Abstraction by Symbolic Indexing Transformations

Thomas F. Melham[1] and Robert B. Jones[2]

[1] Department of Computing Science, University of Glasgow,
Glasgow, Scotland, G12 8QQ.
[2] Strategic CAD Labs, Intel Corporation,
JF4-211, 2511 NE 25th Avenue, Hillsboro, OR 97124, USA.

Abstract. Symbolic indexing is a data abstraction technique that exploits the partially-ordered state space of symbolic trajectory evaluation (STE). Use of this technique has been somewhat limited in practice because of its complexity. We present logical machinery and efficient algorithms that provide a much simpler interface to symbolic indexing for the STE user. Our logical machinery also allows correctness assertions proved by symbolic indexing to be composed into larger properties, something previously not possible.

1 Introduction

Symbolic trajectory evaluation (STE) is an efficient model checking algorithm especially suited to verifying properties of large datapath designs [1]. STE is based on *symbolic ternary simulation* [2], in which the Boolean data domain $\{0, 1\}$ is extended to a partially-ordered state space by the addition of an unknown value 'X'. This gives circuit models in STE a built-in and flexible data abstraction hierarchy.

Symbolic indexing is a technique for formulating STE logic formulas in a way that exploits this partially-ordered state space and reduces the number of BDD variables needed to verify a property. The method can make a dramatic difference in the time and space needed to check a formula, and can be used to verify circuit properties that are infeasible to verify directly [3].

Although symbolic indexing has been known for a long time [4], our experience is that it is not exploited nearly as often as it is applicable. In part, this is because only limited user-level support has been available in libraries provided to verification engineers. But, more importantly, correctness assertions proved by symbolic indexing are not formulated in a way that makes them composable at higher levels. Two formulas written using symbolic indexing might express two circuit properties that imply some desired result but encode these properties using incompatible indexing schemes. Moreover, there is no explicit characterization of the conditions under which more composable formulas can be derived from the indexed ones.

T. F. Melham and R. B. Jones, 'Abstraction by Symbolic Indexing Transformations', in in *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002: Portland, November 2002: Proceedings*, edited by M. D. Aagaard and J. W. O'Leary, Lecture Notes in Computer Science, vol. 2517 (Springer-Verlag, 2002), pp. 1–18. The final publication is available at link.springer.com via `https://doi.org/10.1007/3-540-36126-X_1`

This paper describes some logical machinery aimed at bridging these gaps. We present an algorithm to transform ordinary verification problems into symbolically indexed form, together with an account of the side-conditions that must hold for this transformation to be sound. We also describe how the algorithm can be applied in the presence of environmental constraints, an important consideration in practice. Finally, we provide some experimental results on a CAM (content-addressable memory) circuit.

The work presented in this paper does not completely automate the use of symbolic indexing in the verification flow. Our algorithms require the user to supply an *indexing relation* that expresses the desired abstraction scheme; we do not provide a method whereby an effective indexing relation can be discovered in the first place. Our results do, however, guarantee the soundness, subject to certain well-characterized side-conditions, of using an indexing relation to transform a verification property. This key result paves the way for future work on automatic abstraction techniques for STE, in which an attempt might be made to discover suitable indexing relations automatically.

2 STE Model Checking

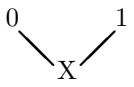
Symbolic trajectory evaluation [1] is an efficient model checking algorithm especially suited to verifying properties of large datapath designs. The most basic form of STE works on a very simple linear-time temporal logic, limited to implications between formulas built from only conjunction and the next-time operator. STE is based on *ternary simulation* [2], in which the Boolean data domain $\{0, 1\}$ is extended with a third value 'X' that stands for an indeterminate value ('0' or '1'). This provides STE with powerful state-space abstraction capabilities, as will be illustrated subsequently.

While the basic STE logic is weak, its expressive power is greatly extended by implementing a *symbolic* ternary simulation algorithm. Symbolic ternary simulation [4] uses BDDs [5] to represent classes of data values on circuit nodes. With this representation, STE can combine many (ternary) simulation runs—one for each assignment of values to the BDD variables—into a single *symbolic* simulation run covering them all.

In this section, we provide a brief overview of STE model checking theory. A full account of the theory can be found in [1] and an alternative perspective in [6].

2.1 Circuit Models

Symbolic trajectory evaluation employs a ternary data model with values drawn from the set $\mathcal{D} = \{0, 1, \mathrm{X}\}$. A partial order relation $\leq$ is introduced, with $X \leq 0$ and $X \leq 1$:



This orders values by information content: X stands for an unknown value and so is ordered below 0 and 1.

We suppose there is a set of *nodes*, $\mathcal{N}$, naming observable points in circuits. A *state* is an instantaneous snapshot of circuit behavior given by assigning a value in $\mathcal{D}$ to every circuit node in $\mathcal{N}$. The ordering $\leq$ on $\mathcal{D}$ is extended pointwise to get an ordering $\sqsubseteq$ on states. We wish this to form a complete lattice, and so introduce a special 'top' state, $\top$, and define the set of states $\mathcal{S}$ to be $(\mathcal{N}{\to}\mathcal{D}) \cup \{\top\}$. The required ordering is then defined for states $s_1, s_2 \in \mathcal{S}$ by

$$s_1 \sqsubseteq s_2 \quad \triangleq \quad s_2{=}\top \quad \text{or} \quad s_1, s_2 \in \mathcal{N}{\to}\mathcal{D} \text{ and } s_1(n) \leq s_2(n) \text{ for all } n \in \mathcal{N}$$

The intuition is that if $s_1 \sqsubseteq s_2$, then s_1 may have 'less information' about node values than s_2, i.e. it may have Xs in place of some 0s and 1s. If one considers the three-valued 'states' s_1 and s_2 as *constraints* or *predicates* on the actual, i.e. Boolean, state of the hardware, then $s_1 \sqsubseteq s_2$ means that every Boolean state that satisfies s_1 also satisfies s_2. We say that s_1 is 'weaker than' s_2. (Strictly speaking, $\sqsubseteq$ is reflexive and we really mean 'no stronger than', but it is common to be somewhat inexact and just say 'weaker than'.) The top value $\top$ represents the unsatisfiable constraint. The *join* operator on pairs of states in the lattice is denoted by '$\sqcup$'.

