Using Abstract Specifications to Verify PowerPC^{TM*} Custom Memories by Symbolic Trajectory Evaluation

Jayanta Bhadra[1,2], Andrew Martin[1], Jacob Abraham[2], and Magdy Abadir[1]

[1] Motorola Inc.
[2] The University of Texas at Austin
Jayanta.Bhadra@Motorola.Com

Abstract. We present a methodology in which the behavior of a switch level device is specified using abstract parameterized regular expressions. These specifications are used to generate a finite automaton representing an abstraction of the behavior of a block of memory comprised of a set of such switch level devices. The automaton, in conjunction with an Efficient Memory Model [1], [2] for the devices, forms a symbolic simulation model representing an abstraction of the array core embedded in a larger design under analysis. Using Symbolic Trajectory Evaluation, we check the equivalence between a register transfer level description and a schematic description augmented with abstract specifications for one of the custom memories embedded in the MPC7450 PowerPC processor.

1 Introduction

At Somerset, Symbolic Trajectory Evaluation (STE) is routinely used to check equivalence between Register Transfer Level (RTL) and switch level views of embedded custom memories [3]. The assertions are generated automatically from the RTL description of the memory under verification using the technique described by Wang [4]. The assertions are verified against a switch level model of the circuit using STE. The switch level model is obtained from transistor netlists using Anamos [5], which partitions the design into channel connected subcomponents, and then analyzes each component as a set of simultaneous switch equations. Although Anamos is quite sophisticated as far as switch level analyzers are concerned, there are still many analog circuit effects that it ignores. That is why, in spite of its sophistication, an Anamos switch level analysis, which views the circuit as a system of switches, is unsuited to a custom static RAM core circuit, which is an inherently analog design. It is quite easy to demonstrate input sequences for which the resulting switch level model predicts one result, while a more sophisticated analog simulation would predict another.

One obvious approach to address these problems would be to increase the sophistication of the switch level model. Given that the circuitry in a custom

* PowerPC is a trademark of the International Business Machines Corporation, used under license therefrom.

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 386–402, 2001.

static RAM core solves a fundamentally analog problem: how to drive a very large load with a very small bitcell, we feel that this approach is unlikely to prove satisfactory in the long term. This is because, an improved switch level model would continue to suffer from the same inaccuracy problem – representing analog devices by switches.

In reality, the array core is designed to operate over a very limited range of input stimuli. Each such "pattern" is validated by extensive analog simulation over a variety of process corners and operating conditions using a circuit simulator such as Spice. These "certified" patterns are known to work. Any other pattern that has not been simulated with Spice, is assumed to fail. This paper presents a notation, based on regular expressions, for describing such "certified" patterns, and the effect that each has on the internal state and outputs of the RAM core. From this notation, we build an automata based simulation model representing an abstraction of the array cores embedded in the larger design under analysis. This abstract model updates its internal state – representing the state of the RAM – as specified, provided that the input conforms to at least one "certified" input pattern. If, however, an "uncertified" input sequence is presented, the RAM internal state and its outputs are set to the "unknown" value "X", where they will remain until altered by a new "certified" input pattern.

In addition to being more conservative than the Anamos generated model, our abstraction of the array cores is also more efficient. The verification methodology, in which the abstraction is used, is based upon symbolic simulation. In the course of a verification, symbolic values are "written" to symbolic addresses within the memory core. Our BDD based implementation (as reported by Krishnamurthy et al in [3]), using an Anamos generated model, must necessarily maintain several unique BDDs for each bitcell in the array. This leads to a BDD table whose size is at best a linear function of the size of the array being verified. In contrast, the abstract approach is able to make use of the well known Efficient Memory Model (EMM) due to Velev et al [1,2], which represents the state of the memory using an association list. Each symbolic write into an array need only add one element to the head of this list. As a result, it is possible to represent the result of a symbolic write using only as many BDD nodes as are required to represent the address and data – typically a logarithmic function of the array size.

The approach presented here represents a fine balance between implementation efficiency and expressive power. The semantics of the regular expressions have been designed to admit a tractable automata based implementation. Moreover, in several places the result is weakened to provide a tractable implementation at the expense of possible false negative verification results. In practice, we do not believe that these false negative results will be problematic.

An interesting facet of the combination of regular expressions representing an implementation, with STE, is the marriage of two opposing forms of non-determinism. The traditional partially ordered state space of STE represents a form of demonic non-determinism. Values that are lower in the partial order represent an undetermined choice between the values which lie above them. A specification is satisfied if and only if the consequent portion is satisfied by the

weakest *trajectory* satisfying the antecedent. That is, every non-deterministic choice that the implementation can make must satisfy the specification. Increasing the amount of non-deterministic choice reduces the number of specifications a given model will satisfy. In contrast, non-determinism that results from alternation in a regular expression describing legal input sequences for a RAM, is angelic. It suffices to find one satisfactory production for any given input sequence. Increasing the amount of non-deterministic choice increases the number of specifications a given model will satisfy.

In this paper we present a methodology by which the user is allowed to represent or *abstract* a block of switch level devices by a set of regular expression specifications. A state machine model is obtained from the specifications and an EMM is used to store values written into the devices. The state machine controls the read/write operations performed on the EMM and provides the outputs from the abstracted block. Our goal is to show that the assertions generated from the RTL description of the circuit corresponds to the switch level description of the circuit augmented with the composition of the state machine model and the EMM. As a future work, we propose to show that the composition of the state machine model and the EMM is a conservative approximation of the abstracted switch level device block. This would let us claim that the switch level model *implies* the RTL model.

2 STE Background

Symbolic Trajectory Evaluation [6] requires a system expressed as a model of the form $\langle S, \preceq, y\rangle$ where S is a set of states, $\preceq$ is a partial order on the states ($\preceq \subseteq S \times S$) and $y : S \to S$ is a state transition function. S must form a complete lattice and y must be monotonic under $\preceq$, i.e., if $s \preceq t$ then $y(s) \preceq y(t)$.

