Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation

Manish Pandey, Randal E. Bryant

Abstract—**We describe the use of symmetry for verification of transistor-level circuits by Symbolic Trajectory Evaluation (STE). We present a new formulation of STE which allows a succint description of symmetry properties in circuits. Symmetries in circuits are classified as *structural symmetries*, arising from similarities in circuit structure, *data symmetries*, arising from similarities in the handling of data values, and *mixed* structural-data symmetries. We use graph isomorphism testing and symbolic simulation to verify the symmetries in the original circuit. Using *conservative approximations*, we partition a circuit to expose the symmetries in its components, and construct reduced system models which can be verified efficiently. Introducing *X-drivers* into switch-level circuits simplifies the task of creating conservative approximations of switch-level circuits. Our empirical results show that exploiting symmetry with conservative approximations can allow one to verify systems several orders of magnitude larger than otherwise possible. We present results of verifying Static Random Access Memory circuits with up to 1.5 Million transistors.**

Keywords—**Formal Verification, Symbolic Trajectory Evaluation, Transistor-Level, Switch-Level, Memory Arrays, Symmetry.**

I. Introduction

Many high performance hardware designs are custom designed at the transistor-level to optimize their area and performance. This makes it necessary to verify them directly at the transistor-level. Verification at this level of abstraction allows one to handle many low-level circuit modeling issues such as bidirectional pass transistors, transistor sizes, node capacitances, and charge sharing. Common examples of such custom-designed hardware units include memory arrays which are found in instruction and data caches of microprocessors, cache tags, and TLBs. Many of these hardware units consist of components which have considerable regularity in their structure, or in the way they handle data. This paper focuses on formalizing this regularity as *symmetry* and shows how symmetry can be used with symbolic trajectory evaluation (STE) to efficiently verity large designs.

Past efforts on verifying memory arrays using STE have been quite successful. Results have been presented on the verification of large systems containing over 10^5 transistors, and over 10^4 memory bits [19], [6]. However, these approaches do not scale up well for much larger systems. As results in this paper show, switch-level analysis can become a serious bottleneck in a STE verification flow that does not utilize symmetry.

By exploiting symmetry with the use of STE, it is possible to verify systems that are orders of magnitude larger than previously possible. We present empirical results for the verification of static random access memory (SRAM) circuits of varying sizes, including one with over 1.5 million transistors. Furthermore, empirical results show that our techniques scale up linearly or sub-linearly with SRAM size, indicating that one could verify circuits much larger than our benchmarks.

Our verification approach builds on three ideas: *circuit partitioning*, *structural analysis*, and *conservative modeling*. Many systems, viewed as a whole, do not possess symmetries that can easily be exploited, but they are made up of smaller components which can. One can exploit the symmetry in the components by partitioning the larger system, verifying the smaller components, and composing the verification results.

We describe two forms of symmetries. *Structural symmetries* arise from similarities in the structure of a system, e.g., by replication of system components. *Data symmetries* arise from similarities in handling of data values in the system. Previous work exploiting symmetry in formal verification in [7], [12] has focussed only on aspects of structural symmetry. Work in in [16] handles structural and a form of data symmetry. Aggarwal et. al describe a form of data symmetry in [1]. Our work contributes a new formulation of data symmetry, and the idea of mixed structural-data symmetry. We have found these other forms of symmetry useful in verifying many common digital building blocks. Consider for example an address decoder, which when given an input address selects the output word having this address. Changing the value of a single decoder input results in a symmetric exchange in the values at the different decoder outputs. Such instances of mixed symmetry cannot be expressed by the other approaches. Structural symmetries in a transistor-level circuit can be detected and verified through a purely *structural analysis* of the system by doing circuit graph isomorphism checks. Other forms of symmetries can be verified through symbolic simulation.

Symmetry in a system imposes a partitioning of the system state space, where permutations of the same state appear in the same partition class. Once symmetry has been checked, one need verify the system for only one representative from each partition class. We exploit this property by constructing a *conservative model* of the system which provides full functionality only for the representative case. This approximation results in large savings in the system excitation function representation size.

In the remainder of this paper, Section II describes past work on applications of symmetry to verification. Section III presents our new formulation of STE, together

Manish Pandey is a research staff member in IBM Austin Research Laboratory, IBM Austin, TX-78758. Randal E. Bryant is the President's Professor of Computer Science in the School of Computer Science at Carnegie Mellon University, Pittsburgh, PA-15213.

with the necessary mathematical background. This section also describes how excitation functions are generated from switch-level circuits. Section IV describes our notion of symmetry as an excitation function preserving transformation. The next three sections describe how symmetry with conservative approximations, and circuit partitioning can be employed to efficiently verify circuits. The application of these techniques to verify a SRAM circuit has been shown in Section VIII. The SRAM verification results are presented in Section IX.

II. Previous Work

Petri nets have long been used to describe systems consisting of communicating concurrent processes. Huber et al. presented early work on exploiting symmetry in petri nets [15]. In [20], Starke proposes an algorithm to compute the generators of the symmetry group for petri nets. Jensen [17] describes the application of symmetry to construct condensed versions of state spaces of concurrent systems described by colored petri nets. The work shows that for reachability properties, the unreduced state space satisfies a property, if and only if the condensed state space satisfies the property. Typically the work in this area focusses on reachability analysis, rather than more general temporal properties, and does not consider the added complexity (and the concomitant payoff) of symbolic state space representations.

Work on exploiting symmetry for automated formal verification techniques is quite recent. Emerson and Sistla [12], [11] show how to exploit symmetry in model checking with the CTL* temporal logic. In a system $\mathcal{M}$ consisting of many isomorphic processes, the symmetry in the system is captured in the group of permutations of process indices defining graph automorphisms of $\mathcal{M}$. Similarly, symmetry in a specification formula f is captured by the group of permutations of process indices that leave f invariant. Given a permutation group G, which is contained in both groups, one can construct a quotient structure $\overline{\mathcal{M}}$, such that for a start state s, and the equivalent state $\overline{s}$ in $\overline{\mathcal{M}}$, $\mathcal{M}, s \models f$ iff $\overline{\mathcal{M}}, \overline{s} \models f$ holds. This is the *correspondence theorem*, which is the central result of their work. This work, however, does not address the complexities that arise from symbolic representations of the state space. Furthermore, this work considers only structural symmetries.

The work by Clarke et al. [7], [8] takes a slightly more general approach than that by Emerson. It views symmetry as a transition relation preserving permutation, not just permutation of indices of identical processes. Given a symmetry group G acting on a Kripke structure M, one can construct a reduced structure M_G. The *correspondence theorem* in this work shows that if there is a CTL* formula f, such that all the atomic propositions in f are invariant under G, the f is true in M if and only if it is true in M_G. This work discusses the construction of the reduced transition relation which is represented symbolically.

There are a number of differences between our work and that by Clarke et al., or Emerson et al. We represent the system transition by means of an "excitation function". Naturally, when we consider symmetry in a system, we examine the symmetry in the excitation function. We model the state of a circuit by associating two "atoms" with each circuit node, a positive atom, and a negative atom (section III-A). Having two atoms for each circuit node allows us to model and exploit a richer set of symmetries in circuits. This contrasts with these other symmetry-based verification approaches which have only one "atom" (atomic proposition) associated with each state holding element in a circuit. Our formulation of symmetry allows us to define structural and data symmetries in a system. We verify that these symmetry properties exist in the system, and then we construct a reduced model from a conservative approximation of the circuit. This avoids the need to construct a model of the complete system and then reducing it by forming a quotient structure. Such an approach allows us to verify huge systems, including one with 262144 state holding elements (or 2^{262144} states). Emerson or Clarke view symmetry as a permutation on a state graph with some desired properties, like preserving the transition relation, or a permutation of process indices which a transition graph automorphism. This approach imposes a significant limitation – capturing phenomena like the symmetrical behavior on the outputs of a decoder is extremely tedious, if not impossible, with their process oriented point of view. Also, in both the approaches outlined above, to be effectively exploited, symmetry must be present in both the state transition graph, and the temporal logic formula being verified — the atomic propositions should be invariant under the action of the symmetry group.

In [16], Ip and Dill discuss the verification of large concurrent systems, where the symmetry in the system is identified by a special scalar-set datatype in the system description language. They view symmetry as an automorphism on the state transition graph. They describe an on-the-fly construction of the reduced state transition graph. They show that safety and liveness properties which hold in the reduced state graph also hold in the original system. Our approach contrasts with this, in that we start with a transistor netlist without any special attributes or datatypes. Of course, once we have this netlist, we manually identify symmetries in the design, and the parts that should be reduced. After this identification, is done, the symmetry checks and the construction of the reduced model are automated. By concentrating on a single instance of the assertion to be verified, we use conservative approximations to aggressively reduce the state space and verify safety properties. In the work by Ip and Dill, the quotient state graph is an exact model that can be used for safety and liveness properties with fairness constraints.

In comparing our work with these previous approaches, one must also take into account the expressiveness of the logic of STE described in this paper with other temporal logic such as CTL. The STE next-time temporal operator allows one to express properties over a finite time period, which equals the maximum depth of the next-time operator. This precludes the specification of liveness properties of systems, which can be specified in CTL. Of course, this

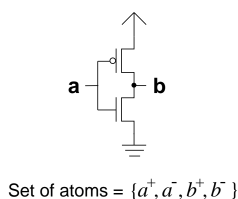

Fig. 1. Set of atoms for an inverter circuit.

expressiveness comes at a price in terms of the size of systems that can be verified.

Aggarwal, Kurshan and Sabnani have exploited symmetry for the verification of the alternating bit protocol, a standard benchmark for protocol verification techniques [1]. Applying reduction techniques specific to this problem, and some simplifying assumptions, they reduce the large state space using machine homomorphisms. They show that for verifying properties about the given state machine, it suffices to verify the desired properties on the reduced machine. This approach to a some degree approximates our notion of data symmetry by considering the data symmetry between 0 and 1 in the transmitted messages. However, the approach does not consider many important issues like detecting, and verifying symmetry, and the automated construction of symmetry reduced state space.

III. Background

A. Model structure and State Domain

Our view of circuits is quite abstract. We view circuits as consisting of a set of nodes and an excitation function which determines how the circuit nodes get updated at every time step. This abstract viewpoint can capture behavior at various levels of abstraction ranging from detailed switch-level models to register transfer level and high-level behavioral models. This section describes the domain of values circuits operate over, and the structure of this domain. Section III-B how the excitation functions are generated and represented.

Intuitively, knowledge about about the state of a circuit is built up of *information atoms.* The state of a circuit consists of values at the circuit nodes. Thus, we need to define atoms associated with every circuit node, as described below.

Definition 1: Let N denote the set of nodes of a circuit. For every node n of a circuit, we define two atoms, n^+ and n^-, indicating that node n has value 1 or 0 respectively. Let $\mathcal{A}$ denote the set of all the atoms of a circuit.

Figure 1 shows the set of atoms for a two node circuit. An atom for a node restricts the value of the node. A set of atoms of a circuit restricts the values on the circuit nodes. For example, the atom set $\{a^+, b^-\}$ indicates a is 1, and b is 0. This motivates the following definition of a circuit state.

Definition 2: Given the set of all the atoms of a circuit, $\mathcal{A}$, we define a circuit state S to be any subset of $\mathcal{A}$, and $\mathcal{S}$ to be the set of all possible states, i.e., $\mathcal{S} = 2^{\mathcal{A}}$.

State set $\mathcal{S}$, together with the subset ordering $\subseteq$ forms a complete lattice, where states are ordered according to their "information content," i.e., how much they restrict the values of the circuit nodes. For example, the structure of the state domain for the circuit in Figure 1 is illustrated in Figure 2. In this diagram we indicate the set of atoms in each state. As the shaded regions indicate, states can be classified as being "partial", "complete", or "conflicting". In a partial state, some nodes have no corresponding atoms while others have at most one. In a complete state, there is exactly one atom for each node. In a conflicting state, there is some node n for which both atoms n^- an n^+ are present. Such a state is physically unrealizable—it requires a signal to be both 0 and 1 simultaneously. Conflicting states are added to the state domain only for mathematical convenience. They extend the semilattice derived from a ternary system model into a complete lattice. Our state lattice has the empty set $\emptyset$ as its least element and the set of all atoms $\mathcal{A}$ as its greatest element. The set union and intersection operations are the *lub* and the *glb* operations, respectively, in the lattice.

