Representation and Symbolic Manipulation of Linearly Inductive Boolean Functions*

Aarti Gupta Allan L. Fisher
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213.

Abstract

We consider a class of practically useful Boolean functions, called Linearly Inductive Functions (LIFs), and present a canonical representation as well as algorithms for their automatic symbolic manipulation. LIFs can be used to capture structural induction in parameterized circuit descriptions, whereby our LIF representation provides a fixed-sized representation for all size instances of a circuit. Furthermore, since LIFs can naturally capture the temporal induction inherent in sequential system descriptions, our representation also provides a canonical form for sequential functions. This allows for a wide range of applications of symbolic LIF manipulation in the verification and synthesis of digital systems. We also present practical results from a preliminary implementation of a general purpose LIF package.

1 Introduction

Symbolic manipulation of Boolean functions has found numerous applications in the area of VLSI design automation [5]. These applications are greatly facilitated by the availability of a canonical Boolean function representation – Bryant's Binary Decision Diagram (BDD) – and symbolic manipulation algorithms that are efficient in practice [4]. In this paper, we focus on automatic symbolic manipulation for a class of *inductive* Boolean functions, i.e. Boolean functions that are defined in terms of induction parameters, called Linearly Inductive Functions (LIFs). We present a canonical representation and symbolic manipulation algorithms for LIFs, with the important feature that their complexity is independent of the parameter of inductive description. We also describe a wide range of applications of symbolic LIF manipulation in the verification and synthesis of digital systems. These apply to two broad categories of circuits:

- inductively-defined hardware, i.e. hardware defined inductively (iteratively, or recursively) in terms of size parameters.
- sequential systems, where time implicitly serves as an induction parameter.

In the first category, we consider circuits described by induction on structure. For example, an n-bit serial adder circuit can be described in terms of some gates and an $(n-1)$-bit adder. In cases where such structural induction can be captured by LIFs, our LIF representation provides a fixed-sized representation for all size instances of the circuit. It also allows us to perform automatic proofs by induction for verification of functional properties. This has practical application in the representation and verification of standard libraries of parameterized designs, which are becoming increasingly important due to emphasis on design reuse [9].
In the second category, LIFs naturally capture the temporal induction inherent in sequential system descriptions, where functions at time t are defined in terms of inputs at time t and functions at time $(t-1)$. In fact, sequential functions, including state and output functions of a deterministic finite state machine (FSM), are LIFs with time as the induction parameter. In this case, *our LIF representation directly provides a canonical representation for a sequential function*, in much the same way as a BDD provides a canonical representation for a combinational function. This sequential canonical form has applications in the verification and synthesis of sequential systems that strongly parallel the applications of BDDs for combinational systems. For example, the canonicity property for FSM outputs can be used to check FSM input/output equivalence, since it does not rely on the number of states or a particular state encoding. LIF manipulations can also be used to capture other interesting automata properties including language equivalence/containment, the set of reachable sets etc.
We start by giving a formal characterization of Linearly Inductive Functions, followed by our representation and symbolic manipulation algorithms.

2 Linearly Inductive Functions

2.1 Characterization

Informally, an inductive Boolean function is defined parametrically in terms of an induction parameter, with the arguments of the function also parameterized accordingly. Formally,
Definition 1: A Linearly Inductive (Boolean) Function (LIF) f satisfies the following three conditions:

*This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

- Condition 1: For $i = 1$, the 1-instance of f (denoted f^1) is a Boolean combination (denoted $\mathcal{B}^1$) of the 1-instance inputs (denoted X^1), i.e. $f^1 = \mathcal{B}^1(X^1)$.
- Condition 2: For all $i > 1$, the i-instance of f (denoted f^i) is a Boolean combination (denoted $\mathcal{B}^i$) of the i-instance inputs (denoted X^i) and some $(i-1)$-instance functions (denoted G^{i-1}), each of which is also an LIF, i.e. $f^i = \mathcal{B}^i\ (X^i, G^{i-1})$.
- Condition 3: For all $i, j > 1, \mathcal{B}^i \equiv \mathcal{B}^j$, i.e. f^i is related to X^i and G^{i-1} in the same manner as f^j is to X^j and G^{j-1}, respectively.

Example 1: The MSB of a serial adder with inputs a, b and a carry input c_in, can be represented parametrically as an LIF sum as follows:

for $i = 1, sum^1 = a^1 \oplus b^1 \oplus c\_in$,
for $i > 1, sum^i = a^i \oplus b^i \oplus carry^{i-1}$,

where the LIF $carry$ is defined as:

for $i = 1, carry^1 = (a^1 \wedge b^1) \vee ((a^1 \vee b^1) \wedge c\_in)$,
for $i > 1, carry^i = (a^i \wedge b^i) \vee ((a^i \vee b^i) \wedge carry^{i-1})$.

