Formal Methods in System Design, 22, 205–224, 2003

BDD Based Procedures for a Theory of Equality with Uninterpreted Functions*

ANUJ GOEL
Department of ECE, The University of Texas at Austin

KHURRAM SAJID
Intel

HAI ZHOU
Department of ECE, Northwestern University

ADNAN AZIZ
Department of ECE, The University of Texas at Austin

VIGYAN SINGHAL
Tempus-Fugit

Received September 3, 1999; Accepted October 17, 2002

Abstract. The logic of equality with uninterpreted functions has been proposed for verifying abstract hardware designs. The ability to perform fast satisfiability checking over this logic is imperative for such verification paradigms to be successful. We present symbolic methods for satisfiability checking for this logic. The first procedure is based on restricting analysis to finite instantiations of the variables. The second procedure directly reasons about equality by introducing Boolean-valued indicator variables for equality. Theoretical and experimental evidence shows the superiority of the second approach.

Keywords: uninterpreted functions, logic of equality, BDDs

1. Verifying high-level designs using the theory of equality

A common problem with automatic formal verification is that the computational resources required for verification increase rapidly with the size of the design. State-of-the art tools for verification of gate-level designs cannot as a matter of course verify designs possessing more than a few hundred binary-valued latches.

This observation motivates the development of tools which can operate on designs at a higher level of abstraction. The basic premise is that abstract designs, being less specified,

*This paper extends results reported by the authors at the following conference: A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. "BDD Based Procedures for a Theory of Equality with Uninterpreted Functions," in *Proc. of Conference on Computer-Aided Verification*, Vancouver, Canada, July 1998.

are simpler and thus easier to verify. Another benefit of this approach is that bugs are caught at earlier stages of the design process.

We are interested in the verification of designs at the high-level. This necessitates reasoning about designs where much of the complexity has been abstracted away, e.g., core-based designs [2]. The use of uninterpreted functions (UIFs) has been proposed as a powerful abstraction mechanism for hardware verification [15, 23]. Essentially, UIFs allow the verification tool to avoid getting bogged down by complex details which are irrelevant to the property being proved. In our work, we will use abstractions where datapath is abstracted away by using unbounded integers, complex combinational functions such as multipliers can be abstracted as uninterpreted functions, complex bypass circuitry required in pipeline designs can be captured by the compare operator, and propositional logic can be used to derive control signals. Moreover, memories can also be incorporated in this framework as partially interpreted functions by adding constraints which relate reads and writes [21].

In this context, the primary verification problem we are interested in is design equivalence; specifically, verifying equivalence between a pipelined and nonpipelined processor. This can be posed as a problem in satisfiability checking for quantifier-free formulas involving both equality and UIFs [8]. As shown by Ackermann [1], this problem can be reduced to satisfiability checking of quantifier-free formulas involving only equality through a suitable generalization of the following: given a formula ϕ containing terms $f(x_1)$ and $f(x_2)$, where f is a UIF, replace $f(x_1)$ and $f(x_2)$ and by fresh variables y_1 and y_2 to obtain a formula ψ; then ϕ is satisfiable iff $(x_1 = x_2 \rightarrow y_1 = y_2) \wedge \psi$ is satisfiable. Satisfiability checking for the theory of equality is more complex than satisfiability checking for propositional logic—the axioms equality need to hold. For example, the formula $(x_1 = x_2) \wedge (x_2 = x_3) \wedge \neg(x_1 = x_3)$ is not satisfiable, since it violates the transitivity of equality.

A number of decision procedures exist for the theory of equality with UIFs and its extensions. Pioneering work was done by Shostak [21], who considered linear arithmetic in conjunction with UIFs. His procedure replaces terms generated from UIFs by new variables as previously described; the formula is then converted to a conjunctive normal form, and each conjunct is checked for satisfiability using Integer Linear Programming. In this way, formula satisfiability (and, by duality, validity) can be checked.

Extensions to the basic algorithm of Shostak have been made in many recent papers on processor verification [3, 8, 15]. Essentially, their approach is a variant of the Davis-Putnam procedure for validity checking over propositional logic, with suitable extensions for handling the properties of equality. One source of their efficiency is the ability to split on subformulas; they also use heuristic rewrite rules for formula simplification. Their target application was the verification of pipelined processors. Their notion of correctness is based on the equivalence of the machine state of the nonpipelined machine after processing an instruction and the state resulting in the pipelined machine after executing the same instruction and flushing it out. (This is the standard "commutative diagram" approach to verification [8].) Equivalence is formulated in terms of the validity of a quantifier free formula involving both equality and UIFs.

One difference of our work with the work of [3] is that while they use formulas to encode the designs, we use BDDs which also incorporate the constraints that are required of the UIFs. If these BDDs can be built and manipulated, the validity checking problem

is considerably simplified, and should work more robustly than a rewrite-based approach. However, a naive method for building these BDDs does not work; BDDs become too big. We present a novel encoding technique so that the validity checking problem can be efficiently represented using BDDs.

Hojati et al. [13, 14] use finite instantiations to handle UIFs (we also discuss a finite instantiation based method in Section 3.1). In [13], they require an explicit invocation of Shostak's method to decide equality between two terms containing UIFs; it is not described if Shostak's algorithm is used directly or another approach is used. Their results were negative from a computational point of view, and they conjectured this was because of the absence of a good variable ordering; our experiments corroborate this. We have developed a new approach for encoding the UIF verification problem with BDDs which results in significantly improved runtime, and enjoys nice theoretical properties—this is the approach presented in this paper (Section 3.2). In our preferred method, constraints due to UIFs (based on Ackermann's reduction) are directly represented by BDDs. We provide experimental evidence that this method performs much better than a finite instantiation based method.

1.1. Symbolic procedures for the theory of equality

We motivate the use of symbolic procedures for the theory of equality by drawing analogies to the problem of verifying the equivalence of gate-level combinational netlists. One approach to the equivalence problem is to form a single "product netlist" wherein corresponding inputs are tied together, and corresponding outputs are XOR-ed. Inequivalence can then be checked by forming a large conjunction of propositional formulas corresponding to the "characteristic functions" of the gates, and a formula asserting that a pair of outputs differ; the designs differ iff the conjunction is satisfiable.

Today, state-of-the-art tools for Boolean verification use BDDs and heavily exploit the structure of the design [16]; the original tools were based on case splitting. Current approaches for verification in the theory of equality with UIFs proceed by case splitting on terms occurring in the formula; heuristic rewriting of subformulas is also performed. Based on experiences with analogous approaches for Boolean verification, we predict that these techniques may not be viable as the examples get larger or more complex, especially when the examples are not hand designs but are outputs of automatic CAD tools, e.g., high-level synthesis tools.

A number of very sophisticated satisfiability checkers exist for propositional logic, e.g., [22]. However, even the best checkers timeout when performing equivalence checking on medium and large sized circuits unless they are structurally very similar [16]. This is indeed the case say when performing ATPG, where the circuits differ only in the existence of a stuck-at fault, where clauses relating the two netlists can be added to "guide" the search, but not so for design verification, where specification and implementation are very different. In part, the failure of propositional satisfiability checkers stems from the huge number of variables (one for each gate) that arise in the formulas, and the loss of the knowledge of the design topology.

Symbolic approaches and their variants have performed well on equivalence checking for circuits which are structured very differently. In particular Reduced Ordered Binary

Decision Diagrams (henceforth BDDs) and their variants have proved to be very useful in verifying gate-level combinational and sequential designs [4, 5].