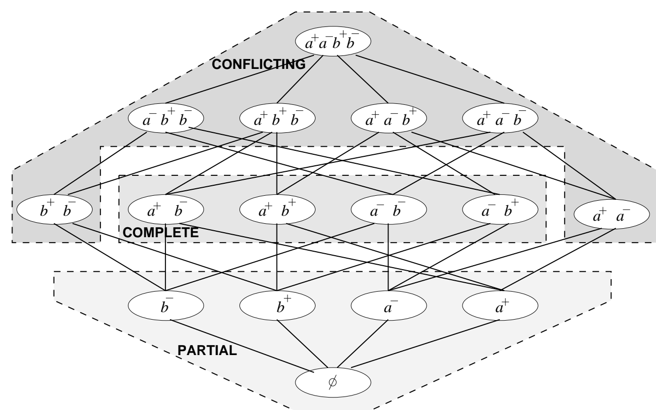

Fig. 2. Structure of State Lattice for Two Node Circuit

Most traditional presentations of switch-level models describe circuit operation over the ternary domain $\mathcal{T}$. Each circuit node takes on one of the three distinct values from the set $\mathcal{T} = \{0, 1, X\}$, where the X value denotes an unknown or an indeterminate value. Associated with $\mathcal{T}$ is the partial order $\sqsubseteq_{\mathcal{T}}$, where $\forall a \in \mathcal{T}.\ a \sqsubseteq_{\mathcal{T}} a$, $X \sqsubseteq_{\mathcal{T}} 0$ and $X \sqsubseteq_{\mathcal{T}} 1$. The partial order $\sqsubseteq_{\mathcal{T}}$ is consistent with the information conveyed by the values in $\mathcal{T}$ since a 0 or a 1 conveys more information than an X in a circuit.

The *atom representation* of the state domain is closely related to the ternary domain. If a circuit state contains n^+, but not n^-, then node n has a value of 1. Similarly, the presence of n^-, and the absence of n^+ implies that n has a value of 0. If the circuit state contains neither n^+, nor n^-, then n has a value of X. For example, in the circuit of Figure 1, the set of atoms $\emptyset$, $\{a^+\}$, $\{b^-\}$, and $\{b^-, a^-\}$, represent the circuit states $(\mathsf{a} = X, \mathsf{b} = X)$, $(\mathsf{a} = 1, \mathsf{b} = X)$, $(\mathsf{a} = X, \mathsf{b} = 0)$, and, $(\mathsf{a} = 0, \mathsf{b} = 0)$, respectively. A circuit state such as $\{b^+, b^-\}$ is a conflicting state. Such conflicting states are mapped to a top element, $\top$, which is added to $\langle \mathcal{T}, \sqsubseteq_{\mathcal{T}} \rangle$ to extend it to a lattice [22].

The behavior of a circuit is defined by its *excitation func-*

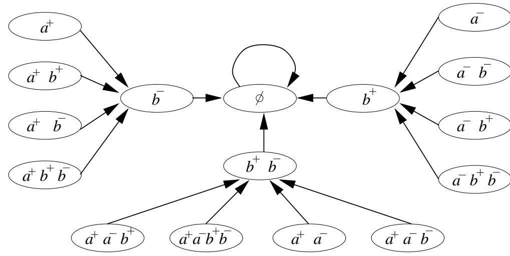

Fig. 3. Excitation function of inverter.

tion $Y: \mathcal{S} \rightarrow \mathcal{S}$. This function serves a role similar to the transition relation or next-state functions of temporal logic model checkers. We require this function to be monotonic over the information ordering, i.e., if two states are ordered $s_1 \subseteq s_2$, then their excitations must also be ordered: $Y(s_1) \subseteq Y(s_2)$. Intuitively, we can view a state as defining a set of constraints on the signal values. We require the excitation function to remain consistent as more constraints are applied. Since input nodes in a circuit are not constrained by the circuit itself, for any state $s \in \mathcal{S}$, $Y(s)$ does not contain any atoms corresponding to the input nodes. Figure 3 shows the excitation function for the inverter of Figure 1. Each oval in Figure 3 is a circuit state. An arrow from one circuit state to another indicates the least constrained new state a circuit can be in after a transition from a previous state. A circuit model is defined by its lattice-structured state-set, and a monotonic excitation function over this state-set. Formally,

Definition 3: A circuit model $\mathcal{M}$ is a tuple $\mathcal{M} = \langle \mathcal{S}, Y \rangle$, where $Y : \mathcal{S} \rightarrow \mathcal{S}$ is a monotonic excitation function.

The behavior of a circuit can be represented as an infinite sequence of states. We define a *circuit trajectory* to be any state sequence $\sigma = \sigma^0 \sigma^1 \ldots$ such that $Y(\sigma^i) \subseteq \sigma^{i+1}$ for all $i \geq 0$. That is, the state sequence obeys the constraints imposed by the circuit excitation function. Below, the sequences σ_1, and σ_2, where $(s)^*$ indicates an infinite repetition of state s, are both trajectories of the inverter of Figure 1.

$$\sigma_1 = \emptyset \quad \{a^+\} \quad \{a^-, b^-\} \quad (\{b^+, a^-\})^*$$
$$\sigma_2 = \{a^-\} \quad \{a^+, b^+\} \quad \{b^-\} \quad (\emptyset)^*$$

As can be seen, these trajectories always obey the constraints imposed by the excitation function of Figure 3 — any state that contains the atom a^+ (a^-, resp.) constrains the succeeding state to contain the atom b^- (b^+, resp.).

$L(\mathcal{M})$ denotes the set of all trajectories of a circuit model $\mathcal{M}$. $L(\mathcal{M}, z)$ denotes the set of all trajectories $\sigma = \sigma^0 \sigma^1 \ldots$ of $\mathcal{M}$ such that $z \subseteq \sigma^0$, i.e., all trajectories which start with a state which is more constrained than z. The set $L(\mathcal{M}, \emptyset)$ equals $L(\mathcal{M})$.

We can extend the $\subseteq$ ordering on elements of $\mathcal{S}$, to sequences of elements of $\mathcal{S}$. This extended ordering is denoted as $\sqsubseteq$. If $\sigma_1 = \sigma_1^0 \sigma_1^1 \ldots$, and $\sigma_2 = \sigma_2^0 \sigma_2^1 \ldots$, then $\sigma_1 \sqsubseteq \sigma_2$ iff for all $i \geq 0$, $\sigma_1^i \subseteq \sigma_2^i$.

B. Creation and Representation of Excitation Functions

The *switch-level model* [3] abstracts digital metal-oxide semiconductor (MOS) circuits as a network of nodes connected together by bidirectional transistor "switches." We employ the technique of symbolic switch-level analysis to generate excitation functions from transistor netlists [4], [5]. This model expresses transistor conductances and node capacitances by discrete strength and size values, and node voltages by discrete states $\{0, 1, X\}$. It can capture many of the important low-level features in MOS circuits such as ratioed, complementary, and precharged logic, and bidirectional pass transistors. Since memory arrays are designed at the transistor-level, switch-level models are particularly appropriate for modeling this class of circuits.

B.1 The Switch-level model

In the switch-level model, a MOS transistor network consists of a set of nodes connected together by transistor switches. Nodes are of two types: *input*, and *storage*. An input node provides strong signals from sources external to the network, like power, ground and data inputs. Storage nodes are internal to the circuit. They have states which are determined by the operation of the network and can retain these states in the absence of applied signals. Each storage node is assigned a size in the set $\{0,...,k\}$ to indicate in a highly idealized way its capacitance relative to other nodes with which it may share charge. The state of a node is represented by one of three logic values: 0, which indicates a low, 1 which indicates a high, and an X which represents an unknown or uninitialized value. Input nodes are assigned a size w. w is greater than the size of all the nodes as well as the strengths of all the transistors in the network.

A MOS transistor is a three terminal device with node connections *gate*, source, and drain. This device acts like a voltage controlled switch, depending on the value at its gate. Normally, there is no distinction between source and drain terminals – the transistor is a symmetric, bidirectional device. We distinguish between three types of transistors: n-type, p-type, and n-type depletion. A transistor acts as a switch between source and drain controlled by the state of its gate node. This switch may be open or closed, or it may have a conductance of unknown value. These three conduction states, open, closed and unknown are represented by the values 0, 1, and X respectively. Each transistor has a strength in the set $\{k+1,...,w-1\}$. The strength of a transistor indicates its conductance when turned on relative to other transistors which may form part of a ratioed path.

Figure 4 shows a switch level circuit, consisting of the nodes a, b, c, k, s, t, p, Vdd, and GND, and the transistors T1, through T8. The node sizes and the transistor strengths are indicated by the numbers in parenthesis. The storage nodes in the circuit are s and t, which have sizes of 1, and 2, respectively. All transistors have a size of 3, except T6, which has a size of 4. The input nodes in the circuit, a, b, c, k, Vdd, and GND have a size of 5. The

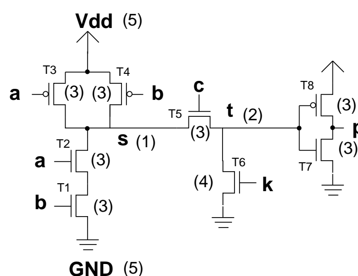

Fig. 4. Example of a switch-level network.

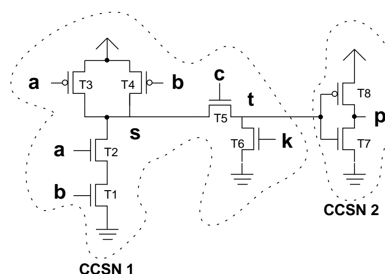

Fig. 5. Circuit partitioned into CCSNs CCSN1 and CCSN2.

x	$x.H$	$x.L$
0	0	1
1	1	0
X	1	1

TABLE I
Dual rail encoding of x.

states of input nodes Vdd, and GND are fixed at 1, and 0, respectively.

Nodes in a switch-level network are connected together by directed *paths* of conducting transistors. Each path originates at a source node, and terminates at a destination node. The path has a *strength*, which roughly indicates the approximate amount of charge that can be supplied along the path from the source to the destination. In case of a path from an input node to a storage node, the strength of the path equals that of the weakest transistor in the path. In case of a path connecting two storage nodes, the strength of the path equals the size of the source node. The state of a node depends on the states of the source nodes of the strongest paths to this node.

C. The behavior of switch-level circuits

Most switch-level analysis and simulation tools partition the transistor-level network into a set of communicating components, termed *channel connected subnetworks* (CCSNs). Each CCSN consists of a set of storage nodes that can share charge, together with the transistors that connect them. Behavior within a CCSN can be difficult to analyze because of the bidirectional nature of transistors, and the multiple signal strengths. The interaction between the CCSNs is simpler. Each CCSN may be viewed as a finite state machine, with inputs, internal state, and outputs. The inputs consist of the transistor gate nodes, and input nodes connected to transistor source or drains. The storage nodes hold the CCSN state, and a subset of the CCSN nodes constitute the set of observable outputs of the CCSN. Given its initial state, and the present inputs, this sequential machine computes a new state, and new outputs. The entire transistor network is thus modeled as a system of communicating sequential machines. Figure 5 shows the circuit of figure 4 partitioned into CCSNs CCSN1, and CCSN2.

The *steady-state response* function of a CCSN describes its sequential behavior. This function specifies how the new CCSN node states are computed, given the initial storage node states, and the values at the inputs and the gates of the CCSN, and given that the transistor states are fixed long enough for the nodes to stabilize.

The *excitation function* of a CCSN gives the steady-state response of the CCSN nodes, when the transistors are held fixed in states determined by the initial storage node states and the inputs. An important property of the excitation function is its *monotonicity* over the $\{0, 1, X\}$ values. This property implies that if some inputs of this function were set to X, and a given output were 0 or 1, then changing the X inputs to 0 or 1 does not alter the output. This property is particularly important for verification, in view of the "information-content" ordering of the three values.

We use the *unit-delay model* to describe circuit delays. In this approach, a change in the state of a transistor gate is reflected as a change in the state of the transistor after a delay of one *time-step*. This time-step is used as the unit of time. In each CCSN, given the logic levels at the inputs, and the storage nodes, a new logic level is computed for each storage node according to the CCSN excitation function. Then, after a delay of one time-step, the storage nodes are assigned the new logic levels just computed.

D. Computing the steady-state response

The steady-state response of a switch-level network can be obtained by a *symbolic Boolean analysis* of the network [5], [4]. In this approach, the problem of determining the network response is cast in terms of determining paths through the channel graph of a CCSN. The result of the analysis is a number of Boolean expressions which state how the ternary state of the CCSN is updated in each time-step.

