Automatic Abstraction in Symbolic Trajectory Evaluation

Sara Adams, Magnus Björk, Tom Melham
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
Email: {sara,magnus,melham}@comlab.ox.ac.uk

Carl-Johan Seger
Strategic CAD Labs, Intel Corporation
Hillsboro, OR 97124, USA
Email: carl.seger@intel.com

***Abstract*—Symbolic trajectory evaluation (STE) is a model checking technology based on symbolic simulation over a lattice of abstract state sets. The STE algorithm operates over families of these abstractions encoded by Boolean formulas, enabling verification with many different abstraction cases in a single model-checking run. This provides a flexible way to achieve partitioned data abstraction. It is usually called 'symbolic indexing' and is widely used in memory verification, but has seen relatively limited adoption elsewhere, primarily because users typically have to create the right indexed family of abstractions manually. This work provides the first known algorithm that automatically computes these partitioned abstractions given a reference-model specification. Our experimental results show that this approach not only simplifies memory verification, but also enables handling completely different designs fully automatically.**

I. Introduction

Symbolic Trajectory Evaluation (STE) is a model checking technology based on symbolic simulation over a lattice of abstract state sets [1]. STE provides a combination of abstraction and algorithmic efficiency for verification of memories and datapath-dominated designs, and has tackled numerous difficult industrial verification problems [2]–[4].

In the abstraction lattice at the heart of STE, each circuit node is assigned a value in the set $\{0, 1, \mathrm{X}\}$, with 'X' representing an unknown or 'don't care' value. An assignment of such values to every circuit node is an abstraction of a set of Boolean circuit states. It is abstract in the sense that it ambiguously stands for any one of a family of Boolean state sets, one for each replacement of every X by 0 or 1. The collection of all such abstractions forms a lattice, ordered by the amount of information about node values.

The STE model-checking algorithm uses three-valued circuit simulation [5] to compute a reachable abstract state-set in this representation, comparing this to a specification written in a weak linear-time temporal logic. The algorithm is space-efficient because it operates over abstractions of sets of states; any parts of the circuit function not relevant to the specification get 'abstracted away' to X. Any correctness result verified in this abstract model transfers over to the real, Boolean model of circuit states. Formally, there is a Galois connection between the three-valued model and the Boolean model of states [6].

On top of the abstraction lattice, STE provides a layer of symbolic representation whereby whole families of abstractions may be checked simultaneously in one run of the model-checker. The abstractions are indexed symbolically by Boolean variables, and formulas of Boolean logic are computed to represent the resulting families of reachable abstract state sets. This mechanism, sometimes called 'symbolic indexing', provides a flexible way to achieve partitioned data abstraction. A typical example is a memory verification, in which an n-element memory is verified with an indexed family of n abstractions, one for each address at which some target data might be located.

The abstractions in an indexed family can overlap in flexible, though not quite arbitrary, ways. This representation can also record interdependencies among node values, and so greatly increases the expressive power of specifications for STE. In implementations, the formulas in the symbolic layer may be manipulated using BDDs or other (usually semi-canonical) representations, and decided using SAT or BDDs.

This abstraction machinery is controlled by the way in which a user writes properties for model checking. By careful coding of the property, the user can guide the symbolic simulation done during model-checking through the right layers of the abstract state lattice to verify the property with contained complexity. A good illustration of success is the content-addressable memory verification done by Pandey and colleagues [2], in which a careful encoding of properties gives a logarithmic reduction in complexity.

Controlling abstraction manually in this way can be difficult, especially if there are assumptions about the operating environment of the verification. The property encoding required for abstraction is often non-obvious and tricky to devise. This paper describes an automatic approach to STE abstraction. We present an algorithm that computes an indexed family of abstractions from the specification to be verified. The abstraction scheme is then encoded in the guards of the verification property using the indexing transformation algorithm of Melham and Jones [7]. The result is an automatic abstraction method for STE, which requires little or no user guidance. We illustrate the effectiveness of our method with verifications of a memory, a CAM, and a simple scheduler.

II. Abstraction in STE

Verification properties in STE are called *trajectory assertions* and have the form $A \Rightarrow C$, where A and C are formulas of a simple linear-time temporal logic. The intuition is that the

antecedent formula A describes some initial conditions of the circuit inputs and states, and the *consequent* C specifies the values expected on circuit nodes as a response. The atomic propositions in A and C take the form '$P \rightarrow n$ is 0' or '$P \rightarrow n$ is 1', where n is the name of a circuit node and the *guard* P is a formula of propositional logic. The guard determines when the proposition is asserted: if P is true, then the node n must have the value 0 (or 1 respectively); if P is false, then there is no such assertion and n can have any value—including, for abstraction efficiency, the don't care value X. Antecedents and consequents are essentially just conjunctions of these atomic propositions, possibly modified by the next-time temporal operator N.

The guards in a trajectory assertion are all formulas of propositional logic over some set of variables; different guards can share variables, but not all the variables need appear in every guard. For each assignment of truth-values to these variables, the trajectory assertion collapses into a property verifiable by three-valued simulation, with Xs on all circuit nodes not forced to 0 or 1 by the antecedent or the circuit itself. Sets of reachable states are approximated by abstractions that assign a value in $\{0, 1, \mathrm{X}\}$ to each circuit node. Given a trajectory assertion, STE simultaneously computes the family of all such three-valued simulations, one for each satisfying assignment to the variables in the guards.

Users control this partitioned abstraction mechanism by appropriate selection of guards. The idea can be illustrated by the following trivial example. Consider a unit-delay AND-gate with three input nodes a, b, and c and output node o. A verification that does not exploit the abstraction lattice is achieved by running STE on this trajectory assertion:

$$\begin{array}{l} \overline{t_1} \rightarrow \mathrm{a} \text{ is } 0 \text{ and } t_1 \rightarrow \mathrm{a} \text{ is } 1 \text{ and} \\ \overline{t_2} \rightarrow \mathrm{b} \text{ is } 0 \text{ and } t_2 \rightarrow \mathrm{b} \text{ is } 1 \text{ and} \\ \overline{t_3} \rightarrow \mathrm{c} \text{ is } 0 \text{ and } t_3 \rightarrow \mathrm{c} \text{ is } 1 \\ \quad \Rightarrow \\ \mathsf{N}(\overline{t_1} \vee \overline{t_2} \vee \overline{t_3} \rightarrow \mathrm{o} \text{ is } 0 \text{ and } t_1 \wedge t_2 \wedge t_3 \rightarrow \mathrm{o} \text{ is } 1) \end{array} \quad (1)$$

This is just symbolic Boolean simulation. The antecedent attaches a distinct, unconstrained, propositional variable to each input node. And the consequent asserts that the expected Boolean function of these variables appears on the output. Note that all Boolean variables in STE appear in the guards; the constants a, b, and c are node names, not variables.

