# Tradeoffs in Canonical Sequential Function Representations*

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

## Abstract

*State space exploration is of prime importance in the study of finite state sequential systems, with several efforts aimed at compact representation of the state space in order to tackle the state explosion problem. In our work with formal verification of inductively-defined hardware, we have identified a useful class of Boolean functions called Linearly Inductive Functions (LIFs). In this paper we explore the relationship between our LIF representation and the classic DFA (deterministic finite state automaton) representation of a sequential function, and examine the associated tradeoffs. We show that our LIF representation corresponds to a minimal reverse DFA, i.e. a minimal DFA which accepts the language consisting of the reverse input strings, where our implicit method for obtaining an LIF representation does not require explicit construction of a forward DFA. Its practical usefulness arises from our demonstration that reverse DFAs for several datapath circuits are exponentially more compact than the classic forward DFAs. Furthermore, in comparison to the traditional representation of DFAs as state transition diagrams, our LIF representations allow for state-sharing while maintaining decomposition, resulting in memory savings in practice.*

## 1 Introduction

State space exploration is of prime importance in the study of finite state sequential systems. It forms the basis of numerous verification techniques such as model checking, input/output equivalence, language containment, reachability analysis etc. Significant effort has therefore been spent on improving its practical performance, and towards tackling the state explosion problem using symbolic methods [2, 4, 5, 6, 12, 14]. In this paper, we describe the relative advantages of our symbolic state-space representation in comparison to the standard deterministic finite state automaton (DFA) representation of sequential systems. Potentially, these advantages can be exploited in areas other than formal verification.

This state-space representation is part of our recent work on verification of inductively-defined hardware, based on canonical representation and automatic symbolic manipulation of classes of inductive Boolean functions [7, 8]. One such class, similar to unilateral iterative arrays [11], is called Linearly Inductive Functions (LIFs). It is useful in capturing induction both in parametric hardware (regular in structure) and in finite state sequential systems (regular in time). In the latter case, our LIF representation provides an encoding-independent canonical representation for a sequential function, in much the same way as a Binary Decision Diagram (BDD) provides a canonical representation for a combinational function [3].

On the other hand, a minimal DFA can also be regarded as a canonical representation of a sequential function, where acceptance/rejection on an input string denotes Boolean value '1'/'0' respectively [10]. Therefore, the natural questions are: *What relationship, if any, is there between our LIF representation and a classic minimal DFA representation of a sequential function? What are the associated tradeoffs?*

In this paper, we show that our LIF representation corresponds to a minimal *reverse* DFA, i.e. a DFA which accepts the reverse input strings. The direct relationship between a classic DFA and a reverse DFA has been studied by other researchers, where the reversal is done using standard automata techniques [15]. However, our LIF representation for a sequential function[1] is significantly different from a classic DFA. First, it forms part of a more general parametric representation for multi-dimensional linearly inductive Boolean functions [7]. Second, our technique for obtaining an LIF representation differs greatly from other symbolic, as well as non-symbolic, automata techniques [8].

The interesting aspect of this research is that a reverse DFA can be exponentially more compact than the forward DFA in some cases. Though this fact is easy to understand in theory, yet no effort has been made to exploit its potential in practice. As we shall show in this paper, several practical datapath circuits such as register files, shift registers, stacks, FIFOs etc. exhibit this exponential compaction when represented as reverse DFAs. Thus, our LIF representations for such circuits offer a distinct advantage over classic DFA representations. Since the problem of state explosion generally becomes worse in the presence of such datapath circuits, due to the multiplicative effect of individual components, this reduction in state space helps in making the exploration task tractable.

Though it is useful to understand the conceptual correspon-

*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.

[1] In the rest of this paper, we will use "LIF representation" to mean "LIF representation for a sequential function".

dence between an LIF representation and a reverse DFA, there are key differences with respect to the traditional representation of a DFA as a state transition diagram. Apart from the standard compactness advantages of a symbolic representation, our LIF representations allow simultaneous exploitation of both decomposition and state-sharing naturally. This is in contrast to traditional DFA representations where one can exploit only one or the other.