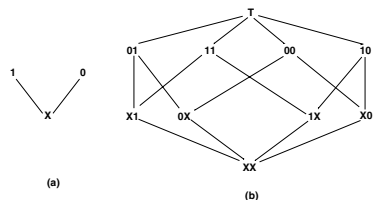

Fig. 1. Lattice Structures

Traditionally, for switch level verification, STE operates over the ternary logic domain $\mathcal{T} = \{0, 1, X\}$, where X denotes an "unknown" value. In order to formalize the concept of an "unknown" value, define a partial order $\preceq$ on $\mathcal{T}$ as illustrated in Fig. 1(a), so that $X \preceq 0$, and $X \preceq 1$. States of an STE model are vectors of elements taken from $\mathcal{T}$. The partial order over $\mathcal{T}$ is extended pointwise to yield a partial order on the space $\mathcal{T}^n$. Unfortunately, $\langle \mathcal{T}^n, \preceq\rangle$ is not a complete lattice, since the least upper bound does not exist for every pair of vectors in $\mathcal{T}^n$. Introduction of a new *top* element, $\top$, solves this problem. Intuitively, $\top$ can

be viewed as an "overconstrained" state, in which some node is both 0 and 1 at the same time. This makes the state space to be $S = \mathcal{T}^n \cup \{\top\}$. For example, Fig. 1(b) shows S for $n = 2$.

In STE, the state of the circuit model includes values on the input, internal and output nodes of the circuit. So n is the collective number of all circuit nodes. The state transition function expresses constraints on the values the nodes can take one time unit later, given the node values at the current time, under some discrete notion of time. Since the value of an input is controlled by the external environment, the circuit does not constrain its value; hence the transition function sets it to X. On the other hand, the value of any other node is determined by the functionality of the circuit and the circuit state. Figure 2 shows the next state function of a unit delay inverter.

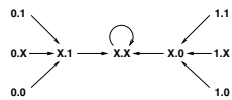

Fig. 2. Transition function of a unit delay inverter (in.out)

A sequence of states, or *behavior*, $\sigma = s_0 s_1 s_2 \ldots$ defines a trajectory if it has at least as much information as given by the application of the next-state function to successive elements,

$$\forall i \geq 0 : (y(s_i) \preceq s_{i+1})$$

The set of all trajectories $\{\sigma\}$ is called the language of the system, L. Specifications are *trajectory assertions* of the form $A_\phi \rightarrow C_\phi$. A_ϕ and C_ϕ are functions (called *trajectory formulas*) from valuations, ϕ, of a set of symbolic variables to predicates over state sequences. STE gives a procedure to determine the set of assignments of values to the variables such that every trajectory of the system that satisfies A_ϕ also satisfies C_ϕ.

The complete lattice, S, represents a form of non-determinism that fundamentally affects the validity of a specification in S. Consider two distinct trajectories $\sigma_1, \sigma_2 \in L$. If $\sigma_1 \preceq \sigma_2$, where $\preceq$ is naturally extended over state sequences, then the states in σ_1 are *not higher* than those in σ_2 in the partial order over S. Each state in σ_1 contains less information, or, more Xs or unconstrained node values than the corresponding state in σ_2. Any specification that is satisfied by σ_1 has to be satisfied by all the trajectories that are obtained by *all* possible choices on the unconstrained node values in the states of σ_1. Constraining nodes to values stronger than those in the states of σ_1 means obtaining trajectories that are necessarily *higher* than σ_1. Since σ_2 is one of the non-deterministic choices that can be made by constraining the node values in the states of σ_1, the specifications that are satisfied by σ_1 constitute a subset of those satisfied by σ_2. So, as one moves *down* the lattice, thus weakening the states in the corresponding trajectories, one decreases the number of specifications that are satisfied by a given circuit model. Hence, the partially ordered state space S represents a demonic non-determinism.

Observation 1 *If Θ is a specification and functions y_1 and y_2 are such that $\forall s \in S, y_1(s) \preceq y_2(s)$, then if the STE model $\langle S, \preceq, y_1 \rangle$ satisfies Θ, then the STE model $\langle S, \preceq, y_2 \rangle$ also satisfies Θ.*

3 Preliminaries of Array Abstraction

3.1 The Broader Picture

We aim at establishing that the state machine model produced from the regular expression definitions, when combined with the surrounding switch level model of the circuit, satisfies the RTL specification. Let M be the block of an array being abstracted, E be its environment circuitry and $\hat{M}$ be the abstraction. Let V_{RTL} be the set of variables in P_{RTL}, the RTL specification, and $\phi \in \mathcal{B}^{V_{RTL}}$ be some assignments to the variables, where $\mathcal{B} = \{0, 1\}$. The verification obligation is to show that the composition of the environment and the abstraction models the property: $\forall \phi \in \mathcal{B}^{V_{RTL}}\{(E \parallel \hat{M}) \models P_{RTL}(\phi)\}$. As a future work, we intend to show that the state machine model is a conservative approximation of the switch level model, thus establishing $(M \rightarrow \hat{M})$, which in turn would imply $(E \parallel M) \rightarrow (E \parallel \hat{M})$. This, in conjunction with the current work, would prove that the switch level model satisfies the RTL specification.

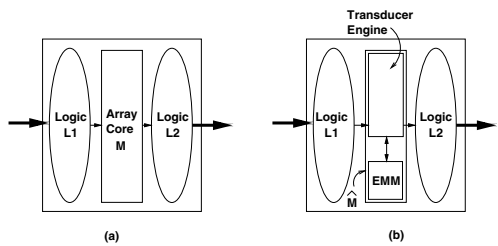

Fig. 3. Abstracting an Array Core

The array core (M) is a piece of logic containing all the bitcells of the array, that accepts certain inputs from some logic module ($L1$ in Fig. 3(a)), stores the values of the bitcells and produces outputs to another logic module ($L2$). The environment E is modeled collectively by $L1$ and $L2$. Figure 3(b) shows the abstract module ($\hat{M}$) that replaces the array core. The sub-module labeled "Transducer Engine" is a state machine model generated from abstract specifications of the array core. It accepts the inputs from $L1$, updates its internal state(s), *writes* or *reads* data to and from the sub-module labeled as "EMM" and provides its outputs to $L2$.

3.2 A Simple Example

Bitcells represent an integral part of an array core in custom-designed memories. They are read from or written to by the application of complex sequences

of values to specific control signals (like precharge, word selects etc). In order to abstract an entire column of bitcells forming a single channel connected subcomponent, one needs to look at the inputs of the column and enumerate the set of legal sequences of control signal assignments for reads and writes on the column. One way to represent such a set is by using what we call *Parameterized Regular Expressions* (PREs). PREs extend regular expressions to include variables and outputs. Variables provide a way in which the description becomes succinct and outputs help define an automaton that produces output strings when input strings are provided. Before we present the PRE syntax details in Sect. 4, let us give a simple example.

In our example the array core is controlled by the signals $wl, precharge, wr\_en$ and din which are the set of word select lines, the precharge line, the write enable and the set of data input lines respectively. We define a predicate "$safe\_state$" to represent the condition in which none of the w word select lines (wl) are high.

$$safe\_state() = \bigwedge_{w} (\neg wl[i]), i = 0, 1, \ldots, w-1$$

The predicate $safe\_state$ is special since both read and write condition sequences start from and terminate in the $safe\_state$. Also $safe\_state$ represents the state of the circuit when nothing is happening as far as read or write is concerned.