Note that LIFs correspond intuitively to inductive functions with a serial style of induction. In fact, since finite state sequential systems can be regarded as computing functions serially over time, we can effectively capture them as LIFs also. More formally, given a symbolic state vector $S = [s_1, s_2, \ldots, s_n]$, and a set of symbolic inputs $X = [x_1, x_2, \ldots, x_m]$, each transition function (s), and each output function (o) of a deterministic FSM can be expressed in the form of an LIF as follows:

Transition functions: for $t > 1, s^t = \mathcal{B}_s(X^t, S^{t-1})$,
Output functions[1]: for $t > 1, o^t = \mathcal{B}_o(X^t, S^{t-1})$,

where the values of these functions at $t = 1$ represent the initial states and outputs respectively.

Example 2: An LIF description for a standard 2-bit counter, with state vector $S = [s_1, s_2]$, an enable input e, and output vector $O = [o_1, o_2]$, can be given is as follows:

for $t = 1, s_1^1 = o_1^1 = 0$,
for $t > 1, s_1^t = o_1^t = (\neg e \wedge s_1^{t-1}) \vee (e \wedge \neg s_1^{t-1})$,
for $t = 1, s_2^1 = o_2^1 = 0$,
for $t > 1, s_2^t = o_2^t = (\neg e \wedge s_2^{t-1}) \vee (e \wedge (s_2^{t-1} \oplus s_1^{t-1}))$.

2.2 An LIF Representation

Given that a BDD is a canonical representation for a Boolean function, we consider an inductive extension of a BDD as a natural candidate for representation of an inductive Boolean function. This inductive extension for an i-instance LIF is like a BDD on i-instance variables, except that its leaf nodes can be either Boolean constants, or pointers to $(i-1)$-instance LIF representations. Observe that Condition 3 for the LIFs (Definition 1) implies that the graph structure representing f^i is *isomorphic* to the graph structure representing

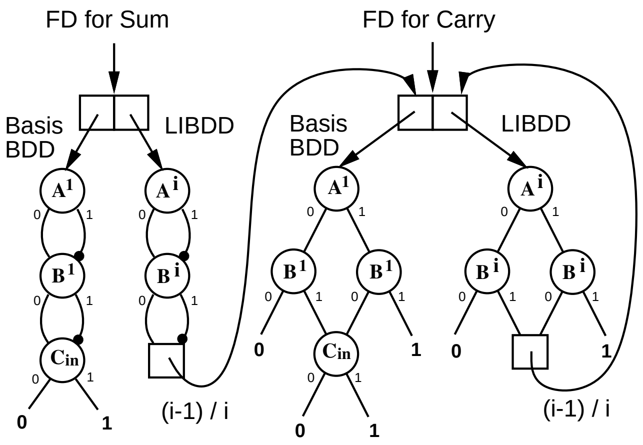

Figure 1: FDs for MSB of a Serial Adder

f^j $(i, j > 1)$, modulo substitution of parameters. In other words, the representation for all f^i, $i > 1$, can be captured by maintaining a substitution semantics on only one parametric structure, and we can identify each LIF with a representation as follows.

Definition 2: An LIF f is represented as a *Function Descriptor (FD)* consisting of:

1. a BDD on 1-instance variables representing the 1-instance of f — called the *Basis BDD* for f, and
2. a parametric i-instance BDD with substitution semantics, representing all i-instances of f, $i > 1$ — called the *Linearly Inductive BDD (LIBDD)* for f.

For example, the FDs for LIFs sum and $carry$ in the serial adder of Example 2, are shown in Figure 1 (where the parameter substitution on pointers from leaf nodes of LIBDDs to FDs is indicated as '$(i-1)/i$').

2.3 Canonicity of LIF Representations

In order for the FD representation to be canonical for an LIF, we need to ensure canonicity of both the Basis BDD and the LIBDD. Since the Basis BDD is a standard BDD, it can be made canonical by construction (with respect to a given variable ordering) [3]. The canonicity of the LIBDD can also be maintained by extending the standard Reduce operation [4], with the modification that leaf nodes are taken to be isomorphic if and only if either they denote the same Boolean constant, or they point to the same canonical FD. Thus, the main task is to establish canonicity of the LIBDD leaf nodes.

Two main problems need to be addressed to accomplish this task, which make it significantly different from the corresponding problem for BDDs. We demonstrate these problems with the help of the following simple example.

Example 3: Consider three user-defined LIFs f, g, and h:

for $i = 1, f^1 = x^1$; for $i > 1, f^i = x^i \wedge g^{i-1}$,
for $i = 1, g^1 = x^1$; for $i > 1, g^i = x^i \wedge h^{i-1}$,
for $i = 1, h^1 = x^1$; for $i > 1, h^i = x^i \wedge (f^{i-1} \vee g^{i-1})$.