To express and compute ternary quantities in the switch-level model in terms of Boolean operations, a "dual rail" encoding is used. Each ternary quantity x, is represented by a pair of Boolean values, $x.L$, and $x.H$, as shown in Table I. For each node n, we introduce two Boolean variables, $n.L$, and $n.H$. The analysis problem can be specified as: for each node n in the circuit, derive the Boolean formulas $N.H$, and $N.L$, which represent the encoded value of the steady-state response at the node as a function of the initial node states.

The goal of symbolic analysis of a network is to derive Boolean expressions indicating the conditions under which conducting paths are formed in the network. To express

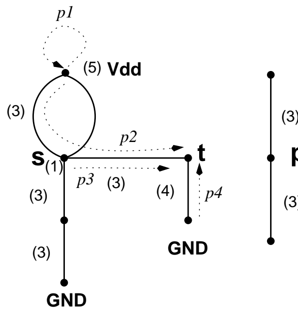

Fig. 6. Rooted paths in channel graph.

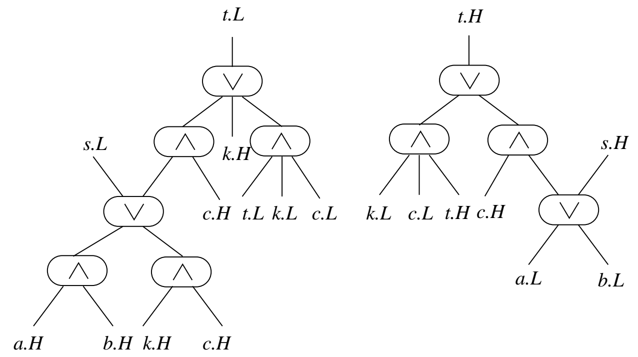

Fig. 7. Results of switch-level analysis.

these concepts more precisely, we first define a *channel graph*, which has a vertex for each circuit node, and an edge for the channel of each transistor in a switch-level circuit. The CCSN defined earlier in section III-C, is a connected component in this graph. The analysis process examines one CCSN at a time. The discussion ahead pertains to the analysis of a single CCSN.

A *rooted path* in the channel graph is a directed path between two nodes. A rooted path p originates at $root(p)$, and it terminates at $dest(p)$. Paths have a length, equal to the number of edges in it. A path can have a length of 0. The *strength* of a path reflects its relative charge transfer capacity. Figure 6 illustrates paths of different types in the channel graph for the circuit of Figure 4. $p1$ is a path of zero length, with a strength of 5. $p2$ is a path of strength 3 from the input node Vdd to the storage node t. $p3$ is a path of strength 1 from s to t. Note that the difference in strengths of $p2$, and $p3$ arises from the fact that $p2$ conveys charge from an input node which can potentially supply unlimited amount of charge, whereas node s can only convey the stored charge it has through path $p3$.

A rooted path is termed a *definite path*, if none of the transistors in the path are in the X state. The steady-state response of a node depends only on the paths to the node that are not *blocked*. Informally, a path is blocked if some intermediate node in the path is the destination of a stronger definite path. The charge from the stronger path overrides the charge from the weaker path. For example, if the transistor corresponding to path $p4$ in Figure 6 is on, then, $p2$ and $p3$, the weaker paths to node t, do not affect the steady-state response of t. If all the unblocked sources of charge to a node drive it to 0 (or 1), then the steady-state response equals 0 (or 1). Otherwise, if unblocked sources drive a node to conflicting values, then the node's response equals X. Let $\{\mathsf{m}_1, \mathsf{m}_2, ..., \mathsf{m}_k\}$ be the the set of nodes which are the origin of unblocked paths to node n. Let $m_1.H$, $m_1.L$, $\ldots m_k.H$, and $m_k.L$ be the pairs of Boolean variables to encode the states of these nodes. If N is the steady state response of node n then, given the dual rail encoding, it may be encoded as

$$N.H = m_1.H \vee m_2.H \vee \ldots \vee m_k.H$$
$$N.L = m_1.L \vee m_2.L \vee \ldots \vee m_k.L$$

The analyzer works with signals of one strength level at a time. It starts with the input signals, which are of the highest strength and works downward, each time adding in the effects of the paths at the next lower strength. For each strength level $w > s \geq 1$, the analyzer sets up and solves using Gaussian elimination three systems of Boolean equations which yield formulas $N.H_s$, $N.L_s$, and $clear_s(n)$ for every storage node n. $N.H_s$ and $N.L_s$ express the steady state response at the node, when all paths of strengths s and higher have been accounted for. $clear_s(n)$ expresses the condition when node n is not the destination of a definite path greater than or equal to s. It is used to set up equations for the next lower strength level. Thus, in the last iteration, expressions $N.H_1$ and $N.L_1$ are obtained, and these equal the steady state response of n.

The Boolean expressions generated in the process of symbolic analysis are represented by directed acyclic graphs (DAGs), where leaves denote Boolean variables and constants, and nodes denote Boolean operations. Each node of the DAG represents a Boolean formula, and often there is considerable amount of sharing in a DAG structure for the steady-state expressions for the storage nodes of a CCSN. Figure 7 shows the steady state expressions for nodes s, and t in component CSSN1 of the circuit in Figure 4.

Section III-A describes the behavior of systems by an excitation function $Y: \mathcal{S} \rightarrow \mathcal{S}$ over sets of atom. The system response obtained by the analysis above can be easily converted to our "atomcentric" point of view by a simple transformation. To ensure consistency between the two representations, this transformation ensures that if the high-rail $n.H$ (low-rail $n.L$) value of a node is 1, then the corresponding circuit state does not include n^- (n^+). Since the presence of n^- precludes the hi-rail from being 1, (dual true for lo-rail, and n^+), the transformation below converts the dual-rail DAGs to an atomcentric DAG.

1. Transform every $x.L$ to x^+, and every $x.H$ to x^-.
2. Convert AND nodes to OR, and OR nodes to AND.

Figure 8 illustrates this transformation on Figure 7.

E. Trajectory Evaluation

In Symbolic Trajectory Evaluation (STE), the specification language consists of a set of *trajectory assertions.* The simplest form of trajectory assertion has the form $[A \Longrightarrow C]$, where A, and C are *trajectory formulas.* A, the *antecedent* of the trajectory assertion describes the stimulus to the circuit over time, and C describes the expected

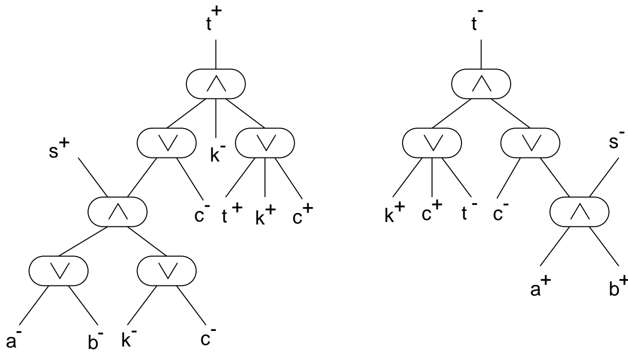

Fig. 8. Atomcentric view of analysis results.

response.

Trajectory formulas (TFs) have the following recursive definition:

1. **Atoms**: For any node n, atoms n^+ and n^- are TFs.
2. **Conjunction**: $(F_1 \wedge F_2)$ is a TF if F_1 and F_2 are TFs.
3. **Domain restriction**: $(E \rightarrow F)$ is a TF if F is a TF and E is a Boolean expression.
4. **Next time**: $(\mathbf{N}F)$ is a TF if F is a TF.

The Boolean expressions occurring in domain restriction operators, having the form $E \rightarrow F$, give these formulas a symbolic character. They can be thought of as "guards," i.e., F must hold for the cases where E evaluates to true. For the theoretical development, however, it is convenient to first consider the form where E is restricted to be either 0 (false) or 1 (true). A *scalar* trajectory formula obeys this restriction throughout its recursive structure. The extension to the symbolic case then simply involves considering the valuation of the expressions for each variable assignment. [22] describes this in greater detail. **N** is the *next time* temporal operator which causes advancement of time by one unit.

The truth of a scalar trajectory formula F is defined relative to a model structure and a trajectory. Let s and $s^0\tilde{s}$ both be members of $L(\mathcal{M})$. $s \models_{\mathcal{M}} F$, the truth of F relative to model $\mathcal{M}$, and a trajectory s is recursively defined as:

1. (a) $s^0\tilde{s} \models_{\mathcal{M}} a^+$ iff $a^+ \in s^0$.
   (b) $s^0\tilde{s} \models_{\mathcal{M}} a^-$ iff $a^- \in s^0$.
2. $s \models_{\mathcal{M}} (F_1 \wedge F_2)$ iff $s \models_{\mathcal{M}} F_1$ and $s \models_{\mathcal{M}} F_2$.
3. (a) $s \models_{\mathcal{M}} (1 \rightarrow F)$ iff $s \models_{\mathcal{M}} F$
   (b) $s \models_{\mathcal{M}} (0 \rightarrow F)$ holds for every s.
4. $s^0\tilde{s} \models_{\mathcal{M}} \mathbf{N}F$ iff $\tilde{s} \models_{\mathcal{M}} F$.

A *defining sequence* of a trajectory formula F, denoted by δ_F, is the weakest possible sequence of states "consistent" the restrictions specified by F. We clarify this below. The recursive definition of this sequence is given as:

1. (a) $\delta_a^+ = \{a^+\}\emptyset\emptyset\emptyset\ldots$
   (b) $\delta_a^- = \{a^-\}\emptyset\emptyset\emptyset\ldots$
2. $\delta_{F_1 \wedge F_2} = lub(\delta_{F_1}, \delta_{F_2})$
3. (a) $\delta_{0 \rightarrow F} = \emptyset\emptyset\emptyset\ldots$
   (b) $\delta_{1 \rightarrow F} = \delta_F$
4. $\delta_{\mathbf{N}F} = \emptyset\delta_F$

While δ_F is not necessarily a trajectory, it can be shown that $s \models_{\mathcal{M}} F$ iff $\delta \sqsubseteq s$. A defining trajectory is the weakest sequence of states that can be constructed, given the constraints specified in a trajectory formula. For example, in the definition above, the defining trajectory for a formula consisting only of a^+, is a sequence, the first element of which is the set $\{a^+\}$, and the remaining elements are the empty set $\emptyset$.

While δ_F is not a trajectory, we may combine it with the successor function Y, to get the *defining trajectory*, τ_F, of F. It can be shown that τ_F is the unique weakest trajectory satisfying F. We outline the construction of τ_F ahead. Let $\delta_F = \delta_F^0\delta_F^1\ldots$. Let $\tau_F = \tau_F^0\tau_F^1\ldots$ be the defining trajectory. Then, the successive elements of τ_F are given by the following construction:

$$\tau_F^i = \begin{cases} \delta_F^0 & \text{if } i = 0 \\ \delta_F^i \cup Y(\tau_F^{i-1}) & \text{otherwise} \end{cases}$$

The truth of a trajectory assertion $[A \Longrightarrow C]$ is defined with respect to a model $\mathcal{M}$, and a set of trajectories L of $\mathcal{M}$. Expressed as, $L \models_{\mathcal{M}} [A \Longrightarrow C]$, it is defined to hold iff for all $s \in L$, $s \models_{\mathcal{M}} A$ implies $s \models_{\mathcal{M}} C$. Often, L equals $L(\mathcal{M})$, the complete set of trajectories of model $\mathcal{M}$. That $[A \Longrightarrow C]$ is true for every trajectory in this set is denoted as $\models_{\mathcal{M}} [A \Longrightarrow C]$. Intuitively, $\models_{\mathcal{M}} [A \Longrightarrow C]$ means that if there is a sequence of states of $\mathcal{M}$ consistent with the excitation function of the system, and A is true with respect to this sequence, then C is also true with respect to this sequence. Alternately, this may be interpreted as A defines a behavior (subset of trajectories) of the system, and C holds true for this behaviour of the system. The existence of a defining trajectory for every trajectory formula considerably simplifies the test for determining the truth of an assertion. A defining trajectory τ_A essentially serves as a representative of the set of trajectories of the system for which the trajectory formula A is true. Given an assertion $[A \Longrightarrow C]$, we can verify that it holds for all elements of $L(\mathcal{M})$ by performing the test $\delta_C \sqsubseteq \tau_A$. Below, the key result of STE, states how the truth of an assertion may be determined. This has been proved by Bryant and Seger in [22].

Theorem 1: $\models_{\mathcal{M}} [A \Longrightarrow C]$ iff $\delta_C \sqsubseteq \tau_A$.