Finally, we also highlight the relationship of symbolic LIF manipulations to other verification techniques. The algorithms for building and manipulating canonical LIF representations provide a framework for automatic symbolic manipulation of sequential functions. These manipulations can be used for various verification tasks such as checking FSM input/output equivalence (since the canonicity property of the LIF representation does not rely on the number of states or a particular state encoding), checking language equivalence/containment, obtaining the set of reachable sets etc. The interesting point is that though we do not tend to view symbolic state-space analysis in terms of an induction proof, yet LIF manipulations capture precisely the inductive characterization of such an analysis.

We start by reviewing the LIF representation of sequential functions [8].

## 2 LIF Representations

Consider the state transition diagram of a standard 2-bit counter as shown in Figure 1. The state transition functions $s1, s2$, and the output functions $o1, o2$ can be written in LIF form[2] as follows, where $e$ is the count enable input:

for $t = 1, s1^1 = o1^1 = 0$,

for $t > 1, s1^t = o1^t = (\neg e^t \wedge s1^{t-1}) \vee (e^t \wedge \neg s1^{t-1})$,

for $t = 1, s2^1 = o2^1 = 0$,

for $t > 1, s2^t = o2^t = (\neg e^t \wedge s2^{t-1}) \vee (e^t \wedge (s2^{t-1} \oplus s1^{t-1}))$

The LIF representations for these sequential functions are shown in Figure 2. Note that each LIF is represented by a function descriptor (FD) consisting of a tuple of structures – a Basis BDD (which represents the basis instance of the function, i.e. at time $t = 1$), and an LIBDD (Linearly Inductive BDD, which represents the inductive instance of the function, i.e. at time $t$). The Basis BDD is a standard BDD on variables that denote inputs at time $t = 1$. The LIBDD is also like a BDD, with inputs at time $t$ serving as variables (shown as circular decision vertices), except that the leaves (shown as boxes) need not represent Boolean constants alone. They may contain pointers to other FDs, with an implicit parameter substitution of $(t-1)/t$, indicating the contribution of LIFs at time $(t-1)$. (As with BDDs, dark circles on pointers indicate negation.) The FDs are made canonical by following a two phase algorithm. In the first phase, called the Generation Phase, new LIFs are

[2]The LIF definition is included in the Appendix; details appear in other papers [7, 8].

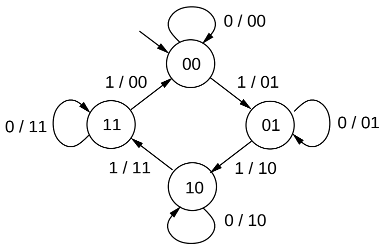


Figure 1: State transition diagram for a 2-bit counter

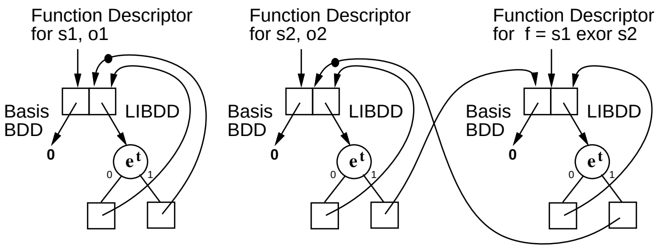


Figure 2: LIF representations for a 2-bit counter

generated in order to represent distinct combinations of the user-given LIFs, e.g. the function $f$ denoting the combination $s1 \oplus s2$ in our example. In the second phase, called the Comparison Phase, an FD is added to the canonical set if it is found unequal to each member of the set through use of an explicit equality check. This check involves recursively checking isomorphism of the LIBDDs, combined with a syntactic check on the Basis BDDs, and allows the final canonical form to be independent of the state encoding of the state transition description. The three FDs shown in Figure 2 are in the final canonical form.