To model dynamic behavior, a sequence of the values that occur on circuit nodes over time will represented by a function $\sigma \in \mathbb{N}{\to}\mathcal{S}$ from time (the natural numbers $\mathbb{N}$) to states. Such a function, called a *sequence*, assigns a value in $\mathcal{D}$ to each node at each point in time. For example, $\sigma\ 3\ reset$ is the value present on the $reset$ node at time 3. We lift the ordering on states pointwise to sequences in the obvious way:

$$\sigma_1 \sqsubseteq \sigma_2 \quad \triangleq \quad \sigma_1(t) \sqsubseteq \sigma_2(t) \text{ for all } t \in \mathbb{N}$$

One convenient operation, used later in stating the semantics of STE, is taking the ith suffix of a sequence. The ith suffix of a sequence σ is written σ^i and defined by

$$\sigma^i\ t \triangleq \sigma\ (t{+}i) \text{ for all } t \in \mathbb{N}.$$

The suffix operation σ^i simply shifts the sequence σ forward i points in time, ignoring the states at the first i time units.

In symbolic trajectory evaluation, the formal model of a circuit c is given by a next-state function $\mathrm{Y}_c \in \mathcal{S} \to \mathcal{S}$ that maps states to states. Intuitively, the next-state function expresses a constraint on the real, Boolean states into which the circuit may go, given a constraint on the current Boolean state it is in. The next-state function must be monotonic and a requirement for implementations of STE is that they extract a next-state function that has this property from the circuit under analysis.[1]

[1] In practice, the circuit model Y_c is constructed on-the-fly by ternary symbolic simulation of a netlist description of the circuit c.

A sequence σ is said to be a *trajectory* of a circuit if it represents a set of behaviors that the circuit could actually exhibit. That is, the set of behaviors that σ represents (i.e. possibly using unknowns) is a subset of the Boolean behaviors that the real circuit can exhibit (where there are no unknowns). For a circuit c, we define the set of all its trajectories, $\mathcal{T}(c)$, as follows:

$$\mathcal{T}(c) \triangleq \{\sigma \mid \mathrm{Y}_c(\sigma\ t) \sqsubseteq \sigma\ (t{+}1) \text{ for all } t \in \mathbb{N}\}$$

For a sequence σ to be a trajectory, the result of applying Y_c to any state must be no more specified (with respect to the $\sqsubseteq$ ordering) than the state at the next moment of time. This ensures that σ is consistent with the circuit model Y_c.

2.2 Trajectory Evaluation Logic

One of the keys to the efficiency of STE and its success with datapath circuits is its restricted temporal logic. A *trajectory formula* is a simple linear-time temporal logic formula with the following syntax:

$$\begin{array}{lll} f, g := n \text{ is } 0 & & \text{- node } n \text{ has value } 0 \\ \quad \mid\ n \text{ is } 1 & & \text{- node } n \text{ has value } 1 \\ \quad \mid\ f \text{ and } g & & \text{- conjunction of formulas} \\ \quad \mid\ P \rightarrow f & & \text{- } f \text{ is asserted only when } P \text{ is true} \\ \quad \mid\ \mathsf{N}f & & \text{- } f \text{ holds in the next time step} \end{array}$$

where f and g range over formulas, $n \in \mathcal{N}$ ranges over the nodes of the circuit, and P is a propositional formula ('Boolean function') called a *guard*. The basic trajectory formulas 'n is 0' and 'n is 1' say that the node n has value 0 or value 1, respectively. The operator and forms the conjunction of trajectory formulas. The trajectory formula $P \rightarrow f$ weakens the subformula f by requiring it to be satisfied only when the guard P is true. Finally, $\mathsf{N}f$ says that the trajectory formula f holds in the next point of time.

Guards are the only place that variables may occur in the primitive definition of trajectory formulas. At first sight, this seems to rule out assertions such as 'node n has value b', where b is a variable. But the following syntactic sugar allows variables—indeed any propositional formula—to be associated with a node:

$$n \text{ is } P \triangleq P \rightarrow (n \text{ is } 1) \text{ and } \neg P \rightarrow (n \text{ is } 0)$$

where $n \in \mathcal{N}$ ranges over nodes and P ranges over propositional formulas.

The definition of when a sequence σ satisfies a trajectory formula f is now given. Satisfaction is defined with respect to an assignment ϕ of Boolean truth-values to the variables that appear in the guards of the formula:

$$\begin{array}{lcl} \phi, \sigma \models n \text{ is } 0 & \triangleq & \sigma(0){=}\top,\ \text{or } \sigma(0) \in \mathcal{N}{\rightarrow}\mathcal{D} \text{ and } \sigma\ 0\ n = 0 \\ \phi, \sigma \models n \text{ is } 1 & \triangleq & \sigma(0){=}\top,\ \text{or } \sigma(0) \in \mathcal{N}{\rightarrow}\mathcal{D} \text{ and } \sigma\ 0\ n = 1 \\ \phi, \sigma \models f \text{ and } g & \triangleq & \phi, \sigma \models f \text{ and } \phi, \sigma \models g \\ \phi, \sigma \models P \rightarrow f & \triangleq & \phi \models P \text{ implies } \phi, \sigma \models f \\ \phi, \sigma \models \mathsf{N}f & \triangleq & \phi, \sigma^1 \models f \end{array}$$

where $\phi \models P$ means that the propositional formula P is satisfied by the assignment ϕ of truth-values to the Boolean variables in P.