The previously mentioned approaches [3, 21] to satisfiability checking for the quantifier free formulas in the theory of equality are all formula based; by analogy to the propositional case, they will not fare well on designs specified as netlists.

It is, therefore, natural to ask the question whether netlists operating on integer-valued inputs where the only operation on integers is equality checking can be verified by BDD-like techniques. The answer is in the affirmative—in this paper we provide two distinct symbolic procedures for the equivalence checking problem. The first approach, presented in Section 3.1, is a naive transformation of the design into a Boolean-valued netlist of gates by appealing to a "small model" property that the existential fragment of the logic of equality enjoys [18]; we refer to this as the finite instantiation approach. The second approach, described in Section 3.2, uses more insight—a Boolean valued variable e_{ij} is introduced for every pair of inputs x_i and x_j that are compared in the design. We provide both theoretical and experimental evidence for the superiority of this encoding.

1.2. Paper organization

The remainder of this paper is structured as follows. We begin in Section 2 by presenting our modeling machinery, which is in the form of *integer equality* (*IE*) netlists; in Section 2.1 we show how satisfiability of quantifier-free formulas in the theory of equality with UIFs can be reduced to satisfiability problems for IE netlists. In Section 3, we describe the theory behind two symbolic algorithms for verifying the equivalence of such designs. Experimental results are presented in Section 4. We conclude with suggestions for future work in Section 5.

2. Definitions

Designs will be specified as *netlists*. Before entering into a formal discussion of syntax and semantics for designs, we provide some illustrative examples. The design of figure 1(a) takes 4 integer-valued inputs—x_1, x_2, x_3, x_4. The signal t_1 is Boolean-valued, and takes the value 1 exactly when x_1 and x_2 are equal. Intuitively, the structure labeled with "=" returns 1 when its inputs are equal, and 0 otherwise. The signal u_1 is integer-valued; it is equal to x_1 when t_1 is 1, and x_2 when t_1 is 0. The structure labeled MUX operates as a multiplexer. The signal t_2 is Boolean-valued; it takes the value 1 exactly when x_4 is equal to u_1.

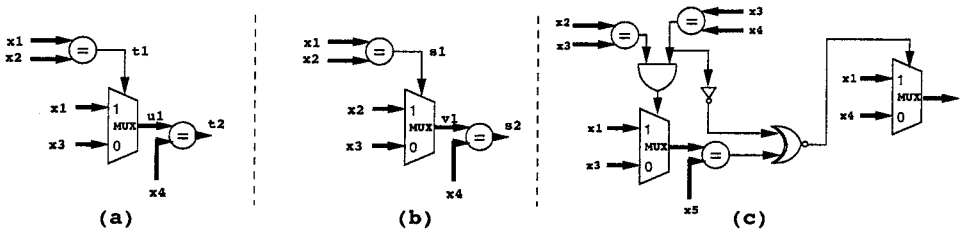

Figure 1. Design examples.

The design of figure 1(b) is identical to the example presented in figure 1(a), except that the 1-input to the multiplexer has been replaced by x_2. Observe however, that the signals t_2 and s_2 take the same value for any input, since the 0-inputs to the corresponding multiplexers are the same, and the 1-input is selected exactly when $x_1 = x_2$.

Definition 1 (IE Netlist Syntax). An *integer equality* (*IE*) *netlist* is a directed acyclic graph, where the nodes correspond to *primitive circuit elements*, and the edges correspond to connections between these elements. Each node is labeled with a distinct variable. The four basic primitive circuit elements are *inputs*, *multiplexers*, *equality checkers*, and *2-input NAND gates*. Some nodes are also labeled as being *outputs*. If an edge (u, v) exists in the IE netlist, u is said to be a *fanin* of v. The set of fanins of a node is assumed to be ordered.

Nodes will be of two *types—Boolean-valued* and *integer-valued*. Inputs and multiplexers are defined to be of integer valued type; equality checkers and 2-input NAND gates are defined to be of Boolean valued type. Input nodes are required to have empty fanin lists; 2-input NAND gate nodes must have two Boolean-valued fanins. Multiplexers are required to have a single Boolean-valued fanin, and two integer-valued fanins; we will often use c to denote the Boolean-valued fanin of a multiplexer node, and u and v to denote the integer-valued fanins. Equality checkers must have two integer-valued fanins.

Figure 1(c) presents an example of an IE netlist.

The restriction to 2-input NAND gates is not serious, since they are functionally complete. Constant-valued nodes and Boolean-valued inputs can also be handled in the framework presented above. The technical issues they bring up are minor, but impinge on the clarity of presentation; for simplicity we ignore them. Additionally, in the interests of exposition, we will often identify the variable corresponding to a node by the node itself.

In the sequel it will be convenient to refer to the *level* of a node.

Definition 2. Given an IE netlist, *level* is the function whose domain is the set of nodes and range is the natural numbers $\omega = \{0, 1, 2, \ldots\}$ defined below:

1. If the node α is of type input, then $level(\alpha) = 0$,
2. otherwise, $level(\alpha) = 1 + \max\{level(\beta) \mid \beta \text{ is a fanin of } \alpha\}$.

The fact that this definition is *sound*, i.e., assigns each node to a unique value can be found in [10, p. 659].

Formally, an *input* to an IE netlist is a function whose domain is the set of input nodes and range is ω. We will often use the Greek letter ι (iota) to denote an input to an IE netlist.

An input can be canonically extended to a function mapping the set of all nodes to ω by evaluating the nodes in topological order as follows:

Definition 3 (IE Netlist Semantics). Let ι be an input, and s an arbitrary node in the netlist. Then ι uniquely defines a value $v_\iota(s)$ to s as follows:

1. If s is an input then $\nu_\iota(s) = \iota(s)$.
2. If s is the output of an equality node with fanins v and w then $\nu_\iota(s) = 1$ if $\nu_\iota(v) = \nu_\iota(u)$, and 0 otherwise.
3. If s is the output of a multiplexer node with fanins c, v and w, then $\nu_\iota(s) = \nu_\iota(v)$ if $\nu_\iota(c) = 1$, and $\nu_\iota(w)$ otherwise.
4. If s is the output of a 2-input NAND with fanins v and w, then $\nu_\iota(s) = 1$ if $\nu_\iota(v) = 0$ or $\nu_\iota(u) = 0$, and 0 otherwise.

It follows from induction on the level of s that ν_ι assigns a unique value to s.

In this manner, an IE netlist D on inputs $a_1, a_2, \ldots, a_n$ and outputs $b_1, b_2, \ldots, b_m$ *defines* a function $f_D : \omega^n \mapsto \omega^m$. Intuitively, two designs are functionally equivalent if in any environment they can be used interchangeably; a necessary and sufficient condition for this is that they define identical functions. Note that an IE netlist can operate on inputs drawn from an arbitrary set, and not just integers, since no operation other than equality is applied to the integer-valued nodes.

Observe that for an input ι, the value taken by any integer-valued node in the IE netlist will be the value assigned to some input node by ι. This is because there are no functions which can be applied to the integers propagated in the IE netlist; integers can only be tested for equality. Indeed, a stronger claim can be asserted—for a given input, the value taken by an integer valued node can be traced back to a *specific* input node. With this in mind, we define the function $F^s(\iota)$ as follows:

Definition 4 (Flows function). Let s be an integer valued node, and ι an input. We define $F^s(\iota)$ as follows:

1. If s is an input then $F^s(\iota) = s$.
2. Otherwise, s must be a multiplexer. Let its fanins be c, v, and w. Then $F^s(\iota) = F^v(\iota)$ when $\nu_\iota(c) = 1$, and $F^s(\iota) = F^w(\iota)$ when $\nu_\iota(c) = 0$.