Thus, to verify the truth of an assertion, all we need to do is construct a defining sequence, and a defining trajectory, and check that the former is weaker than the latter. Furthermore, we need check only an initial prefix of the two sequences, which is equals the "depth" of C. The depth of a formula F, denoted by $d(F)$, equals the maximum nesting of the next time **N** operator. This is stated as corollary 1.

Corollary 1: $\delta_C \sqsubseteq \tau_A$ iff $\delta_C^i \sqsubseteq \tau_A^i$ for $0 \leq i < d(C)$.

IV. Symmetries of a circuit

We express both circuit operation and the specifications in terms of sets of atoms. We can therefore express symmetries in a circuit and the corresponding transformations of the specification in terms of bijective mappings over atoms, named *state transformations.*

Definition 4: A state transformation, σ, is a bijection over the set of atoms: $\sigma : \mathcal{A} \rightarrow \mathcal{A}$. We can extend σ to be a bijection over states by defining $\sigma(s)$ for state s as $\cup_{a \in s}\{\sigma(a)\}$.

As the term suggests, a state transformation σ takes a system state s and alters it to a new state $\sigma(s)$. Since σ is bijective, σ^{-1} exists. Also, if σ_1, and σ_2 are state transformations, then their composition $\sigma_1\sigma_2$ is also a state transformation.

Two types of state transformations, which alter circuit state in a "structured" manner are particularly interesting. These are the *structural*, and *data* transformations. Below, $s[a_1/b_1, ..., a_n/b_n]$ denotes the state obtained by simultaneously substituting atoms $a_1, \ldots, a_n$ for the atoms $b_1, \ldots, b_n$, respectively, in state s.

Definition 5: A *structural* transformation is a state transformation which swaps the atoms for two different nodes. For nodes n_1 and n_2, we write $\mathsf{n}_1 \leftrightarrow \mathsf{n}_2$ to denote the transformation consisting of the swappings: n_1^+ with n_2^+ and n_1^- with n_2^-. Given $\sigma = \mathsf{n}_1 \leftrightarrow \mathsf{n}_2$, and a circuit state s, $\sigma(s) = s[n_1^+/n_2^+, n_1^-/n_2^-, n_2^+/n_1^+, n_2^-/n_1^-]$

Definition 6: A *data* transformation involves swapping the two atoms for a single node. For node n, we write $n^\pm$ to denote the transformation consisting of the swapping of n^+ with n^-. Given $\sigma = n^\pm$, and circuit state s, $\sigma(s) = s[n^+/n^-, n^-/n^+]$

Intuitively, a structural transformation exchanges two nodes of a circuit (by swapping their atoms), and a data transformation complements the value at a circuit node by altering a positive atom of a node to a negative atom, and vice versa. Thus, these transformations provide us with a convenient mechanism to express circuit structure and circuit data handling related issues. Composing structural and data transformations allows us to express a variety of circuit transformations. To simplify notation, we will denote more complex transformations as a list of elementary transformations.

Our unified view of state transformations and excitations as functions mapping states into states allows us to succinctly express symmetry in a circuit as an excitation preserving transform. This closely parallels the definition of symmetry in [7] as a transition relation preserving state permutation.

Definition 7: A state transformation σ is a *symmetry property* of a circuit with excitation function Y when $\sigma(Y(s)) = Y(\sigma(s))$ for every state s.

That is, the excitation of the circuit on the transformed state $\sigma(s)$ matches the transformation of the excitation of s.

Lemma 1: Symmetry transformations have the following properties.

1. If σ is a symmetry property, and Y is an excitation function then $Y = \sigma Y \sigma^{-1} = \sigma^{-1} Y \sigma$.

2. σ is a symmetry property if and only if its inverse σ^{-1} is a symmetry property.

3. If σ_1 and σ_2 are symmetry properties, then their composition $\sigma_1\sigma_2$ is also a symmetry property.

Proof: 1. Since σ is a symmetry property, for every state s, $\sigma(Y(s)) = Y(\sigma(s))$, i,e, $\sigma Y = Y\sigma$. Therefore $\sigma^{-1}\sigma Y = \sigma^{-1} Y \sigma$, i.e., $Y = \sigma^{-1} Y \sigma$.

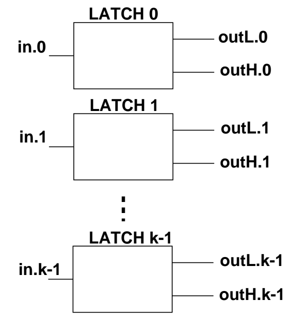

Fig. 9. Illustration of the symmetries of a circuit

2. Since σ is a symmetry property, $\sigma Y = Y\sigma$. Therefore, $\sigma^{-1}(\sigma Y)\sigma^{-1} = \sigma^{-1}(Y\sigma)\sigma^{-1}$, which reduces to $Y\sigma^{-1} = \sigma^{-1}Y$. The other direction can be proved similarly.

3. σ_1 and σ_2 are symmetry properties. Therefore, $Y = \sigma_1^{-1} Y \sigma_1$, and $Y = \sigma_2^{-1} Y \sigma_2$. Substituting $\sigma_1^{-1} Y \sigma_1$ for Y in the second equation yields $Y = \sigma_2^{-1}(\sigma_1^{-1} Y \sigma_1)\sigma_2$, i.e., $Y = (\sigma_1\sigma_2)^{-1} Y \sigma_1\sigma_2$, i.e., $\sigma_1\sigma_2$ is a symmetry property. □

Depending on their constituent state transformations, symmetry properties may be classified into one of the following three categories:

- *Structural symmetry* — consists entirely of structural transformations.
- *Data symmetry* — consists entirely of data transformations.
- *Mixed symmetry* — consists of both structural and data transformations.

Consider, for example, the circuit shown in Figure 9. This circuit consists of k identical latches. In each latch outL is a complement of the input, and outH has the same value as the input. Since the latches are identical, this circuit has a structural symmetry corresponding to the swapping of any pair of latches i and j, such that $0 \leq i, j < k$:

$$[\mathsf{in}.i \leftrightarrow \mathsf{in}.j, \mathsf{outL}.i \leftrightarrow \mathsf{outL}.j, \mathsf{outH}.i \leftrightarrow \mathsf{outH}.j]\,. \quad (1)$$

Each individual latch also stores data values 0 and 1 in a symmetric way, expressed for Latch 0 by the data symmetry:

$$[\mathsf{in}.0^\pm, \mathsf{outL}.0^\pm, \mathsf{outH}.0^\pm]\,. \quad (2)$$

Finally, each latch can also be viewed as a one-bit decoder—it sets one of its outputs high based on its input data. Such behavior for Latch 0 is expressed by a mixed symmetry:

$$[\mathsf{in}.0^\pm, \mathsf{outL}.0 \leftrightarrow \mathsf{outH}.0]\,. \quad (3)$$

V. Verification under symmetry

Verifying a circuit involves checking a family of assertions against the circuit model. The presence of symmetry properties in the circuit often allows us to dramatically reduce the number of assertions that need to be verified. This is because if an assertion G holds, and σ is a symmetry property of the circuit, then the assertion $\sigma(G)$ also holds. This is elaborated below.

We can extend σ to be a bijection over state sequences by applying σ to each state in the sequence. We can also extend state transformation σ to be a bijection over temporal formulas. First we define the extension of σ to Trajectory formulas(TFs).

Definition 8: If F is a TF, then $\sigma(F)$ is recursively defined as

1. If F is the atom a, then $\sigma(F)$ is the transformed atom $\sigma(a)$.
2. $\sigma(F_1 \wedge F_2) = (\sigma(F_1) \wedge \sigma(F_2))$, where F_1 and F_2 are TFs.
3. $\sigma(E \rightarrow F) = (E \rightarrow \sigma(F))$, where F is a TF and E is a Boolean expression.
4. $\sigma(\mathbf{N}F) = (\mathbf{N}\sigma(F))$ where F is a TF.

The effect of applying σ to a TF F is to replace every atom a in F by $\sigma(a)$. This idea is carried further, where σ may be applied to an assertion. The result of applying σ to the assertion $[A \Longrightarrow C]$ is the assertion $[\sigma(A) \Longrightarrow \sigma(C)]$, which is also denoted by $\sigma([A \Longrightarrow C])$.

Since a symmetry property is an excitation function preserving transformation, it follows fairly intuitively that the structure of the defining sequence and the defining trajectory of a TF F should remain invariant with respect to σ. This is formalized below in lemma 2, and lemma 3.

Lemma 2: If temporal formula F has defining sequence δ_F, then its transformation $\sigma(F)$ will have defining sequence $\delta_{\sigma(F)} = \sigma(\delta_F)$.

Proof: This can be proved by induction over the structure of TFs. Details are shown in [18].

Lemma 3: If σ is a symmetry property of a circuit model $\mathcal{M}$, then its defining trajectories for any temporal formula F will obey the symmetry: $\tau_{\sigma(F)} = \sigma(\tau_F)$.

Proof: We can prove this result by induction on the sequence of elements in the defining trajectory $\tau_F = \tau_F^0\tau_F^1 \ldots \tau_F^i\tau_F^{i+1} \ldots$. Details are shown in [18].

This brings us to the central theorem of this paper, which is stated below.

Theorem 2: For an assertion $[A \Longrightarrow C]$, and a symmetry property σ of model $\mathcal{M}$, $\models_{\mathcal{M}} [A \Longrightarrow C]$ if and only if $\models_{\mathcal{M}} [\sigma(A) \Longrightarrow \sigma(C)]$.

Proof: The proof follows directly from the definition of a symmetry property, lemma 2, and lemma 3.

$\models_{\mathcal{M}} [A \Longrightarrow C]$
$\Leftrightarrow \tau_A \subseteq \delta_C$ (Theorem 1)
$\Leftrightarrow \sigma(\tau_A) \subseteq \sigma(\delta_C)$ (Bijection σ preserves subset relation)
$\Leftrightarrow \tau_{\sigma(A)} \subseteq \sigma(\delta_C)$ (Lemma 3)
$\Leftrightarrow \tau_{\sigma(A)} \subseteq \delta_{\sigma(C)}$ (Lemma 2)
$\Leftrightarrow \models_{\mathcal{M}} [\sigma(A) \Longrightarrow \sigma(C)]$ (Theorem 1)
□

Thus, proving that σ is a symmetry property of a circuit allows us to infer the validity of a transformed assertion once we verify the original. For example, suppose we verify that Latch 0 in Figure 9 operates correctly for input value 1, and also prove that the transformations defined by Equations 1 and 2 are indeed symmetry transformations. Then we can infer from Equation 1 that for all j, Latch j operates correctly for input value 1, and from Equation 2 that Latch 0 operates correctly for input value 0. Furthermore, by composing these two transformations, we can infer that for all j, Latch j will operate correctly for input value 0.

VI. Verification of symmetry properties

To exploit symmetry, we verify an assertion $[A \Longrightarrow C]$, and given a set of symmetry properties, $S = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, we can conclude that $[\sigma_1(A) \Longrightarrow \sigma_1(C)]$, $[\sigma_2(A) \Longrightarrow \sigma_2(C)]$, $\ldots [\sigma_3(A) \Longrightarrow \sigma_3(C)]$ all hold. However, before drawing this conclusion, one must verify that every element of S is actually a symmetry property. The typical set of symmetry properties we work with is a group. Below, we give two basic definitions from group theory which we use ahead.

Definition 9: Set S is a symmetry group for a model structure $\mathcal{M} = \langle \mathcal{S}, Y \rangle$ iff every element of S is a symmetry property of $\mathcal{M}$, and the following properties hold:

1. The identity element σ_e is in S, where $\sigma_e(a) = a$.
2. Every element of $\sigma \in S$ has an inverse $\sigma^{-1} \in S$ such that $\sigma\sigma^{-1} = \sigma^{-1}\sigma = \sigma_e$

Definition 10: A set $\langle S \rangle$ is termed a generator of a symmetry group S if repeated compositions of the elements in $\langle S \rangle$ can generate every element of S.

The definitions above imply that rather than test every element of a set S for the symmetry property, it suffices to check only the generators of S. Thus, it is possible to prove the correctness of an entire set of assertions by simply verifying that each member of a set of generators for a group of transformations is a symmetry property.