STE's abstraction lattice lets us reduce the number of variables needed to verify this gate. The key observation is that if any one input is 0, then the output will be 0 regardless of the other inputs. We can exploit this with Xs to introduce abstraction in the model-checking run. For the AND gate, there are four cases to check; we can enumerate or 'index' these with two variables, say x_1 and x_2. We write the following:

$$\begin{array}{l} \overline{x_1} \wedge \overline{x_2} \rightarrow \mathrm{a} \text{ is } 0 \text{ and} \\ x_1 \wedge \overline{x_2} \rightarrow \mathrm{b} \text{ is } 0 \text{ and} \\ \overline{x_1} \wedge x_2 \rightarrow \mathrm{c} \text{ is } 0 \text{ and} \\ x_1 \wedge x_2 \rightarrow \mathrm{a} \text{ is } 1 \text{ and } \mathrm{b} \text{ is } 1 \text{ and } \mathrm{c} \text{ is } 1 \\ \quad \Rightarrow \\ \mathsf{N}(\overline{x_1} \vee \overline{x_2} \rightarrow \mathrm{o} \text{ is } 0 \text{ and } x_1 \wedge x_2 \rightarrow \mathrm{o} \text{ is } 1) \end{array} \quad (2)$$

Model-checking this with STE will simultaneously check four cases, each with different but sometimes overlapping abstractions of the reachable states arising. Any property verified in STE with a node set to X also holds when the node is either 0 or 1, so this assertion covers all input cases and is complete.

The advantage of this kind of abstraction is that it makes the representation of sets of states more compact, so that BDD or SAT computations in the model-checking are more tractable. The reduction in the number of propositional variables can be substantial in real applications.

A. Indexing Transformations

Melham and Jones [7] describe an algorithm by which trajectory assertions can be transformed to introduce more abstraction. Suppose $A \Rightarrow C$ is an assertion. The algorithm replaces the guards in A and C with new propositional formulas over a set of fresh variables in such a way that if the transformed assertion holds then so does $A \Rightarrow C$.

The algorithm takes as input a relation that specifies the abstraction scheme to be applied. For the three-input AND-gate, one possible abstraction relation is

$$\begin{array}{l} ((\overline{x_1} \wedge \overline{x_2}) \rightarrow \overline{t_1}) \wedge ((x_1 \wedge \overline{x_2}) \rightarrow \overline{t_2}) \wedge \\ ((\overline{x_1} \wedge x_2) \rightarrow \overline{t_3}) \wedge ((x_1 \wedge x_2) \rightarrow (t_1 \wedge t_2 \wedge t_3)) \end{array} \quad (3)$$

The variables x_1 and x_2 index the abstraction cases, and the variables t_1, t_2, and t_3 appear in the directly-formulated Boolean trajectory assertion (1). Using this relation, the algorithm will compute the encoded trajectory assertion (2).

The Melham-Jones algorithm works by taking certain preimages of an assertion's guards under the supplied abstraction relation. The relation takes the form $R[\mathcal{X}, \mathcal{T}]$, where $\mathcal{T}$ is a set of *target variables* that occur in the guards of $A \Rightarrow C$ (they need not be all the variables) and $\mathcal{X}$ is a set of fresh *indexing variables*. For a given guard P, we define the *weak preimage* P_R and *strong preimage* P^R by:

$$P_R[\mathcal{X}] = \exists \mathcal{T}.\, R[\mathcal{X}, \mathcal{T}] \wedge P[\mathcal{T}]$$

$$P^R[\mathcal{X}] = P_R[\mathcal{X}] \wedge \neg \exists \mathcal{T}.\, R[\mathcal{X}, \mathcal{T}] \wedge \neg P[\mathcal{T}]$$

where $\exists \mathcal{T}$ denotes existential quantification over all the variables in the set $\mathcal{T}$. Intuitively, $P_R[\mathcal{X}]$ is true for all indices $\mathcal{X}$ that *allow* P to hold, and $P^R[\mathcal{X}]$ is true for all indices $\mathcal{X}$ that *force* P to hold. Given an assertion $A \Rightarrow C$, the algorithm applies the strong preimage to the guards in A and the weak preimage to the guards in C. This weakens the verification assumptions A by introducing Xs, while maintaining the strength of the verification requirements C.

It is a technical side-condition required for soundness of the abstraction that the supplied relation satisfies the coverage condition $\forall \mathcal{T}.\, \exists \mathcal{X}.\, R[\mathcal{X}, \mathcal{T}]$. This ensures all the target variable values are indexed. Coverage is a bit more tricky when there are environmental constraints on the verification [7].

The indexing algorithm can be used on specifications that specify timing delays, such as the AND-gate above, as well as purely combinational ones. There is no explicit representation of time in the indexing relation itself—sequential STE specifications normally use distinct variables in the guards of

input nodes for the different time points of interest, so these are just different target variables in the indexing relation.

B. The Automatic Abstraction Method

The core of the method in this paper is an algorithm that automatically computes abstractions for use with the indexing transformation just described. The output of the algorithm is an abstraction relation of the kind illustrated by (3), the AND-gate relation of the previous section, but vastly more complex and unintuitive for realistic examples. The practical benefit is that a user does not have to invent the abstraction and manually encode it into the trajectory assertion to be verified.

The algorithm takes as input a *specification* for the circuit to be verified, in the form of a Boolean expression that states the required I/O function. This specification is typically a component of the STE assertion to be verified, so in principle our method need not require anything beyond the manually-written properties that any verification needs. (But see the discussion of 'symbolic constants' below.) We work from specifications rather than circuits because they give a clean reference model of the algorithm the circuit uses, unencumbered by implementation detail [8]. Although the I/O specification provided as input to our algorithm is purely combinational, the resulting abstraction relation is applicable to sequential circuits, for the reasons explained in the previous section.

The algorithm is presented in Sections III and IV, where we also prove that the relations generated meet the required coverage condition by construction. In Section V we then describe several optimizations to the preimage calculations done to transform trajectory assertions. Some of these exploit the special form that our abstractions relations have because of how they are generated. We prove the correctness of the most complex of these optimizations.

The experimental results in Section VI show that our method handles both embedded memories—the classic target for manual symbolic indexing—and the much less intuitive example of a scheduler. The paper concludes with some discussion of related work and our plans for future extensions.

III. Computing Abstraction Relations

The approach we take in computing an abstraction relation is to use the structure of a specification function. Specifically, we assume that we have been given a Boolean expression constructed using only two-input AND operators, NOT operators, and named variables. We call this datatype a *bexpr* for short. For the time being, we suppose the expression has a single output and is tree-structured.