As in Definition 3, induction on the level of s demonstrates that $F^s(\iota)$ is assigned a unique value for each ι.

We will say the input x_i *flows* to s under the input assignment ι exactly when $F^s(\iota) = x_i$. For example, the input x_1 flows to u_1 for the design in figure 1(a) under the input $x_1 = 2$, $x_2 = 2, x_3 = 3, x_4 = 4$. Note that x_2 does not flow to u_1 under this assignment, even though the value taken at u_1 is the same as that at x_2.

The flow function has the following property:

Lemma 2.1. *Let ι an input, and s be an integer valued node. Then if $F^s(\iota) = x_k$, we must have $\nu_\iota(s) = \iota(x_k)$.*

Proof: We use induction on the level of s.

Induction Hypothesis: Lemma 2.1 holds for each integer valued node of level $\leq q$.

Base Case: $level(s) = 0$. From Definition 2, we see s must be of type input. By Definition 4 [Case 1], $F^s(\iota) = s$; furthermore, by Definition 3 [Case 1], $\nu_\iota(s) = \iota(s)$, proving the base case.

Induction Step: $level(s) = q + 1$. We assume the IH for all integer valued nodes of level $\leq q$.

Let s be any integer valued node whose level is $q + 1$. Examining Definition 2, we see that s must be a multiplexer node.

Let the fanins of s be c, v, and w. Note that the level of c, v, and w must be less than the level of s, and hence we can apply the induction hypothesis to them.

Suppose $\nu_\iota(c) = 1$. Let $F^v(\iota) = x_k$. By Definition 4 [Case 2], $F^s(\iota) = F^v(\iota) = x_k$. Furthermore, by Definition 3 [Case 3], $\nu_\iota(s) = \nu_\iota(v)$. By the IH, $\nu_\iota(v) = \nu_\iota(x_k)$, so $\nu_\iota(s) = \nu_\iota(x_k)$. The case when $\nu_\iota(c) = 0$ is symmetric. Thus the induction step holds.

By the principle of mathematical induction, we see Lemma 2.1 holds for all nodes. □

2.1. Relating designs, equality with UIFs, and IE netlists

As stated in the introduction, we are concerned with designs which operate on unbounded integers, wherein the datapath has been abstracted away using UIFs, and equality is the only operation which is applied to integer nodes; design inequivalence can then be cast as the satisfiability of a quantifier-free formula involving equality and UIFs. IE netlists cannot directly represent UIFs; however, the outputs of the UIF blocks can be replaced by new inputs. When comparing the resulting IE netlists, these new inputs must satisfy the constraint that if the inputs to two instances of the same UIF are equal, then the outputs of the two instances are equal; this constraint can be added to the IE circuit using simple circuitry (an equality checker and a gate). As is the case for Shostak's procedure [21], the soundness and completeness of this construction follows from [1].

3. IE netlist satisfiability checking

IE Netlist Satisfiability Checking consists of taking an IE-netlist and determining if an input assignment exists for which a specified Boolean-valued output can take the value 1.

Note that the usual "product construction" for checking the equivalence of gate-level netlists can be applied to the problem of equivalence checking for IE netlists; this is illustrated in figure 2. Observe that the construction results in exactly one Boolean-valued

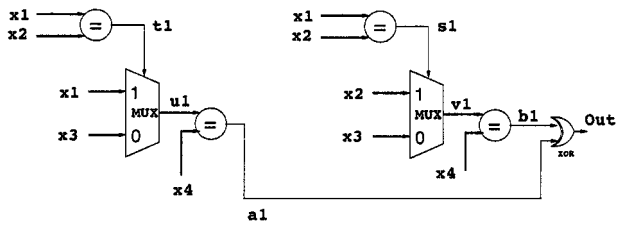

Figure 2. Product construction for equivalence checking.

output, and so the equivalence problem for IE netlists can be easily reduced to IE netlist satisfiability checking.

It is natural to ask if the IE netlist satisfiability checking problem is decidable, and if so, what computational complexity class it lies in.

3.1. Finite model approach

The existence of a decision procedure follows immediately from the fact that a "finite model" folk-theorem holds for the existential fragment of the theory of equality: an existential formula in the language of equality is satisfiable iff it is satisfiable in some model whose universe has cardinality equal to the number of variables occurring in the formula.

We review this result below; details can be gleaned from [18]. A tolerance for mathematical logic will be needed to appreciate the next few paragraph; Chapter 2 of Enderton [11] provides an excellent introduction.

The set of *quantifier free* formulas of the first order logic of pure equality consists of Boolean combinations of formulas of the form $(x_i = x_j)$. A formula σ is said to be an *existential sentence* if it is of the form $\exists x_{i_1} \exists x_{i_2} \ldots \exists x_{i_k} \phi$ where ϕ is a quantifier free formula, and no variables occur in ϕ save for those in the set $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$.

The formulas $\exists x_1 \exists x_2 (\neg(x_1 = x_2))$ and $\exists x_3 \exists x_4 \exists x_2 ((x_3 = x_2) \wedge (x_4 = x_3) \rightarrow (\neg(x_2 = x_4))$ are existential sentences; $\exists x_1 ((x_1 = x_2) \rightarrow (x_1 = x_1))$ is not an existential sentence (since x_2 is not a quantified variable), $\exists x_1 \exists x_2 (x_1 = x_2) \rightarrow \exists x_3 \exists x_4 (\neg(x_3 = x_4))$ is not existential sentence (since not all quantifier symbols appear at the beginning).

Let σ be an existential sentence in this language, with n distinct variable symbols occurring in it. Then it is easily seen (for example by an exercise in [18]) that σ is satisfiable iff it is satisfiable when the variables are taken to range over a set (the "universe") of cardinality n. The bound is "tight", i.e., there are existential sentences σ on n variables so that no less than n elements are required in the universe. This result follows from the fact that since there are only n variables in the formula, and they are only being compared, in any interpretation in which the sentence is true, at most n distinct values are being assigned to the variables.

3.1.1. Reduction to Boolean network satisfiability. The problem of determining if there is an input to an IE netlist which sets a designated Boolean-valued output to 1 can be reduced to checking the satisfiability of an existential sentence in the first order logic of pure equality; the encoding is very similar to that used to convert an instance of Boolean network satisfiability to an instance of CNF formula satisfiability [17]. As an example, the output of the IE netlist in figure 2 can be 1 iff the formula $\exists x_1 \exists x_2 \exists x_3 \exists x_4 ((\phi \wedge \neg\psi) \vee (\neg\phi \wedge \psi))$ is satisfiable, where ϕ abbreviates the expression $(((x_1 = x_2) \rightarrow (x_4 = x_1)) \wedge (\neg(x_1 = x_2) \rightarrow (x_4 = x_3)))$ and ψ abbreviates the expression $(((x_1 = x_2) \rightarrow (x_4 = x_2)) \wedge (\neg(x_1 = x_2) \rightarrow (x_4 = x_3)))$.