## 3 LIF Representations and DFAs

Let $DFA(\mathcal{L})$ be a DFA that accepts a regular language $\mathcal{L}$. Also for a language $\mathcal{L}$, define $\mathcal{L}^R$ to be the reverse language, i.e. $\mathcal{L}^R = \{x^R | x \in \mathcal{L}\}$, where $x^R$ denotes the reverse of string $x$. For example, for a regular language $\mathcal{L} = (0+1)^* 1 (0+1)^k$, the reverse language $\mathcal{L}^R = (0+1)^k 1 (0+1)^*$. If $\mathcal{L}$ is regular, then so is $\mathcal{L}^R$, and let $DFA(\mathcal{L}^R)$ accept the language $\mathcal{L}^R$. Also, let a sequential function $l$ denote a regular language $\mathcal{L}$, where $l$ is allowed to use any state encoding. We can now state the following:

**Theorem**: *LIF representation of* $l \equiv$ *minimal* $DFA(\mathcal{L}^R)$.

The proof follows from the observation that a traversal down the LIF representation for $l$, is the same as following a transition path through a DFA, where:

- FDs are regarded as states (for a finite number of user-given LIFs, i.e. transition functions, there can be only a finite number of LIFs denoting the Boolean combinations, resulting in a finite number of FDs),
- variables of the LIBDDs are regarded as inputs, and paths through LIBDDs are regarded as labeling the inputs,

- pointers from LIBDD leaf nodes to FDs are regarded as transitions to new states,
- the FD for $l$ is regarded as the starting state, and
- the Basis BDD of the last FD in the traversal is regarded as the final acceptance condition.

For example, consider the 2-bit counter example in Figure 2. Recall that the LIBDD for the output $o2$ represents the $t$-instance of $o2$ ($o2^t$) inductively. Thus, for a given input string $x = x^1 x^2 \ldots x^t$, the value of $o2^t$ is obtained by following the LIBDD, branching on the input value $e^t = x^t$, and arriving at a LIBDD leaf node. Suppose $x^t = 1$, and we arrive at the leaf node with a pointer to the FD for $f$. Recall that, through the parameter substitution mechanism, this leaf node represents $f^{(t-1)}$. Again, the value of $f^{(t-1)}$ is obtained by following the LIBDD for $f$, branching on the input value $e^{(t-1)} = x^{(t-1)}$, and following its leaf node pointer. This process is repeated until the Basis BDD provides the final Boolean constant value depending on the basis input $e^1 = x^1$. This Boolean constant denotes the value of the output $o2^t$ on the input string $x$, where a '1' indicates acceptance, and a '0' indicates rejection, by the DFA.

Therefore, a traversal down the LIF representation is like a DFA traversal leading to acceptance or rejection of the input string. However, note from the example that we have consumed the last input first, i.e. the input string is followed in the reverse order $x^t x^{(t-1)} \ldots x^2 x^1$ when starting from the FD for $l$. Thus, the LIF representation for $l$ corresponds to a DFA which accepts $\mathcal{L}^R$, i.e. LIF representation for $l \equiv DFA(\mathcal{L}^R)$. Furthermore, since the FDs (states of $DFA(\mathcal{L}^R)$) are ensured to be pairwise non-equivalent by the LIF canonicity algorithm, this reverse DFA is minimal. This discussion also highlights the fact that our method for obtaining a canonical LIF representation can be regarded as an implicit method for obtaining a reverse DFA, in that *it directly uses state transition equations and does not rely on an explicit representation of the forward DFA*. Though the resulting reverse DFA is explicit, it is very useful to avoid construction of the forward DFA, especially for circuits where the reverse DFA is known to be more compact (described later in this paper).

### 3.1 Relationship with DFA State Minimization

The minimality of the reverse DFA corresponding to an LIF representation follows directly from the latter's canonicity property. Exploring this correspondence further, we can regard the Comparison phase of our LIF canonicity algorithm as a state minimization procedure, where each FD is checked for equality against members of the canonical set. However, our overall LIF canonicity algorithm is different from the standard minimization algorithms [1, 9, 10] in several respects. First, since we start with state transition equations, we do not have to deal with an explicit unminimized state space. The Generation phase of the algorithm, which has no counterpart in standard algorithms, generates only as many explicit LIFs (states) as are needed for a particular output and captures canonicity with respect to user-defined LIFs [8]. This phase is followed by the Comparison phase, which involves explicit equality checks between pairs of LIFs. Thus, complexity of the comparison phase is $O(n^2)$, where $n$ is the number of LIFs at the end of the generation phase, not the number of unminimized states. Second, the entire algorithm is incremental in nature, relying on reuse of previous results, and allows interleaving of the Generation and Comparison phases. This is consistent with the aim of supporting symbolic manipulation of a dynamic system of sequential functions, where more functions are added as more tasks are performed. Again, the standard algorithms, as stated, are useful for one-time minimization and not for incremental operation.