The key feature of this logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written $[f]^\phi$. It is defined recursively as follows:

$$\begin{array}{lcl}
[m \text{ is } 0]^\phi\ t & \triangleq & \lambda n.\ 0 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[m \text{ is } 1]^\phi\ t & \triangleq & \lambda n.\ 1 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise X} \\
[f \text{ and } g]^\phi\ t & \triangleq & ([f]^\phi\ t) \sqcup ([g]^\phi\ t) \\
[P \rightarrow f]^\phi\ t & \triangleq & [f]^\phi\ t \text{ if } \phi \models P, \text{ otherwise } \lambda n.\,\mathrm{X} \\
[\mathsf{N} f]^\phi\ t & \triangleq & [f]^\phi\ (t{-}1) \text{ if } t{\neq}0, \text{ otherwise } \lambda n.\,\mathrm{X}
\end{array}$$

The crucial property enjoyed by this definition is that $[f]^\phi$ is the unique weakest sequence that satisfies f for the given ϕ. That is, for any ϕ and σ, $\phi, \sigma \models f$ if and only if $[f]^\phi \sqsubseteq \sigma$.

The algorithm for STE is also concerned with the weakest *trajectory* that satisfies a particular formula. This is the *defining trajectory* for a formula, written $[\![f]\!]^\phi$. It is defined by the following recursive calculation:

$$\begin{array}{lcl}
[\![f]\!]^\phi\ 0 & \triangleq & [f]^\phi\ 0 \\
[\![f]\!]^\phi\ (t{+}1) & \triangleq & [f]^\phi\ (t{+}1) \sqcup \mathrm{Y}_c(\ [\![f]\!]^\phi\ t)
\end{array}$$

The defining trajectory of a formula f is its defining sequence with the added constraints on state transitions imposed by the circuit, as modeled by the next-state function Y_c. It can be shown that $[\![f]\!]^\phi$ is the unique weakest trajectory that satisfies f.

2.3 Symbolic Trajectory Evaluation

Circuit correctness in symbolic trajectory evaluation is stated with *trajectory assertions* of the form $A \Rightarrow C$, where A and C are trajectory formulas. The intuition is that the *antecedent* A provides stimuli to circuit nodes and the *consequent* C specifies the values expected on circuit nodes as a response.

A trajectory assertion is true for a given assignment ϕ of Boolean values to the variables in its guards exactly when every trajectory of the circuit that satisfies the antecedent also satisfies the consequent. For a given circuit c, we define $\phi \models A \Rightarrow C$ to mean that for all $\sigma \in \mathcal{T}(c)$, if $\phi, \sigma \models A$ then $\phi, \sigma \models C$. The notation $\models A \Rightarrow C$ means that $\phi \models A \Rightarrow C$ holds for all ϕ.

The fundamental theorem of trajectory evaluation [1] follows immediately from the previously-stated properties of $[f]^\phi$ and $[\![f]\!]^\phi$. It states that for any ϕ, the trajectory assertion $\phi \models A \Rightarrow C$ holds exactly when $[C]^\phi \sqsubseteq [\![A]\!]^\phi$. The intuition is that the sequence characterizing the consequent must be 'included in' the weakest sequence satisfying the antecedent that is also consistent with the circuit.

This theorem gives a model-checking algorithm for trajectory assertions: to see if $\phi \models A \Rightarrow C$ holds for a given ϕ, just compute $[C]^\phi$ and $[\![A]\!]^\phi$ and compare them point-wise for every circuit node and point in time. This works because both A and C will have only a finite number of nested next-time operators N, and so only finite initial segments of the defining trajectory and defining sequence need to be calculated and compared.

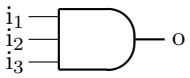
Much of the practical utility of STE comes from the key observation that it is possible to compute $[C]^\phi \sqsubseteq [\![A]\!]^\phi$ not just for a specific ϕ, but as a symbolic constraint on an arbitrary ϕ. This constraint takes the form of a propositional formula (e.g. a BDD) which is true exactly for variable assignments ϕ for which $[C]^\phi \sqsubseteq [\![A]\!]^\phi$ holds. Such a constraint is called a *residual*, and represents precisely the conditions under which the property $A \Rightarrow C$ is true of the circuit.

3 Symbolic Indexing in STE

Two important properties follow from the STE theory just presented. Consider an STE assertion $A \Rightarrow C$. Suppose we replace the antecedent A with a new antecedent B that has a defining sequence *no stronger* than that of A (i.e. $[B]^\phi \sqsubseteq [A]^\phi$ for all ϕ). Then by monotonicity of underlying the circuit model we will also have that $[\![B]\!]^\phi \sqsubseteq [\![A]\!]^\phi$ for all ϕ. Hence if we can prove $\models B \Rightarrow C$, then the original STE assertion $\models A \Rightarrow C$ also holds. This is called *antecedent weakening*. Likewise, if we replace the consequent C with a new consequent D that has a defining sequence *no weaker* than that of C (i.e. $[C]^\phi \sqsubseteq [D]^\phi$ for all ϕ) and we can prove $\models A \Rightarrow D$, then the original STE assertion $\models A \Rightarrow C$ also holds. This is called *consequent strengthening*.

Symbolic indexing is the systematic use of antecedent weakening to perform data abstraction for certain circuit structures. It exploits the partially-ordered state space of STE to reduce the complexity of the BDDs needed to verify a circuit property. Intuitively, symbolic indexing is a way to use BDD variables only 'when needed'.

The idea can be illustrated using the following trivial example. Consider the three-input AND gate shown below:



With direct use of STE, an assertion that could be used to verify this device is

$$\models (\mathrm{i}_1 \text{ is } a) \text{ and } (\mathrm{i}_2 \text{ is } b) \text{ and } (\mathrm{i}_3 \text{ is } c) \Rightarrow (\mathrm{o} \text{ is } a \wedge b \wedge c) \tag{1}$$

In primitive form, this would be expressed as follows:

$$\begin{aligned} \models\ & \neg a \rightarrow (\mathrm{i}_1 \text{ is } 0) \text{ and } a \rightarrow (\mathrm{i}_1 \text{ is } 1) \text{ and} \\ & \neg b \rightarrow (\mathrm{i}_2 \text{ is } 0) \text{ and } b \rightarrow (\mathrm{i}_2 \text{ is } 1) \text{ and} \\ & \neg c \rightarrow (\mathrm{i}_3 \text{ is } 0) \text{ and } c \rightarrow (\mathrm{i}_3 \text{ is } 1) \text{ and} \\ & \quad \Rightarrow \\ & \neg a \vee \neg b \vee \neg c \rightarrow (\mathrm{o} \text{ is } 0) \text{ and } a \wedge b \wedge c \rightarrow (\mathrm{o} \text{ is } 1) \end{aligned} \tag{2}$$

The strategy here is to place unique and unconstrained Boolean variables on each input node in the device, and symbolically simulate the circuit to check that the desired function of these variables will appear on the output node.