We introduce the algorithm by describing a simplified version, so we can convey the basic idea without too much detail. Consider the algorithm in Fig. 1. The function *simple_bp* takes a bexpr for the specification and two BDDs, h and l, that represent the conditions under which the output should be high or low, respectively. It computes a BDD representing an abstraction relation by propagating conditions backwards through the circuit. The index variables of this relation are

```
1. simple_bp(e,h,l) =
2.  if is_VAR(e) then
3.    t := bexpr2bdd(e)
4.    return((h → t) ∧ (l → ¬t));
5.  elseif is_NOT(e) then
6.    return bp(strip_NOT(e),l,h);
7.  else // AND
8.    x := fresh_index_var();
9.    (e1,e2) := destruct_AND(e)
10.   r1 := simple_bp(e1,h,l∧x);
11.   r2 := simple_bp(e2,h,l∧¬x);
12.   return(r1 ∧ r2);
```

Fig. 1. Simple back-propagation algorithm.

generated in line 8 and occur in h and l. The target variables originate from the bexpr and are introduced in line 3.

The algorithm is recursive over the structure of e, the specification bexpr. In the base case, when the e is a variable, the abstraction relation simply says that the target variable is true whenever h holds and false whenever l holds. An invariant of the algorithm is that h and l never both hold. For NOT operations, we simply reverse h and l and continue. Finally, for a two-input AND operator, more work is needed. Both inputs must be high whenever h holds, so we simply pass h to the recursive calls in lines 10 and 11. But there are two ways l can be forced: either the first input or the second input has to be low. This choice is captured by creating a fresh index variable x. It is used to select which of the inputs are going to be the decisive low signal. Note that we never require both inputs to be low in order to achieve a maximum abstraction.

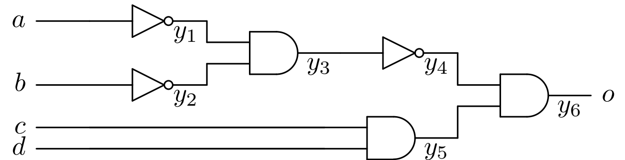

Fig. 2. Example circuit to illustrate our algorithm.

To illustrate the algorithm, consider the circuit in Fig. 2. First assume we want to compute the abstraction relation for the case the output is high. In other words, we want to compute a family of abstract inputs that all yield an output of high and that cover all such inputs. The resulting abstraction relation is

$$(x_1 \rightarrow a) \wedge (\overline{x_1} \rightarrow b) \wedge c \wedge d.$$

Intuitively, the index variable x_1 decides whether a or b has to be high, and both c and d must be high. Similarly, if we call *simple_bp* with h=F and l=T we get the abstraction relation

$$(x_1 \rightarrow \overline{a}) \wedge (x_1 \rightarrow \overline{b}) \wedge (x_2\overline{x_1} \rightarrow \overline{c}) \wedge (\overline{x_2}\,\overline{x_1} \rightarrow \overline{d}).$$

We can combine the above two cases by calling *simple_bp* with h=x_0 and l=$\overline{x_0}$, where x_0 is a fresh Boolean indexing

variable. In this combined case, we get

$$(x_3 x_0 \to a) \wedge (x_1 \overline{x_0} \to \overline{a}) \wedge (\overline{x_3}\, x_0 \to b) \wedge (x_1 \overline{x_0} \to \overline{b}) \wedge$$
$$(x_0 \to c) \wedge (x_2 \overline{x_1}\, \overline{x_0} \to \overline{c}) \wedge (x_0 \to d) \wedge (\overline{x_2}\, \overline{x_1}\, \overline{x_0} \to \overline{d}).$$

There are several shortcomings of this simple algorithm that make it inefficient. First, it builds a monolithic Boolean expression for the abstraction relation. But a partitioned relation is often more feasible and easier to use. Second, the algorithm is very generous in using fresh Boolean variables. Third, by recognizing only two-input AND gates, the algorithm will essentially use a unary encoding, rather than a binary encoding. And by recognizing only conjunctions, the algorithm uses more Boolean variables than needed—for example, simply recognizing XNOR gates can reduce the number of variables by a factor of two. Finally, the algorithm is sometimes too aggressive in computing the abstraction relation. It is quite common that the user can select some signals not to be part of the abstraction. Thus, the algorithm needs to be controlled when such *symbolic constants* are present.

IV. Improved Algorithm

In Fig. 3, we provide a much improved algorithm that tackles all of the shortcomings of the simple algorithm.

1. $bp(\mathcal{C}, e, h, l, name) =$
2. **if** $freevars(e) \subseteq \mathcal{C}$ **OR** is_VAR(e) **then**
3. **return** $\{(\text{bexpr2bdd}(e), h, l)\}$;
4. **elseif** is_XNOR(e) **then**
5. (i_1, i_2) := sort_inp_args($\mathcal{C}$,e);
6. **if** $freevars(i_1) \subseteq \mathcal{C}$ **then**
7. c := bexpr2bdd(i_1);
8. **return** $bp(\mathcal{C}, i_2, h\,c \vee l\,\overline{c}, h\,\overline{c} \vee l\,c, name)$;
9. **else**
10. $\{x_1,x_2\}$:= get_case_exprs(name,2);
11. $\{n_1,n_2\}$:= make_unique_names(name,2);
12. **return**($bp(\mathcal{C}, i_1, h\,x_1 \vee l\,x_1, h\,x_2 \vee l\,x_2, n_1) \cup$
13. $bp(\mathcal{C}, i_2, h\,x_1 \vee l\,x_2, h\,x_2 \vee l\,x_1, n_2)$);
14. **elseif** is_NOT(e) **then**
15. **return** $bp(\mathcal{C}, \text{strip_NOT}(e), l, h, name)$;
16. **else** // AND
17. (*cis*,*oinps*) := find_big_ands(e);
18. c:= $\bigwedge_{c_i \in cis}$ bexpr2bdd(c_i);
19. res := **if** $cis = \emptyset$ **then** $\emptyset$ **else** $\{(c,h,\text{F})\}$;
20. cases := get_case_exprs(name,$|oinps|$);
21. **if** $h \equiv F$ **then**
22. names := mk_same_names(*name*, $|oinps|$);
23. **else**
24. names := mk_unique_names(*name*, $|oinps|$);
25. **foreach** $b \in oinps,\ s \in cases,\ n \in names$
26. res := $res \cup bp(\mathcal{C}, b, h, l \wedge s \wedge c, n)$;
27. **return** *res*;

Fig. 3. Main back-propagation algorithm.