Observe that when reducing an instance of IE netlist satisfiability to an instance of satisfiability of an existential sentence in the first order logic of pure equality, we can always produce an existential sentence for which the number of variables is equal to the number of input nodes in the netlist: that is because an IE netlist can be "flattened", i.e., output node functions can always be defined by quantifier-free logical expressions of the

input nodes. (A formal proof of this fact can be established using induction on the level of nodes.) Thus if an IE netlist with n input nodes is satisfiable, it is satisfiable for an input whose range is of size n.

Hence the integer valued variables can be replaced by n-valued variables; each such variable can then be encoded by a vector of $\lceil \lg n \rceil$ Boolean-valued variables. Thus an instance of IE netlist satisfiability can be polytime reduced to an instance of Boolean network satisfiability.

This reduction also shows that satisfiability of IE netlists is in NP; furthermore, it is readily seen that the presence of 2-input NAND gates implies that an instance of CNF satisfiability can be polynomial time reduced to an instance of IE netlist satisfiability, and hence IE satisfiability is NP-complete [12].

3.2. A better encoding

In this section we develop a superior reduction of IE netlist satisfiability to Boolean network satisfiability. We introduce a set of Boolean valued variables—one for each distinct comparison which is made between input nodes. We will show that the IE netlist functionality can be characterized by Boolean-valued functions of these Boolean variables.

Specifically, for an IE netlist D on inputs $x_1, \ldots, x_n$ introduce Boolean variables e_{ij} for $1 \leq i < j \leq n$. We now define for a Boolean-valued node s in D a Boolean function f^s defined over the set of variables $\mathcal{E} = \{e_{ij} \mid 1 \leq i < j \leq n\}$, and for an integer-valued node t, a vector of n Boolean-valued functions $\langle f_1^t, f_2^t, \ldots, f_n^t \rangle$ defined over the same set of variables:

Definition 5 (e_{ij} Encoded Functions).

1. If s is an input, say x_k, we define $f_k^s = 1$ and for $j \neq k$, $f_j^s = 0$.
2. If s is a 2-input NAND gate with inputs u and v, then $f^s = (f^u \cdot f^v)'$.
3. If s is the output of a mux with control input c, and data inputs v, w, then for each $k \in \{1, 2, \ldots, n\}$, we define $f_k^s = f^c \cdot f_k^v + (f^c)' \cdot f_k^w$.
4. If s is the output of an equality node with inputs u, and v then

$$f^s = \sum_{i=1}^{n} \left(f_i^u \cdot f_i^v \right) + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(f_i^u \cdot f_j^v \cdot e_{\min(i,j)\max(i,j)} \right)$$

Intuitively, the e_{ij} variables *indicate* whether the i-th and j-th integer inputs are equal or not. We will soon prove that for a Boolean valued node s, the function f^s represents the condition on the input under which the node evaluates to 1, and that for an integer valued node t, f_k^t represents the condition under which the circuit node assumes the value of the k-th input.

Note the distinction between the input that flows to t under ι, and the value $v_\iota(t)$: it may be the case that $v_\iota(t)$ is equal to the value of more than one input, but there will still be a unique input x_k which flows to t.

Example. Consider the IE netlist shown in figure 2. The functions at the nodes are as follows:

$$\left\langle f_1^{x_1}, f_2^{x_1}, f_3^{x_1}, f_4^{x_1}\right\rangle = \langle 1,0,0,0\rangle \quad \left\langle f_1^{x_2}, f_2^{x_2}, f_3^{x_2}, f_4^{x_2}\right\rangle = \langle 0,1,0,0\rangle$$
$$\left\langle f_1^{x_3}, f_2^{x_3}, f_3^{x_3}, f_4^{x_3}\right\rangle = \langle 0,0,1,0\rangle \quad \left\langle f_1^{x_4}, f_2^{x_4}, f_3^{x_4}, f_4^{x_4}\right\rangle = \langle 0,0,0,1\rangle$$
$$f^{t_1} = e_{12}$$
$$\left\langle f_1^{u_1}, f_2^{u_1}, f_3^{u_1}, f_4^{u_1}\right\rangle = \langle e_{12}, 0, e'_{12}, 0\rangle \quad \left\langle f_1^{v_1}, f_2^{v_1}, f_3^{v_1}, f_4^{v_1}\right\rangle = \langle 0, e_{12}, e'_{12}, 0\rangle$$
$$f^{a_1} = e_{12}\cdot e_{14} + e'_{12}\cdot e_{34} \quad f^{b_1} = e_{12}\cdot e_{24} + e'_{12}\cdot e_{34}$$
$$f^{Out} = e_{12}\cdot(e_{14}\cdot e'_{24} + e'_{14}\cdot e_{24})$$

Note that f^{Out} does not depend on e_{34}.

Encoding the network using these functions allows us to directly store the relationship between the value taken by a node and the inter-relationships between the inputs. Of course, as we see later in this section, to prevent false positives, we will need to introduce procedures that ensure the axioms of equality, particularly transitivity.

The claim that the functions defined above characterize the IE netlist is formalized by the following two lemmas:

Lemma 3.1 (*Completeness*)**.** *Let ι be an input and s a node in the IE netlist. Define the Boolean-valued function ϵ whose domain is the set $\mathcal{E} = \{e_{ij} \mid 1 \le i < j \le n\}$ by $\epsilon(e_{ij}) = 1$ iff $\iota(x_i) = \iota(x_j)$. Then if s is Boolean-valued, $f^s(\epsilon) = \nu_\iota(s)$, and if s is integer-valued, then $f_k^s(\epsilon) = 1$ iff $F^s(\iota) = x_k$.*

The proof of Lemma 3.1, though fairly straightforward, consists of a tedious series of case analyses; readers are encouraged to bypass it on a first reading.

Proof: We use induction on the level of the node.

Induction Hypothesis: Lemma 3.1 holds for every node s for which $level(s) \le q$.

Base Case: $level(s) = 0$. From Definition 2, s must be an input node, say x_k. By Definition 4, we see $F^{x_k}(\iota) = x_k$. From Definition 5 [Case 1], we have $f_k^s = 1$, and for $j \neq k$, we have $f_l^s = 0$. Thus the base case holds.

Induction Step: $level(s) = q + 1$. We assume the IH for all nodes of level $\le q$.

There are three cases for s:

Case 1: s is a 2-input NAND, say with fanins u and v. Note that the level of u and v is at most q, and so we can apply the IH to them. By Definition 5 [Case 2], $f^s(\epsilon) = (f^u(\epsilon)\cdot f^v(\epsilon))'$. Applying the IH to u and v, we see $f^s(\epsilon) = (\nu_\iota(u)\cdot\nu_\iota(v))'$. But by Definition 3 [Case 4], $\nu_\iota(s) = (\nu_\iota(u)\cdot\nu_\iota(v))'$, and so $f^s(\epsilon) = \nu_\iota(s)$; thus the induction step holds for Case 1.

Case 2: s is a multiplexer, say with fanins c, v, and w. Note that the level of c, v, and w is at most q, and so we can apply the induction hypothesis to them. Examining Definition 5 [Case 3], we see $f_k^s(\epsilon) = f^c(\epsilon)\cdot f_k^v(\epsilon) + (f^c(\epsilon))'\cdot f_k^w(\epsilon) = 1$.

Suppose $f^c(\epsilon) = 1$. By the IH, $f^c(\epsilon) = \nu_\iota(c)$. Examining Definition 4 [Case 2] we see $F^s(\iota) = F^v(\iota)$. Applying the induction hypothesis to v, we see $F^v(\iota) = x_k$ iff $f_k^v(\epsilon) = 1$. But since we have taken $f^c(\epsilon) = 1$, it follows that $F^s(\iota) = x_k$ iff $f_k^s(\epsilon) = 1$.