[1] This form is for a Mealy machine model; a Moore machine model can also be handled easily.

Note that these LIFs are defined in a mutually-recursive manner. Therefore, a bottom-up approach starting from canonical leaf nodes, as is used in building a standard BDD, will not work. Furthermore, the definition of h^i involves a Boolean combination of $(i-1)$-instance functions f^{i-1} and g^{i-1} (as allowed by Condition 2 of the LIF Definition). However, an LIBDD allows only leaf nodes to point to other FDs, i.e. branching on nodes with pointers to FDs is not permitted. So we need to find some way of representing this Boolean combination function as an FD. Our approach for handling these problems is to conceptually separate the task into two separate phases (described in the following sections):

- Generation Phase — where we work with "tentative" (as yet non-canonical) FDs, such that leaf nodes of a tentative LIBDD may point to tentative FDs, and where new LIFs are generated to represent the required Boolean combination functions as FDs.
- Comparison Phase — where tentative FDs are made canonical by comparison against those in the canonical set.

2.3.1 Generation Phase

In a manner similar to BDDs, we start from the user-given definitions of LIFs, where each LIF is identified with a tentative FD, and generate the tentative LIBDDs bottom-up. For each required Boolean combination of two user-given LIFs, we rename the combination to a *new* LIF with an associated tentative FD. (Note that a Boolean combination of LIFs is also an LIF.) However, we have to guard against generating an infinite series of new LIFs due to renaming. *The key idea is to initially maintain canonical forms for Boolean combinations with respect to user-defined LIFs.* In other words, canonicity is initially maintained with respect to user-defined LIFs, not with respect to the underlying variables. For example, two combinations p and q, defined as $p = f \vee g$ and $q = f \wedge g$, are *not* equivalent with respect to LIFs f and g. However, p and q may be equivalent with respect to the underlying variables x and y, e.g. if $f = g = x \vee y$. Note that the reverse implication is always true, i.e. equivalence with respect to user-defined LIFs implies equivalence with respect to the underlying variables. Thus, our initial LIF equivalence classes are conservative with respect to the underlying variables.

We maintain canonicity with respect to user-defined LIFs by working in a separate Boolean meta-space, where a meta-variable is assigned to each user-defined LIF. BDDs on these meta-variables, called Meta-BDDs, denote arbitrary Boolean combinations of user-defined LIFs. Due to their canonicity property, Meta-BDDs are used to ensure that all equivalent combinations are renamed to the same new LIF. Since there exist a finite number of unique combinations (2^{2^u}) for a finite number of user-defined LIFs (u), this also guarantees termination of our renaming process. For each new LIF generated to represent a Boolean combination, we construct its tentative FD by using a standard BDD Apply operation [4] on the Basis BDDs, and a modified Apply on the LIBDDs for the argument LIFs. (The simple modification allows for terminating recursion at the leaf nodes by generating new LIFs representing distinct Boolean combinations, as and when needed.) In summary, a tentative FD is generated for each user-defined LIF, and for each required, distinct Boolean combination of user-defined LIFs.

For our Example 3, the tentative FDs (TFDs) for f and g are shown in Figure 2(a). (In our figures, the first slot of a TFD is implicitly understood to point to the Basis BDD, and the parameter substitutions are also implicit.) In order to capture the user definition of h, we rename the Boolean combination of f and g to a new LIF p, defined $p^i = f^i \vee g^i$, such that leaf node of the tentative LIBDD for h points to the TFD for p. Next, we construct the TFD for p by using Apply on the Basis BDDs and LIBDDs for f and g. In this process, we need a new LIF defined $q^i = g^i \vee h^i$ (resulting from Apply on leaf nodes of the LIBDDs for f and g). Again, we construct the TFD for q, and we now need $r^i = h^i \vee p^i$, as shown in the same figure. Note that the new LIFs (p, q, and r) represent distinct Meta-BDDs consisting of the meta-variables f, g and h as shown in Figure 2(b). At this stage, the tentative LIBDD for r points to a potentially new LIF $s^i = p^i \vee q^i$. However, from meta-space manipulations, we find that s represents the same Meta-BDD as r. Thus, the leaf node of the LIBDD for r should point back to the TFD for r, and LIF generation is complete. If we had not caught this redundancy, the process of renaming would have continued forever.

2.3.2 Comparison Phase

Once all required, potentially mutually-recursive, tentative FDs have been generated, the task is to convert them to canonical FDs. We maintain a separate set of canonical FDs (which is empty to start with) in a manner similar to a standard scheme for BDDs [3]. The idea is to convert each tentative FD to a canonical one, either by finding an equivalent FD in the canonical set, or by adding a new member to this set if it is unequal to all existing members.