The algorithm is recursive and takes a set of symbolic constants $\mathcal{C}$, a bexpr *e*, the high and low conditions *h* and *l*, and a base name *name* used to create unique variables. It returns a list of triples, where each triple consists of a Boolean function f over the symbolic constants $\mathcal{C}$ and the generated indexing variables $\mathcal{X}$, and two Boolean functions denoting the cases in which f should be high or low, respectively. In Section V we will show how this format leads to a very efficient algorithm for computing the preimage operations needed for STE.

In more detail, on line 2-3, we deal with the two cases that the expression e only depends on variables in $\mathcal{C}$ or e is a target variable. For either case, we create a singleton triple representing the partial relation.

On line 4, we test whether the last bexpr represents an XNOR gate or not. If yes, on line 5 we sort the two inputs to the XNOR gate by their support so that if one input only depends on variables in $\mathcal{C}$, it will be the first input i_1. If at least one of the inputs depends only on variables in $\mathcal{C}$, we do not need to introduce any new indexing variables, but can simply call *bp* recursively suitably modifying the *h* and *l* functions given the value of the symbolic expression. If both inputs to the XNOR depend on target variables, then on line 10-11 we create two case expressions and two new base names. The case expressions are simply x_i and $\overline{x_i}$ for some variable x_i created from the *name* argument. On lines 12 and 13 we then call *bp* recursively on inputs i_1 and i_2, with suitable high and low functions and distinct base names. Finally, the union of the two sets of triples returned is formed.

On lines 14-15 we deal with the case when e is a NOT expression. Here we simply call *bp* recursively switching *h* and *l*.

Finally on lines 16-27 we deal with the case when the final operator of e is an AND gate. On line 17, we traverse the bexpr to find as large an AND gate as possible by calling the routine find_big_ands. This routine not only finds all inputs that are conjoined, it also separates the inputs into two groups: the ones that only depend on variables in $\mathcal{C}$ and those that do not. The conjunction of the first group is formed in line 18, since they do not need indexing variables. Only a consistency requirement, computed on line 19, is needed to make sure the assignment of the indexing variables does not contradict the value of this expression. Technically, we are only required to add the implication $h \to c$, but rather than having a separate set of consistency requirements, we include the triple (c, h, F) in the relation.

On line 20 we compute a set of mutually exclusive Boolean expressions over some fresh indexing variables that will be used to encode which of the inputs should be set to low. On line 21, we deal with a common special case that allows us to call *bp* recursively without new base names, thus greatly increase the sharing of indexing variables. Finally, on lines 25 and 26 we call *bp* recursively on every input expression modifying the low condition according to the case expressions.

A. Coverage

Although the intuition behind the algorithm is fairly simple, it is critical to ensure that the trajectory assertions after abstraction verify all of the cases that were verified by the original assertions. We guarantee this by proving coverage in Theorem 1. Note that apart from indexing variables $\mathcal{X}$ we also need to consider the symbolic constants $\mathcal{C}$. Recall that $\mathcal{X}$ are used to index different cases of assertions for $\mathcal{T}$, while the symbolic constants $\mathcal{C}$ basically introduce some additional, temporary target variables. For coverage we hence show that for every assignment of target variables and symbolic constants we can find an indexing corresponding to it.

Theorem 1 (Coverage): Given a Boolean expression represented as a bexpr-tree e, let $rl = bp(\mathcal{C}, e, x_0, \overline{x_0}, name)$ be the result of the algorithm in Fig. 3, where *name* is chosen to ensure that none of the generated variables are called x_0. If

$$R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = \bigwedge_{(e_i,h_i,l_i)\in rl} (h_i \to e_i) \wedge (l_i \to \overline{e_i}),$$

then $\forall \mathcal{T}.\forall \mathcal{C}.\exists \mathcal{X}.\ R[\mathcal{X}, \mathcal{C}, \mathcal{T}]$.

Proof: Let the target variables $\mathcal{T}$ and the symbolic constants $\mathcal{C}$ be arbitrary, but fixed. Then we can simulate the bexpr-tree. That is, we can determine all input and output values of each constructor of e. In particular, the simulation result of e and the values of all consequents e_n in the relation $R[\mathcal{X}, \mathcal{C}, \mathcal{T}]$ are known. By induction on the depth k of the bexpr-tree e we show that there exists a variable assignment for $\mathcal{X}$ such that

$$\forall n.\ \ ((e_n = \mathrm{F}) \to (h_n = \mathrm{F})) \wedge ((e_n = \mathrm{T}) \to (l_n = \mathrm{F})) \quad (4)$$

Then $R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = \mathrm{T}$ as required.

Base case: The only bexpr-tree e of depth 0 is a single variable t. In this case the relation is $R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = (x_0 \to t) \wedge (\overline{x_0} \to \overline{t})$. Property (4) is satisfied by choosing $x_0 = t$.

Inductive step: Suppose that for all bexpr-trees of depth $\leq k$ there is a variable assignment such that (4) holds.

We examine the last constructor of the bexpr-tree e of depth $k+1$ and determine an assignment for the variables $\mathcal{Y}$ introduced by the algorithm for that constructor. W.l.o.g. assume the last constructor is an AND or a NOT that completes an XNOR. In both cases assume at least one input depends on a variable not in $\mathcal{C}$—otherwise, no new indexing variables are introduced and the induction hypothesis provides the required variable assignment for $\mathcal{X}$. We consider the two constructor cases, AND and NOT, in I and II below.

I. Suppose the last constructor is AND. We analyse e further to extract all consecutive AND constructors, resulting in a multiple input AND of maximal size, which is achieved by *find_big_ands* in line 17 of Fig. 3. We will now determine a variable assignment as desired depending on the simulation result of e. For this we split the inputs of the AND gate into two groups: *cis*, which only depend on variables in $\mathcal{C}$, and i_j, which depend on some indexing variables.

Ia. First suppose $cis = \emptyset$. Further suppose the simulation result of e is true. We choose $x_0 = \mathrm{T}$, and hence the output conditions are $h_e = \mathrm{T}, l_e = \mathrm{F}$. Observe that all inputs i_j of the AND have to be true in the simulation and $\forall j.\ h_{i_j} = \mathrm{T}, l_{i_j} = \mathrm{F}$. Arbitrary values can be chosen for $\mathcal{Y}$, they are only necessary for the case where e is false. Note that the variables $\mathcal{Y}$ are never used at any other point in the algorithm due to the use of new, unique base names. An assignment for the remaining $\mathcal{X}$ is determined as follows: By induction hypothesis there exists an assignment for the bexpr-tree of each input i_j. The algorithm guarantees that each input-tree uses a unique set of variables by specifying a unique base name for the variables. Hence the assignments for each input tree, and our assignment for $\mathcal{Y}$, can be merged to an assignment as required.