For example, (1) in section IV represents a total of $k(k-1)/2$ symmetry transformations, corresponding to the pairwise exchange of any two latches. In general, one could argue that this circuit would remain invariant for any permutation π of the latches. Consider the transformation σ_π mapping the 6 atoms for each Latch i (two each for nodes $\mathsf{in}.i$, $\mathsf{outL}.i$ and $\mathsf{outH}.i$) to their counterparts in Latch $\pi(i)$. We could prove that each such transformation is a symmetry property, but this would require $k!$ tests. Instead, we can exploit the fact that any permutation π can be generated by composing a series of just two different permutation types. The "exchange" permutation swaps values 0 and 1, while the "rotate" permutation maps each value i to $i + 1 \bmod k$. Thus, proving that the state transformations given by these two permutations are symmetry properties allows us to infer that σ_π is a symmetry property for an arbitrary permutation π.

So, once the generators of a symmetry group are identified, the next step is to verify that they are indeed symmetry properties. We verify structural symmetries of a circuit by circuit graph isomorphism checks, and we verify data and mixed symmetries using symbolic simulation based techniques. We describe these techniques in sections VI-A, and VI-B.

A. Structural symmetry property verification

We can verify structural symmetries in our circuit models by checking for isomorphisms in the circuit-graph network. Since the circuit analysis tool we use (Anamos [5]) derives its representation of the excitation function from

the network, any isomorphisms in the network graph imply structural symmetries in the excitation function. While it is also possible to verify structural symmetries using symbolic simulation, such an approach requires performing switch-level analysis on the circuit. This analysis can be prohibitively expensive (Table II) for circuit components such as array cores with their large CCSNs.

Consider the problem of determining if $n_1 \leftrightarrow n_2$ is a symmetry property in Circuit 1 (Figure 10). If this symmetry were to hold in the circuit, Circuit 2 which is obtained from Circuit 1 by swapping the labels of nodes n_1, and n_2 should be isomorphic to Circuit 1. To perform this isomorphism test we use a graph coloring based algorithm described in [10], [9]. This algorithm, which is described below, converts a circuit graph network into a *pseudo-canonical* form. This allows a fast and efficient test for isomorphism between two networks.

Given the circuit graph for a switch-level circuit, we first construct a *coloring graph* which contains a *transistor vertex* for each transistor in the circuit, and a *node vertex* for each node and primary inputs in the circuit. The edges of this graph connect transistor vertices and node vertices, and they correspond to the node-transistor interconnections in the circuit. Since there are no edges which connect only two node vertices, or two transistor vertices, the coloring graph is bipartite. Once the coloring graph is constructed, the vertices of the graph are "colored" with integers. Based on isomorphism invariant vertex properties such as the number of edges incident on a vertex, all the vertices in the graph are assigned an initial color. Vertices are recolored repeatedly using a hashing function which combines the colors of the neighboring vertices, until all the vertices are colored uniquely. Then the nodes and transistors of the circuit are sorted to yield what is termed as the *quasi-canonical form* of the circuit network. This coloring and sorting technique guarantees that two networks that are not isomorphic will be colored differently. However, in some remote cases, it is also possible that two isomorphic networks may not be colored uniquely within a given fixed number of coloring iterations. However, we have not encountered this problem in our experiments (Section IX).

The problem we need to solve is slightly different from the one solved above. We need to swap two nodes in the circuit, and test if the two circuits, before and after the swap, are isomorphic. This does not work directly, as the isomorphism checks work purely on the structure of the network, and they ignore the node names. So the two circuits, before and after the node name swap will still reduce to the same quasi-canonical form. We work around this by using *structural labels*.

A structural label is a circuit graph (or an interconnection of nodes and transistors) which satisfies the following two properties:

- The structural label is not isomorphic to any subgraph of the original circuit.
- The label is not isomorphic to any other structural label.

These properties allow us to use these labels to physically tag circuit nodes. Structural labels serve to uniquely

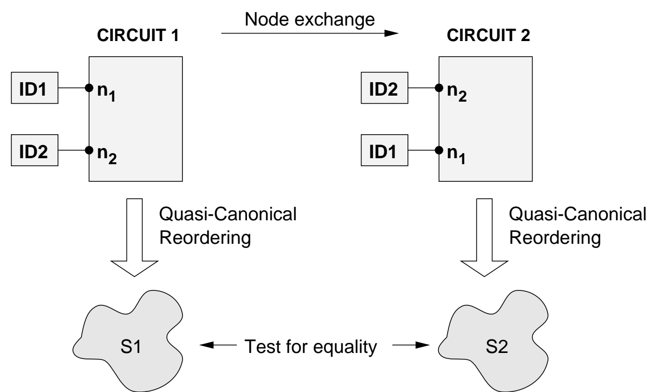

Fig. 10. Verification of structural symmetry $n_1 \leftrightarrow n_2$

tag a node in the circuit being tested for symmetry properties. So, structural labels are attached not only to the set of nodes being swapped, but also to some of the remaining circuit nodes. While verifying symmetry properties by graph isomorphism checks, it is necessary to attach structural labels to all the input and output nodes of a circuit. However, it is not necessary to consider the state nodes during our symmetry tests. If the circuit being verified satisfies all the assertions, then it has the desired IO behavior, and no further tests are necessary for the internal nodes. Of course, if some assertion fails, then further checks are necessary, and we may well discover a problem related to the state nodes.

Thus, to solve the problem of determining whether $n_1 \leftrightarrow n_2$ holds in Circuit 1, we can attach structural labels ID1 and ID2 to nodes n_1 and n_2 in Circuit 1, and then flip these labels in Circuit 2. We then can reduce the two circuits to the quasi-canonical form, and compare them (Figure 10).

The worst case time complexity for the graph coloring algorithm for a circuit with n transistors is $O(n^2 log(n))$. However, in practice we have seen both the time and memory scale nearly linearly with n. (Table III).

B. Data and mixed symmetry property verification

Data and mixed symmetries can be verified by symbolic simulation. Data symmetries involve "switching polarities" of values on a node. Mixed symmetries involve "switching polarities" of the value on a node, and "exchanging" the values of two different nodes. To verify these symmetries it is necessary to check whether for every combination of circuit node values, the exchange and the polarity switching of the input node values results in changes in the output node values as specified by the symmetry property. Symbolic simulation is the ideal tool for such checks involving all possible combination of Boolean values. One symbolic Boolean variable is introduced for each circuit node. The circuit is simulated with these Boolean variables, and then with new Boolean values corresponding to the changes specified by the symmetry property. The results of the two simulations are compared to verify if the circuit obeys the symmetry property.

Consider, for example, the symmetry of Latch 0 specified

in equation 2:

$$\left[\mathsf{in}.0^{\pm}, \mathsf{outL}.0^{\pm}, \mathsf{outH}.0^{\pm}\right].$$

We symbolically simulate the circuit with the symbolic value a at the input in.0, and inspect the values at the output nodes outL.0, and outH.0. If the above symmetry holds for the latch, then complementing the value at node in.0, should result in the values at circuit nodes outL.0, and outH.0 being complemented.

Consider also the mixed symmetry of Latch 0, specified in equation 2:

$$\left[\mathsf{in}.0^{\pm}, \mathsf{outL} \leftrightarrow \mathsf{outH}.0\right].$$

This symmetry can also be verified by symbolically simulating the circuit with a symbolic value a at the input in.0, and observing the outputs, outL.0, and outH.0. The equation above specifies that complementing the value at in.0 should result in the values at nodes outL.0, and outH.0 being exchanged, which can be easily checked.

VII. Conservative approximations of circuits

Intuitively, a conservative approximation of a circuit is a "reduced" version of the circuit which, given any current circuit state, imposes fewer constraints on the next state the circuit can take. In terms of values, on circuit nodes, the reduced circuit produces more Xs than the original circuit, i.e., fewer atoms in the next state. We formalize this concept below.

Definition 11: Let $\mathcal{M}'$ and $\mathcal{M}$ be circuit models over the same state set, having excitation functions Y' and Y, respectively. We say that $\mathcal{M}'$ is a *conservative approximation* of $\mathcal{M}$ if for every state s, $Y'(s) \subseteq Y(s)$. We denote this by $\mathcal{M}' \preceq \mathcal{M}$.

We exploit conservative approximations to perform two important tasks:

- Create reduced models which take less memory to represent.
- Partition circuits to expose symmetric regions of a design.

Proving an assertion for a reduced circuit model, allows us to infer that the assertion holds for the original circuit. Theorem 3 below justifies this. The advantage of this is reduced circuit models are often a fraction of the size of the original circuit model, which results in smaller verification memory requirements. Conservative approximations provide a systematic way to reason about partitioned circuits, allowing us to verify the complete circuit by proving properties about each partition. This is particularly useful when the partitioning can expose highly symmetric regions of the circuit. In addition, if we can prove that a circuit has some structural symmetry, then we can create a "weakened" version of the circuit containing just enough circuitry to verify the behavior for one representative of the symmetry group.

A. Verification of Reduced Models

Below, we show that for any trajectory assertion $[A \Longrightarrow C]$, and models $\mathcal{M}$, and $\mathcal{M}'$, such that $\mathcal{M}' \preceq \mathcal{M}$, if the assertion $[A \Longrightarrow C]$ holds for $\mathcal{M}'$, then it should also hold for $\mathcal{M}$. We start with the proof of a simple result on the trajectories of $\mathcal{M}'$ and $\mathcal{M}$.

Lemma 4: If F is any trajectory formula, and models $\mathcal{M}$, and $\mathcal{M}'$ are such that $\mathcal{M}' \preceq \mathcal{M}$, then the defining trajectories for the two models, τ'_F and τ_F, must be ordered $\tau'_F \sqsubseteq \tau_F$.

Proof: Let $\tau_F = \tau_F^0\tau_F^1\tau_F^2\ldots$, and $\tau'_F = \tau'^0_F\tau'^1_F\tau'^1_F\ldots$. Using induction we can show that for all $i \geq 0$, $\tau_F^i \subseteq \tau'^i_F$. Details are in given in [18].

Therefore, as expected, "weakening" the model also weakens the trajectories of the model, and this immediately leads to the theorem below.

Theorem 3: For any assertion $[A \Longrightarrow C]$, and $\mathcal{M}' \preceq \mathcal{M}$, if $\models_{\mathcal{M}'} [A \Longrightarrow C]$, then $\models_{\mathcal{M}} [A \Longrightarrow C]$.

Proof: Since $\models_{\mathcal{M}'} [A \Longrightarrow C]$, $\delta_C \sqsubseteq \tau'_F$ (Theorem 1). This result, when combined with $\tau'_F \sqsubseteq \tau_F$ (follows from lemma 4 above), gives $\delta_C \sqsubseteq \tau_F$, i.e., $\models_{\mathcal{M}} [A \Longrightarrow C]$. □

Thus, proving an assertion for a conservative approximation to a circuit model allows us to infer that the assertion holds for the original circuit.

B. Partitioning circuits via conservative approximations

We can view the partitioning of a circuit into different components as a process of creating multiple conservative approximations. For example, suppose we partition a circuit with nodes N into components having nodes N_1 and N_2, respectively, as illustrated in Figure 11. The set of nodes forming the interface between the components comprise the set $N_1 \cap N_2$. In this example, we assume the communication is purely unidirectional—N_1 generates signals for N_2. Suppose we wish to prove a property described by an assertion $[A \Longrightarrow C]$, where the atoms of C are contained only in N_2. We could then create conservative models $\mathcal{M}_1$ and $\mathcal{M}_2$ using the subset construction given by Equation 4 below. Taken individually, each of the two models is too weak to prove the assertion. Using the technique of waveform capture described ahead, we can record the output values generated by model $\mathcal{M}_1$ and use them in verifying the assertion with model $\mathcal{M}_2$. We describe this technique in greater detail below.

We start with the idea of creating conservative approximations by removing nodes of a circuit. Let N' be a subset of the set of circuit nodes N, and $\mathcal{A}'$ be the corresponding set of atoms. Then we can view the removal of those nodes not in N' as yielding a conservative approximation to the circuit with an excitation function Y' such that:

$$Y'(s) \quad = \quad Y(s \cap \mathcal{A}') \cap \mathcal{A}'. \tag{4}$$

Intuitively, $s \cap \mathcal{A}'$ eliminates atoms of all nodes other than in N', i.e., sets all the excluded nodes to X. The intersection of $Y(s \cap \mathcal{A}')$ with $\mathcal{A}'$ ensures that in the response, Y', atoms of all nodes other than in N' are eliminated.

Below, we first discuss the idea of *waveform capture*, and show how to construct a trajectory formula corresponding to the signal waveforms on a set of nodes.

Let N' be a subset of nodes in a circuit $\mathcal{M}$. Let $\tau_A = \tau_A^0\tau_A^1\tau_A^2\ldots$ be the defining trajectory for the circuit

$\mathcal{M}$ for a trajectory formula A. The technique of waveform capture records the occurrence of atoms on the nodes in N' as specified by the elements of the sequence τ_A, and it creates a temporal formula W_A describing this occurrence of atoms.

