difficulties.

NOR Arrays

One approach to the fabrication of ASICs is the sea-of-NORs. The hardware infrastructure for this approach is a regular tableau of NOR gates, and an interconnect matrix used to connect them. The final result is a conventional NOR-based ASIC, with some of the tableau gates connected to form a multilevel NOR circuit.

NOR arrays are also used as a structure for memory. Each column of the NOR array memory consists of a large NOR gate, with the memory word lines driving all the transistors on a particular row. NOR array memories are ROMs, they are intended to be hardwired. A common use of NOR arrays is as look-up tables.

Finally, a NOR array can be used as the substrate for a PLA, rather than the SOP or POS structures mentioned above.

The CCA is not a layout of logic gates with potential wiring, it is rather a layout of memory locations which independently do not embody a logical functionality. It is the spatial array of all the memory locations, the relational locations of the marked memory cells, which defines the logic functionality. One possible CCA design could rely on a NOR array memory as a substrate. However, the functionality of the CCA is substantively different than that of a NOR array memory. The NOR array memory is intended to store data. When queried, it returns the stored data at each NOR intersection. In contrast, the CCA, when queried, returns the evaluation of the function stored in the array.

Folded NORs

A technique to improve the efficiency of PLAs is called the folded NOR approach. In this technique, signals exiting the two-level logic are feed back into (folded into) the array at specific points. This permits some

structure sharing between circuit elements. Successive levels of NOR gates can be fed-back, or cascaded, in order to implement multilevel logic designs.

The CCA does not use folded NORs, it does not return inputs into the array for additional processing (except in the case of information stored in registers for a later clock tick). Comparing the logical structure of each approach makes their differences clear.

Look-up Tables

Look-up tables are also called truth tables. They are commonly implemented in a PROM architecture. Similar to the two-level form of PLAs, look-up tables experience an explosion of physical wires as the number of inputs increases. In the case of a look-up table, each variable has two possible values, {0,1}. Variables in combination multiply the possibilities, so that two variables require four table entries, three variables require eight entries, and in general, N variables require 2^N entries in the look-up table. For example, the four variable function

OR[AND[a,b], AND[c,d]]

would have the following look-up table:

a	b	c	d	((a AND b) OR (c AND d))
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Physical look-up tables have very much of the same layout. A sixteen bit memory array is initialized with 1s in the locations identified in the function evaluation column above. When the input vector is identical to a row associated with a function evaluation of 1, the output of the look-up table is 1, otherwise it is 0.

The size of memory resource required for a look-up table is 2^N, for N variables. This approach very clearly becomes physically impossible for large N. Ten inputs require about one thousand storage locations, while 20 inputs require one million storage locations.

The spatial array of the CCA does not use values indexed by input vectors, and does not encounter the explosion of table entries.

Cross-point Switches

A cross-point switch is a programmable array of interconnect points. These switches are used for steering bit-streams between i/o ports of appropriate processing devices, such as routers. Switching matrices support one-to-one and one-to-many switching.

The CCA architecture does not steer logic to particular locales (other than function outputs). Switching matrices do not compute a logical function value, they only route signals.

The Circuit Configuration Array

The BILD and Comesh circuit configuration arrays have been described above. Following is a list of conventional hardware architectures and the substantive differences between each architecture and the innovative BM architectures.

Conventional architecture	*CCA Similarity*	*CCA Difference*
ASIC	multilevel	programmable
NOR Sea-of-Gates	multilevel	programmable
FPGA	programmable	no look-up tables
CPLD	programmable	no macrocells
PLA or SPLD	efficient	multilevel
NOR Memory Array	array	functional evaluation
Folded NOR	programmable	no signal feedback
Look-up Table	array	no stored values
Cross-point Switch	array	functional evaluation