STE's unknown value X allows us to reduce the number of variables needed to verify the desired property. Because of the functionality of the AND gate, only the four cases enumerated in the table below need to be verified:

case	i_1	i_2	i_3	o
0	0	X	X	0
1	X	0	X	0
2	X	X	0	0
3	1	1	1	1

If at least one of the AND inputs is 0, the output will be 0 regardless of the values on the other two inputs. In these cases, X may be used to represent the unknown value on the other two input nodes. If all three inputs are 1, then the output is 1 as well. Antecedent weakening, and the fact that the four cases enumerated above cover all input patterns of 0s and 1s, means this is sufficient for a complete verification.

Symbolic indexing is the technique of using Boolean variables to enumerate or 'index' groups of cases in this efficient way. For the AND gate, there are just four cases to check, so these can be indexed with two Boolean variables, say p and q. These cases can then be verified simultaneously with STE by checking the following trajectory assertion:

$$
\begin{aligned}
\models\ & \neg p \wedge \neg q \rightarrow (i_1 \text{ is } 0) \text{ and } p \wedge q \rightarrow (i_1 \text{ is } 1) \text{ and} \\
& p \wedge \neg q \rightarrow (i_2 \text{ is } 0) \text{ and } p \wedge q \rightarrow (i_2 \text{ is } 1) \text{ and} \\
& \neg p \wedge q \rightarrow (i_3 \text{ is } 0) \text{ and } p \wedge q \rightarrow (i_3 \text{ is } 1) \text{ and} \\
& \qquad \Rightarrow \\
& \neg p \vee \neg q \rightarrow (o \text{ is } 0) \text{ and } p \wedge q \rightarrow (o \text{ is } 1)
\end{aligned}
\tag{3}
$$

If this formula is true, then we have definitive—but somewhat indirectly stated—formal evidence that the AND gate does what is required. Antecedent weakening says that whenever (3) allows an input circuit node to be X, that node could have been set to either 0 or 1 and the input/output relation verified would still hold. It can also be established by inspection of the cases enumerated in the antecedent that the given combinations of explicit constant 0s and 1s and implicit Xs covers the whole input space. This (informal) reasoning tells us that the indexed formula (3) amounts to a complete verification of the expected behavior.

The advantage of symbolic indexing is that it reduces the number of Boolean variables needed to verify a property. In the AND gate the reduction is trivial—two variables instead of three. But much greater reductions are possible in real applications, and there are certainly circuits that can be verified in STE by indexing but cannot be verified directly. Memory structures are one notable example that arise frequently.

4 Indexing Transformations

The technical contribution of this paper addresses two problems with using symbolic indexing in practice. First, how can we gain the efficiency of symbolic indexing and yet still obtain properties that make direct, non-indexed statements about circuit correctness? Second, what side conditions must hold to ensure the soundness of such a process?

We show how to construct indexed STE assertions from direct ones, given a user-supplied specification of the indexing scheme to be employed. For example, applying the method to the AND gate formula (1) above produces the indexed formula (3). This provides an accessible interface to the indexing technique. The user no longer needs to generate indexed antecedents and consequents explicitly, but can describe the indexing scheme abstractly and let a computer program construct the correct indexed formulas.

Moreover, if the resulting indexed assertions are proven true, then the original assertion is also true by construction (subject to a certain side condition). This means that the original assertion can subsequently be used in higher-level reasoning. For example, it might be composed via theorem proving with other assertions verified using a different indexing scheme.

4.1 Indexing Relations

The user's interface to our indexing method is an *indexing relation* that specifies the indexing scheme to be applied to the problem at hand. The relation is a propositional logic formula of the form $R(xs, ts)$. It relates the Boolean variables ts appearing in the original problem and the Boolean variables xs that will index the cases being grouped together in the abstraction. The original problem variables ts are called the index *target variables* and the variables to be introduced xs are called the *index variables*.

For the AND gate, the index targets are a, b, c and the index variables are p and q. The indexing relation R is:

$$R(p, q, a, b, c) \equiv (\overline{pq} \supset \overline{a}) \wedge (p\overline{q} \supset \overline{b}) \wedge (\overline{p}q \supset \overline{c}) \wedge (pq \supset abc)$$

As can be seen, this relation represents in logical form an enumeration of the four cases in the table of Section 3. Note that the indexing relation is not one-to-one (though other indexing relations may be). This reflects the Xs that appear in the table in Section 3, and indeed is essential to making the indexing a data abstraction at all.

4.2 Preimage and Strong Preimage

It is convenient to specify two operations on predicates using an indexing relation. The first is the ordinary *preimage* operation. Given a relation R and a predicate P on the target variables, the preimage P_R is defined by

$$P_R \triangleq \exists ts.\, R(xs, ts) \wedge P(ts)$$

The second is the *strong* preimage of a predicate. Given a relation R and a predicate P on the target variables, the strong preimage P^R is defined by

$$P^R \triangleq P_R \wedge \neg\,[\exists ts.\, R(xs, ts) \wedge \neg P(ts)]$$

The strong preimage is $P^R(xs)$ holds of some index xs precisely when xs is in the preimage of P and *not* in the preimage of the negation of P.

These operations are illustrated in Figure 1. The solid circle is the preimage

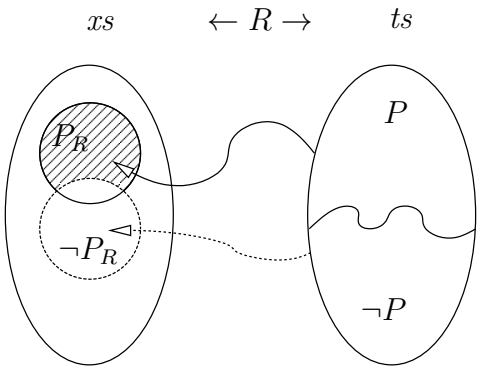


Fig. 1. Index Relation Preimages

P_R of P and the dotted circle the preimage $(\neg P)_R$ of the negation of P. The strong preimage P^R is the shaded region—i.e. that part of P_R that does not also lie within $(\neg P)_R$.