A symmetric argument holds when $f^c(\epsilon) = 0$. Hence the induction step holds for Case 2.

Case 3: s is an equality node, say with fanins u and v. Note that the level of u and v is most q, and so we can apply the IH to them.

We need to prove that $f^s(\epsilon) = \nu_\iota(s)$. We do this in two stages; in the first we show $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$, and in the second we show $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

Stage 1: We prove that $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

By Definition 5 [Case 4],

$$f^s(\epsilon) = \sum_{i=1}^{n} \left(f_i^u(\epsilon) \cdot f_i^v(\epsilon) \right) + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(f_i^u(\epsilon) \cdot f_j^v(\epsilon) \cdot \epsilon\left(e_{\min(i,j)\max(i,j)}\right) \right) \tag{1}$$

Since by hypothesis the expression on the right of Eq. (1) evaluates to 1, one of the following two possibilities must be true:

Possibility 1: For some i, we have $f_i^u(\epsilon) \cdot f_i^v(\epsilon) = 1$. Then by applying the IH to u and v, we see $F^u(\iota) = x_i$ and $F^v(\iota) = x_i$. But by Lemma 2.1, this implies $\nu_\iota(u) = \nu_\iota(x_i)$, and $\nu_\iota(v) = \nu_\iota(x_i)$. By Definition 3 [Case 2], this implies that $\nu_\iota(s) = 1$. Hence for Possibility 1, $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

Possibility 2: For some i and j, where $i \neq j$, we have $f_i^u(\epsilon) \cdot f_j^v(\epsilon) \cdot \epsilon(e_{\min(i,j)\max(i,j)}) = 1$. Then by applying the IH to u and v, we must have $F^u(\iota) = x_i$ and $F^v(\iota) = x_j$. By Lemma 2.1, this implies $\nu_\iota(u) = \nu_\iota(x_i)$, and $\nu_\iota(v) = \nu_\iota(x_j)$. Since $\epsilon(e_{\min(i,j)\max(i,j)}) = 1$, from the definition of ϵ in the statement of Lemma 3.1, we know $\iota(x_i) = \iota(x_j)$. Thus $\nu_\iota(v) = \nu_\iota(u)$; examining Definition 3 [Case 2], we see $\nu_\iota(s) = 1$. Hence for Possibility 2 also, $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

Stage 2: Now we prove $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

Since $\nu_\iota(s) = 1$, going by Definition 3 [Case 2], we must have $\nu_\iota(u) = \nu_\iota(v)$. Let $F^u(\iota) = x_i$ and $F^v(\iota) = x_j$. Then by Lemma 2.1, we have $\nu_\iota(u) = \nu_\iota(x_i)$, and $\nu_\iota(v) = \nu_\iota(x_j)$. Since by supposition, $\nu_\iota(u) = \nu_\iota(v)$, we must have $\nu_\iota(x_i) = \nu_\iota(x_j)$. Now applying the IH to u and v, we see $f_i^u(\epsilon) = 1$ and $f_j^v(\epsilon) = 1$.

If $i = j$, we have $f_i^u(\epsilon) \cdot f_j^v(\epsilon) = 1$; examining the right hand side of Eq. (1) we see $f^s(\epsilon) = 1$. Otherwise, it must be that $i \neq j$; since $\nu_\iota(x_i) = \nu_\iota(x_j)$, we must have $\epsilon(e_{\min(i,j)\max(i,j)}) = 1$. Hence $f_i^u(\epsilon) \cdot f_j^v(\epsilon) \cdot \epsilon(e_{\min(i,j)\max(i,j)}) = 1$, and so again from Eq. (1) it follows that $f^s(\epsilon) = 1$.

Hence we have proved that $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

Thus the induction step holds for Case 3.

By the principle of mathematical induction, it follows that Lemma 3.1 holds for all nodes. □

The functions computed above are not "sound"; values taken by them may not be achievable in the IE netlist. This is because there is no guarantee that the basic axioms equality are satisfied; figure 2 provides an example. As shown previously, the output of the product network is assigned the function $e_{12} \cdot (e_{14} \cdot e'_{24} + e'_{14} \cdot e_{24})$. However, closer inspection shows that it is not possible to find an input ι so that the ϵ extension causes e_{12} and e_{14} to be 1 and e_{24} to be 0 or e_{12} and e_{24} to be 1 and e_{14} to be 0 simultaneously; the transitivity of equality would be violated.

Definition 6. An assignment ϵ to the e_{ij} variables is said to be *consistent* if it satisfies

$$\bigwedge_{i \neq j \neq k} \left(\epsilon\left(e_{\min(i,j)\max(i,j)}\right) \cdot \epsilon\left(e_{\min(j,k)\max(j,k)}\right) \rightarrow \epsilon\left(e_{\min(i,k)\max(i,k)}\right)\right)$$

Intuitively, a consistent assignment is one which satisfied the transitivity of equality.

For consistent assignments, the converse of Lemma 3.1 holds:

Lemma 3.2 (*Soundness*)**.** *Let ϵ be a consistent assignment to the e_{ij} variables. Let s be a node in the IE netlist. Then there exists an input ι such that if s is Boolean-valued, then $f^s(\epsilon) = v_\iota(s)$, and if s is integer-valued then $f^s_k(\epsilon) = 1$ iff $F^s(\iota) = x_k$.*

The proof is based on the fact that ϵ yields an equivalence relation on the inputs, from which the desired input ι can be constructed. Again, the proof is relatively straightforward, but tedious; first time readers are advised to skip it.

Proof: First, define the undirected graph G whose vertex set consists of the input nodes, i.e., $\{x_1, x_2, \ldots, x_n\}$; an edge (x_i, x_j) is present in G exactly when $\epsilon(e_{\min(i,j)\max(i,j)}) = 1$.

Take x_i, x_j, and x_k to be distinct vertices such that the edges (x_i, x_j) and (x_j, x_k) appear in the graph G. Recall that the edge (x_i, x_j) is present exactly when $\epsilon(e_{\min(i,j),\max(i,j)}) = 1$. Similarly, the edge (x_j, x_k) appears exactly when $\epsilon(e_{\min(j,k),\max(j,k)}) = 1$. But since ϵ is assumed to be consistent, this implies that $\epsilon(e_{\min(i,k)\max(i,k)}) = 1$, i.e., the edge (x_i, x_k) has to be present in the graph.

From the above argument we see that every pair of vertices in a connected component of G must be joined by an edge, i.e., the connected components of G are cliques. Consequently, any two vertices x_a, x_b for which $\epsilon(e_{\min(a,b)\max(a,b)}) = 0$ must lie in different connected components.

Let the connected components of this graph be $V_1, V_2, \ldots, V_m$. Define ι to be the input which assigns to input nodes the index of the component which they lie in.

Proposition 3.3. *The input ι discharges the requirement of Lemma* 3.2, *i.e., if s is Boolean-valued, then $f^s(\epsilon) = v_\iota(s)$, and if s is integer-valued, $f^s_k(\epsilon) = 1$ iff $F^s(\iota) = x_k$.*

We use induction on the level of the node to prove the above proposition.

Induction Hypothesis: The claim holds for every node s such that $level(s) \leq q$.