Let $\mathcal{A}_{N'}$ be the set of atoms corresponding to the nodes in N'. By eliminating all atoms from τ_A that are not in N', we construct a new sequence,

$$(\tau_A^0 \cap \mathcal{A}_{N'})\ (\tau_A^1 \cap \mathcal{A}_{N'})\ (\tau_A^2 \cap \mathcal{A}_{N'})\ \ldots .$$

From this sequence, we construct a trajectory formula

$$W_A = W_A^0 \wedge (\mathbf{N} W_A^1) \wedge (\mathbf{N}^2 W_A^2) \ldots \quad (5)$$

such that $W_A^i = a_1 \wedge a_2 \wedge \ldots \wedge a_k$, where $a_i \in (\tau_A^i \cap \mathcal{A}_{N'})$. The lemma below states the obvious consequence of such a construction.

Lemma 5: Let A be a trajectory formula, and let N' be a subset of nodes of the circuit described by $\mathcal{M}$. If W_A is the trajectory formula constructed from τ_A, as described above, then $\models_{\mathcal{M}} [A \Longrightarrow W_A]$.

Proof: Consider $\delta_{W_A} = \delta_{W_A}^0 \delta_{W_A}^1 \delta_{W_A}^2 \ldots$, the defining sequence of W_A. From the construction above, and the definition of defining trajectories, $\delta_{W_A}^i = \delta_{W_A^i}$, i.e., $\delta_{W_A}^i = (\tau_A^i \cap \mathcal{A}_{N'})$. Obviously, $(\tau_A^i \cap \mathcal{A}_{N'}) \subseteq \tau_A^i$. Therefore, $\delta_{W_A} \sqsubseteq \tau_A$, which proves $\models_{\mathcal{M}} [A \Longrightarrow W_A]$. □.

Now we discuss the use of waveform capture to verify a property $[A \Longrightarrow C]$ for a larger design by partitioning it into smaller components and verifying these components individually. Prior to the discussion however, we prove theorem 4, which justifies waveform capture and combination. To simplify the proof of the theorem, we first show some useful results in lemma 6, 7, and 8. Lemma 7 and 8 also appear in a modified form in [14].

Lemma 6: If $\models_{\mathcal{M}} [A \Rightarrow C]$, then $\models_{\mathcal{M}} [A \Rightarrow A \wedge C]$.

Proof: Since, $\models_{\mathcal{M}} [A \Rightarrow C]$, therefore, $\Rightarrow \delta_C \sqsubseteq \tau_A$, i.e., for $i \geq 0$, $\delta_C^i \subseteq \tau_A^i$. From definition of τ_A, for $i = 0$, $\tau_A^0 = \delta_A^0$, and for $i > 0$, $\tau_A^i = lub(\delta_A^i, Y(\tau_A^{i-1}))$, i.e., for $i \geq 0$, $\delta_A^i \sqsubseteq \tau_A^i$. Therefore, for $i \geq 0$, $(\delta_A^i \cup \delta_C^i) \subseteq \tau_A^i$, i.e., $\delta_A \cup \delta_C \subseteq \tau_A$. Therefore, $\models_{\mathcal{M}} [A \Rightarrow A \wedge C]$.

Lemma 7: If A and B are trajectory formulas, then $\delta_B \sqsubseteq \tau_A \Rightarrow \tau_B \sqsubseteq \tau_A$

Proof: This can be proved by induction on elements of the sequences $\tau_A = \tau_A^0 \tau_A^1 \tau_A^2 \ldots$ and $\tau_B = \tau_B^0 \tau_B^1 \tau_B^2 \ldots$ [18].

Lemma 8: If $\models_{\mathcal{M}} [A \Rightarrow B]$, and $\models_{\mathcal{M}} [B \Rightarrow C]$, then $\models_{\mathcal{M}} [A \Rightarrow C]$.

Proof: From $\models_{\mathcal{M}} [B \Rightarrow C]$, and $\models_{\mathcal{M}} [A \Rightarrow C]$, we know that $\delta_B \subseteq \tau_A$, and $\delta_C \subseteq \tau_B$ are true. From $\delta_B \subseteq \tau_A$, and lemma 7, we can conclude that $\tau_B \subseteq \tau_A$. Therefore, $\delta_C \subseteq \tau_B \subseteq \tau_A$, i.e., $\models_{\mathcal{M}} [A \Rightarrow C]$. □

Theorem 4: If $\mathcal{M}_C$ and $\mathcal{M}_D$ are conservative approximations of $\mathcal{M}$, and $\models_{\mathcal{M}_C} [A \Rightarrow F]$, and $\models_{\mathcal{M}_D} [A \wedge F \Rightarrow C]$, then $\models_{\mathcal{M}} [A \Rightarrow C]$.

Proof: If $\models_{\mathcal{M}_C} [A \Rightarrow F]$, then $\models_{\mathcal{M}} [A \Rightarrow F]$ (from Theorem 3), and thus $\models_{\mathcal{M}} [A \Rightarrow A \wedge F]$ (from Lemma 6). Similarly, if $\models_{\mathcal{M}_D} [A \wedge F \Rightarrow C]$, then $\models_{\mathcal{M}} [A \wedge F \Rightarrow C]$ (Theorem 3). From this, using Lemma 8, we can conclude that $\models_{\mathcal{M}} [A \Rightarrow C]$.

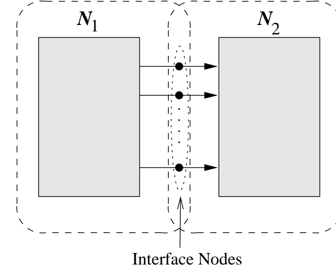

Fig. 11. Illustration of Circuit Partitioning.

Let τ_A^1 be the defining trajectory generated by model $\mathcal{M}_1$ for antecedent A. We construct a trajectory formula W_A describing the occurrence of the atoms corresponding to the nodes in $N_1 \cap N_2$ as described above. One refinement that should be performed is to record the values up to the maximum depth of the next-time operators in C (Corollary 1). Our construction ensures that $\models_{\mathcal{M}_1} [A \Longrightarrow W_A]$, and therefore, $\models_{\mathcal{M}} [A \Longrightarrow W_A]$. Using model $\mathcal{M}_2$, we then verify the assertion $[A \wedge W_A \Longrightarrow C]$, and if it holds, we know that $\models_{\mathcal{M}} [A \wedge W_A \Longrightarrow C]$ also holds. Effectively, we "play back" the waveforms on the interface nodes. Using theorem 4, we show that for any model $\mathcal{M}$ and any temporal formula F, if $\models_{\mathcal{M}} [A \Longrightarrow F]$ and $\models_{\mathcal{M}} [A \wedge F \Longrightarrow C]$, then $\models_{\mathcal{M}} [A \Longrightarrow C]$, and therefore this pair of verifications is sufficient to prove the desired property. An extension of this technique can handle partitioning with bidirectional communication[18].

C. Verification of partitioned designs

Consider a circuit $\mathcal{M}$. From $\mathcal{M}$ we create two conservative approximations $\mathcal{M}_1$, and $\mathcal{M}_2$, with the set of nodes N_1, and N_2, respectively. We assume that $\mathcal{M}_1$ generates signals for $\mathcal{M}_2$. Given a trajectory formula A, we perform waveform capture over the nodes at the interface of $\mathcal{M}_1$, and $\mathcal{M}_2$ to construct a trajectory formula W_A. From lemma 5 we know that $\models_{\mathcal{M}_1} [A \Longrightarrow W_A]$ holds. If $\models_{\mathcal{M}_2} [A \wedge W_A \Longrightarrow C]$ is true, then theorem 4 states that $[A \Longrightarrow C]$ holds for the entire design $\mathcal{M}$. From these properties of individual components, and the symmetry in the components, we can infer a family of properties for the entire design, as shown below.

Let σ_1 and σ_2 be symmetries of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. If $\models_{\mathcal{M}_1} [A \Longrightarrow W_A]$, and $\models_{\mathcal{M}_2} [A \wedge W_A \Longrightarrow C]$ are both true, then the following hold (theorem 2):

$$\models_{\mathcal{M}_1} [\sigma_1(A) \Longrightarrow \sigma_1(W_A)] \quad (6)$$

$$\models_{\mathcal{M}_2} [\sigma_2(A) \wedge \sigma_2(W_A) \Longrightarrow \sigma_2(C)] \quad (7)$$

To compose the results of equations 6 and 7 by applying theorem 4, the assertions and the symmetry transformations need to obey the following two constraints:

- *Interface constraint:* The signals at the interface of $\mathcal{M}_1$ and $\mathcal{M}_2$ should match, i.e., $\sigma_1(W_A) = \sigma_2(W_A)$.
- *Antecedent constraint:* The trajectory formula A should be invariant under the symmetry transformations σ_1, and σ_2, i.e., $\sigma_1(A) = \sigma_2(A)$.

Given these constraints, equations 6 and 7 may be written as $\models_{\mathcal{M}_1} [\sigma_1(A) \implies \sigma_1(W_A)]$ and $\models_{\mathcal{M}_2} [\sigma_1(A) \wedge \sigma_1(W_A) \implies \sigma_2(C)]$ respectively, which leads to the following:

$$\models_{\mathcal{M}} [\sigma_1(A) \implies \sigma_2(C)] \tag{8}$$

Thus, given symmetry properties σ_1 and σ_2, and the two constraints, we have established a new assertion. In this manner, one can establish a family of properties for the entire design by using symmetries on individual components. It should be noted that in general for a trajectory formula F, $\sigma_1(F) = \sigma_2(F)$ does not imply $\sigma_1 = \sigma_2$.

The antecedent constraint, $\sigma_1(A) = \sigma_2(A)$ requires that σ_1 and σ_2 transform atoms in A identically. This requirement is easy to satisfy. For instance, let all the atoms in A come from the nodes of $\mathcal{M}_1$ and not from $\mathcal{M}_2$. Given this, σ_2, should also include those structural and data transformations in σ_1 which transform atoms of A. If σ_2 does not contain these transformations, then it can be trivially extended to $\sigma_2' = [\sigma_2, \sigma_{a_1}, ..., \sigma_{a_k}]$ by composing σ_2 with the structural and data transformations $\sigma_{a_1}, ..., \sigma_{a_k}$ in σ_1 which act on A, but are not present in σ_2. Since these additional transformations $\sigma_{a_1}, ..., \sigma_{a_k}$ do not alter the state of $\mathcal{M}_2$, σ_2' is also a symmetry of $\mathcal{M}_2$. Thus, from equations 6 and 7 we can establish that $\models_{\mathcal{M}} [\sigma_1(A) \implies \sigma_2'(C)]$. In general, we have to handle the case where A includes atoms of nodes from both $\mathcal{M}_1$ and $\mathcal{M}_2$. In such a case, both σ_1 and σ_2 may be extended to σ_1' and σ_2' as described above to ensure that $\sigma_1'(A) = \sigma_2'(A)$, while remaining symmetries of $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. This allows one to establish $\models_{\mathcal{M}} [\sigma_1'(A) \implies \sigma_2'(C)]$ for the overall design. In some cases, the antecedent constraint is not required. The subsection ahead describes this case.

C.1 A special case of verifying partitioned designs

Let $\mathcal{M}_1$, and $\mathcal{M}_2$ be conservative approximations of $\mathcal{M}$. Let σ_1 and σ_2 be the symmetries of $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. When the assertions and the circuit have a specific structure, one verify partitioned designs by imposing only the interface constraint.

Consider the problem of verifying an assertion of the form $[A_1 \wedge A_2 \wedge A' \Rightarrow C]$ for $\mathcal{M}$. Let A_1 contain atoms only from $\mathcal{M}_1$, and A_2 contain atoms only from $\mathcal{M}_2$. Furthermore, let A' contain only those atoms which are not transformed by the symmetries under consideration for the design. Common examples of such atoms are those corresponding to clock inputs, control signals, scan signals etc.

As earlier, Given the trajectory formula $A_1 \wedge A'$, we perform waveform capture for $\mathcal{M}_1$ over the nodes at the interface of $\mathcal{M}_2$ to construct a trajectory formula W_A. Using this waveform, and the trajectory formulas A', and A_2, one can establish that $[W_A \wedge A_2 \wedge A' \Rightarrow C]$ holds for $\mathcal{M}_2$. These assertions on $\mathcal{M}_1$, and $\mathcal{M}_2$ allow one to prove that $[A_1 \wedge A_2 \wedge A' \Rightarrow C]$ holds for $\mathcal{M}$. Below, theorem 5 states this.