4.3 Transforming STE Formulas with Indexing Relations

Our indexing transformation for an STE assertion $A \Rightarrow C$ applies the strong preimage operation to the guards of the antecedent A and the preimage operation to the guards of the consequent C. For given trajectory formula f and indexing relation R, we write f_R for the preimage of f under R and f^R for the strong preimage of f under R. The definitions of these operations are given by recursion over the syntax of trajectory formulas in the obvious way:

$$\begin{array}{ll} (n \text{ is } 0)^R & \triangleq n \text{ is } 0 \\ (n \text{ is } 1)^R & \triangleq n \text{ is } 1 \\ (f \text{ and } g)^R & \triangleq f^R \text{ and } g^R \\ (P \to f)^R & \triangleq P^R \to f^R \\ (\mathsf{N}\, f)^R & \triangleq \mathsf{N}\, f^R \end{array} \qquad \begin{array}{ll} (n \text{ is } 0)_R & \triangleq n \text{ is } 0 \\ (n \text{ is } 1)_R & \triangleq n \text{ is } 1 \\ (f \text{ and } g)_R & \triangleq f_R \text{ and } g_R \\ (P \to f)_R & \triangleq P_R \to f_R \\ (\mathsf{N}\, f)_R & \triangleq \mathsf{N}\, f_R \end{array}$$

Two theorems about the preimage and strong preimage operations on trajectory formulas are used in the sequel. The first is that applying the strong preimage of an indexing relation to the guards of an STE formula is a weakening operation:

Theorem 1 *For all R, f and ϕ, if $\phi \models R$, then $[f^R]^\phi \sqsubseteq [f]^\phi$.*

This is really the core of our abstraction transformation. Taking the *strong* preimage under an indexing relation can strictly weaken the guards of the formula by 'subtracting out' the indexes of cases in which the guard can be false. This achieves an abstraction by introducing Xs into the defining sequence of the formula.

The second theorem is that applying the preimage of an indexing relation to the guards of an STE formula is a strengthening operation:

Theorem 2 *For all R, f and ϕ, if $\phi \models R$, then $[f]^\phi \sqsubseteq [f_R]^\phi$.*

Each of these theorems follows by a straightforward induction on the structure of the trajectory formula f.

4.4 Transforming STE Assertions with Indexing Relations

The theorems just cited, combined with the STE antecedent weakening and consequent strengthening properties of Section 2, allow an arbitrary property $A \Rightarrow C$ to be indexed by an indexing relation R. Intuitively, we can use an indexing scheme to weaken the antecedent by grouping some of its separate Boolean input configurations using Xs (thereby assuming *less* about circuit behavior). If we use the same indexing to strengthen the consequent, and the resulting STE assertion holds, then we can also conclude the original STE assertion.

To guarantee soundness, a technical side condition must be satisfied—namely that the indexing scheme R completely 'covers' the target variables:

$$\forall ts.\, \exists xs.\, R(xs, ts) \tag{4}$$

This says that for any values of the target variables ts (the variables that appear in A and C), there is an assignment to the index variables xs that indexes it. This condition ensures that every verification case included in the original problem is also covered in the indexed verification—which is clearly necessary, for otherwise the indexed verification would be incomplete.

Before considering the soundness of our transformation, we introduce a notation for the truth of a trajectory formula under a propositional assumption about its Boolean variables. If P is a propositional Boolean formula (for example an indexing relation) and $A \Rightarrow C$ a trajectory assertion, we write $P \models A \Rightarrow C$ to mean that for any valuation ϕ for which $\phi \models P$, we have that $\phi \models A \Rightarrow C$. Informally, we are saying that $A \Rightarrow C$ is true whenever the condition P holds. More detail on how such an assertion can be checked in practice is given in Section 5.1.

Soundness of our abstraction transformation is given by the following theorem.

Theorem 3 *If we can show that* $R(xs, ts) \models A^R \Rightarrow C_R$ *and the indexing relation coverage condition* $\forall ts.\, \exists xs.\, R(xs, ts)$ *holds, then we may conclude* $\models A \Rightarrow C$.

Proof. By the following derivation.

1.	$R(xs, ts) \models A^R \Rightarrow C_R$	[assumption]
2.	$R(xs, ts) \models A \Rightarrow C_R$	[1 and Theorem (1)]
3.	$R(xs, ts) \models A \Rightarrow C$	[2 and Theorem (2)]
4.	$\exists xs.\, R(xs, ts) \models A \Rightarrow C$	[3, because xs do not appear in A or C]
5.	$\forall ts.\, \exists xs.\, R(xs, ts)$	[side condition]
6.	$\models A \Rightarrow C$	[4 and 5]

Note that although the variables ts do not appear in the trajectory assertion $A^R \Rightarrow C_R$ of line 1, the variables xs do. The condition given by $R(xs, ts)$ is therefore significant to verification of this assertion. Indeed in this context it is equivalent to $\exists ts.R(xs, ts)$, which restricts the verification to values of xs that actually do index something.

If the STE algorithm produces a residual when checking the formula shown in line 1, then this will of course be given in terms of the index variables rather than the target variables from the original problem. The user must therefore analyze the residual by taking its *image* under the indexing relation, mapping it back into the original target variables for inspection there.

5 Indexing under Environmental Constraints

Few verifications take place in isolation from complex environmental and other operating assumptions. In this section, we extend our indexing algorithm to incorporate such conditions. We present two methods for indexing under an environmental constraints. The first is the simpler option, and requires little or no user intervention. The second is an alternative that can be applied to certain problems for which the direct approach is infeasible.

Both methods use the technique of *parametric representation* of environmental constraints, which we now briefly introduce.

5.1 Parametric Representation

The *parametric representation* of Boolean predicates is useful for restricting verification to a care set and for reducing complexity by input-space decomposition [7,8,9]. The technique is independent of the symbolic simulation algorithm in STE, does not require modifications to the circuit, and can be used to constrain both input and internal signals.