In the following discussion, high and low mean logical one and zero respectively. Consider a write operation where data d is written into location i. Initially, the $safe\_state$ condition holds. After that, the $safe\_state$ condition should keep holding and at the same time the precharge should be high for at least 10 time steps. Following that the $safe\_state$ condition should keep holding even if the precharge goes low. Then exactly one word select line (line i) should go high and the write enable should be low. Then the same word select line should remain high in conjunction with the write enable being high, precharge being low and some data (d) being on the data lines. This is maintained for at least 6 time steps and then the write happens in the addressed bitcell. After that the condition is weakened to keep the same word select line high and if the write enable is high then the data lines are also maintained at d.

This family of conditions is expressed by the following PRE in which one can specify sequences of conditions by using semi-colons; provide annotations for the time for which a particular condition should hold; and specify when the writes to the bitcells actually occur. In the following PRE, the predicate $one\_hot(wl)$ represents the condition that exactly one word select line is high and all others are low.

$var\ i : 0..3;;$ (* address width *)
$var\ d : 0..31;;$ (* data width *)
$write(i, d) = safe\_state();$
$(safe\_state() \wedge precharge)^{10};$
$safe\_state()^{*};$
$(one\_hot(wl) \wedge wl[i] \wedge \neg wr\_en)^{*}\ ;$
$(one\_hot(wl) \wedge wl[i] \wedge wr\_en \wedge (din = d) \wedge \neg precharge)^{5};$

$$(one\_hot(wl) \wedge wl[i] \wedge wr\_en \wedge (din = d) \wedge \neg precharge)^*;$$
$$one\_hot(wl) \wedge wl[i] \wedge wr\_en \wedge (din = d) \wedge \neg precharge :: WR(i, d);$$
$$(one\_hot(wl) \wedge wl[i] \wedge (wr\_en \Rightarrow (din = d)))^*;;$$

The annotation *WR* signifies that the write (updating the bitcell contents) completes at this point of execution. Similar to the write operation, we can specify the read operation using condition sequences over input signals of the bitcell column. A bitcell column can thus be abstracted by a set of PREs representing all its transactions (reads and writes).

4 Parameterized Regular Expressions and STE

In this section, we will define a syntax for Parameterized Regular Expressions (PREs) and show how a finite automaton can be generated from a PRE definition. We will then prove that the finite automaton is suitable for use in a symbolic simulator employing STE.

4.1 PRE Definition and Finite Automaton

A PRE expresses a family of legal behaviors. Each behavior represents (a) a sequence of assignments to the inputs of the abstraction, $\hat{M}$, (b) a sequence of assignments to the outputs of $\hat{M}$ and (c) a set of writes to the memory in $\hat{M}$. All of these are associated with precise timing information detailing durations and relative times. The alternations in the PRE definition result in an angelic form of non-determinism. A sequence of conditions on the input signals that matches any of the alternative behaviors defined by a PRE, is accepted as legal. Thus by increasing the alternation in a PRE definition one can increase the number of legal behaviors. PREs are defined with respect to a set of boolean valued variables, A, a finite input alphabet Σ_i, and a finite output alphabet Σ_o which is partially ordered by the relation $\preceq_{\Sigma_o}$. Σ_o forms a complete lattice with respect to $\preceq_{\Sigma_o}$ with $\bot_{\Sigma_o}$ and $\top_{\Sigma_o}$ as the bottom and the top elements respectively.

Definition 1 PRE:

$$\begin{aligned} R \ ::= \ & P \ ; \ Q \\ | \ & P \ + \ Q \\ | \ & P^* \\ | \ & P^n \\ | \ & a :: o \end{aligned}$$

Here P and Q are PREs, and the operators $;$ $+$ and $*$ are sequence, choice, and Kleene Star respectively. P^n represents a short hand for the sequence $P; P; P; \ldots$ (n *times*). In the terminal "$a :: o$", a is a predicate over variable assignments in $\mathcal{B}^A$ and input symbols in Σ_i; o is a function mapping variable assignments in $\mathcal{B}^A$ to output symbols in Σ_o, where $\mathcal{B} = \{0, 1\}$. Given a PRE R, the Non-Deterministic Finite Automaton (NDFA) with ϵ is a tuple $N_\epsilon(R) = \langle s, t, S, T, W, A, \Sigma_i, \Sigma_o \rangle$ where A, Σ_i, Σ_o are the set of variables, the input alphabet and the

output alphabet of the PRE respectively, S is a finite set of states, $s \in S$ is the start state, $t \in S$ is the end state, T is a set of transitions, $T \subseteq S \times \mathcal{B}^A \times (\Sigma_i \cup \{\epsilon\}) \times S$ and W is an output function, $W : S \times \mathcal{B}^A \rightarrow \Sigma_o \cup \{\epsilon\}$. A transition in T is of the form $s_1 \xrightarrow{I,v} s_2$, where $s_1, s_2 \in S$ are the source and destination states respectively, v is an assignment of values from $\mathcal{B}$ to the variables, and $I \in \Sigma_i \cup \{\epsilon\}$ is either an input symbol, or the special symbol ϵ. A *run* of N_ϵ is a sequence $vs_0, vs_1, vs_2, \ldots$ where $v \in \mathcal{B}^A$ is a variable assignment, $\forall j \geq 0 : s_j \xrightarrow{I_j,v} s_{j+1} \in T$, where $\forall j \geq 0 : I_j \in \Sigma_i \cup \{\epsilon\}$, $\forall j \geq 0 : s_j \in S$ and $s_0 = s$, the starting state of N_ϵ. The corresponding output of N_ϵ is the sequence $\sigma_0\sigma_1\sigma_2 \ldots \in \Sigma_o^*$, such that $\forall j \geq 0 : \sigma_j = W(s_j, v)$. The role of v can be understood by considering a different run $v's_0, v's_1, v's_2, \ldots$, such that $v' \neq v$. Although the state sequence involved in this run is same as that of the former, the outputs obtained from the two runs are different as $v' \neq v$.

