BDD Based Procedures for a Theory of Equality with Uninterpreted Functions

Anuj Goel[1], Khurram Sajid[2], Hai Zhou[1], Adnan Aziz[1], and Vigyan Singhal[3]

[1] The University of Texas at Austin
[2] Intel Corporation
[3] Cadence Berkeley Labs

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 this verification paradigm 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 design. 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.

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 are not capable of routinely verifying designs possessing more than a hundred to two hundred binary-valued latches.

This observation motivates the development of tools which can operate on designs at a higher level of abstraction. Loosely speaking, the basic premise is that abstract designs, being less specified, are simpler and consequently 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 a lot of complexity has been abstracted away. The use of uninterpreted functions (UIFs) has been proposed as a powerful abstraction mechanism for hardware verification [10, 14]. 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 [13].

In this context, the primary verification problem we solve is design equivalence; this includes such applications as verifying equivalence of pipelined and nonpipelined processors. This can be posed as a problem in satisfiability checking for quantifier-free formulas involving both equality and UIFs. 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 $f(x_2)$, 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. The additional complexity of validity checking for the theory of equality over propositional logic arises from the fact that the properties of equality need to be taken into account. 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 [13], 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, 10, 2]. 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 [3].) Equivalence is formulated as in terms of the validity of a quantifier free formula involving both equality and UIFs.

One difference of our work with the work of [2] 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 [8, 9] use finite instantiations to handle UIFs (we also discuss a finite instantiation based method in Section 3.1). In [8], 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; the original tools were based on case splitting (e.g., ATPG-based methods) [11]. Currently, all 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.

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 with `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.

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 $x_1 = x_2$. Figure 1(c) is a more complex design containing complex Boolean gates.

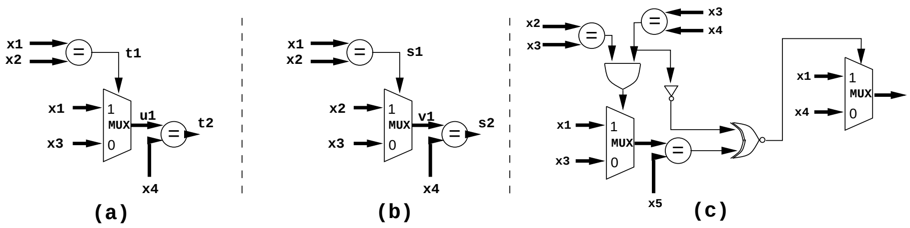

Fig. 1. Design examples.

Definition 1. An *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 w_i. The four basic primitive circuit elements are *primary inputs, multiplexers, equality checkers,* and *2-input NAND gates*. Some nodes are also labeled as being *primary outputs*. If an edge (u, v) exists in the IE netlist, u is said to be a *fanin* of v.

Nodes will be of two *types* — Boolean-valued and integer-valued. Nodes corresponding to primary inputs and multiplexers are integer typed, and nodes corresponding to equality checkers and 2-input NAND gates are Boolean. Multiplexers are required to have a single Boolean-valued input, and two integer-valued inputs; equality checkers should have two integer-valued inputs. A 2-input NAND gate has two Boolean-valued inputs.

Note that 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.

For an IE netlist, given an *input* (i.e., a function mapping primary input nodes to integer values), one can uniquely compute the values of each node in the IE netlist by evaluating the functions at gates in topological order, starting at the primary inputs. More precisely, let ι be an input; then ι uniquely defines a value $\nu_\iota(s)$ to the signal s in the IE netlist through the following recursive rules:

Definition 2. IE Netlist Semantics

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

In this way, a IE netlist D on inputs $a_1, a_2, \ldots, a_n$ and outputs $b_1, b_2, \ldots, b_m$ *defines* a function $f_D : \omega^n \to \omega^m$ (here $\omega = \{0, 1, 2, \ldots\}$ is the set of natural numbers). Intuitively, two designs are functionally equivalent if in any environment they can be used interchangeably; a necessary and sufficient condition for this is for them to have identical defined functions. Note that an IE netlist can operate on arbitrary inputs and not just integers, since no operation other than equality is applied to the integer-valued nodes.

Observe that for a primary input assignment ι, the value taken by any integer-valued node in the IE netlist will be the value taken by some primary input. This is because there are no functions which can be applied to the integers propagated in the IE netlist; integers can only be compared. Indeed, a stronger claim can be asserted — the value taken by the node can be traced back to a *specific* primary input which caused it. The proof of the claim is by an inductive argument starting at the PIs, where it vacuously holds. Any other integer-valued node must be the output of a multiplexer; the result follows by applying induction to the mux inputs.

We'll define the input x_i to *flow* to s under the input assignment ι when the value taken by s under ι is traced back to x_i. For example, the input x_1 flows to

u_1 for the design in Figure 1(a) under the input $x_1 = 2, x_2 = 2, x_3 = 3, x_4 = 4$; 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.

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 variables; design inequivalence can then be cast as the satisfiability of a quantifier-free formula involving equality and UIFs. IE netlists can not directly represent UIFs; however, the outputs of the UIF blocks can be replaced by new primary 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 [13], 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 primary output, and so the equivalence problem for IE netlists can be easily reduced to the IE netlist satisfiability checking.