Getting back to our Example 3, assume that the set of canonical FDs is initially empty. So we pick a candidate, say the TFD for f, and attempt to make it canonical. Its LIBDD leaf points to the TFD for g. Since our current canonical set contains only the TFD for f, we have to first check if TFD for g is equal to TFD for f before adding it to the set. It is easy to see from their TFDs (Figure 2) that $g = f$, *provided* $h = g$.

We represent such dependencies in the form of a directed graph on nodes that represent equalities between (unordered) pairs of FDs. In this graph, which we call a

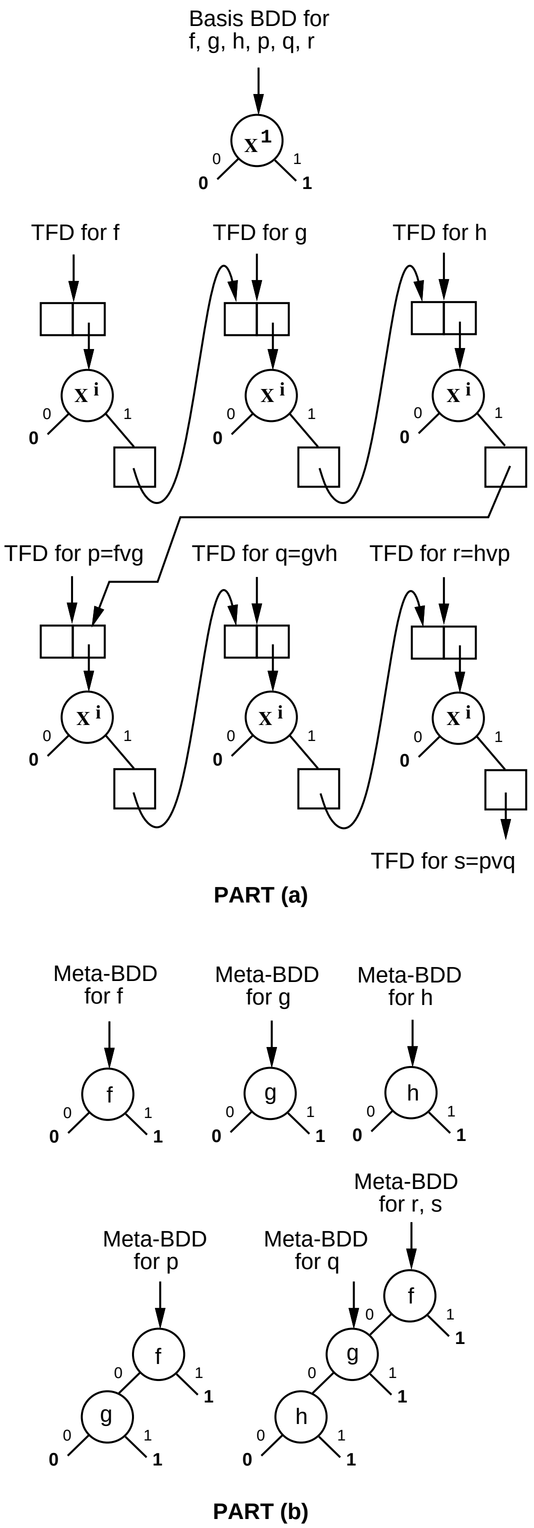

Figure 2: TFDs and Meta-BDDs for Example 3

$EQ(s,t)$: To check equality of FDs for LIFs s and t

1. *Create node (s,t) (if it doesn't exist), and explore it as follows.*
2. *If it is known that $s=t$ from table of results, label node (s,t) 'True', and return.*
3. *If it is known that $s \neq t$ from table of results, label node (s,t) 'False', and return.*
4. *Check equality of Basis BDDs for s and t. If not equal, $s \neq t$, label node (s,t) 'False' and return.*
5. *Check equality of LIBDDs for s and t as follows:*
   - (a) *Find pairs of corresponding LIBDD leaf nodes for s and t by recursive LIBDD traversal.*
   - (b) *For each pair of leaf nodes p and q:*
     - *Create a node (p,q) (if it doesn't exist).*
     - *Create an edge from (s,t) to (p,q).*
     - *If node (p,q) was just created, explore it recursively using $EQ(p,q)$.*

     If any node (p,q) is labeled 'False', label node (s,t) 'False' and return.
   - (c) *Reduce graphs of LIBDDs for s and t based on equality results of leaf nodes.*
6. *At this point, $s=t$. Label node (s,t) 'True'.*

Figure 3: Algorithm for $EQ(s,t)$

comparison-graph, an edge exists from a node v to a node w, if the truth of v requires the truth of w. We also maintain a table of results of equality checks for all pairs of FDs. A high-level algorithm $EQ(s,t)$, for checking equality of FDs for LIFs s and t, is shown in Figure 3. (In the context of sequential functions, this algorithm can be viewed as checking state equivalence; its exact relationship to standard techniques is beyond the scope of this paper.)