Hopcroft and Ullman give a description of regular expressions and the corresponding construction of automata from them [7]. The regular expressions described by Hopcroft result in automata that are language acceptors. PREs, in contrast, are transducers – automata that output a string when provided with an input string. Moreover, PREs have explicit variables which are absent in the classical automata theory. Nonetheless, a few simple modifications enable us to construct an NDFA with ϵ transitions (N_ϵ) in a fashion that resembles the classical automata theory. For a PRE R, $N_\epsilon(R)$ can be constructed inductively using the definition of PREs. For the PRE of the form "$a :: o$", we construct an NDFA containing only two states s and t and a set of transitions $T = \{s \xrightarrow{I,v} t | a(I, v)\}$. The resultant NDFA is $\langle s, t, \{s, t\}, T, W, A, \Sigma_i, \Sigma_o\rangle$, where A is a set of variable names, $W(s, v) = \perp_{\Sigma_o}$ and $W(t, v) = o(v)$.

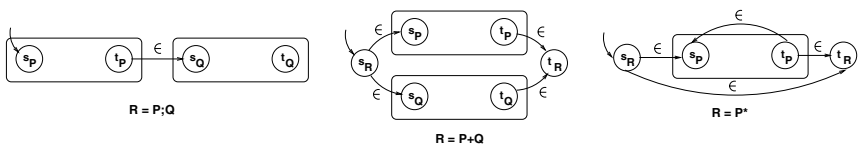

Fig. 4. Inductive Construction of N_ϵ

The method for inductively constructing NDFAs from PREs of the form $P; Q$, $P + Q$ and P^*, given the NDFAs of P and Q, can be derived by simple modifications of the techniques described in Hopcroft [7] as illustrated in Fig. 4. Let PREs P and Q generate ϵ-*NDFA*s $N_\epsilon(P) = \langle s_P, t_P, S_P, T_P, W_P, A, \Sigma_i, \Sigma_o\rangle$ and $N_\epsilon(Q) = \langle s_Q, t_Q, S_Q, T_Q, W_Q, A, \Sigma_i, \Sigma_o\rangle$ where the respective state sets are disjoint, i.e., $S_P \cap S_Q = \phi$. The general inductive construction procedure can be characterized by one that adds a few new transitions to *merge* the NDFAs and unites the state spaces and the rest of the transitions. In the following discussion, we assume that $s_R, t_R \notin S_P, S_Q$. If $R = P; Q$ then $N_\epsilon(R) = \langle s_P, t_Q, S_P \cup S_Q, T_P \cup T_Q \cup \{t_P \xrightarrow{\epsilon} s_Q\}, W_R, A, \Sigma_i, \Sigma_o\rangle$ where $W_R(x, v) = \epsilon$ if $x = s_Q$, $W_P(x, v)$ if $x \in S_P$, and $W_Q(x, v)$ otherwise. If $R = P + Q$ then $N_\epsilon(R) = \langle s_R, t_R, S_P \cup$

$S_Q \cup \{s_R, t_R\}, T_P \cup T_Q \cup \{s_R \xrightarrow{\epsilon} s_P, s_R \xrightarrow{\epsilon} s_Q, t_P \xrightarrow{\epsilon} t_R, t_Q \xrightarrow{\epsilon} t_R\}, W_R, A, \Sigma_i, \Sigma_o\rangle$ where $W_R(x, v) = \epsilon$ if $x \in \{s_P, s_Q, t_R\}$, $\perp_{\Sigma_o}$ if $x = s_R$, $W_P(x, v)$ if $x \in S_P$, and $W_Q(x, v)$ otherwise. If $R = P^*$, then $N_\epsilon(R) = \langle s_R, t_R, S_P \cup \{s_R, t_R\}, T_P \cup \{s_R \xrightarrow{\epsilon} s_P, s_R \xrightarrow{\epsilon} t_R, t_P \xrightarrow{\epsilon} s_P, t_P \xrightarrow{\epsilon} t_R\}, W_R, A, \Sigma_i, \Sigma_o\rangle$ where $W_R(x, v) = \perp_{\Sigma_o}$ if $x = s_R$, ϵ if $x \in \{t_R, s_P\}$, and $W_P(x, v)$ otherwise.

4.2 Obtaining a Deterministic Automaton from N_ϵ

Given a PRE, we aim at constructing an STE-usable Deterministic Finite Automaton (DFA)[1] . Given a regular expression described by Hopcroft [7], we can construct an NDFA with ϵ transitions (state space say S) and then convert it to an NDFA without ϵ transitions (state space S) and then determinize it using standard subset construction methods to obtain a DFA (state space 2^S) that acts as an acceptor of the language of the regular expression. However, since we are dealing with variables and outputs and our final automaton is going to be a transducer over infinite strings, the conversion from a PRE definition to an STE-usable DFA involves a series of automata constructions. Parallels can be drawn between our procedure and the standard procedure.

We begin by removing the so-called ϵ transitions to yield an automaton that consumes one input symbol and produces one output symbol during each state transition. At this "ϵ-removal" step we also introduce some additional non-determinism to enable non-deterministic *restarting* of runs in the automaton [2]. We then perform a determinization step, yielding a deterministic automaton, D. The standard determinization procedure would result in an exponential blowup in the size of the state space, which would be impractical. Instead, we use a sparse state space representation and construct a determinized conservative approximation to the original automaton N. Finally, we introduce the standard STE lattice structure over the input, state and output spaces to yield a determinized transition system suitable for use as a model for STE. The construction thus produced is a conservative approximation of the precise intended semantics of PREs, and thus in theory may give false negative verification results. Our admittedly limited experience to date suggests that this does not become a problem in practice.

Given a state q in N_ϵ, let ϵ-*closure*(q) denote the set of all states that are reachable from q via only ϵ transitions. Let ϵ-*closure*(Q), where Q is a set of states, be $\bigcup_{q \in Q} \epsilon$-*closure*$(q)$, that is, the union of the ϵ-closures of all the elements of Q.

Given a PRE R and $N_\epsilon(R) = \langle s, t, S, T, W, A, \Sigma_i, \Sigma_o\rangle$, define an NDFA without ϵ transitions, N with state space $S \times \mathcal{B}^A$. States of N consist of two components – a control component represented by S and a data component represented

[1] A DFA that has i) a state space that forms a complete lattice under a certain partial order and ii) a monotonic next state function.

[2] The non-determinism is added at this stage rather than being included in the original automaton N_ϵ purely as a technical convenience to facilitate the inductive definition of N_ϵ.

by assignments of boolean values to the variables in A. Define the starting state $s_N = \{s_0\} \times \mathcal{B}^A$ as the set of starting states of N, signifying that the variables can be non-deterministically assigned values when the execution of N starts. The transitions of N are derived from two sources. The first, T_{N_1}, results from the non-ϵ transitions of N in the classical way: $(s, v) \xrightarrow{I} (t, v) \in T_{N_1}$ if and only if $\exists r : r \in \epsilon\text{-}closure(s) \wedge r \xrightarrow{I,v} t \in T$. The second source, T_{N_2}, is introduced to allow the automaton to "restart" at any point, assigning new values to the variables, $(s, v_1) \xrightarrow{I} (t, v_2) \in T_{N_2}$ if and only if there is a starting state $(s_0, v_2) \in s_N$ such that $(s_0, v_2) \xrightarrow{I} (t, v_2) \in T_{N_1}$.