Consider a Boolean predicate P that constrains input and state variables vs. Suppose we express the required behavior of the circuit as a trajectory assertion $A \Rightarrow C$ over the same variables, but expect this assertion to hold only under the constraint P. That is, we wish to establish that $P \models A \Rightarrow C$. One way of

doing this is to use STE to obtain a residual from $\phi \models A \Rightarrow C$ and then check that P implies this. But this is usually not practical; the complexity of directly computing $\phi \models A \Rightarrow C$ with a symbolic simulator is too great.

A better way is to evaluate $\phi \models A \Rightarrow C$ only for those variable assignments ϕ that actually do satisfy P. The parametric representation does exactly this, by encoding the care predicate implicitly by means of *parametric functions*. Given a satisfiable P, we compute a vector of Boolean functions $Qs = \texttt{param}(P, vs)$ that are substituted for the variables vs in the original trajectory assertion.[2] These functions are constructed so that $P \models A \Rightarrow C$ holds exactly when $\models A[Qs/vs] \Rightarrow C[Qs/vs]$ holds. An algorithm for `param` and its correctness proof are found in [9].

Suppose M is an arbitrary expression—either a propositional logic formula or a trajectory formula—and P is a predicate over the variables vs appearing in M. We write '$M[P]$' for $M[\texttt{param}(P, vs)/vs]$. A complicating factor is that the parametric functions will, in general, contain fresh variables vs' distinct from the original variables vs. When necessary, we will write $M[P](vs')$ to emphasize the appearance of these in the resulting expression.

5.2 Method 1: Direct Parametric Encoding

We wish to apply an indexing relation R to a verification problem $P \models A \Rightarrow C$ that includes a constraint P. With our first method, a fully automatic procedure uses the parametric representation to 'fold' the constraint P into both the trajectory assertion being checked and the relation R. Indexed verification then proceeds as before.

Suppose we wish to check an STE assertion $P \models A \Rightarrow C$ under an environmental constraint P and using an indexing relation $R(xs, ts)$. First, we compute a parametrically-encoded STE assertion $\models A[P] \Rightarrow C[P]$ and indexing relation $R[P]$. We then just supply these to the symbolic indexing algorithm of Section 4.

The soundness of the optimization provided by our transformation is justified as follows. Note that we also write the encoded indexing relation $R[P]$ as $R[P](xs, ts')$, where ts' are the fresh variables introduced by the parametric encoding process.

Theorem 4 *If* $R[P](xs, ts') \models A[P]^{R[P]} \Rightarrow C[P]_{R[P]}$ *and the indexing relation coverage condition* $\forall ts'.\, \exists xs.\, R[P](xs, ts')$ *holds, then* $\models A \Rightarrow C$.

Proof. By the following derivation.

1. $R[P](xs, ts') \models A[P]^{R[P]} \Rightarrow C[P]_{R[P]}$ [assumption]
2. $R[P](xs, ts') \models A[P] \Rightarrow C[P]_{R[P]}$ [1 and Theorem (1)]
3. $R[P](xs, ts') \models A[P] \Rightarrow C[P]$ [2 and Theorem (2)]
4. $\exists xs.\, R[P](xs, ts') \models A[P] \Rightarrow C[P]$ [3, because xs do not appear in A or C]
5. $\forall ts'.\, \exists xs.\, R[P](xs, ts')$ [side condition]

[2] As usual, we write $f[Qs/vs]$ to denote the result of substituting Qs for all occurrences of vs (respectively) in a formula f.

6. $\models A[P] \Rightarrow C[P]$ [4 and 5]
7. $P \models A \Rightarrow C$ [parametric theorem (see [8])]

As before, if the STE run that checks line 1 produces a non-trivial residual this must first be mapped back through the relation $R[P]$ to derive a residual in terms of the target variables of $\models A[P] \Rightarrow C[P]$. But these will, of course, be the fresh variables introduced by the parametric encoding, so we must also undo this encoding in turn to get back to the user's variables of the original assertion $A \Rightarrow C$.

5.3 Method 2: Analyzing Indexed Residuals

While the method presented above is straightforward, it is often infeasible in practice to construct the parameterized indexing relation $R[P]$. Our second method avoids this, while still allowing us to use a constraint predicate P.

We initially run the STE model-checking algorithm on $A^R \Rightarrow C_R$. This will then produce a residual that describes the *indexed* situations under which the property holds. The predicate P is then itself indexed with R, to produce an indexed predicate P_R. This is then checked to ensure it implies the indexed residual obtained from STE. This process is sound only for certain indexing relations R, and the main technical innovation here consists in identifying the required side conditions on R.

The first side condition is similar to the coverage side condition (4) in Section 4.4. It requires the indexing relation to cover all values of the target variables that satisfy the constraint P:

$$\forall ts.\ P(ts) \supset \exists xs.\ R(xs, ts) \tag{5}$$

The second side condition is new. It is that the preimage P_R and the preimage $(\neg P)_R$ must be disjoint, making $P_R = P^R$. The intuition for this condition is provided by considering Figure 1, where P_R and $(\neg P)_R$ overlap. We wish to index the condition P in order to check that it implies the residual—and we must do this by either taking the preimage P_R or the strong preimage P^R. If the preimage P_R is selected, and there is an overlap, then false negatives may occur. Every point in the overlap will be included in the verification, but also maps via R to elements of $\neg P$, and the property may simply not hold for some of these 'don't care' elements. On the other hand, false positives could occur if the strong preimage P^R is selected. In this case, there may be points in P that are indexed only from points in the overlap area, but for which the verification property fails. The solution is to ban the overlap.

One way to ensure $P_R = P^R$ is to make the preimage $(\neg P)_R$ empty. The following condition does this by restricting R from indexing anything in $\neg P$:

$$\forall ts.\ (\exists xs.\ R(xs, ts)) \supset P(ts) \tag{6}$$

If we choose an indexing relation R that exactly partitions P,

$$\forall ts.\ P(ts) \equiv \exists xs.\ R(xs, ts) \tag{7}$$

both side conditions are satisfied.