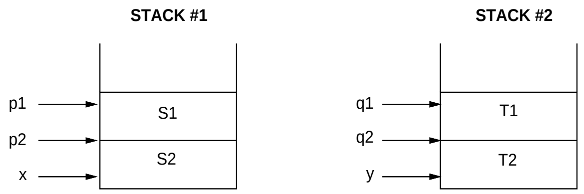

Figure 3: System of two stacks

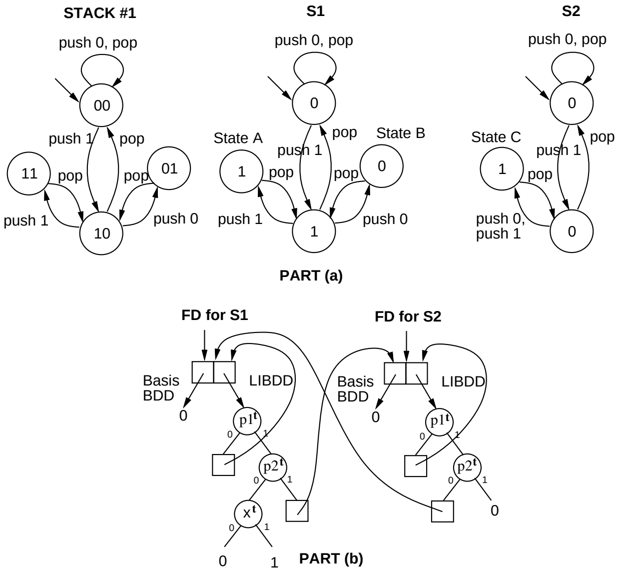

Figure 4: Stack Example

### 3.2 Comparison with State Transition Diagrams

Though LIF representations conceptually correspond to DFAs, they exhibit significant advantages in comparison to traditional state transition diagram representations of DFAs. As remarked earlier, our LIF representation supports symbolic manipulation of a dynamic system of sequential functions. Since the emphasis is on representation of individual functions, we naturally use the advantages of decomposition, i.e. representing only the relevant parts of a state-space per output. The same effect can be achieved with a state transition diagram representation also, by minimizing the state space per output. However, unless the system is naturally decomposable, one loses the shared information in this process. Thus, in the traditional DFA representation, one can exploit either decomposition or state-sharing, but not both[3]. In the LIF representation, on the other hand, the minimal states (FDs) are tagged by the Boolean combinations (meta-BDDs) of the state transition variables (user-given LIFs) that they represent [8]. This information is enough to allow state-sharing among individual output representations.

As an illustration, consider a system with two 2-bit deep stacks as shown in Figure 3. (We consider two 2-bit stacks for exposition only, the argument holds for arbitrary $n$-bit deep stacks). Each stack has two state bits ($s1, s2$ and $t1, t2$ for Stacks #1 and #2, respectively), one data input ($x$ and $y$, respectively) and two control inputs ($p1, p2$ and $q1, q2$, respectively — $00, 01$:no-op, 10:push, and 11:pop). Suppose both stacks are well-behaved, i.e. the stack is never pushed when full and never popped when empty. Clearly, a minimal DFA representation of the system has 16 states. However, one can minimize the state space per stack, as also per state bit, as shown in Figure 4 Part(a) for Stack #1, and individually for state bits $s1$ and $s2$. (Stack #2, $t1$ and $t2$ are similar.) Note that minimization per stack results in a reduction due to the natural decomposition between the two stacks. However, when minimization is performed per state bit, some state-sharing information is lost. For example, State $C$ of $s2$ = State $A \vee$ State $B$ of $s1$, i.e. on any input sequence State $C$ is reached in $s2$ whenever either State $A$ or State $B$ is reached in $s1$. Such information is useful not only for sharing data structures in the representation, but also for symbolic manipulation of sequential functions. What one would like is a way to exploit decomposition without sacrificing the advantages of state-sharing. This is provided by our LIF representation, shown for the entire system in Figure 4 Part(b). Note that FDs for the two stacks are naturally decoupled, while allowing state-sharing among state bits of the same stack.