Definition 2 NDFA:
Given a PRE R and $N_\epsilon = \langle s_0, t, S, T, W, A, \Sigma_i, \Sigma_o \rangle$, define an NDFA N obtained from N_ϵ to be the tuple $N = \langle s_N, S_N, T_N, W_N, A, \Sigma_i, \Sigma_o \rangle$ where A remain unchanged from N_ϵ, $S_N = S \times \mathcal{B}^A$ is the set of states and $s_N = \{s_0\} \times \mathcal{B}^A$ is the set of starting states of N. Define the transition relation $T_N = T_{N_1} \cup T_{N_2}$, where T_{N_1} and T_{N_2} are as already defined. Define the output function $W_N : S_N \to \Sigma_o$ to be $W_N(\langle s, v \rangle) = \perp_{\Sigma_o}$ if $W(s, v) = \epsilon$, else, $W(s, v)$ for all $s \in S, v \in \mathcal{B}^A$.

The standard subset construction method for computing a deterministic automaton, say $\hat{D}$, from N, results in a state space $S_{\hat{D}} = 2^{S \times \mathcal{B}^A}$, which is doubly exponential in A. To arrive at a more efficient solution, we define a DFA D with a state space $S_D = 2^S \times \mathcal{Q}^A$, where $\mathcal{Q} = \{0, 1, X, \top\}$. Thus, a state in S_D consists of a control component from S and a data component consisting of an assignment from $\mathcal{Q}$ to every variable in A. S_D encodes the state information of $S_{\hat{D}}$ in a conservative but concise manner. The subset construction method represents a set of states (P) from N by a single state in $\hat{D}$. The state in D that maps to P is obtained by computing the greatest lower bound of the states in P in the partially ordered state space S_D (the partial order defined later) and hence is a conservative approximation of the entire set P. Every state in the state space S_D can be associated with a set of states in $S_{\hat{D}}$, but not vice versa; thus making S_D more sparsely populated than $S_{\hat{D}}$.

Define a partial order $\preceq_D$ over S_D as $\forall \langle s_1, \phi_1 \rangle, \langle s_2, \phi_2 \rangle \in S_D$, $\langle s_1, \phi_1 \rangle \preceq_D \langle s_2, \phi_2 \rangle$ if and only if $s_1 \subseteq s_2$ and $\phi_1 \preceq \phi_2$. For example, if $S = \{s, t\}$ and $A = \{v_0, v_1\}$, and states $s_0 = \langle \{s\}, 00 \rangle$, $s_1 = \langle \{s\}, 01 \rangle$ and $s_2 = \langle \{s\}, 0\top \rangle$ are members of S_D, then, $s_0 \preceq_D s_2$ and $s_1 \preceq_D s_2$. The partially ordered state space S_D represents a form of angelic non-determinism. The state s_2, that is *higher* in the partial order, represents a non-deterministic choice between related *lower* states s_0 and s_1. A specification that is satisfied by a trajectory σ containing either s_0 or s_1, is also satisfied by one that can be obtained from σ by replacing occurrence of s_0 or s_1 by s_2. As one goes higher up in the partial order the number of specifications that are validated by a given circuit model increases.

The automaton D has a starting state which assigns $\top$ to all the variables in A, representing an angelic non-deterministic choice between all possible values. An execution along a path in D *weakens* the value of each variable as it is assigned in order to satisfy a predicate along a particular arc. A state D represents a set of states in N. A transition in N takes place when its predicate is satisfied by

the values of the inputs as well those of the variables associated with the source and destination states. A transition in D assigns the greatest lower bound of the values that are necessary to take the *corresponding set* of transitions in N. Thus the assignment of variables in D is a conservative approximation. Also, the output generated from a state in D is the greatest lower bound of the outputs generated by the corresponding set of states in N, thus making it weaker than the weakest output produced by any state in the state set in N. In practice, being conservative in this construction does not affect the verification.

Definition 3 DFA:
Let R be a PRE with ϵ-NFA and NDFA respectively defined as $N_\epsilon = \langle s_0, t, S, T, W, A, \Sigma_i, \Sigma_o \rangle$ and $N = \langle s_N, S_N, T_N, W_N, A, \Sigma_i, \Sigma_o \rangle$ where the symbols are as described earlier. Define a DFA obtained from R as $D = \langle \langle \{s_0\}, \top^A \rangle, S_D, T_D, W_D, A, \Sigma_i, \Sigma_o \rangle$ where $\top^A$ is the assignment of $\top$ to all the variables in A and the state space $S_D = 2^S \times \mathcal{Q}^A$. The transition function $T_D : S_D \times \Sigma_i \to S_D$ is defined using a function ψ

$$\psi(\alpha, \phi, i) = \{(t, a) | \exists s \in \alpha : a \preceq \phi \wedge (s, a) \xrightarrow{i} (t, a) \in T_N\}$$

Define the state transition function, T_D, as

$$\forall \alpha, \phi : \langle \alpha, \phi \rangle \in S_D, \alpha \neq \{\} : T_D(\langle \alpha, \phi \rangle, i) = \langle \alpha', \phi' \rangle$$

where, $\alpha' = \{t | \exists a : (t, a) \in \psi(\alpha, \phi, i)\}$, is the set of projections onto the states obtained from ψ, and, $\phi' = glb\{a | \exists t : (t, a) \in \psi(\alpha, \phi, i)\}$, is the glb of the set of projections onto the assignments to variables obtained from ψ. Define the output function, $W_D : S_D \to \Sigma_o$ as $W_D(\langle \alpha, \phi \rangle) = glb\{W_N(\langle t, \phi \rangle) | t \in \alpha\}$, where W_N is extended from its previous definition by simple monotonic quarternary extension of its domain, making it $W_N : S \times \mathcal{Q}^A \to \Sigma_o$.

The state $\langle \{\}, \top^A \rangle$ can be reached only by accepting an input symbol that neither starts any fresh legal input string nor satisfies any of the outgoing transitions from any state of D. This in turn means that $\langle \{\}, \top^A \rangle$ is the state which acts as a recognizer of an illegal input symbol. Also the output generated from $\langle \{\}, \top^A \rangle$ is $\perp_{\Sigma_o}$. However, we can encounter certain strings that might have a legal prefix of length zero or more followed by substrings that do not start any legal transaction. These illegal substrings need to have an execution on the automaton. This is served by *adding* some *new edges* to D.