The soundness of the optimization provided by our transformation is justified as follows. Note again that we write $R(xs, ts)$ as just 'R' when we do not need to emphasize the particular variables involved.

Theorem 5 *Let Q be the residual condition under which the model-checking assertion $R(xs, ts) \models A^R \Rightarrow C_R$ holds. Suppose that $\forall ts.\ P(ts) \equiv \exists xs.\ R(xs, ts)$ and that $P_R \supset Q$. Then $P \models A \Rightarrow C$.*

Proof. By the following derivation.

1. $Q \wedge R(xs, ts) \models A^R \Rightarrow C_R$	[assumption]
2. $P_R \supset Q$	[assumption]
3. $(\exists ts.\ R(xs, ts) \wedge P(ts)) \supset Q$	[2 and definition of P_R]
4. $P(ts) \wedge R(xs, ts) \models A^R \Rightarrow C_R$	[1 and 3, by logic]
5. $P(ts) \wedge R(xs, ts) \models A \Rightarrow C_R$	[4 and Theorem (1)]
6. $P(ts) \wedge R(xs, ts) \models A \Rightarrow C$	[5 and Theorem (2)]
7. $P(ts) \wedge \exists xs.\ R(xs, ts) \models A \Rightarrow C$	[6, because xs do not appear in A or C]
8. $\forall ts.\ P(ts) \equiv \exists xs.\ R(xs, ts)$	[side conditions]
9. $P(ts) \models A \Rightarrow C$	[7 and 8, by logic]

6 Experimental Results

We have implemented the above algorithm as an experimental extension to Forte, a formal verification environment developed in Intel's Strategic CAD Labs. Forte combines STE model checking with lightweight theorem proving in higher-order logic and has successfully been used in large-scale industrial trials on datapath-dominated hardware [10,11,12].

The implementation of our algorithm is highly optimized, to ensure that the cost of computing an indexed STE property does not exceed the benefit gained by the abstraction. As usual with symbolic treatment of relations in model-checking algorithms, the main computational overhead arises from the existential quantifier of the preimage. We use the common strategy of partitioning the indexing relation to allow early quantification. The implementation is also carefully engineered to eliminate redundant computations.

One circuit structure we studied is the simple CAM shown in Figure 2. This compares a 64-bit query against the contents of an n-entry memory, producing a bit that indicates whether the query value is in the memory or not. CAM devices have previously been verified using symbolic indexing by Pandey et al. [3], who devised an indexing scheme with a logarithmic reduction in the number of variables needed—bringing an otherwise infeasible verification within reach of STE.

Our experiments on CAMs showed that we could add our indexing transformation to get a verification of directly-stated CAM properties with acceptable computational overhead. As an example, we present results for the following simple property: if the query value is equal to the contents of one of the CAM

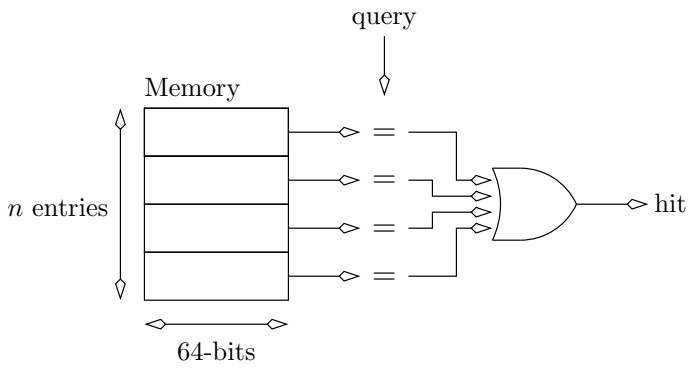


Fig. 2. Simple Content-Addressable Memory (CAM)

memory entries, then the 'hit' output will be true. The formalization of this property in STE involves the use of an environmental constraint to express the condition that the query is equal to one of the CAM entries. The verification therefore employs the methods of Section 5. Of course, this is not a complete characterization of correct behavior for the CAM device. However, it is typical of the kind of property for memory arrays that cannot be verified directly but that yields to the symbolic indexing technique.

Figure 3 shows the CPU time required to verify this property for different numbers of entries in the CAM memory, from 4 up to 64. All runs were performed on a 400 MHz Intel Pentium® II Processor running RedHat® Linux, and user time was determined with the system `time` command. The verification of this property by symbolic indexing, including our indexing transformation algorithm, is much faster than the best-known alternative, namely using the parametric representation to case-split on the location of the hit while simultaneously weakening other circuit nodes. The numbers reported are for the model-checking portions of the verification. Both approaches require similar amounts of deductive reasoning, namely coverage analysis for case splitting and the coverage side condition for symbolic indexing.

As shown in Figure 4, our automatic indexing transformation did not add significant computational overhead to the indexed verification, a requirement for our technique to be feasible in practice. The computational overhead for our indexing algorithm is roughly constant at 50-60% of the total verification time.

7 Conclusions

We have presented algorithms that facilitate easier application of symbolic indexing in STE model checking. Our approach provides a simpler interface for the STE user, making it easier to include the technique in the verification flow.

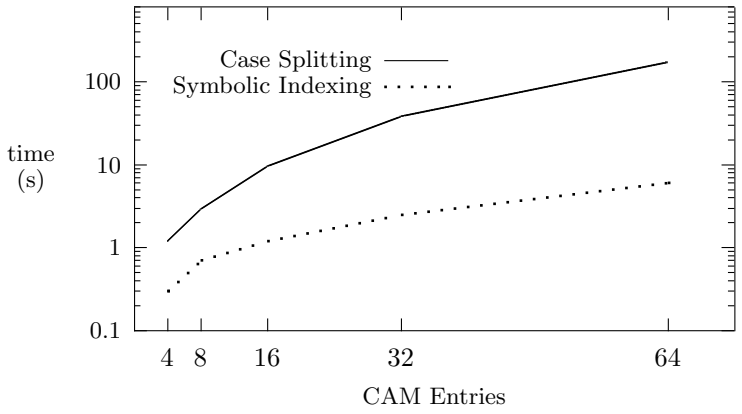


Fig. 3. Symbolic Indexing vs. Case Splitting

Our theoretical results also provide the logical foundation for composing multiple indexed results into larger properties. The method allows us to transform an STE formula into the more efficiently-checkable indexed form, but still conclude the truth of the original formula. A top-level verification can, therefore, be decomposed into separate sub-properties that are verified under different, and possibly incompatible, indexing schemes.