For each top-level call to $EQ(s,t)$, a comparison-graph is constructed which labels the nodes 'True' or 'False'. Note that Step 5 can be implemented as a standard BDD algorithm on the recursive binary structure of an LIBDD, and Step 5(b) for the LIBDD leaf nodes may recursively lead to new equality checks on pairs of FDs. The termination condition for this recursion involves an interesting interplay between the mutually-recursive and inductive aspects of LIF definitions, and is captured by the following assertion.

Theorem: The equality of FDs corresponding to a node v in a comparison-graph is true if and only if there is no directed path from v to a node labeled 'False'.

Proof Outline: (Details can be found in [10].) The proof is a simple inductive argument on the length of either a simple path connecting node v to a node labeled 'True', or a path connecting v to a cycle of nodes. This inductive argument justifies the use of previous hypotheses (in the form of previously labeled 'True' nodes or previously encountered

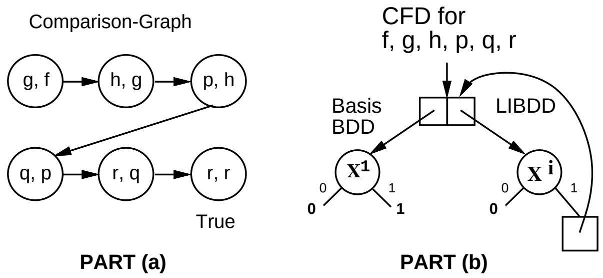

Figure 4: Comparison-graph and CFD for Example 3

cycle nodes) for a proof of equality of two FDs. Since this argument is built in into our method for checking equality, it allows us to avoid an explicit reasoning by induction for verification purposes. □

For our Example 3, we obtain the comparison-graph as shown in Figure 4, Part(a). Since there is no 'False' node in the graph, all nodes (g, f), (h, g), (p, h), (q, p), and (r, q) represent equalities. Thus, FDs for all other LIFs (g, h, p, q, r) are equal to the canonical FD for f, and our task is complete, with the final canonical FD (CFD) as shown in Figure 4, Part(b).

2.4 Symbolic Manipulation of LIFs

In this section we describe symbolic manipulation of LIFs by use of appropriate operations on their FDs.

- **Boolean Operations**: The result h of a Boolean operation on given LIFs f and g is also an LIF. The canonical representation of h is obtained by the method described (in the previous section) for arbitrary Boolean combinations of user-defined functions. We use an integrated meta-space to both build the initial set of canonical function descriptors and to add Boolean combinations later as needed.
- **Restriction** of an i-instance function f^i with respect to an i-instance variable $x^i = 0/1$, is performed by a standard BDD Restriction operation [4], applied to the Basis BDD or the LIBDD for f. (Note that the leaf nodes of the LIBDD remain unaffected since they represent functions independent of x^i.)
- **Composition** can be represented in several different ways. In case of f^i composed as an i-instance input x^i for g^i, it can be expressed in terms of Boolean operations and restriction [4], as follows:

  $g^i|_{x^i=f^i} = f^i \wedge g^i|_{x^i=1} \vee \neg f^i \wedge g^i|_{x^i=0}$

  We can also handle other useful cases, such as composition of all i-instance inputs ($i \geq 1$) of an LIF with i-instance functions etc.

Note that these manipulations typically require simple extensions of the standard BDD operations on the Basis BDDs and LIBDDs, followed by canonicity maintenance of the resulting (and newly generated) FDs. The complexity of these operations is *independent of* i (the parameter of inductive description), and depends instead on the number of FDs in the system. On the other hand, previous results for functions similar to our LIFs — with bounded circuit widths [1, 15], and iterative systems [16] — indicate that size complexity of the corresponding BDD representations is linear in i.

3 Applications

Since the inductive characterization of LIFs can be used to capture both structural induction in inductively-defined hardware, as well as temporal induction in sequential systems, the applications of symbolic LIF manipulation naturally fall under these two broad categories. In this section, we describe some of these applications, with an emphasis on how they directly follow from our LIF representations.