Base Case: $level(s) = 0$. From Definition 2, s must be an input node, say x_k. From Definition 5 [Case 1], $f_i^s(\iota) = 1$ when $i = k$, and $f_i^s(\iota) = 0$ when $i \neq k$. By Definition 4 [Case 1], $F^s(\iota) = x_k$. Thus the base case holds.

Induction Step: $level(s) = q + 1$. We assume the IH for all nodes of level $\leq q$.

There are three cases for s:

Case 1: s is a 2-input NAND, say with fanins u and v. Note that the levels of u and u are at most q, and so we can apply the IH to them. By Definition 5 [Case 2], $f^s(\epsilon) = (f^u(\epsilon) \cdot f^v(\epsilon)'$. Applying the IH to u and v, we see $f^u(\epsilon) = \nu_\iota(u)$ and $f^v(\epsilon) = \nu_\iota(v)$. But by Definition 3 [Case 4], we have $\nu_\iota(s) = (\nu_\iota(u) \cdot \nu_\iota(v))' = (f^u(\epsilon) \cdot f^v(\epsilon))' = f^s(\epsilon)$. Hence the induction step holds for Case 1.

Case 2: s is a multiplexer, with fanins c, v, and w. Note that the levels of c, v and w are at most q, and so we can apply the IH to them. by Definition 5 [Case 3], for each k, we have $f_k^s(\epsilon) = f^c(\epsilon) \cdot f_k^v(\epsilon) + f^c(\epsilon)' \cdot f_k^w(\epsilon)$.

Suppose $f^c(\epsilon) = 1$. By the IH, $\nu_\iota(c) = f^c(\epsilon)$. Also, by Definition 4, $F^s(\iota) = F^v(\iota)$. But by the IH, $F^v(\iota) = x_k$ iff $f_k^v(\epsilon) = 1$. Hence, $F^s(\iota) = x_k$ iff $F_k^v(\epsilon) = 1$ iff $f_k^s(\epsilon) = 1$.

A symmetric argument holds when $f^c(\epsilon) = 0$. Hence the induction step holds for Case 2.

Case 3: s is an equality node, say with fanins u and v. Note that the levels of u and v are at most q, and so we can apply the IH to u and v.

We need to prove that $\nu_\iota(s) = f^s(\epsilon)$. We do this in two stages; in the first we show $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$, and in the second we show $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

Stage 1: We prove $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

By Definition 5 [Case 4],

$$f^s = \sum_{i=1}^{n} \left(f_i^u \cdot f_i^v\right) + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(f_i^u \cdot f_j^v \cdot e_{\min(i,j)\max(i,j)}\right) \tag{2}$$

Since we have assumed $f^s(\epsilon) = 1$, we must have one of the following two possibilities:

Possibility 1: for some i, we have $f_i^u = 1$ and $f_i^v = 1$. By the IH, this implies that $F^u(\iota) = x_i$ and $F^v(\iota) = x_i$. By Lemma 2.1, we know that $\nu_\iota(u) = \nu_\iota(v) = \nu_\iota(x_i)$. By Definition 3 [Case 2], we have $\nu_\iota(s) = 1$, so assuming Possibility 1, $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

Possibility 2: for some i and j, where $i \neq j$, we have $f_i^u = 1$, $f_j^v = 1$ and $\epsilon(e_{\min(i,j),\max(i,j)}) = 1$. By the IH, $f_i^u = 1$ and $f_j^v = 1$ imply $F^u(\iota) = x_i$ and $F^v(\iota) = x_j$, respectively. By Lemma 2.1, this implies that $\nu_\iota(u) = \nu_\iota(x_i)$, and $\nu_\iota(v) = \nu_\iota(x_j)$.

In the graph G used to construct ι, since $\epsilon(e_{\min(i,j),\max(i,j)}) = 1$, both x_i and x_j must lie in the same connected component. Hence they are assigned the same value by ι. Consequently, we must have $\nu_\iota(x_i) = \nu_\iota(x_j)$. This implies that $\nu_\iota(u) = \nu_\iota(v)$, and so by Definition 3 [Case 2], we have $\nu_\iota(s) = 1$, i.e., assuming Possibility 2, $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

Thus we have shown $f^s(\epsilon) = 1 \Rightarrow \nu_\iota(s) = 1$.

Stage 2: Now we prove $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

By Definition 3 [Case 2], $\nu_\iota(s) = 1$ implies that $\nu_\iota(u) = \nu_\iota(v)$. Let $F^u(\iota) = x_i$, and $F^v(\iota) = x_j$. By Lemma 2.1, $\nu_\iota(u) = \nu_\iota(x_i)$, and $\nu_\iota(v) = \nu_\iota(x_j)$, hence $\nu_\iota(x_i) = \nu_\iota(x_j)$, i.e., by Definition 3 [Case 1], $\iota(x_i) = \iota(x_j)$.

Consider $f_i^u(\epsilon)$; by the IH, $f_i^u(\epsilon) = 1$ iff $F^u(\iota) = x_i$. Similarly, by the IH, $f_j^v(\epsilon) = 1$ iff $F^v(\iota) = x_j$.

There are two possibilities:

Possibility 1: $i = j$. Then the corresponding term in $\sum_{i=1}^{n}(f_i^u \cdot f_i^v)$ in Eq. (2) becomes 1, and so $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$ for Possibility 1.

Possibility 2: $i \neq j$. Since $\iota(x_i) = \iota(x_j)$ we must have had $\epsilon(e_{\min(i,j),\max(i,j)}) = 1$. Thus the corresponding term in $\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}(f_i^u \cdot f_j^v \cdot e_{\min(i,j)\max(i,j)})$ must evaluate to 1, and so $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$ for Possibility 2.

Thus we have shown $\nu_\iota(s) = 1 \Rightarrow f^s(\epsilon) = 1$.

Hence the induction step holds for Case 3.

By the principle of mathematical induction, it follows that Lemma 3.2 holds for all nodes. □

3.3. Satisfiability checking using the e_{ij} encoding

It follows from Lemmas 3.1 and 3.2 that the functions in Definition 5 characterize the IE netlist. In particular, they suggest the following approach to satisfiability checking for IE netlists: build BDDs for the e_{ij}-encoded Boolean functions, and then check if there is a consistent assignment under which the output BDD evaluates to 1.

Unfortunately, finding a consistent satisfying assignment for a BDD over the e_{ij} variables will not be easy. The problem we are concerned about can be formulated as follows.

BDD Satisfiability under Consistency (BDD ConSAT)

INSTANCE: A BDD on variables e_{ij}, $1 \leq i < j \leq n$

QUESTION: Is the BDD satisfiable under some minterm ϵ satisfying the consistency requirement: $\bigwedge_{i\neq j\neq k}(\epsilon(e_{ij}) \cdot \epsilon(e_{jk}) \rightarrow \epsilon(e_{ik}))$

Theorem 3.4. *BDD ConSAT is NP-Complete.*

Proof: Given an assignment for the e_{ij} variable, both the BDD and the consistency requirement can be evaluated in polynomial time. This tells us the simple fact that BDD SAT is in NP.

We now show BDD ConSAT to be NP-hard by transforming the problem of PATH WITH FORBIDDEN PAIRS [12] to it.

INSTANCE: Directed graph $G = (V, A)$, specified vertices $s, t \in V$, collection $C = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ of pairs of vertices from V.
QUESTION: Is there a directed path from s to t in G that contains at most one vertex from each pair in C?

This problem remains NP-complete even under the restriction that G is acyclic with no in- or out-degree exceeding 2 and all the given pairs are disjoint. Our transformation will use a version with this restriction.