We have demonstrated the efficiency of an implementation of our algorithms by verifying a simple property of a CAM, a hardware structure commonly encountered in microprocessor designs. The indexing scheme applied in this example comes from past work by Pandey et al. [3]. Of course, the single property chosen as an illustration in Section 6 doesn't provide a complete characteriza-

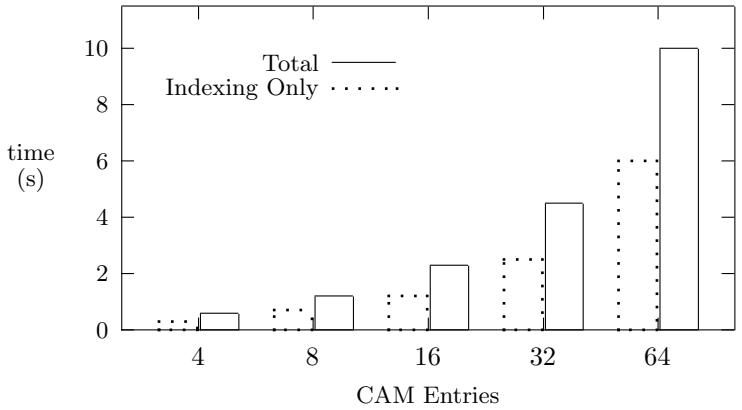


Fig. 4. Overhead of Automatic Indexing Algorithm

tion of the desired behavior of a CAM. Our contribution has been to show that we can both obtain the computational advantages of this indexing scheme *and* justifiably conclude a direct statement of the desired property—with negligible additional cost.

Our algorithm requires a user-supplied abstraction scheme, presented formally as a Boolean relation. Of course the indexing scheme could also be provided as a set of (possibly overlapping) predicates over the the target variables in the original formula. For example, the indexing scheme in Section 3 for the AND gate can also be given by the following set of predicates:

$$\{\neg a,\ \neg b,\ \neg c,\ a \wedge b \wedge c\}$$

These cover the whole input space and precisely characterize the four cases to be verified in terms of the 'target' variables in the original property. A formal indexing relation can just be an arbitrary enumeration of these predicates in terms of a suitable number of index variables and can easily be generated automatically.

But this still leaves the problem of discovering the indexing scheme in the first place. Part of our current research is directed at finding techniques to automatically discover abstractions that can leverage the indexing algorithms presented here.

Finally, we observe that our transformation is a pre-processing step for STE model checking. In this paper, we have assumed a BDD-based STE algorithm. But of course the data abstraction capability of STE's partially-ordered state spaces is orthogonal to the propositional logic technology employed. It is therefore reasonable to suppose that our method would also work with STE algorithms based on SAT [13], provided the formula representation supports our preimage and strong preimage operations. It would also be very interesting to see how our algorithms could be applied to *generalized STE* [14], a promising new model checking method that combines the efficiency of STE's partially-ordered state spaces with a much more expressive and flexible framework for stating properties.

8 Acknowledgments

We thank the anonymous referees for their careful reading of the paper and very helpful comments. John Harrison and Ashish Darbari also provided useful remarks on notation.

References

1. Seger, C.J.H., Bryant, R.E.: Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods in System Design **6** (1995) 147–189
2. Bryant, R.E.: A methodology for hardware verification based on logic simulation. Journal of the ACM **38** (1991) 299–328

3. Pandey, M., Raimi, R., Bryant, R.E., Abadir, M.S.: Formal verification of content addressable memories using symbolic trajectory evaluation. In: ACM/IEEE Design Automation Conference, ACM Press (1997) 167–172
4. Bryant, R.E., Beatty, D.L., Seger, C.J.H.: Formal hardware verification by symbolic ternary trajectory evaluation. In: ACM/IEEE Design Automation Conference, ACM Press (1991) 397–402
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
6. Chou, C.T.: The mathematical foundation of symbolic trajectory evaluation. In Halbwachs, N., Peled, D., eds.: Computer Aided Verification (CAV). Volume 1633 of Lecture Notes in Computer Science., Springer-Verlag (1999) 196–207
7. Jain, P., Gopalakrishnan, G.: Efficient symbolic simulation-based verification using the parametric form of Boolean expressions. IEEE Transactions on Computer-Aided Design of Integrated Circuits **13** (1994) 1005–1015
8. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Formal verification using parametric representations of Boolean constraints. In: ACM/IEEE Design Automation Conference, ACM Press (1999) 402–407
9. Jones, R.B.: Applications of Symbolic Simulation to the Formal Verification of Microprocessors. PhD thesis, Department of Electrical Engineering, Stanford University (1999)
10. O'Leary, J.W., Zhao, X., Gerth, R., Seger, C.J.H.: Formally verifying IEEE compliance of floating-point hardware. Intel Technical Journal (First quarter, 1999) Available at developer.intel.com/technology/itj/.
11. Kaivola, R., Aagaard, M.D.: Divider circuit verification with model checking and theorem proving. In Aagaard, M., Harrison, J., eds.: Theorem Proving in Higher Order Logics. Volume 1869 of Lecture Notes in Computer Science., Springer-Verlag (2000) 338–355
12. Aagaard, M.D., Jones, R.B., Seger, C.J.H.: Combining theorem proving and trajectory evaluation in an industrial environment. In: ACM/IEEE Design Automation Conference, ACM Press (1998) 538–541
13. Bjesse, P., Leonard, T., Mokkedem, A.: Finding bugs in an Alpha microprocessor using satisfiability solvers. In Berry, G., Comon, H., Finkel, A., eds.: Computer Aided Verification (CAV). Volume 2102 of Lecture Notes in Computer Science., Springer-Verlag (2001) 454–464
14. Yang, J., Seger, C.J.H.: Introduction to generalized symbolic trajectory evaluation. In: Proceedings of 2001 IEEE International Conference on Computer Design. (2001) 360–365