3.1 Inductively-defined Hardware

Our interest in symbolic LIF manipulation sprang from the goal of formal verification of inductively-defined hardware. The motivation was to combine reasoning by induction and symbolic tautology-checking in a way that would incorporate the advantages of both. Though reasoning by induction is a popular proof method, successfully employed by several research groups [7, 12, 13, 17, 18], most available techniques are semi-automated at best. A complete proof typically requires user interaction and heuristic search. On the other hand, symbolic tautology-checking has been very successful, both for functional verification of combinational circuits and for behavioral verification of sequential circuits [2, 6, 8, 15]. However, these methods have so far been used to verify only fixed-sized instances of circuits, not those defined parametrically in terms of size parameters.

As described in Section 2.3.2, our LIF representations can be regarded as inductive extensions of BDDs, where canonicity is enforced by equality checking *with a built-in principle of induction*. The hope was that by building in the principle of induction into our representations, we would be able to avoid an explicit proof for verification purposes. In fact, a proof by induction for a specification property g, expressed as a Boolean formula on a set of LIFs, naturally translates to a run of $EQ(g, \text{'1'})$. As mentioned earlier, this algorithm automatically takes various induction hypotheses into account in order to prove the induction step. This not only allows us to completely automate a proof by induction, but also eliminates the heuristic search typically associated with more powerful proof-theoretic frameworks. In order to fully realize this potential in practice, we have explored additional parametric circuit representation mechanisms, which allow us to represent common practical circuits as LIFs. These mechanisms have been described in detail elsewhere [11], along with examples that include serial adders, decoders, register files etc. In the same paper, we also describe a verification example for checking the mutual exclusion property of decoder outputs.

This aspect of our work is related to recent efforts by Rho and Somenzi on unilateral iterative systems [16]. In comparison to their linear-sized BDD representations of iterative array outputs, our representations are independent of the size parameter. Furthermore, our method for obtaining LIF canonicity directly gives a check for functional equivalence without converting it to an automata problem. Also, unlike cell automata which implicitly consider all combinations of circuit outputs, we represent only those combinations that are required (in a sort of lazy evaluation style). However, we are still working on extending our results to iterative sequential systems.

3.2 Sequential Systems

Recall that our LIF representation is canonical in terms of the underlying variables, i.e. for a sequential function represented as an LIF, our representation is canonical in terms of the sequences of inputs, *not relying upon the number of states, or a particular state encoding*. This opens up numerous possibilities for application in the verification and synthesis of sequential systems:

- Sequential logic synthesis — Since LIFs represent functions of sequences of inputs, they permit easy computation and storage of input, output, and implicit don't-care sequences for direct use in synthesis.
- FSM input/output equivalence — By representing the state transition functions and the output functions of FSMs as LIFs, input/output equivalence can be ensured by checking equality of the output LIFs.
- Language equivalence/containment — For cases where the language acceptance criterion can be represented as a predicate on the symbolic state vector (represented as an LIF), these problems reduce to LIF equivalence/implication problems respectively.
- Reachability analysis — Reachability of a particular state (or a set of states), can be checked by checking satisfiability of the LIF denoting the corresponding state predicate.
- Symbolic representation of the set of all reachable states — Since the symbolic state vector S can be represented as a vector of LIFs, the characteristic function of its range ($Img(S)$ [8]) provides a symbolic representation of the set of all reachable states.
- Synchronous composition of FSMs/finite state automata — This can be easily obtained through standard manipulations for LIF composition.

These LIF applications highlight interesting parallels between BDD-style manipulations for combinational circuits, and the corresponding LIF manipulations for sequential circuits, as summarized in Table 1. In fact, the correspondence between their applications is a natural consequence of the correspondence between the facts that BDDs represent Boolean functions over a set of inputs, and LIFs represent Boolean functions over a set of *sequences* of inputs. Another interesting point is that though we tend not to think of symbolic state-space analysis in terms of an induction proof, our LIF manipulations capture the precise inductive characterization of such an analysis. This also opens up possibilities for combining structural induction proofs and temporal induction proofs within a unified framework based on symbolic LIF manipulation.

4 Implementation and Results

In this section, we describe the salient features of our preliminary implementation of an LIF package, and provide the results obtained so far.

Conceptually, our LIF package can be viewed as spanning three different Boolean subspaces — Meta-BDDs , Basis BDDs , and LIBDDs — each of which is implemented in the style of a standard BDD package (developed by David Long at CMU). In addition to the standard Boolean manipulations on LIFs, we support two additional operations — *lif_equal(f,g)* (checks if LIF f is equal to LIF g or its negation), and *make_canonical(f)* (checks equality of LIF f against members in the canonical set, adds it to the set if f is distinct, and recursively makes canonical the leaf nodes of the LIBDD for f). We have currently adopted an overall lazy-evaluation approach, i.e. canonicity is not enforced unless requested by the user. (Also, the FD for an LIF is not constructed unless needed.) The reason is that unlike BDDs, where canonicity can be syntactically ensured by construction, canonicity for LIFs involves further effort. This effort may or may not be required for all applications. Thus, the flexibility of enforcing canonicity currently lies with the user.