Ib. Now suppose $cis = \emptyset$, but the simulation result of e is false. We choose $x_0 = \mathrm{F}$, and hence $h_e = \mathrm{F}, l_e = \mathrm{T}$. Observe that at least one input i_j of the AND has to be false in the simulation, w.l.o.g. $i_1 = \mathrm{F}$. The algorithm specifies that $h_{i_j} = l_e = \mathrm{F}$ and $l_{i_j} = case_j[\mathcal{Y}] \wedge l_e$. We choose a variable assignment for $\mathcal{Y}$ such that $case_1 = \mathrm{T}$ and hence $case_j = \mathrm{F}\ \ \forall j \neq 1$.

By induction hypothesis there exists a variable assignment for the input tree of i_1 as required. It does not include $\mathcal{Y}$, so the variable assignment can be merged with our assignment to $\mathcal{Y}$. Observe that for all other inputs $h_{i_j} = l_{i_j} = \mathrm{F},\ j \neq 1$. But the algorithm forms new h_{in}, l_{in} using conjunction with h_{out}, l_{out} only, so we can deduce that all h, l on the input trees of $i_j,\ j \neq 1$ will be false throughout, in particular at the terminating points e_n included in the relation. This means that all variable assignments, and in particular the one chosen for i_1, satisfy (4) trivially.

Ic. Next suppose $cis \neq \emptyset$ and $c = \bigwedge_{c_i \in cis} c_i$ is false. Then at least one of the concrete inputs is false, and accordingly the simulation output is false. Hence we choose $x_0 = \mathrm{F}$ and receive $h_e = \mathrm{F}$. The extra requirement $h_e \to c$ is therefore trivially satisfied. Also notice that in this case we need not force any other inputs to be low, i.e. $l_{i_j} = \mathrm{F}$ for all j. So the indexing variables $\mathcal{Y}$ can be chosen arbitrarily with the same reasoning as in Ia.

Id. Finally suppose $cis \neq \emptyset$ and $c = \bigwedge_{c_i \in cis} c_i$ is true. Then the extra requirement $h_e \to c$ is trivially satisfied. If the simulation output is true, we choose $x_0 = \mathrm{T}$ and $\mathcal{Y}$ can be chosen arbitrarily as seen in Ia. If the simulation output is false, we choose $x_0 = \mathrm{F}$. Given that we know that the concrete inputs are all true this means the indexing has to force one of the remaining inputs to be false. As seen in Ib, w.l.o.g. we choose $\mathcal{Y}$ such that $case_1 = \mathrm{T}$ and use the induction hypothesis to complete the variable assignment.

II. Now suppose the last constructor is a NOT that completes an XNOR, where i_1 and i_2 are the two subtrees of e being compared. This is detected by *is_XNOR* in line 2 of Fig. 3. Further assume that neither input only depends on $\mathcal{C}$. Recall that for XNORs the algorithm sets the input conditions $h_{i_1} = y \wedge h_e \vee y \wedge l_e$ and $l_{i_1} = \overline{y} \wedge h_e \vee \overline{y} \wedge l_e$ and $h_{i_2} = y \wedge h_e \vee \overline{y} \wedge l_e$ and $l_{i_2} = \overline{y} \wedge h_e \vee y \wedge l_e$ using the output conditions and a single indexing variable y solely introduced for that step. Intuitively, the indexing variables x_0 and y enumerate the four possible

cases that can occur for the 2-bit XNOR. We now specify which assignments correspond to the explicit instances.

Suppose the simulation result shows that $i_1 = \mathrm{F}, i_2 = \mathrm{F}, e = \mathrm{T}$. We choose $x_0 = \mathrm{T}$, and hence $h_e = \mathrm{T}, l_e = \mathrm{F}$. By further choosing $y = \mathrm{F}$ we receive $h_{i_1} = h_{i_2} = \mathrm{F}$ and $l_{i_1} = l_{i_2} = \mathrm{T}$, which corresponds to the input values. By induction hypothesis there exists an assignment for the bexpr-tree i_1 and i_2. These two assignments and our assignment to x_0 and y are merged and results in the full assignment required. Notice that i_1 and i_2 use distinct base name for their variables and never x_0 or y, so no conflicts can occur.

The other three possible cases are similar. If simulation result shows that $i_1 = \mathrm{T}, i_2 = \mathrm{T}, e = \mathrm{T}$, we choose $x_0 = \mathrm{T}$ and $y = \mathrm{T}$. If it is $i_1 = \mathrm{F}, i_2 = \mathrm{T}, e = \mathrm{F}$, we choose $x_0 = \mathrm{F}$ and $y = \mathrm{F}$. Finally, if $i_1 = \mathrm{T}, i_2 = \mathrm{F}, e = \mathrm{F}$, then we choose $x_0 = \mathrm{F}$ and $y = \mathrm{T}$. □

B. DAGs and multiple outputs

In the previous we assumed that we are given a bexpr-tree. The algorithm can be extended to handle DAGs as follows. Every fanout point is marked with a new variable v and the graph is cut at these points. The algorithm is then run for each resulting tree with distinct variables to guarantee independence of indexing variables. Observe that in some cases v will be an output, and in others an initial input. The resulting triplets are merged as follows.

Assume T_o is a set of triplets computed by the algorithm, where the output was previously a fanout point v. W.l.o.g. let v_h and v_l be two fresh, distinct variables used for h and l respectively when running the algorithm.

Further assume that T_i is a set of triplets computed where v is one of the initial inputs, i.e. v is a target variable. Let $(v, h_v, l_v) \in T_i$ be the triplet corresponding to v.

We then determine T'_o by substituting v_h with h_v and v_l with l_v in every triplet of T_o. This disposes of the variables v_h and v_l. Additionally, due to the fact that h_v and l_v are mutex, contradicting assignments for v_h and v_l are eliminated. The union of all T_i and T'_o for all cut points then represents the set of all triplets needed for the desired relation.

This procedure is motivated by the fact that we want to have the same value on every branch of a fanout. Coverage is achieved with the same argument as before. The requirement of using unique variables for each run guarantees that assignments for the indexing variables can be chosen independently and merged without conflicts.

Similarly, we can handle bexprs with multiple outputs. We simply run the algorithm for each output independently. For this we introduce a unique indexing variable x_0^i for each output and require unique base names $name^i$. This ensures independence of the results and allows us to form the conjunction of the relations, where coverage follows as before.

V. Efficient Preimage Implementation

The algorithm in Fig. 3 produces a partitioned abstraction relation, which allows early existential quantification to be used in the preimage computations explained in Section II-A. We can, for example, use the technique described in [9]. But we can do even better by exploiting the special form that our relations have by virtue of how they are generated.