Theorem 5: Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be conservative approximations of $\mathcal{M}$. If $\models_{\mathcal{M}_1} [A_1 \wedge A' \Rightarrow W_A]$, and $\models_{\mathcal{M}_2} [A' \wedge W_A \wedge A_2 \Rightarrow C]$, then $\models_{\mathcal{M}} [A_1 \wedge A_2 \wedge A' \Rightarrow C]$.

Proof: From the definition of a trajectory, and theorem 1 it is easy to that if $\models_{\mathcal{M}_1} [A_1 \wedge A' \Rightarrow W_A]$ is true, then $\models_{\mathcal{M}_1} [A_1 \wedge A' \Rightarrow W_A \wedge A']$ is also true.

Also, $\models_{\mathcal{M}} [P \Rightarrow Q]$ implies $\models_{\mathcal{M}} [P \wedge R \Rightarrow Q \wedge R]$ (result from [14]). Applying this result to $\models_{\mathcal{M}_1} [A_1 \wedge A' \Rightarrow W_A \wedge A']$ with trajectory formula A_2, we obtain $\models_{\mathcal{M}_1} [A_1 \wedge A' \wedge A_2 \Rightarrow W_A \wedge A_2 \wedge A']$.

This result and $\models_{\mathcal{M}_2} [A' \wedge W_A \wedge A_2 \Rightarrow C]$, upon application of theorem 3, and lemma 8 yield $\models_{\mathcal{M}} [A_1 \wedge A_2 \wedge A' \Rightarrow C]$. □

Since σ_1, and σ_2 are symmetries of $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, given $\models_{\mathcal{M}_1} [A_1 \wedge A' \Rightarrow W_A]$, and $\models_{\mathcal{M}_2} [A' \wedge W_A \wedge A_2 \Rightarrow C]$, the following holds:

$$\models_{\mathcal{M}_1} [\sigma_1(A_1) \wedge \sigma_1(A') \Rightarrow \sigma_1(W_A)] \tag{9}$$

$$\models_{\mathcal{M}_2} [\sigma_2(A') \wedge \sigma_2(W_A) \wedge \sigma_2(A_2) \Rightarrow \sigma_2(C)] \tag{10}$$

If A' is not transformed by σ_1 and σ_2, and the signals at the interface of $\mathcal{M}_1$ and $\mathcal{M}_2$ match, i.e., $\sigma_1(W_A) = \sigma_2(W_A)$, then applying theorem 5 to equations 9, and 10 yields the following:

$$\models_{\mathcal{M}} [\sigma_1(A_1) \wedge \sigma_2(A_2) \wedge A' \Rightarrow \sigma_2(C)] \tag{11}$$

Thus, from the individual component symmetries, we have established a new property of the system.

D. *Creation of conservative approximations*

Creation of a conservative approximation of a circuit intuitively means creating a reduced version of the circuit which produces more Xs than does the original circuit. The limiting case of the conservative approximation of a circuit is one whose every node produces only an X for every input sequence. Such a "strict" approximation is of little use, however. We would like to create conservative approximations in a more controlled manner, so that we can selectively disable desired portions of the circuit. The following two techniques are our means of doing so:

- Attach "X-drivers" to internal circuit nodes.
- Strengthen transistors which are adjacent to X-drivers.

An X-driver is a strong source of Xs. Attaching an X-driver to a node is equivalent to converting the node to an input node set to the constant value X. The X-driver is analogous to the Vdd and ground nodes, which are strong sources of 1 and 0, respectively. An accompanying technique we employ to create a conservative approximation is strengthening transistors adjacent to X-drivers. Intuitively, this strengthening aids the propagation of Xs through the circuit. As will be shown later, both these techniques monotonically move all the circuit nodes towards X.

As an example, suppose we wish to create a reduced model for the circuit in Figure 5 by eliminating nodes `a`, `b`, and `s`. Then we could describe the remaining portions of CCSN1 by the excitation expressions shown in Figure 12. One can see that these expressions were obtained from those of Figure 5 by simplifying the result of setting the

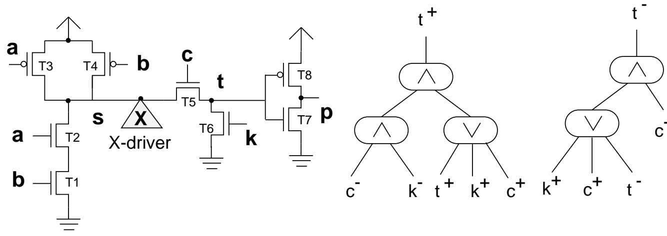

Fig. 12. Conservative approximation of CCSN1.

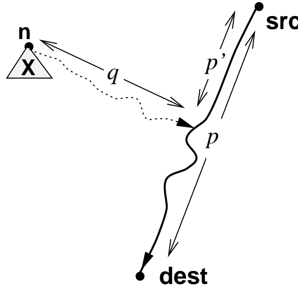

Fig. 13. Paths in a switch-level circuit conservative approximation

leaves for all eliminated atoms to false. This conservative approximation could be used to verify circuit operation for the cases where node c is set to 0. We have modified Anamos to generate these simplified expressions directly, avoiding the need to ever generate a complete model. In particular, we would replace node s in the example circuit by a X-driver.

One final task which remains is to show that the two circuit transformation techniques discussed above actually create a conservative approximation. As mentioned in section III-D, the steady state response of a node in a switch-level network depends on all the unblocked paths to that node. In figure 13, consider the path p from src to dest. If the path were unblocked before the X-driver is attached to node n, then, two possibilities arise after the X-driver is introduced. The first is that the path still remains unblocked, and the node response stays the same. The second is that a stronger path q from n blocks p, i.e., $|q| > |p'|$, where p' is a prefix of p originating at the root. In this case, the stronger path q will dominate the response at dest, i.e., dest will monotonically move towards X. The effect of strengthening transistors adjacent to a X-driver is similar — stronger paths from the X-driver may dominate existing paths, sending their destination towards X. Note that the argument above also accounts for the case when n is on p. In such a case, the length of path q is 0.

VIII. Putting it together: Verifying a SRAM circuit

Consider the 16-bit (1 bit/word) SRAM circuit shown in Figure 14. This circuit consists of the the following major components — row decoder, column address latches, column multiplexer (Mux) and the memory cell array core. To simplify the discussion here, many essential SRAM components like precharge column, write-drivers etc. have not been shown in the figure. This is a standard organization

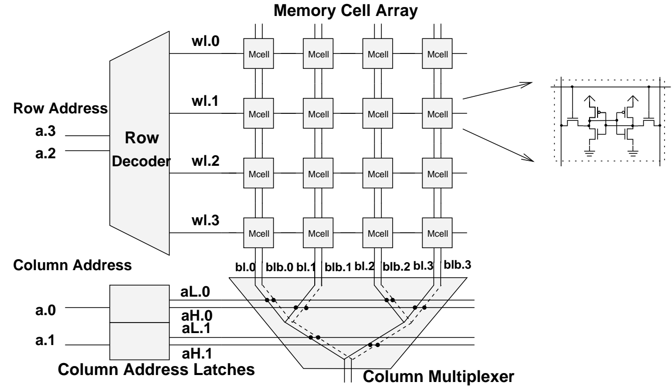

Fig. 14. SRAM circuit

followed in many larger industrial SRAM arrays [13].

To verify this circuit we must show that the read and write operations work correctly. To verify the write operation, one must show that if we write data to a memory location, then the correct memory location gets updated. Let din be the input node, m[0], ..., m[15] be the memory storage nodes, and read and write be the control signals. For simplicity, we ignore all clocks and we assume that from setting data, address and control signals to memory location update takes only two time steps. Let $(\mathsf{din} = d)$ be the shorthand for the trajectory formula $(d \rightarrow din^{+}) \wedge (\neg d \rightarrow din^{-})$, where d is a symbolic Boolean variable. The assertion below states that if we write data d at address 0 (all address lines are held low), it ends up at m[0], the node for memory location 0 two time steps later.

$$[(\mathsf{din} = d) \wedge a.3^{-} \wedge a.2^{-} \wedge a.1^{-} \wedge a.0^{-} \wedge write^{+} \wedge read^{-} \Longrightarrow \mathbf{NN}(\mathsf{m[0]} = d)]$$

Details on introducing symbolic Boolean variables for addresses to write compact assertions may be found in [19], [18]. Other properties of interest, such as if a memory location is addressed and read from, then the correct value must appear at the output, and that non-addressed memory locations do not change can be expressed with STE assertions. [19], [18] describe these in more detail.

The machinery we have built in the previous sections allows us to verify the read or write operation for only one location and, from the symmetry in the SRAM circuit, conclude that the operation works for every location. We expand on this below, starting with a discussion of SRAM symmetries.

A. Symmetries of a SRAM

Consider the decoder in Figure 15. For any memory operation, the value of the row address assigned to nodes a.2 and a.3, causes one of the word lines wl.0, wl.1, wl.2 and wl.3 to be active. The figure shows that wl.0 is active for row address 00. The same waveform occurs on the active word line regardless of the address. This mixed symmetry

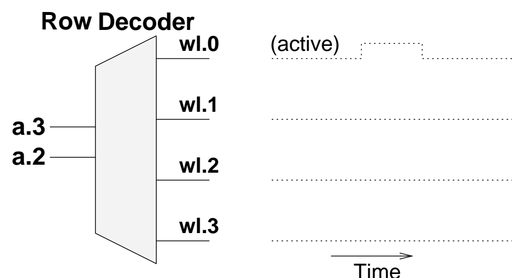

Fig. 15. Row decoder and signal waveforms on word lines for row address 00.

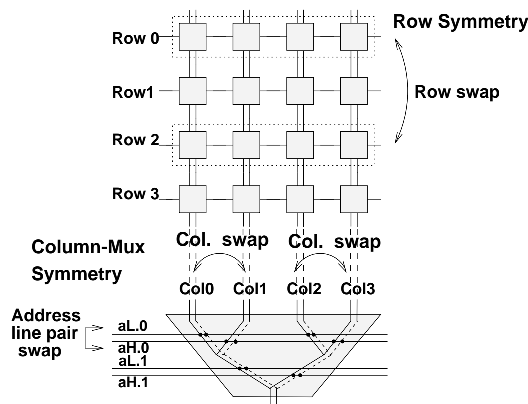

Fig. 16. Structural symmetries of the SRAM core.

of the decoder is expressed by the group of transformations generated by transformations σ_0 and σ_1:

$$\sigma_0 = [\mathsf{a.2}^{\pm}, \mathsf{wl.0} \leftrightarrow \mathsf{wl.1}, \mathsf{wl.2} \leftrightarrow \mathsf{wl.3}]$$
$$\sigma_1 = [\mathsf{a.3}^{\pm}, \mathsf{wl.0} \leftrightarrow \mathsf{wl.2}, \mathsf{wl.1} \leftrightarrow \mathsf{wl.3}]$$

Transformation σ_i indicates that complementing bit i of the row address causes an exchange of signal waveforms for each pair of word lines j and k such that the binary representations of j and k differ at bit position i. The column address latches obey the "decoder" symmetry expressed by Equation 3.

The mixed symmetries of the decoder and the column address latches can be verified by symbolic simulation, where a single run of the simulator with n symbolic Boolean values at the circuit inputs is equivalent to 2^n runs of a conventional simulator with 0-1 values. For example, to verify that σ_0 is a symmetry of the decoder, we symbolically simulate the decoder with symbolic values s_0 and s_1 at the decoder inputs a.2, and a.3 in Figure 15. As the simulation proceeds, we check that a substitution of $\overline{s_0}$ for s_0 in the symbolic waveform for wl.0 (resp., wl.2) matches the symbolic waveform on wl.1 (resp., wl.3).