## 4 Forward vs. Reverse DFA

Returning back to the conceptual level, given that our LIF representation corresponds to a minimal reverse DFA, the next question is: How does this compare with a minimal *forward* DFA, i.e. the minimal classic DFA?

In theory, a reverse DFA can be exponentially more compact (or, bigger) than a forward DFA. The reason is easily understood as follows. Imagine obtaining the reverse DFA from a forward one, by reversing all edges in the state transition diagram, and exchanging initial states with final states. However, this process may lead to nondeterminism, e.g.

[3] without doing significantly extra work

| Circuit | Latches | CPU Time (sec.) | Unique LIFs | Sharing Quotient |
|---|---|---|---|---|
| dk27 | 3 | 1.1 | 13 | 0.76 |
| dk17 | 3 | 1.2 | 27 | 0.43 |
| lion9 | 3 | 0.9 | 6 | 1.00 |
| beecount | 3 | 1.4 | 15 | 0.50 |
| dk14 | 3 | 1.5 | 17 | 0.35 |
| dk512 | 4 | 1.3 | 36 | 0.50 |
| ex3 | 4 | 1.3 | 39 | 0.52 |
| ex6 | 4 | 2.0 | 68 | 0.15 |
| reg file | 16 | 0.7 | 16 | 1.00 |
| reg file | 64 | 1.9 | 64 | 1.00 |
| reg file | 256 | 24.7 | 256 | 1.00 |
| reg file | 1024 | 458.1 | 1024 | 1.00 |
| shift reg | 16 | 0.3 | 16 | 0.06 |
| shift reg | 64 | 0.5 | 64 | 0.02 |
| shift reg | 256 | 4.7 | 256 | 0.01 |
| shift reg | 1024 | 123.4 | 1024 | 0.01 |
| stack | 11 | 0.4 | 16 | 0.20 |
| stack | 36 | 1.2 | 64 | 0.06 |
| stack | 135 | 15.3 | 256 | 0.02 |
| stack | 521 | 247.5 | 1024 | 0.01 |
| FIFO | 11 | 0.8 | 15 | 0.23 |
| FIFO | 36 | 1.7 | 63 | 0.06 |
| FIFO | 135 | 19.1 | 255 | 0.02 |
| FIFO | 521 | 282.9 | 1023 | 0.01 |

Unique LIFs: total # LIFs with state-sharing
Sharing Quotient: # LIFs with/without state-sharing

Table 1: LIF Representation of Sequential Circuits

two edges into a state on the same input will now become outgoing edges on that input. Further determinization may therefore involve an exponential blow-up. Note that an exponential blow-up in one direction implies an exponential compaction in the other. For example, consider the language $\mathcal{L} = (0+1)^*1(0+1)^k$, i.e. all $0/1$ strings where the last $(k+1)^{th}$ input is a 1. A minimal $DFA(\mathcal{L})$ requires at least $2^k$ states, since it needs to check all possible $k$-length strings after a 1. Now consider the reverse language $\mathcal{L}^R = (0+1)^k1(0+1)^*$, i.e. all $0/1$ strings where the first $(k+1)^{th}$ input is a 1. Clearly a minimal $DFA(\mathcal{L}^R)$ requires only $k+3$ states in order to check the $(k+1)^{th}$ input and accept/reject. Thus, the reverse DFA in this example is exponentially more compact than the forward DFA.