First, it is easy to show that the abstraction relation $R[\mathcal{X}, \mathcal{T}]$ generated by our algorithm can be written as a conjunction $S[\mathcal{X}] \wedge T[\mathcal{X}, \mathcal{T}]$, where $S[\mathcal{X}]$ contains no target variables. Terms in $S[\mathcal{X}]$ are generated by line 3, when $\mathit{freevars}(e) \subseteq \mathcal{C}$, or by the consistency requirement in line 19 of Fig. 3. The conjunct $T[\mathcal{X}, \mathcal{T}]$ has the form

$$\bigwedge_i (h_i[\mathcal{X}] \rightarrow t_i) \wedge (l_i[\mathcal{X}] \rightarrow \overline{t_i}),$$

where t_i ranges over the target variables $\mathcal{T}$, and no target variable occurs in any h_i or l_i. We define $T_i[\mathcal{X}, t_i]$ to be $(h_i[\mathcal{X}] \rightarrow t_i) \wedge (l_i[\mathcal{X}] \rightarrow \overline{t_i})$, so that $T[\mathcal{X}, \mathcal{T}] = \bigwedge_i T_i[\mathcal{X}, t_i]$.

The main theorem of this section shows how this structure is exploited. We first introduce some notation. Given a guard P, let $\mathcal{F} = \mathit{freevars}(P)$ and let $\mathcal{I}$ be the set of indices i such that $t_i \in \mathcal{F}$. Let $R{\downarrow}P = \bigwedge_{i \in \mathcal{I}} T_i$. Intuitively, $R{\downarrow}P$ denotes the part of R that mentions the target variables in P. Note that $P_{R\downarrow P} = \exists \mathcal{F}. R{\downarrow}P \wedge P$.

Theorem 2: Given a guard P and a relation R produced by the algorithm in Fig. 3, define $D = S \wedge \bigwedge_i \overline{h_i \wedge l_i}$. Then:

$$P_R = \begin{cases} D \wedge P & \text{if } \mathit{freevars}(P) \cap \mathcal{T} = \emptyset \\ D \wedge \overline{l_i} & \text{if } P = t_i \\ D \wedge \overline{h_i} & \text{if } P = \overline{t_i} \\ D \wedge P_{R\downarrow P} & \text{otherwise} \end{cases}$$

$$P^R = D \wedge \overline{\overline{P}_R}$$

Because D does not depend on the guard P, it can be precomputed when R is created and used for all guards in the trajectory assertion to be transformed. By building hash tables that map t_i to h_i and l_i, the two middle cases can be computed very quickly. The fourth case, seen only rarely in practice, can be optimized using quantification scheduling.

The remainder of this section is devoted to proving the above theorem. First, given a relation R, we define $dom(R)[\mathcal{X}] = \exists \mathcal{T}.\, R[\mathcal{X}, \mathcal{T}]$. Intuitively, $dom(R)$ holds for all valuations of the indexing variables that represent used, or consistent, indexing cases.

The following lemma states that $dom(R)$ is equal to the expression D used in Theorem 2.

Lemma 1: $dom(R) = S \wedge \bigwedge_i \overline{h_i \wedge l_i}$.

Proof: By definition, $dom(R) = \exists \mathcal{T}. R$. Using the known shape of R, we expand this to $\exists \mathcal{T}. S \wedge \bigwedge_i T_i$. Since S does not depend on the target variables $\mathcal{T}$, and each T_i depends only on target variable t_i, we can push the quantifier inwards to obtain $S \wedge \bigwedge_i \exists t_i. T_i$. By eliminating the quantifier, this can easily be shown to equal $S \wedge \bigwedge_i \overline{h_i \wedge l_i}$. □

The next lemma states that only the parts of a relation that mention the target variables that occur in a guard need to be included when computing the preimage.

Lemma 2: $P_R = dom(R) \wedge P_{R\downarrow P}$.

Proof: First, we express P_R in a form that resembles the goal. By definition, $P_R = \exists\mathcal{T}.R \wedge P$. Using the known shape of R, this can be rewritten to $\exists\mathcal{T}.S \wedge (\bigwedge_i T_i) \wedge P$. We then partition the variables $\mathcal{T}$ into the ones that occur in P, $\mathcal{U}$, and the rest of them, $\mathcal{V}$, and also split the big conjunction similarly, to obtain $\exists\mathcal{V}.\exists\mathcal{U}.S \wedge (\bigwedge_{i\notin\mathcal{I}} T_i) \wedge (\bigwedge_{i\in\mathcal{I}} T_i) \wedge P$. Each T_i mentions only one target variable (namely t_i), and S does not mention any target variables, so the quantifiers can be pushed inwards: $S \wedge (\exists\mathcal{V}.\bigwedge_{i\notin\mathcal{I}} T_i) \wedge \exists\mathcal{U}.(\bigwedge_{i\in\mathcal{I}} T_i) \wedge P$. So $P_R = S \wedge (\exists\mathcal{V}.\bigwedge_{i\notin\mathcal{I}} T_i) \wedge P_{R\downarrow P}$ by the definition of $R\downarrow P$.

Now consider the subformula $\exists\mathcal{V}.\bigwedge_{i\notin\mathcal{I}} T_i$. Again, each T_i mentions only target variable t_i, so the quantifier can be pushed inwards to get $\bigwedge_{i\notin\mathcal{I}} \exists t_i.T_i$. It is easy to show that $\exists t_i.T_i = \overline{h_i \wedge l_i}$, and so $\exists\mathcal{V}.\bigwedge_{i\notin\mathcal{I}} T_i = \bigwedge_{i\notin\mathcal{I}} \overline{h_i \wedge l_i}$. We use this new equivalence to simplify the formula obtained earlier. Hence, $P_R = S \wedge (\bigwedge_{i\notin\mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R\downarrow P}$.

It remains to show $S \wedge (\bigwedge_{i\notin\mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R\downarrow P} = dom(R) \wedge P_{R\downarrow P}$. We do this by proving implication in both directions.

Leftwards: According to Lemma 1, the right side can be rewritten to $S \wedge (\bigwedge_i \overline{h_i \wedge l_i}) \wedge P_{R\downarrow P}$. This formula trivially implies the left side, since the only difference is that the left side contains fewer conjuncts.

Rightwards: Assume that $S \wedge (\bigwedge_{i\notin\mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R\downarrow P}$ holds. For a moment, we focus on the fact that the subformula $P_{R\downarrow P}$ holds, which, by definition, is equal to $\exists\mathcal{U}.(\bigwedge_{i\in\mathcal{I}} T_i) \wedge P$. We can weaken this to the formula $\exists\mathcal{U}.(\bigwedge_{i\in\mathcal{I}} T_i)$, and push the quantifier inwards yielding $\bigwedge_{i\in\mathcal{I}} \exists t_i.T_i$. This formula can easily be shown to equal $\bigwedge_{i\in\mathcal{I}} \overline{h_i \wedge l_i}$.