In the remainder of this section, we present resource requirement results for representing LIFs in canonical form for several circuits (as measured on a Sparc station 1+, with 40M main memory). Since the focus of this paper is on canonical LIF representation and symbolic manipulation, we do not provide results for particular applications. However, we believe that these numbers are a strong indicator of the practicality of our approach, since the canonicity procedure is the core step of all applications outlined in Section 3.

In Table 2 we report results for LIF representation of some common combinational circuits defined parametrically. The table shows both the memory requirements and the CPU time required to obtain canonical LIFs representing the circuit outputs. Since the degree of mutual-recursion in such LIFs is typically small, the effort for enforcing canonicity is minimal, and we get fairly efficient performance. In the same table, we have also shown the memory requirements for a BDD representation of the corresponding 4-bit circuit slice. Note that our requirements are quite modest in com-

BDD Manipulations	LIF Manipulations
Canonicity over inputs	Canonicity over sequences of inputs
Combinational function equivalence	Sequential function equivalence
Combinational don't-care conditions	Sequential don't care conditions (sequences)
Function equivalence	Language equivalence
Function implication	Language containment
Satisfiability	Satisfiability: Reachability
— Function true for some input	— State predicate true for some sequence of inputs
Tautology	Tautology: Invariant properties
— Function true for all inputs	— State predicate true for all sequences of inputs

Table 1: Correspondence of BDD and LIF Manipulations

Circuit	LIF		BDD (4-bit Slice) Mem. (Kb)	
	Time (sec.)	Mem. (Kb)	MSB	All bits
i-bit Adder	0.5	1.1	0.7	1.1
i-bit Decoder	0.5	0.6	0.2	0.8
Register File ($2^i \times j$-bits)				
Cell Outputs	0.6	2.5	0.7	44.3
Read Outputs	0.6	3.5	1.9	7.7

Table 2: LIF Representation of Combinational Circuits

parison, and are especially effective in capturing *all* outputs of multiple-output parametric circuits.

Our results for LIF representation of sequential circuits are not as uniform, which is not surprising since temporal induction in such circuits is rarely as regular as structural induction in combinational circuits. We have summarized these results for the MCNC sequential benchmarks [14] in Table 3. In addition to the CPU time needed to make the output LIFs of these circuits canonical, we also report the number of latches (L), inputs (I), outputs (O), gates (G), unique LIFs (Uniq), and LIF pairs found equal (EQ). The number of unique LIFs gives some measure of the inherent complexity of the problem, since it represents the distinct combinations of states that give rise to the same outputs (needed for canonical representation of the sequential outputs). It can be easily observed that the CPU time correlates well with this measure. The number of LIF pairs found equal is also an interesting statistic as it indicates the number of equalities that would have been missed without an inductive approach to checking equality of sequential functions. In this sense, it justifies the extra effort needed to enforce canonicity.

As can be seen from Table 3, we are currently able to handle most of the MCNC sequential benchmarks. The timing requirements also seem reasonable in practice. At present, we have not used any heuristics for ordering either the parameterized variables (in the Basis BDD and the LIBDD spaces), or the meta-variables (denoting the user-defined LIFs in the meta-space). Since the same parameterized

Circuit	L	I	O	G	Time (sec.)	Uniq LIFs	EQ Pairs
lion	2	2	1	22	0.3	3	1
train4	2	2	1	21	0.3	3	0
dk15	2	3	5	65	0.4	10	0
mc	2	3	5	40	0.4	11	1
tav	2	4	4	34	0.4	7	1
shiftreg	3	1	1	22	0.4	4	0
dk27	3	1	2	35	0.4	13	0
lion9	3	2	1	37	0.3	6	0
ex5	3	2	2	60	0.6	29	2
dk17	3	2	3	57	0.5	27	0
beecount	3	3	4	54	0.4	15	0
dk14	3	3	5	86	0.5	17	3
s27	3	4	1	47	0.4	5	11
s8	3	4	1	36	0.6	1	0
mod12	4	1	1	44	0.6	0	1
bbtas	4	2	2	44	0.4	14	2
train11	4	2	1	50	0.5	14	0
ex3	4	2	2	68	0.7	39	2
ex7	4	2	2	76	1.2	62	3
bbara	4	4	2	70	0.5	15	0
opus	4	5	6	84	88.1	817	200
ex6	4	5	8	113	2.8	92	48
mark1	4	5	16	100	5.5	127	41
s386	4	7	7	120	178.1	1135	84
dk512	5	1	3	73	0.7	32	4
ex2	5	2	2	123	5.3	278	107
cse	5	7	7	163	187.7	1177	508
bbsse	5	7	7	117	51.1	807	52
s1a	5	8	6	273	0.8	0	6
s208	5	11	2	69	2.7	88	47
kirkman	5	12	6	132	58.2	711	455
s420	5	19	2	66	1.7	46	67
dk16	6	2	3	263	36.1	423	525

Table 3: Results for MCNC Sequential Circuits

variable ordering is used at every level of recursion in our equality-checking, the problems with a bad ordering get compounded in our case, resulting in a potential explosion. We hope to either manually input good orderings, or include heuristics into our system soon, which would help in handling the remaining MCNC benchmarks and larger circuits like the ISCAS benchmarks.