Apart from the theoretical insight, it is instructive to examine this issue in practice also. Results for obtaining LIF representations for some example circuits are shown in Table 1, with measurements on an SGI Indigo with 32 Mb. main memory. In addition to the CPU time needed, we also report number of latches, unique LIFs, and sharing quotient. The number of unique LIFs is the total number of LIFs (minimal states) needed for a representation of all outputs with state-sharing. The sharing quotient indicates the quotient of the number of LIFs with, and without, state-sharing between outputs. A low number indicates a large amount of state-sharing. The examples in the top part of the table are from the MCNC sequential benchmarks [13] consisting largely of controllers, and those in the bottom part are 1-bit slices of simple gate-level datapath circuits. As can be seen from the number of unique LIFs, there is a blow-up in most of the MCNC circuits[4], i.e. unique LIFs $> 2^{Latches}$. On the other hand, for the datapath circuits, the number of unique LIFs is linear in the circuit size. Since the minimal classic DFA for these datapath circuits is $O(2^{Latches})$, the LIF representation offers an exponential compaction in these cases, which translates to substantial savings for the circuit sizes usually encountered in practice. In most examples, we can also see the benefits of state-sharing between outputs, as evidenced by the sharing quotient column.

One possible explanation for the observed difference between these controller and datapath circuits can be understood by drawing a parallel between LIF representations and BDDs. LIF representations can be viewed as layered BDD-like graphs, with the layer of $t$-instance inputs placed above the layer of $(t-1)$-instance inputs. It is well known that smaller BDDs are obtained by placing the more "important" variables (with more fan-out, controlling variables etc.) first in the variable ordering [3]. Translating this requirement for LIFs implies that smaller LIF representations are obtained if $t$-instance inputs are more critical than $(t-1)$-instance inputs, i.e. if outputs depend more critically on more recent inputs than on less recent inputs. It is quite likely that this temporal preference is not exhibited by the controller circuits experimented with, thereby resulting in bigger LIF representations. On the other hand, the datapath circuits shown exhibit this temporal preference, e.g. for a FIFO of length $k$, the last $k$ data inputs are more relevant. Thus, the LIF ordering is well-suited for these circuits, resulting in compact representations.

## 5 Related Work

As explained in the previous sections, LIF representations correspond to reverse DFA representation of sequential systems. Therefore, a traversal of these LIF representations (either an explicit depth-first traversal, or an implicit breadth-first traversal) amounts to backward traversal of the state space. In this respect LIF-based verification techniques are different from other symbolic techniques that are based on forward traversal of the state space [4, 5, 14]. The latter typically start from the initial state, and implicitly obtain the symbolic representation of the next-states satisfying a particular property. On the other hand, LIF-based verification techniques typically start from a sort of final state satisfying some property, and work backwards implicitly through the transitions.

[4] Though the number of unique LIFs shown here is the total for all outputs, we observed a blow-up in the numbers for individual outputs too, thereby indicating that lack of decomposition is not its primary cause.

| Circuit | Filkorn's Equivalence (Sun 3/60) Time(sec.) | LIF Manipulations (SGI Indigo) | |
|---|---|---|---|
| | | Equivalence Time (sec.) | Canonicity Time (sec.) |
| Counter 2 | NA | 0.3 | 0.3 |
| Counter 4 | NA | 0.3 | 0.3 |
| Counter 8 | NA | 0.4 | 2.7 |
| Counter 16 | 21 | 0.8 | — |
| Counter 32 | 66 | 3.9 | — |
| Counter 64 | 300 | 30.7 | — |

Table 2: Results for Counter Circuits

In terms of other techniques based on symbolic backward traversal [6, 12], we have found a strong similarity between them and our LIF-based method of checking I/O equivalence. In fact, an implicit breadth-first version of our LIF equality checking algorithm is operationally the dual of Filkorn's method based on computing non-equivalent classes of states [6]. However, at the higher level, it is easy to view the LIF method as an induction proof, which uses a general purpose equality checking algorithm with basis conditions and induction hypotheses [8]. It does not require a reasoning in terms of states and transitions. Furthermore, our algorithm saves partial results that are needed for manipulating and obtaining canonical representations for LIFs. Thus, its use is not limited to a single verification task, but admits a general framework for automatic symbolic manipulation of LIFs, including sequential functions.

Again, as in the case of canonical representations, LIF-based backward traversal verification techniques perform better for sequential systems with compact reverse DFAs. Also, since backward traversal is affected by the unreachable part of the state space, unlike forward traversal that depends only on the reachable part, these techniques work well for systems with small (or nil) unreachable parts, e.g. counters. The same factors may also explain Filkorn's observation that his method can handle certain classes of synchronous circuits better than methods based on forward traversal.