Combining this formula with the original assumption results in $S \wedge (\bigwedge_i \overline{h_i \wedge l_i}) \wedge P_{R\downarrow P}$. By Lemma 1, this is equal to $dom(R) \wedge P_{R\downarrow P}$, which concludes the proof. □

Theorem 2 can now be established using Lemmas 1 and 2. It is easy to show that if a guard P contains no target variables, then $P_R = P^R = dom(R) \wedge P$. This justifies the first case of Theorem 2. For the second and third case of Theorem 2, we only need to specialize Lemma 2 with $P = t_k$ and $P = \overline{t_k}$. The fourth case follows immediately from the lemmas. The identity $P^R = dom(R) \wedge \overline{\overline{P}_R}$ is easily shown using the definitions of the preimage operations.

One last observation allows us to improve the preimage calculations further. Notice that the expression $D = dom(R) = \exists\mathcal{T}.R[\mathcal{X},\mathcal{T}]$ is present as a conjunct in every case. As shown by Melham and Jones [7], the coverage condition $\forall\mathcal{T}.\exists\mathcal{X}.\, R[\mathcal{X},\mathcal{T}]$ must be supplied as an environmental constraint in the STE run. In other words, we are only interested in cases where this constraint is true. Since the formula D is implied by the constraint, D can safely be assumed to be true, and hence removed from the formulas. This significantly reduces the complexity of the preimage operations.

VI. Experimental Results

We illustrate the use of our algorithm on three classes of circuits; two that are traditional targets for the use of symbolic indexing and one that illustrates the power of our algorithm in applying abstraction in more subtle ways. All results were obtained with BDD-based STE in the Forte [3] environment on a 2 Gbyte laptop.

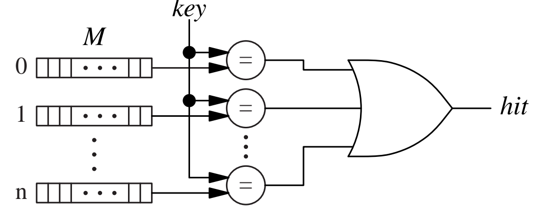

Fig. 4. High-level model of CAM.

Our first example is a traditional content-addressable memory (CAM), as illustrated in Fig. 4. The verification we are interested in is the behavior of the *hit* signal. It should be high iff there is an entry in the CAM with the same content as the key being presented. The specification function we use is the obvious one: try matching the key against each entry and take the disjunction of the results to yield the *hit* value. More precisely, the specification function we use is

$$hit = \bigvee_i (M[i] = key)$$

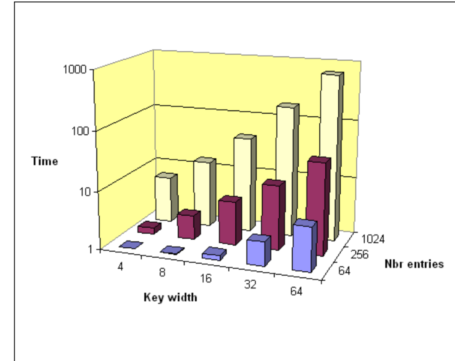

Fig. 5. Results for CAM verification.

We restricted the abstraction by declaring the key inputs to be symbolic constants. With this formulation, in Fig. 5 we show the time required for verifying a CAM of varying size and number of entries. The time for running the verification is split roughly in three equal parts: running the abstraction algorithm, computing the preimages, and finally running the STE verification. The results indicate that running time grows linearly with the size of the CAM. Indeed, our algorithm automatically generates the indexing suggested in [2], i.e., we yield an indexing that previously had to be developed with careful reasoning. It should be pointed out that this design could not be verified for larger sizes without using symbolic indexing: For BDD based verification, we ran out of memory; for SAT based verification, the verification process timed out.

It is interesting to note that if we apply our algorithm to the CAM specification but do not declare any symbolic constants, we obtain a much finer grained symbolic indexing scheme. For example, one case included in this finer indexing family

covers the case in which bit i of the key is different from bit i in every entry in the CAM. This input should lead to the hit signal being low. Not only does this approach yield a symbolic indexing scheme that is far less efficient, the STE verification failed on our design due to over-abstraction. Effectively, we obtained an X on the hit signal when we expected a 0. The reason for this was that in our design the key input to the CAM was protected by a (simulated) error correcting circuit. This circuit needed all inputs to the key to be fully defined to produce non-X outputs. In practice, it appears that such over-abstractions are relatively easy to avoid, but more work in this direction is clearly needed.

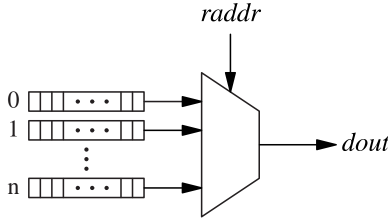

Fig. 6. High-level model of memory.

Our second example is a read operation for a memory, which is abstractly shown in Fig. 6. The specification function essentially consists of the natural sequence of "if read-address is i then return the content of address i else try $i + 1$". In contrast, the implementation uses a decoder circuit, pre-charged logic etc. to implement the read operation. Here, we state that the read address should not be abstracted, but kept fully symbolic. As can be seen in the graph in Fig. 7, the run-time behaviour of the STE verification is exceedingly good. A closer inspection of the symbolic indexing computed reveals that it is virtually identical to the one suggested in [2]. Given that we did not need to provide any information except stating that the address should be symbolic, this result clearly demonstrates the efficiency and practicality of our approach.

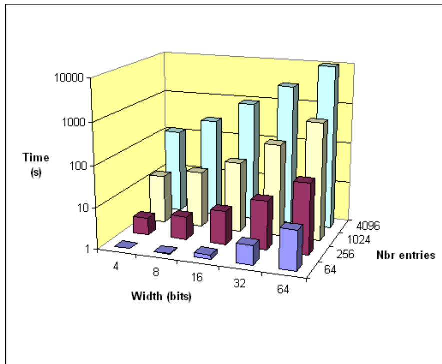

Fig. 7. Results for memory read.