Whenever the starting state of D_A is reached, the automaton is starting afresh and the EMM is initialized to the empty list. This is accomplished by the $\perp_{\Sigma_o}$ output at the starting state. Also all the variables are reset to $\top$. D_A can reach the sink state $\langle \{\}, \top^A \rangle$ via transitions inherited from D, where the EMM is reset to the empty list by producing the output $\perp_{\Sigma_o}$. All the variables are also reset to $\top$. The new edges in D_A enables it to continue to remain in $\langle \{\}, \top^A \rangle$ and maintain the variables and the EMM at $\top$ and empty list respectively until the start of a fresh legal transaction is recognized. This mechanism enables D_A to recognize illegal input substrings and produce the required output strings.

Definition 4 Augmented DFA:
Let R be a PRE with ϵ-NFA, NDFA and DFA respectively defined as $N_\epsilon = \langle s_0, t, S, T, W, A, \Sigma_i, \Sigma_o \rangle$, $N = \langle s_N, S_N, T_N, W_N, A, \Sigma_i, \Sigma_o \rangle$, and $D = \langle \langle \{s_0\}, \top^A \rangle, S_D, T_D, W_D, A, \Sigma_i, \Sigma_o \rangle$ where the symbols are defined as earlier. Define the augmented DFA $D_A = \langle \langle \{s_0\}, \top^A \rangle, S_D, T_A, W_D, A, \Sigma_i, \Sigma_o \rangle$, where $T_A : S_D \times \Sigma_i \to S_D$ is defined as: $T_A(\langle \{\}, \top^A \rangle, i) = T_D(\langle \{s_0\}, \top^A \rangle, i)$ and for all other states $s \in S_D : T_A(s, i) = T_D(s, i)$.

4.3 Obtaining an Automaton for STE

Until now the input variables were assumed to be assigned boolean values. In order to function in a ternary simulation environment we have to extend the input alphabet to the ternary domain – one that is partially ordered. This results into a further extension of the state space. As the input *weakens*, one encounters values that are lower in the partial order of the input space. Weaker values represent an undetermined choice between stronger input values that lie above. A state transition that is *certain* for a given input might become *uncertain* as the input weakens. Hence the state space extends to $S_F = (\mathcal{T}^S \times \mathcal{Q}^A) \cup \{\top\}$, where $\mathcal{T} = \{0, 1, X\}$ and $\mathcal{Q} = \{0, 1, X, \top\}$, in order to incorporate this non-determinism. Weaker state assignments represent an undetermined choice between stronger state assignments. For instance, being "*uncertainly* present" in a state is weaker than either being "definitely present" or being "definitely not present" in the state. Also, being "uncertainly present" represents the non-deterministic choice between being "definitely present" and being "definitely not present" in the state. A specification is satisfied if and only if the consequent portion is satisfied by the weakest trajectory (sequence of states) satisfying the antecedent. As the trajectory weakens (via the weakening of the constituent states), every nondeterministic choice that is possible in the state space must satisfy the specification. So increasing the amount of nondeterministic choice reduces the number of specifications a given model will satisfy. Thus the state space S_F represents a form of *demonic* non-determinism, wherein, increasing non-determinism results into non-acceptance of input strings. Parallels can be drawn between the non-determinism exhibited by S_F and the state space of an STE circuit model, as discussed in Sect. 2. Later in this section, we will show that S_F is suitable for STE. An automaton having S_F as its state space, when viewed as a transducer, has a weaker state (that is lower in the partial order) producing outputs that are weaker than the stronger states lying above it.

We extend Σ_i to another finite alphabet Σ_i' such that Σ_i' is partially ordered by the relation $\preceq_{\Sigma_i'}$, $\Sigma_i \subseteq \Sigma_i'$ and Σ_i' is a complete lattice with $\top_{\Sigma_i'}$ and $\bot_{\Sigma_i'}$ as the top and bottom elements respectively. Also, two unequal symbols $a, b \in \Sigma_i$ are incomparable in $\preceq_{\Sigma_i'}$. Next, we extend the definition of D_A into one which is obtained by simple monotonic ternary extension of the inputs of D_A.

Definition 5 Final DFA:
Let R be a PRE with ϵ-NFA, NDFA and augmented DFA respectively defined as $N_\epsilon = \langle s_0, t, S, T, W, A, \Sigma_i, \Sigma_o \rangle$, $N = \langle s_N, S_N, T_N, W_N, A, \Sigma_i, \Sigma_o \rangle$, and

$D_A = \langle\langle\{s_0\}, \top^A\rangle, S_D, T_A, W_D, A, \Sigma_i, \Sigma_o\rangle$. *Define the final DFA* $D_F = \langle s_f, S_F, y, W_F, A, \Sigma_i', \Sigma_o\rangle$ *where* A, Σ_i *and* Σ_o *are as defined before. The state space is* $S_F = \mathcal{T}^S \times \mathcal{Q}^A$, *and the next state function is* $y : S_F \times \Sigma_i' \to S_F$, *which is defined using the functions* φ *and* ξ. *In the following definitions* $\alpha \in \mathcal{T}^S$, $\alpha' \in 2^S$ *and* $\phi \in \mathcal{Q}^A$. *Let* $\varphi(\alpha, \phi, i) =$

$$\{\langle\alpha_D, \phi_D\rangle | \exists\alpha' \in 2^S, \exists\phi' \in \mathcal{Q}^A, \exists i' \in \Sigma_i : \alpha \preceq \xi(\alpha') \wedge \phi \preceq \phi'$$

$$\wedge\ i \preceq_{\Sigma_i'} i' \ \wedge\ T_A(\langle\alpha', \phi'\rangle, i') = \langle\alpha_D, \phi_D\rangle\}$$

where if the states in S *be arbitrarily one-to-one mapped to the set* $\{0, 1, 2, \ldots, |S| - 1\}$ *by the function* λ *then,*

$$\xi(\alpha') = \alpha'_0\alpha'_1 \ldots \alpha'_{|S|-1} \ \text{ such that } \ \alpha'_{\lambda(i)} = 1 \ \text{ if } \ i \in \alpha', \ \ \alpha'_{\lambda(i)} = 0 \ \text{ otherwise.}$$