5 Conclusions and Future Directions

We have presented a canonical representation and symbolic manipulation algorithms for a class of inductive Boolean functions called LIFs. This class of functions is useful in capturing the inductive characterization of several practical combinational circuits, as well as sequential circuits. In the former case, our representation provides a fixed-sized representation for all size instances of a parametric circuit. In the latter case, it provides a canonical representation for a sequential function, in much the same way as a BDD provides a canonical representation for a combinational function. We also describe a preliminary implementation for supporting LIF manipulations, and report results on its practical feasibility for both classes of circuits.

The significance of our method for obtaining canonical LIF representations is that it is based upon an implicit inductive argument, which immediately lends itself to several applications in the area of verification. It allows complete automation of proofs by induction, both for functional verification of combinational circuits, as well as for behavioral verification of sequential circuits. The canonical sequential function representation also has potential applications in other areas such as logic synthesis and optimization.

In terms of future directions, we are planning to study the effects of variable orderings within our LIF implementation, especially since the problem seems to be as critical as it is for BDD-based manipulations. This will enable us to explore interesting practical applications. We are also working on other classes of inductive Boolean functions, where the underlying manipulation principles are similar to those presented here for LIFs. Another interesting direction is the use of our representations to reason about functional equivalence of different inductive descriptions, e.g. equivalence of a serial parity circuit and a parallel tree parity circuit. Preliminary work indicates that by deliberate manipulation of the two representations, it may be possible to automate a proof of their equivalence. Such proofs are of practical importance in verification of correctness preserving transformations, an essential component of most automated high-level synthesis tools.

References

[1] C. L. Berman. Ordered Binary Decision Diagrams and circuit structure. In *Proc. IEEE Int. Conf. on Comp. Design*, pages 392–395. IEEE Comp. Soc. Press, 1989.

[2] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In L. M. J. Claesen, editor, *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, volume II, pages 151–158. North-Holland, 1990.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conf.*, pages 40–45. IEEE Comp. Soc. Press, June 1990.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, August 1986.

[5] R. E. Bryant. Symbolic Boolean manipulation with Ordered Binary Decision Diagrams. Technical Report CMU-CS-92-160, School of Comp. Sci., Carnegie Mellon Univ., July 1992. Also to be published in ACM Computing Surveys.

[6] J. Burch, E. M. Clarke, K. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th ACM/IEEE Design Automation Conf.*, pages 46–51. IEEE Comp. Soc. Press, June 1990.

[7] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. North-Holland, 1987.

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373. Springer-Verlag, 1990.

[9] A. J. de Geus. High level design: A design vision for the 90's. In *Proc. IEEE Int. Conf. on Comp. Design*, page 8. IEEE Comp. Soc. Press, 1992.

[10] A. Gupta and A. L. Fisher. Representation and manipulation of inductive Boolean functions. Technical Report CMU-CS-92-129, School of Comp. Sci., Carnegie Mellon Univ., April 1992.

[11] A. Gupta and A. L. Fisher. Parametric circuit representation using inductive Boolean functions. In *Proceedings of the Conference on Computer-Aided Verification*, June 1993.

[12] W. A. Hunt, Jr. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–129. North-Holland, 1987.

[13] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proc. 8th Ann. ACM Symp. on Principles of Dist. Computing*, pages 239–247. ACM Press, 1989.

[14] R. Lisanke, editor. *FSM Benchmark Suite*. Microelectronics Center of North Carolina, Research Triangle Park, North Carolina, 1987.

[15] K. L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem.* PhD thesis, School of Comp. Sci., Carnegie Mellon Univ., May 1992.

[16] J.-K. Rho and F. Somenzi. Inductive verification of iterative systems. In *Proc. 29th ACM/IEEE Design Automation Conf.*, pages 628–633. IEEE Comp. Soc. Press, June 1992.

[17] D. Verkest, P. Johannes, L. Claesen, and H. De Man. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proc. European Design Automation Conf.* IEEE Comp. Soc. Press, 1990.

[18] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80. Springer-Verlag, 1990.