Our final example is a scheduler abstractly shown in Fig. 8. Its functionality is to compute the address of the oldest entry that is ready. Providing a specification function that actually computes the oldest ready entry is fairly involved. However, supplying a relation that checks that a proposed address is the

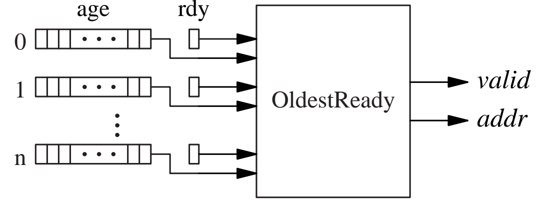

Fig. 8. High-level model of scheduler.

oldest ready is much easier. Thus we write the specification as a relation in terms of the symbolic constants: address $\vec{a}$ and its age $\vec{o}$. The complete specification relation is given by:

$$\bigvee_i \left(\vec{a} = i \wedge rdy[i] \wedge age[i] = \vec{o} \wedge \bigwedge_{j \neq i} \left(\overline{rdy[j]} \vee age[j] < \vec{o} \right) \right)$$

In applying our algorithm, we state that $\vec{a}$ and $\vec{o}$ are symbolic constants. In Fig. 9 we show data on the run-time behavior of this verification effort, revealing that using this technique realistically sized schedulers can be verified. It is worth pointing out that trying to verify the same circuit without symbolic indexing, i.e., with variables in every state holding register and input, fails for circuits larger than 16 entries and width of the age registers of at least 4. In other words, not even a fairly trivial scheduler can be handled without abstraction. At the same time, with the method of this paper, this verification is completely straightforward and fully automatic.

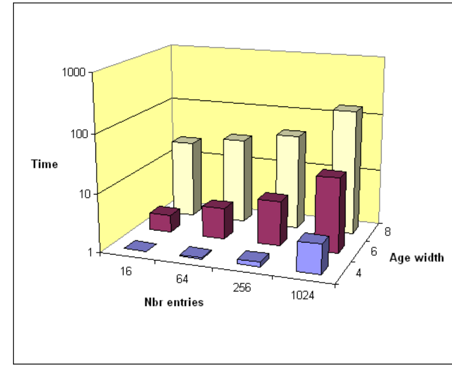

Fig. 9. Results for scheduler verification.

VII. Related Work

There is a rich and growing literature on automatic abstraction for model checking and verification. Some cornerstone techniques are *localization reduction* [10], *counterexample-guided abstraction refinement* [11], and *predicate abstraction* [12]. These and other methods have been extensively studied in the context of symbolic model-checking for hardware. By contrast, the literature on *automatic* abstraction for STE or GSTE is rather thin.

One notable exception is the work of Tzoref and Grumberg [13] on abstraction refinement for STE. This addresses the problem of over-abstraction in STE that manifests itself in symbolic simulation by a node carrying X where the

consequent expects 0 or 1. This is common with manually-formulated assertions, usually because some input variable node has not been driven by the antecedent [8]. A heuristic algorithm is presented in [13] that assigns fresh, distinct Boolean variables to input nodes in a clever way targeted at eliminating the Xs that make the model-checking run fail.

Roorda [14] presents a SAT-based method that assists in manual abstraction refinement for STE. The algorithm provides the user with hints for refining abstractions that over-approximate circuit behaviour. For a given assertion and circuit, Roorda's algorithm finds a minimal set of extra circuit inputs and an assignment of Booleans to them that will eliminate Xs at relevant circuit outputs. The user can then reformulate their assertion to drive these nodes with variables.

Both [13] and [14] are aimed at strengthening verification properties by driving more inputs. This refines the abstraction level of the specification to eliminate Xs, but in contrast with our work, does not in itself introduce complex, indexed families of abstractions.

Finally, our algorithm bears a resemblance to the D-algorithm [15] and other automatic test pattern generation methods. Both work backwards through a combinational circuit finding ways in which outputs might be forced high or low.

VIII. Conclusion

The algorithm we presented in this paper is, to the best of our knowledge, the first automatic abstraction algorithm for STE that yields results equivalent to carefully hand-crafted assertions. This has the potential to significantly increase the use of this type of abstraction in STE-based verification.

Further improvement of our approach is planned. A fruitful field for future research will be to examine how to best encode the abstractions. That is, when should we reuse indexing variables, and when should we introduce fresh ones? This is particularly interesting when handling DAGs and multiple outputs, and when examining SAT-based STE verification. Some first results seem to indicate this type of abstraction does not speed up SAT-based STE verification the same way that it does BDD-based ones. Perhaps this is caused by an unsuitable encoding, or some entirely new approach is needed. We intend to investigate this issue much more thoroughly.

A second important research direction concentrates on how to refine the abstraction our algorithm computes in case it over-approximates the circuit function. In this paper, we introduced a simple and somewhat crude solution by allowing users to state that certain signals should not be abstracted. This is a perfectly workable solution for many examples, but more automatic abstraction refinement methods are needed to deploy our approach more widely. Applying the algorithm presented in [13], slightly modified so it can handle indexing, is one promising option to consider.

Finally, another obvious extension will be to incorporate the presented ideas into the GSTE algorithm [16]. We also intend to apply our method to a much wider class of problems to determine its strengths and weaknesses. We suspect this will yield more insight into how to further enhance our approach.

Acknowledgments

This research was conducted while Carl-Johan Seger was a Visiting Fellow at Balliol College, Oxford University.

References

[1] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.

[2] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation," in *Design Automation Conference*. ACM Press, Jun. 1997, pp. 167–172.

[3] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, Sept. 2005.

[4] T. Schubert, "High Level Formal Verification of Next-Generation Microprocessors," in *DAC'03: Proceedings of the 40th conference on design automation*. ACM Press, 2003, pp. 1–6.

[5] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, Apr. 1991.

[6] C.-T. Chou, "The Mathematical Foundation of Symbolic Trajectory Evaluation," in *Computer Aided Verification*, ser. LNCS, vol. 1633. Springer-Verlag, 1999, pp. 196–207.

[7] T. F. Melham and R. B. Jones, "Abstraction by Symbolic Indexing Transformations," in *Formal Methods in Computer-Aided Design: FMCAD 2002*, ser. LNCS, vol. 2517. Springer-Verlag, 2002, pp. 1–18.

[8] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical Formal Verification in Microprocessor Design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, Jul./Aug. 2001.

[9] P. Chauhan, E. M. Clarke, S. Jha, J. H. Kukula, T. R. Shiple, H. Veith, and D. Wang, "Non-linear Quantification Scheduling in Image Computation," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001, pp. 293–298.

[10] R. P. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[12] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," in *CAV'97*, ser. LNCS, vol. 1254. Springer-Verlag, 1997, pp. 72–83.

[13] R. Tzoref and O. Grumberg, "Automatic refinement and vacuity detection for symbolic trajectory evaluation," in *Computer Aided Verification, 18th International Conference: CAV 2006*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 190–204.

[14] J.-W. Roorda and K. Claessen, "SAT-based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation," in *Computer Aided Verification, 18th International Conference: CAV 2006*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 175–189.

[15] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, Jul. 1966.

[16] J. Yang and C.-J. H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 3, pp. 345–353, 2003.