Define the starting state as $s_f = \langle\xi(\{s_0\}), \top^A\rangle$ *and the next-state function as* $y(\langle\alpha, \phi\rangle, i) = \langle\hat{\alpha}, \hat{\phi}\rangle$ *where*

$$\hat{\alpha} = glb\{\xi(\alpha_D) | \exists\phi_D : \langle\alpha_D, \phi_D\rangle \in \varphi(\alpha, \phi, i)\}$$

$$\hat{\phi} = glb\{\phi_D | \exists\alpha_D : \langle\alpha_D, \phi_D\rangle \in \varphi(\alpha, \phi, i)\}$$

Let $\xi(\alpha) = \alpha_0\alpha_1 \ldots \alpha_{|S|-1}$. *Define a function* ρ *as* $\rho(\alpha) = 1$ *if and only if* $\exists k \in \{0, 1, \ldots, |S| - 1\}$ *such that* $\alpha_k = 1$; *otherwise* 0. *Finally, the output function,* $W_F : S_F \to \Sigma_o$, *is defined as*

$$W_F(\langle\alpha, \phi\rangle) = \ \text{if } \rho(\alpha) = 1 \text{ then } glb\ \{W_N(\langle k, \phi\rangle) | \alpha_{\lambda(k)} \preceq 1\}, \text{ otherwise, } \perp_{\Sigma_o}$$

where α_i *is the* i*-th bit of* $\xi(\alpha)$.

Intuitively, the automaton D_F does not output the bottom element, $\perp_{\Sigma_o}$, if and only if D_F is *definitely* in at least one of the states of N. In such a case, D_F outputs the *glb* of the outputs of all the states in which it is *definitely* or *possibly* present in. D_F has a partially ordered state space. States that are weaker lie *below* the ones that are stronger. In situations where D_F reaches a weak state where it is not *certainly* present in at least one of the states of N then it outputs $\perp_{\Sigma_o}$. This situation results from inputs that are not strong enough to push D_F to be present in stronger states. The output $\perp_{\Sigma_o}$ is also produced in cases where an illegal input sequence has been detected by D_F. In such cases because of the output $\perp_{\Sigma_o}$, the EMM is re-initialized to an empty list, where it is maintained until the start of a fresh legal string is recognized. Unlike D_F, the output of N is never supposed to be $\perp_{\Sigma_o}$ in any situation other than when N is present in its start state. The fundamental reason why the outputs generated by D_F is weakened under the certain situations arises because of a fundamental difference between D_F and N. R and hence N only refer to *exact, legal and finite* input strings which can *definitely* take N to some states in N whereas D_F has the capability of recognizing *weak, or illegal, or infinite* input strings as well.

4.4 Using the Final DFA in an STE Engine

In order to establish that the final DFA is suitable for symbolic simulation purposes in an STE environment, we need to demonstrate that its state space forms a complete lattice under a partial order relation and that its next-state function is monotonic under that relation. We define a partial order relation on the state space S_F.

Definition 6 *Define $\sqsubseteq$, a partial order over $S_F \cup \{\top\}$. Let $\langle a, \phi_a\rangle, \langle b, \phi_b\rangle \in S_F$. Define $\langle a, \phi_a\rangle \sqsubseteq \langle b, \phi_b\rangle$ if and only if $a \preceq b \wedge \phi_a \preceq \phi_b$ and $\forall \langle a, \phi_a\rangle \in S_F : \langle a, \phi_a\rangle \sqsubseteq \top$.*

Lemma 1 *$\langle S_F \cup \{\top\}, \sqsubseteq\rangle$ forms a complete lattice.*

Proof Outline. The proof follows from a) both $\langle \mathcal{T}^S, \preceq\rangle$ and $\langle \mathcal{Q}^A, \preceq\rangle$ are partial orders, b) $\sqsubseteq$ is defined using $\preceq$ and c) $\forall \langle a, \phi_a\rangle \in S_F : \langle a, \phi_a\rangle \sqsubseteq \top$. □

Lemma 2 *The function y is monotonic under the $\sqsubseteq$ partial order.*

Proof Outline. Let us take two elements of S_F, say, $\langle \alpha, \phi_\alpha\rangle$ and $\langle \beta, \phi_\beta\rangle$ such that $\langle \alpha, \phi_\alpha\rangle \sqsubseteq \langle \beta, \phi_\beta\rangle$, then we are required to prove that $\langle \hat{\alpha}, \hat{\phi_\alpha}\rangle \sqsubseteq \langle \hat{\beta}, \hat{\phi_\beta}\rangle$ where, for an input $i \in \Sigma_i'$, $y(\langle \alpha, \phi_\alpha\rangle, i) = \langle \hat{\alpha}, \hat{\phi_\alpha}\rangle$ and $y(\langle \beta, \phi_\beta\rangle, i) = \langle \hat{\beta}, \hat{\phi_\beta}\rangle$. The proof can be divided into two parts: showing $\hat{\alpha} \preceq \hat{\beta}$ and $\hat{\phi_\alpha} \preceq \hat{\phi_\beta}$.

Using $\langle \alpha, \phi_\alpha\rangle \sqsubseteq \langle \beta, \phi_\beta\rangle$, one can show $\varphi(\beta, \phi_\beta, i) \subseteq \varphi(\alpha, \phi_\alpha, i)$. That, in turn can be used to show that $\hat{\phi_\alpha} \preceq \hat{\phi_\beta}$ and $\hat{\alpha} \preceq \hat{\beta}$ and hence the proof. □

4.5 Outputs from PREs

Until this point, we have been able to define PREs, give an operational account of their semantics based on a sequence of automata constructions without being specific about the precise nature of the output alphabet. In practice, however, PRE outputs are used to provide inputs to the EMMs, and to represent the outputs of the circuit elements that they are intended to abstract. The same generic format *write*(*enable*, *addr*, *data*) can be used to represent both cases. In this format *enable* is a ternary value, while *addr* and *data* are both vectors of ternaries. The entire PRE output is a vector of such values. The tuples have the obvious meaning when writing to an EMM – the *enable* bit determines whether to write to the EMM or not, while the vectors *addr* and *data* supply the address and data for the write respectively. In case of primary outputs of the block being abstracted, the tuple is a degenerate case in which the address is of length 0, that is, an output is represented as an EMM with only a single address. If the enable bit is on, new data is output, if it is off, the most recent output is repeated. In both the general case of the EMM output and the special case of a primary output the output symbols are partially ordered by bitwise extension of the standard ternary partial order.