Given such an instance of PATH WITH FORBIDDEN PAIRS, we can construct an instance of BDD ConSAT as follows.

First, we will modify the instance of PATH WITH FORBIDDEN PAIRS such that each vertex appearing in the pairs has exactly one out-edge. This can be done as follows. For each vertex v, which appears in the pairs and whose out-degree is not 1, we will split it into two vertices v_1 and v_2. All in-edges now end on v_1 and all out-edges now start from v_2 and there is one edge goes from v_1 to v_2. We also substitute v_1 for v in the pairs. It is obvious that the new instance still obeys the restriction and it has a "yes" answer if and only if the original one has one.

Now we will transform the modified DAG into a BDD by labeling and adding vertices and edges. First we will add one vertex labeled $e_{n+1,n+2}$ with an out-edge labeled 0 going to s. We also label vertex t as constant 1. For each pair (a_i, b_i), we will label them as $e_{i,n+1}$, $e_{i,n+2}$, respectively, and their out-edges as 1. For any vertex which is still not labeled, we will label it as $e_{1,k}$, where k is an index different with any previously used one. We will also add a new vertex and label it as constant 0, and add an additional edge to this vertex from each vertex whose out-degree is only 1. The unlabeled edges are then labeled 1 or 0 in such a way that exactly one out-edge from each vertex is labeled 1 and exactly one is labeled with 0. Because of the restriction we added on the instance of PATH WITH FORBIDDEN PAIRS, it is easy to check that what we have constructed is actually a BDD, though it may have redundancy.

Based on our construction, we can now prove that the instance of PATH WITH FORBIDDEN PAIRS has yes answer if and only the constructed BDD is satisfiable under the consistency requirement.

($\Rightarrow$) If there is a path from s to t in G that contains at most one vertex from each pair in C, then corresponding vertices will form a path in BDD, which, when adding $e_{n+1,n+2}$ at the head, forms a path from $e_{n+1,n+2}$ to 1. This path gives an assignment which satisfies the BDD. We need only prove it obeys the consistency requirement. This is trivial because only those vertices appearing in a pair can give trouble but the path contains at most one of them.

($\Leftarrow$) If the BDD is satisfiable under the consistency requirement, then there is a path from $e_{n+1,n+2}$ to 1. It corresponds to a path in G from s to t. This path can only contains at most one vertex from each pair. Otherwise, the assignment will make $e_{i,n+1} = 1$, $e_{i,n+2} = 1$ and $e_{n+1,n+2} = 0$, which is contradictory with the fact that the assignment obeys the consistency requirement. □

3.4. *Heuristically finding a consistent minterm*

We now develop a heuristic for solving the BDD ConSAT problem. First, observe that a cube c whose literals are drawn from the set of variables $\mathcal{E} = \{e_{12}, e_{13}, \ldots, e_{(n-1)n}\}$ naturally gives rise to a partial assignment ϵ_c to the variables. For example, the cube $\kappa = e_{12} \cdot e'_{14} \cdot e_{23}$ corresponds to the partial assignment ϵ_κ where $\epsilon_\kappa(e_{12}) = 1$, $\epsilon_\kappa(e_{14}) = 0$, $\epsilon_\kappa(e_{23}) = 1$.

Lemma 3.5. *For any cube c, if the resulting partial variable assignment ϵ_c is consistent, then there is a minterm in the cube which is consistent.*

Proof: The result follows from the following construction: start with the partition of the set $\{1, 2, \ldots, n\}$ into n distinct equivalence classes; recursively merge equivalence classes to which i and j belong if $\epsilon_c(e_{ij}) = 1$. Call the resulting partition P_ϵ. Since ϵ_c is consistent, there cannot be a and b so that a and b lie in the same equivalence class of P_ϵ but $\epsilon_c(e_{ab}) = 0$. Hence the minterm $\hat{\epsilon}$ given by $\hat{\epsilon}(e_{ij}) = 1$ iff i and j lie in the same equivalence class of P_ϵ is consistent; furthermore, it lies in c. □

The proof is constructive, and yields an algorithm for checking cube satisfiability; efficient querying and updating of the partition can be performed by a variant of the union-find algorithm [10]. Thus, a procedure for finding a consistent minterm in a BDD is to iterate over a set of cubes (a "cover") which contains all the minterms in the BDD. Such a cover can be derived from the BDD by recursive application of the Shannon decomposition, starting from the top variable.

The iteration time is potentially exponential in the size of the BDD; the search can be made far more efficient by bounding the search. If cube c_1 contains cube c_2, and c_1 has no consistent assignments, then c_2 has no consistent assignments. When iteratively generating cubes, we prune the search by finding early contradictions; this is the source of a major speedup. This is similar to the procedure of Chan et al. [9] for pruning BDDs over variables corresponding to complex arithmetical constraints. One source of relative efficiency for us is that because we are dealing purely with equality, we can incrementally check inconsistency as we explore the BDD.

Another potential way to prune the search is to identify nodes appearing in the BDD for which the corresponding subfunction rooted at that node has no satisfying assignments; we have not experimented with this.

4. Experiments

We implemented the procedure for constructing the e_{ij}-encoded functions from an IE netlist on top of VIS [7], which is a popular gate-level BDD-based verification tool. (For the finite instantiation approach, there was no code to write, since VIS has the capability of building BDDs for binary netlists.)

In order to perform a comparison of the two symbolic methods for IE netlist satisfiability checking we first created a series of examples. These correspond to verifying processors using commutative diagrams [15]. Specifically, they arise in the verification of a pipelined

processor; the approach taken is that of Burch and Dill, wherein a pipelined processor is flushed after executing one instruction; the resulting state is compared with the state resulting from execution of the same instruction on a nonpipelined implementation. Our examples are derived from the comparison of the pipelined and non-pipelined version of the 3-stage pipelined ALU used in [8]; this design has uninterpreted functions which correspond to the ALU and Reads/Writes to the register file.

Constraints corresponding to the UIFs are added to the designs: for the ALU, each constraint ensures that if the inputs to a pair of ALUs is the same, the outputs with be the same; for Reads/Writes, each constraint ensures that if we read a memory address that has been written to, we will read the same data was that written. The five examples correspond to different number of constraints. The entire set of constraints is not necessary to show that the designs are equivalent; PIPE1, PIPE2, PIPE3 all contain enough constraints to prove equivalence. (We were able to find a minimal set of constraints by starting with no constraints, and iteratively adding constraints to eliminate false negatives.) PIPE3 has more constraints than PIPE2, which in turn has more than PIPE1; this is reflected in the increased computational effort to perform verification. The constraints used in PIPE4, PIPE5 are not enough to prove equivalence, but they do have some superfluous constraints, resulting in higher verification times. A feel for complexity of the designs can be had from the fact that they had approximately 28 inputs, 60 equality blocks, 200 2-input NAND gates, and 40 Mux elements.

Table 1 shows the results we obtained. For both approaches, we report the computational resources expended in verification—memory in the form of peak and final BDD size, and total computation time. These experiments were performed on a Pentium-200 with 64 Mbytes running Linux. The column headed *Sat.?* indicates whether the netlist output was satisfiable. Note that for the finite instantiation approach, the resulting BDD has only one node (the 0 node) when the output is not satisfiable; in contrast, the e_{ij}-encoded function for the output may be nonzero, but it will have no consistent minterms.