In Table 2, we have shown practical results for our LIF-based methods on counter circuits of various sizes. As can be seen from the second and third columns, our method for equivalence checking is competitive with Filkorn's (taking into account the relative CPU speeds). Note also from the fourth column that this is a relatively inexpensive operation in comparison to obtaining canonical representations (which we were able to accomplish only up till a size of 12 due to memory limitations).

Another related work has used the reverse DFA for generating linear-sized BDD representations for outputs of iterative arrays [15]. However, it uses a direct method for obtaining a reverse DFA from an explicit forward DFA, unlike our implicit method which requires only the state transition equations. Furthermore, since their focus was on inductive verification, the compactness advantages of reverse DFA representations for practical circuits were not examined. As for verification, there is similarity in that automata equivalence is the common underlying principle in checking equivalence of unilateral combinational iterative arrays (which correspond to our single-dimensional LIFs). However, we have not handled sequential iterative arrays yet, for which they have established several interesting results.

## 6 Conclusions and Future Directions

Linearly Inductive Functions (LIFs) is a useful class of functions that can capture both structural and temporal induction in digital systems. In this paper, our focus was on the relationship between an LIF representation and a classic DFA representation of a sequential function, where we showed that our LIF representation corresponds to a minimal reverse DFA. In fact, our canonicity algorithm for obtaining an LIF representation provides an implicit method for obtaining a reverse DFA from state transition equations, without the explicit construction of a forward DFA. Our experiments show that the LIF (reverse DFA) representation is especially useful in handling datapath circuits such as register files, shift registers, stacks, FIFOs etc., because it exhibits exponential compaction in comparison to the classic (forward) DFA representation in these cases. Our experiments also support the memory savings advantage of simultaneously exploiting decomposition and state-sharing with our LIF representations, in comparison to the traditional representation of DFAs as state transition diagrams.

The correspondence between LIF representations and reverse DFAs allows several useful insights. Verification techniques based on traversal of the LIF representations can be viewed as performing backward state-space traversal. Similar to the case of forward traversal, this can be done either explicitly (using depth-first techniques) or implicitly (using symbolic breadth-first techniques). Given the different factors affecting forward and backward traversal, it should be interesting to examine ways of combining both within a single technique. Apart from its direct application to formal verification, the LIF representation is part of a general framework for symbolic manipulation of sequential functions, with potential in other areas such as synthesis, simulation and testing.

## References


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

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

[4] J. Burch, E. M. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *Proc. 5th Ann. IEEE Symp. on Logic in Comp. Sci.*, pages 428–439. IEEE Comp. Soc. Press, June 1990.

[5] 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.

[6] T. Filkorn. A method for symbolic verification of synchronous circuits. In D. Boorione and R. Waxman, editors, *Proc. 10th Int. Symp. on Comp. Hardware Description Lang. and their Applications*. IFIP, North-Holland, April 1991.

[7] A. Gupta and A. L. Fisher. Parametric circuit representation using inductive Boolean functions. In *Proceedings of the Conference on Computer-Aided Verification*, volume 697 of *LNCS*, pages 15–28. Springer-Verlag, June 1993.

[8] A. Gupta and A. L. Fisher. Representation and symbolic manipulation of linearly inductive Boolean functions. In *Proc. IEEE Int. Conf. on Comp.-Aided Design*, pages 192–199. IEEE Comp. Soc. Press, November 1993.

[9] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computation*, pages 189–196. Academic Press, New York, 1971.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, Reading, MA, 1979.

[11] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.

[12] S. Krischer. The backward walk approach in FSM verification. In *Proc. Int. Symp. on Comp. Hardware Description Lang. and their Applications*. IFIP, North-Holland, April 1993.

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

[14] K. L. McMillan. *Symbolic Model Checking, An approach to the state explosion problem*. PhD thesis, School of Comp. Sci., Carnegie Mellon Univ., Pittsburgh, PA, May 1992. Also to be published in ACM Distinguished Dissertation Series.

[15] 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.


## Appendix

**LIF Definition**: A Linearly Inductive (Boolean) Function (LIF) $f$ is a parametric Boolean function, defined inductively in terms of a parameter $i$ as follows:

- 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.