As defined in Sect. 4.1, the terminal in a PRE definition is of the form "$a :: o$", where a is a predicate on the input symbols and assignments to variables of the abstraction $\hat{M}$ and o is the output defined as functions mapping from assignments to variables to output symbols of $\hat{M}$.

$$\begin{aligned} o \ ::= \ & /*\, empty \,*/ \\ & \mid WR(i,d) \\ & \mid output[output\_pin\_name := val, \ldots] \end{aligned}$$

The annotations WR and *output* enable the user to specify write operations to the EMM in $\hat{M}$ and the signal values to be assigned at the output pins of $\hat{M}$. The write annotation "$WR(i,d)$" stands for a memory write that happens with variable i as the address and variable d as the data and produces a "write tuple" $write(1,i,d)$ where the 1 stands for the enabled bit being high signifying a *definite* write. The *output_pin_name* and *val* are the name and the specified value of an output pin respectively. *val* can be any of $\{0,1,X,RD(j),\neg RD(j)\}$ where $RD(j)$ represents a read of the value of the array entry symbolically indexed by a variable j. When o produces the *empty* terminal, the predicate a can be viewed as augmented with the write tuple $write(0,\top,\top)$ to represent *no change* to the memory contents.

The EMM is modeled using an associative list mapping symbolic data to symbolic addresses. The state of the EMM is a finite lattice formed by ternary partial order $\preceq$. The *write* command, $write(enable, address, data)$, is monotonic over the EMM state space with respect to the ternary partial order. The "*no change*" command, which is the identity function for the state space of the EMM, is $write(0,\top,\top)$. The output $\perp_{\Sigma_o}$ from the PRE meant for an EMM translates to the write tuple $write(X,X,X)$, which resets the EMM to an empty list.

The outputs can be treated as an EMM with one bit of memory. When an output becomes $0,1,X$, or some value a (result of an array read) the corresponding EMM commands would be $write(1,,0)$, $write(1,,1)$, $write(1,,X)$ and $write(1,,a)$ respectively. The address in this case does not matter and hence is omitted.

If in a particular terminal in a PRE nothing is specified about a memory write or value assignments to an output, we assume that the corresponding write tuple generated for the corresponding EMM is the identity $write(0,\top,\top)$, preserving its contents.

5 Experimental Results

We prototyped a tool building on that reported by Krishnamurthy et al [3] and augmented it with the technique presented in this paper. We conducted our experiments on a 360MHz Sun UltraSparc-II with 512MB of memory. Our example circuit was a segment array from the MPC7450 PowerPC microprocessor. The array had 512 bitcells, a read port and a write port. At first the PRE definitions of the read and the write operations were written. Then the RTL description was verified against the switch level model augmented with the abstract PRE

descriptions in about 4 minutes time. The method reported by Krishnamurthy et al [3] takes about a minute to do the same correspondence check without the PREs. The time difference is due to some BDD reordering taking place and we speculate that this is because of the way we handle variables in the implementation. This issue is currently being addressed.

The real effectiveness of the methodology was demonstrated by a fault injection experiment. We injected a fault by tying the precharge to ground, so that the bitcells are never precharged before a read or a write operation takes place. The earlier method [3] failed to discover the bug whereas our prototype was able to discover it and generate a witness execution sequence exposing it. The bug that we found is an instance of an "uncertified" input pattern that is not allowed as an input to the memory but is permitted by the switch level model. Thus, the bug is a member of a class of circuit level bugs that are abstracted away by the switch level model but are exposed by the current methodology. This experiment points out the importance of our methodology as it can serve as a "stricter check-point" when the custom memories have passed the earlier verification flow [3]. Although we believe that more time is consumed by our prototype because of implementation details, it hardly seems fair to compare a method that checks a schematic against an RTL with another that checks a composition of a schematic and an abstraction of the rest of the schematic against an RTL, since they are doing two fundamentally different things.

6 Related Work, Conclusions, and Future Directions

Checking the correspondence between the switch and the gate level views of the memory has been addressed by many researchers [5], [10], [11], [12], [4], [3]. While others focussed on using STE to verify equivalence by proving functional properties on both the RTL and the schematic view of the circuit, Krishnamurthy et al reported work on generating the assertions automatically from the RTL and to cross-check them against the schematic [3]. This removes the onus off the user to come up with a so-called "complete" set of assertions. Our work advances this approach by enabling the schematic view to be "weaker" in order for us to discover more buggy behavior.

We have provided a way in which memory can be modeled using regular expressions which represent a family of conditions that are visible to the portion of the memory that is being abstracted out. We have shown that the state transition model obtained from such a specification can be used in an STE framework. We have conducted experiments on an industrial strength circuit and demonstrated the applicability of our approach.

Future work will address the issue of checking whether the state machine model produced is a conservative approximation of the actual bitcell behavior. This could be done by verifying the specifications against a switch level bit-cell model, or even better, by verifying against a suite of Spice simulations. One way to get behavioral specifications is to obtain them automatically from the Spice

simulations performed on these switch level designs. Work in this area is another candidate for future work.

References

1. M. N. Velev, R. E. Bryant, A. Jain. Efficient "Modeling of Memory Arrays in Symbolic Simulation". *CAV, 1997*, Proceedings. LNCS, Vol. 1254, Springer, 1997, pp. 388-399.
2. M. N. Velev, R. E. Bryant. "Efficient Modeling of Memory Arrays in Symbolic Ternary Simulation". *TACAS, 1998*.
3. N. Krishnamurthy, A. K. Martin, M. S. Abadir, J. A. Abraham. "Validating PowerPC Microprocessor Custom Memories" *IEEE Design and Test of Computers*, Vol. 17, No. 4, Oct-Dec 2000, pp. 61-76.
4. L.-C. Wang, M. S. Abadir, N. Krishnamurthy. "Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Evaluation". *35th ACM/IEEE DAC*, June, 1998.
5. R. E. Bryant. "Algorithmic Aspects of Symbolic Switch Network Analysis". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(4), July 1987.
6. C.-J. H. Seger and R. E. Bryant. "Formal verification by symbolic evaluation of partially-ordered trajectories". *Formal Methods in System Design*, 6(2):147-189, March, 1995.
7. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979, pp. 1-45.
8. R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, 35(8), August 1986.
9. R. E. Bryant. "Verifying a Static RAM Design by Logic Simulation", *Fifth MIT Conference on Advanced Research in VLSI*, 1988, pp. 335-349.
10. N. Ganguly, M. S. Abadir, M. Pandey. "PowerPC array verification methodology using formal techniques". *International Test Conference 1996*, pp.857-864.
11. M. Pandey, R. Raimi, D. L. Beatty, R. E. Bryant. "Formal Verification of PowerPC arrays using Symbolic Trajectory Evaluation". *33rd ACM/IEEE DAC*, June 1996, pp.649-654.
12. M. Pandey, R. Raimi, R. E. Bryant, M. S. Abadir. "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation". *34th ACM/IEEE DAC*, June 1997.