It is noteworthy that for the finite instantiation approach, the default BDD variable ordering always resulted in memory overflows; dynamic variable reordering [20] had to be enabled for the process to complete. Even so, the example PIPE5.v exhausted available memory. For the e_{ij} encoding, variables were allocated dynamically and added to the end of the order; no variable re-ordering was needed.

Table 1. Comparing symbolic procedures for equality.

	Finite instantiations			e_{ij} encoding			
Ex.	Max BDD	Final BDD	Time	Max BDD	Final BDD	Time	*Sat.?*
PIPE1	3,932	1	12.5	62	36	0.3	No
PIPE2	42,875	1	137.2	218	146	0.3	No
PIPE3	131,889	1	447.0	536	355	0.4	No
PIPE4	141,016	79,336	590.7	413	376	0.5	Yes
PIPE5	∞	?	∞	1523	1335	0.5	Yes

We observed that the number of BDD variables needed for the e_{ij} encoding was never more than twice the number of inputs and hence substantially smaller than for the finite instantiation approach, which always required $n \cdot \lceil \log(n) \rceil$ Boolean variables (where n is the number of inputs). This may seem surprising, since the e_{ij} encoding could require as many as $n \cdot (n-1)/2$ Boolean variables. However, not all inputs are compared in the design; in fact input comparisons are "sparse". Thus, if variables are only created on demand, the number of variables required is relatively small.

The running time for the e_{ij}-encoded approach includes both the time to build the functions, and to search the output BDD for a consistent minterm; the latter was very fast, taking of the order of tens of milliseconds. The results clearly are in favor of the e_{ij} encoding; hence, we propose it as the method of choice for BDD-based satisfiability checking.

The runtimes are higher than those reported in [8]; this is not surprising given the large over-heads associated with initialization of the data structures we use for design representation. The results demonstrate that BDD methods are feasible, contradicting prevailing beliefs. In the next section, we discuss recent enhancements which make the BDD-based approach competitive with the existing formula-based approaches.

5. Conclusion

In summary, our major contribution is the extension of BDD techniques to the existential fragment of the theory of equality. On the theoretical side, we have developed semantic foundations and addressed complexity issues. Our experiments justify the use of symbolic procedures; encoding each comparison of inputs by a Boolean variable is superior to the direct mapping of inputs to an appropriately-sized vector of Boolean-valued variables.

There are many ways in which this work can be extended. Perhaps the most important is the incorporation of the "miter" concept for identifying equivalent nodes; this has been extremely successful in the Boolean verification world [16], enabling the verification of million gate circuits. We are currently working on incorporating other interpreted functions and relations, such as addition and inequality; this is motivated by the observation that the abstraction of designs to UIFs with equality is too "coarse" for certain applications (e.g., replacing increment circuitry for a program counter by a UIF may result in false negatives). It may be possible to get by with a simple approximation; for example, certain properties may depend only on the associative and commutative properties of plus.

Recently, Pnueli et al. [19] and Bryant et al. [24] have described extensions to the approach we have put forth in this paper. Pnueli et al. [19] demonstrated that the cardinality of the domain we used in the finite instantiation approach can be reduced for certain easily identifiable classes of formulas, e.g., formulas where the variables can be partitioned into groups which are in some precise sense "independent". We had experimented with reduced domains; however, for our application, namely pipeline verification, it was still extremely inefficient. The application treated in [19] is quite different, namely that of certifying compiler output, for which the functions are not as "deep" as those arising in pipeline verification; this may be the reason their approach succeeded. The approach taken by Bryant et al. [24] is to syntactically identify "p-formulas", which are in a certain sense "monotone". They prove a result to the effect that for this fragment, a formula is true iff it is true for a "maximally

diverse" interpretation; This allows them to dramatically simplify the problem of validity checking. Their procedure has been successfully used to verify designs containing highly nontrivial control logic, e.g., dual issue super scalar processors.

In subsequent work, Bryant et al. [6] demonstrated that the BDD representation of the transitivity constraints for a set $\mathcal{E}$ of e_{ij} variables can be exponential in $|\mathcal{E}|$; in certain respects, this is a stronger result than given in Theorem 3.4. They describe a procedure for minimizing the size of the set of transitivity constraints which need to be considered when testing whether a function of the e_{ij} variables is satisfiable. The number of constraints to be considered is $O(N^3)$ when $\mathcal{E}$ is "dense," i.e., there is an e_{ij} for each $1 \leq i < j \leq N$, but can be exponential when $\mathcal{E}$ is sparse. However, by studying the structure of a graph defined over $\mathcal{E}$ they were able to simultaneously minimize (in a heuristic sense) the number of constraints and the number of variables added to $\mathcal{E}$. This led to dramatic improvements in performance for their examples.

References

1. W. Ackermann, *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. VSI Alliance. Virtual Socket Interface Proposal 1.0. `http://www.vsi.org/`, September 1996.
3. C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality," in *Formal Methods in CAD*, November 1996.
4. R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, Vol. C-35, pp. 677–691, August 1986.
5. R. Bryant and Y.A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Design Automation Conference*, pp. 535–541, June 1995.
6. R. Bryant and M. Velev, "Boolean satisfiability with transitivity constraints," in *Computer Aided Verification*, July 2000.
7. R.K. Brayton et al., "VIS: A system for verification and synthesis," in *Computer Aided Verification*, July 1996.
8. J. Burch and D. Dill, "Automatic verification of microprocessor control," in *Computer Aided Verification*, July 1994.
9. W. Chan, R. Anderson, P. Deame, and D. Notkin, "Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints," in *Computer Aided Verification*, July 1997.
10. T.H. Cormen, C.E. Leiserson, and R.H. Rivest, *Introduction to Algorithms*, MIT Press, 1989.
11. H. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.
12. M.R. Garey and D.S. Johnson, *Computers and Intractability*, W.H. Freeman and Co., 1979.
13. R. Hojati, A. Isles, D. Kirkpatrick, and R. Brayton, "Verification using finite instantiations and uninterpreted functions," in *Formal Methods in CAD*, November 1996.
14. R. Hojati, A. Kuehlmann, S. German, and R. Brayton, "Validity checking in the theory of equality using finite instantiations," in *Proceedings of the International Workshop on Logic Synthesis*, May 1997.
15. R.B. Jones, D. Dill, and J.R. Burch, "Efficient validity checking for processor validation," in *International Conference on Computer-Aided Design*, pp. 2–6, 1995.
16. A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conference*, June 1997.
17. T. Larrabee, "Efficient generation of test patterns using Boolean difference," in *International Test Conference*, pp. 795–801, 1989.
18. C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.
19. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel, "Deciding equality formulas by small-domains instantiations," in *Computer Aided Verification*, July 1999.
20. R. Rudell, "Dynamic variable ordering for binary decision diagrams," in *International Conference on Computer-Aided Design*, pp. 42–47, November 1993.

21. R.E. Shostak, "A practical decision procedure for arithmetic with function symbols," *Journal of the ACM*, Vol. 26, No. 2, pp. 351–360, 1979.
22. J. Silva and K. Sakallah, "GRASP—A new search algorithm for satisfiability," in *International Conference on Computer-Aided Design*, Santa Clara, CA, November 1996.
23. M. Srivas and M. Bickford, "Formal verification of a pipelined microprocessor," *IEEE Software*, Vol. 7, No. 5, pp. 52–64, September 1990.
24. M. Velev and R. Bryant, "Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors," in *Design Automation Conference*, June 1999.