Figure 16 illustrates the two structural symmetries of the SRAM core and column Mux combination. The *row symmetry* arises from the invariance of the core-mux circuit structure under permutations of the rows of the core. The *column-mux symmetry* arises from the invariance of the circuit structure under a swap of column address latch output pairs accompanied by a corresponding exchange of columns. For example, in Figure 16, a swap of aH.0 and

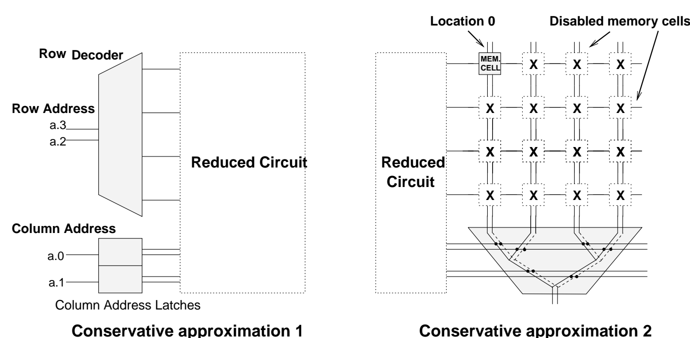

Fig. 17. Conservative approximations of the SRAM.

aL.0 accompanied by a swap of column 0 with 1, and a swap of column 2 with 3 is a symmetry of the circuit.

We verify the core-Mux symmetries in two parts. First we verify that arbitrary row and column permutations are symmetries of the core. Verification that the exchange and rotate permutation generators for rows and columns are symmetries suffices for this. This gives a total of 4 symmetry checks for the core. Next we verify the column-mux symmetry for the Mux. In the figure, the generators of the four different column address line pair permutations are the two permutations associated with each column address latch output pair. Therefore, two symmetry checks verify the column-mux symmetry. In general n symmetry checks must be done for the Mux in a SRAM with n column address line pairs.

B. Verification steps

In order to verify the SRAM circuit we go through the following sequence of steps.

1. **Circuit partitioning** — We partition the SRAM circuit into two parts. The first part consists of the decoder with the column address latches. The second part consists of the memory core and the column Mux.
2. **Symmetry verification** — Using symbolic simulation we verify the symmetries of the decoder and column latches. Using circuit graph isomorphism checks we verify the symmetries of the core and the column Mux.
3. **Conservative approximations** — We create two conservative approximations of the SRAM (Figure 17). In the first model, the memory core and the column Mux are "disabled". In the second model, the decoder, the column address latches are disabled, and all the memory cells except that for location 0 are disabled. Figure 18 shows how we conservatively disable a SRAM cell by attaching X-drivers, and strengthening transistors adjacent to the X-drivers. The figure also shows the optimization that chains of N or P transistors from Vdd or ground to an X-driver may be eliminated from the circuit — their presence does not alter circuit behavior.
4. **Waveform capture** — Given the assertion $[A \Longrightarrow C]$ specifying an operation for memory location 0, we use the antecedent A to symbolically simulate conservative approximation 1. During the process of symbolic simulation we record the signal waveforms on the outputs of the decoder

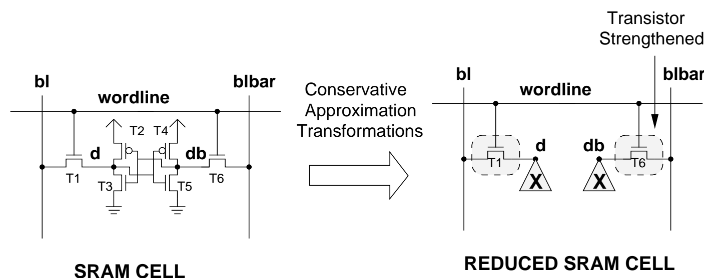

Fig. 18. Creation of a conservative approximation of a SRAM Cell

SRAM Size (bits)	Time (CPU Secs.)	Memory (MB)
1K	1.7	0.69
4K	2.1	0.74
16K	2.5	0.88
64K	3.2	1.10
256K	4.2	1.52

TABLE IV
Decoder and col. latch symmetry checks.

and the column address latches. We construct a trajectory formula W, which captures the signal values on the outputs recorded above. As discussed earlier, it can be shown that $[A \Longrightarrow W]$ is true.
5. **Verification of SRAM core** — Finally, with conservative approximation 2, we show that given the waveform W, and the antecedent A, the consequent C is true, i.e., $[A \wedge W \Longrightarrow C]$. From the earlier discussion in section VII, if $[A \Longrightarrow W]$ and $[A \wedge W \Longrightarrow C]$ are both true, then we can conclude that $[A \Longrightarrow C]$ is true, i.e., the memory operation is verified for location 0. Given the symmetries of the circuit we can then conclude that the operation works correctly for every memory location.

IX. Experiments and Results

All the time and memory figures in this section have been measured on a Sun SparcStation-20. We used the Anamos switch-level analyzer to generate switch-level models [5]. We modified Anamos to make it possible to attach *X-drivers* to circuit nodes to generate reduced models (conservative approximations) of switch-level circuits. Table II shows the results of model generation for SRAM circuits of varying sizes for the full and reduced circuit models. The full circuit model contains about 73 operations for each memory bit. However, for circuits larger than 16K, it was not possible to generate the full circuit model within reasonable time or memory bounds (empty table entries). Conservative approximations of SRAM circuits, on the other hand, can be generated for much larger circuits for a miniscule fraction of the cost of the full model. The reduced model size grows proportional to the square root of the SRAM size, and its generation time and memory is proportional to the SRAM size.

To verify a structural symmetries, we do graph isomorphism checks as outlined in section VI. We have modified the isomorphism checking code [2] from Anamos for our purpose. Table III reports the running time and memory taken for converting one instance of the memory core or column mux permutation into a canonical circuit, and the total time to do all the isomorphism checks. The total time and memory requirements scale linearly with the SRAM size. Table III reports the resources required to check the structural symmetries. The memory core has four symmetry generators. We verify that each of these is a symmetry (columns 2,3,4). Column 4 indicates the number of symmetry checks that are required. The column multiplexer has a number of symmetry generators equal to the number of column address lines. We also verify each of these generators (columns 5,6,7). The total time is reported in column 8. Table IV shows the time and memory required to check the decoder and column address latch symmetries by symbolic simulation.

We used the Voss verification system [21] to verify the reduced SRAM circuit. Table V shows the running time and the memory required for verifying the write operation for location 0. In addition, we verify two other properties — that the read operation reads the value stored at location 0 (Table VI), and that operations at other addresses do not change the data in location 0 (Table VII). The time and memory required to verify these other operations is similar to that of the write. Much of the verification time and memory is taken to read in the reduced circuit model and representing it. So, it is not surprising that the time and memory requirements grow roughly proportional to the square root of the memory size.

The total verification time for a SRAM circuit is the sum of the times in tables II, III, IV and V. For example, to verify a 64K SRAM, 170.7 secs. are required to generate the reduced circuit model, a total of 952.4 + 3.2 secs. are required to verify the circuit symmetries, and an additional 6.0 + 6.6 + 6.1 secs. are required to verify the reduced model for all the operations. This gives a total verification time of 1145.0 secs. It is interesting to note that symmetry checks dominate much of this time. In the verification process, the only time we ever work with the complete circuit is the symmetry check phase. This partially explains the reason for the relatively large time and memory requirements of this phase. However, the circuit isomorphism code we have used is a simple modification of that in Anamos. There is considerable scope for reducing time and memory by developing a specialized circuit isomorphism checker without the baggage from Anamos. Circuit netlist comparison programs such as schematic-layout checkers routinely use these isomorphism algorithms to handle graphs with tens of millions of nodes and edges.

X. Conclusion

We show that exploiting symmetry allows one to verify systems several orders of magnitude larger than otherwise possible. We have verified memory arrays with multi-million transistors. The techniques we have developed also

SRAM size (bits)	No. of Transistors	Model Size (Bool. ops)		Anamos Time (CPU Secs.)		Anamos Memory (MB)	
		Full	Reduced	Full	Reduced	Full	Reduced
1K	6690	79951	2781	120	4.1	9.6	0.9
4K	25676	307555	5462	863	14.1	36.8	2.1
16K	100566	1205239	10895	7066	43.2	144.2	6.0
64K	397642	—	21960	—	170.7	—	22.0
256K	1581494	—	44545	—	732.7	—	80.0

TABLE II
Generation of SRAM model: Full vs. Reduced model.

SRAM Size (bits)	Memory Core			Column Multiplexer			Total Isomorph. Check Time (Secs.)
	CPU Time (Secs.)	Memory (MB)	No. of checks	CPU Time (Secs.)	Memory (MB)	No. of checks	
1K	2.6	1.6	4	0.3	0.19	5	11.9
4K	11.1	6.5	4	0.5	0.38	6	47.4
16K	51.2	26.0	4	1.3	0.74	7	214.1
64K	232.1	104.0	4	3.0	1.44	8	952.4
256K	1135.6	416.0	4	6.6	3.50	9	4601.8

TABLE III
Symmetry checks for memory core and column multiplexer.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.5	0.79
4K	2.0	1.05
16K	3.0	1.80
64K	6.0	2.84
256K	18.5	4.26

TABLE V
Verification of reduced SRAM writes.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.7	0.79
4K	2.2	1.05
16K	3.0	1.80
64K	6.2	2.84
256K	18.7	4.26

TABLE VI
Verification of reduced SRAM reads.

SRAM Size (bits)	Verif. Time (CPU Secs.)	Verif. Memory (MB)
1K	1.8	0.87
4K	2.2	1.12
16K	3.1	1.82
64K	6.6	2.90
256K	19.6	4.35

TABLE VII
Verification that unaddressed location unchanged.

successfully overcome the switch-level analysis bottleneck for such circuits. We believe that with our work the problem of SRAM verification is solved. With more computational resources, and some fine-tuning of our programs, the results of our experiments indicate that we can verify multi-megabit SRAM circuits. The techniques we have presented can be used in a rather straightforward manner to exploit symmetries in other hardware units like set associative cache tags, where every set is identical in structure. One direction for future work in the short run would be to extend these ideas to verify content addressable memories. In the longer run, it would be interesting to apply these ideas to verify hardware units other than memory arrays. Candidates for such an application include a processor datapath, where one can find the presence of structural symmetries because of bit-slice repetition, and data symmetries arising from the datapath operations.

References

[1] S. Aggarwal, R. P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification*, volume 3, pages 91–34. IFIP, Elsevier Science Publishers B.V. (North Holland), 1983.

[2] Derek L. Beatty and Randal E. Bryant. Fast incremental circuit analysis using extracted hierarchy. In *25th ACM/IEEE Design*

Automation Conference, pages 495–500, June 1988.

[3] Randal E. Bryant. A switch-level model and simulator for MOS digital systems. *IEEE Transactions on Computers*, C-33(2):160–177, Feb. 1984.

[4] Randal E. Bryant. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):618–633, July 1987.

[5] Randal E. Bryant. Boolean analysis of MOS circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):634–649, July 1987.

[6] Randal E. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on Computer-Aided Design*, CAD-10(1):94–102, January 1991.

[7] Edmund M. Clarke, Robert Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.

[8] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of 5th International Conference on Computer Aided Verification*, pages 450–462, 1993.

[9] C. Ebeling. GeminiII: A second generation layout validation program. In *Proceedings of the International Conference on Computer Aided Design*, 1992.

[10] C. Ebeling and O. Zazicek. Validating VLSI circuit layout by wirelist comparison. In *Proceedings of the International Conference on Computer Aided Design*, pages 172–173, 1982.

[11] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In *Proceedings of 5th International Conference on Computer Aided Verification*, pages 463–478, 1993.

[12] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[13] Stephen T. Flannagan, Perry H. Pelley, Norman Herr, Bruce E. Engles, Taisheng Feng, Scott G. Nogle, John W. Eagan, Robert J. Dunnigan, Lawrence J. Day, and Robert I. Kung. 8-ns CMOS 64k × 4 and 256k × 1 SRAMs. *IEEE Journal of Solid-State Circuits*, pages 1049–1054, October 1990.

[14] Scott Hazelhurst and Carl-Johan H. Seger. A simple theorem prover based on symbolic trajectory evaluation and OBDDs. Technical Report 93-41, Department of Computer Science, University of British Columbia, 1993.

[15] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level Petri nets. *Theoretical Computer Science (Netherlands)*, 45(3):94–102, 1986.

[16] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[17] Kurt Jensen. Condensed state spaces for symmetrical colored petri nets. *Formal Methods in System Design*, 9:7–40, 1996.

[18] Manish Pandey. Formal verification of memory arrays. Technical Report CMU-CS-97-162, Department of Computer Science, Carnegie Mellon University, August 1997. PhD Thesis.

[19] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant. Formal verification of PowerPC(TM) arrays using symbolic trajectory evaluation. In *33rd ACM/IEEE Design Automation Conference*, pages 649–654, June 1996.

[20] P.H.Starke. Reachability analysis of Petri nets using Symmetries. *Systems Analysis - Modeling - Simulation (Germany)*, 8(4-5):293–303, 1991.

[21] Carl-Johan H. Seger. Voss—a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, 1993.

[22] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.