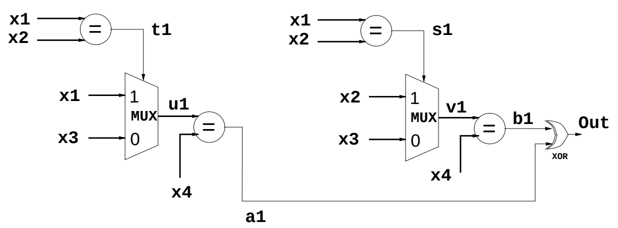

Fig. 2. Product construction for equivalence checking.

It is natural to ask at this point if there is a decision procedure for IE netlist satisfiability checking, and if so, what the computational complexity of the problem is.

3.1 Finite Model Approach

The existence of a decision procedure follows immediately from the fact that a "finite model" folk-theorem holds for 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.

Reduction to Combinational 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 the procedure for reducing Boolean-valued netlist satisfiability to satisfiability of a CNF formula from propositional logic. Hence the integer valued variables can be replaced by n-valued variables, which in turn can be encoded in $\lceil \log(n) \rceil$ Boolean-valued variables. Thus the satisfiability problem can be efficiently (polytime) transformed to a problem of checking the satisfiability of Boolean-valued netlists.

3.2 A Better Encoding

In this section we develop a superior encoding of IE netlist satisfiability instances into Boolean netlist satisfiability. We introduce a minimal set of Boolean variables — one for each distinct comparison which is made between primary inputs. We will show that the design 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 $1 \leq i < j \leq n$. For a Boolean-valued node s in D, we will construct a Boolean function f^s over the set of variables $\{e_{12}, e_{13}, \ldots, e_{23}, e_{24}, \ldots, e_{1(n-1)}, e_{(n-1)n}\}$; for an integer-valued node t, we will construct a vector of n Boolean-valued functions $[f_1^t \, f_2^t \, \ldots \, f_n^t]$ over the same set of variables.

Intuitively, the variables e_{ij}'s indicate whether the i-th and j-th integer inputs are equal or not. For a Boolean node, the function f at the node is a Boolean function of these indicator variables, and it represents the condition under which the circuit node evaluates to 1. For an integer node, such as a mux, the k-th component of the n-tuple function f represents the condition under which the circuit node assumes the value of the k-th integer input. Note the distinction between the primary input that flows to s under ι, and the value $\nu_\iota(s)$; for the input ι, it may be the case that $\nu_\iota(s)$ is equal to the value taken by more than one primary inputs, but there will still be a unique input x_k which flows.

Definition 3. e_{ij} **Encoded Functions**

1. If s is a primary input, say x_k, then $f_k^s = 1$ 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 c, and data inputs u, v, then $f_k^s = f^c \cdot f_k^u + (f^c)' \cdot f_k^v$.
4. If s is the output of an equality node with inputs u, and v then

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

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

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

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

Encoding the network using these e_{ij}'s allows us to directly store the relationship between the function nodes and the equality of pairs of inputs. For many validity checking applications, it is the equality of intermediate circuit functions which is exploited in simplifying or complicating (by pipeline bypass logic, for example, in a pipelined implementation) logic circuitry. Encoding the equality by pairwise variables allows us to represent relationship between equalities directly by having single BDD variables for each of these e_{ij} variables. Of course, as we see later in this section, to prevent false negatives, we will need to introduce procedures that ensure the transitivity of equality.

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

Lemma 1 (Completeness). Let ι be an input and s a node in the design. Let ϵ be the extension of ι to the e_{ij} variables, i.e., $\epsilon(e_{ij}) = 1$ exactly when $\iota(x_i) = \iota(x_j)$. Then if s is Boolean-valued, $f^s(\epsilon) = \nu^\iota(s)$; if s is integer-valued, then $f_k^s(\epsilon) = 1$ exactly when x_k flows to s under ι.

The proof follows by an easy induction on the depth of the node from the primary inputs.

The functions computed above are not "sound"; values taken by them may not be achievable in the design. This is because the there is no guarantee that the basic axioms of 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 results in ι 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 4. An assignment ϵ to the e_{ij} variables is said to be *consistent* if it satisfies $\bigwedge_{1 \le i < j < k \le n} [\epsilon(e_{ij}) \cdot \epsilon(e_{jk}) \rightarrow \epsilon(e_{ik})]$.

Intuitively, a consistent assignment is one which satisfies the transitivity of equality; for consistent assignments, the converse of Lemma 1 holds:

Lemma 2 (Soundness). Let ϵ be a consistent assignment to the e_{ij} variables. Let s be a node in the design. If s is Boolean-valued, there is an input ι so that $f^s(\epsilon) = \nu^\iota(s)$; if s is integer-valued and $f_k^s(\epsilon) = 1$ then there is an input ι so that the input x_k flows to s under ι.

The proof is based on the fact that ϵ yields an equivalence relation on the primary inputs, from which the desired input can be constructed.

3.3 Satisfiability using the e_{ij} encoding

It follows from these two lemmas that the functions in Definition 3 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: $\bigwedge_{1 \leq i<j<k \leq n} [\epsilon(e_{ij}) \cdot \epsilon(e_{jk}) \rightarrow \epsilon(e_{ik})]$

Theorem 1. *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 [7] to it.

INSTANCE: Directed graph $G = (V, A)$, specified vertices $s, t \in V$, collection $C = \{(a_1, b_1), \dots, (a_n, b_n)\}$ 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- and 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 by v_1 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 $e_{n+1,n+2}$ and 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 let each vertex whose out-degree is 1 have another edge entering it. The unlabeled edges will be labeled 1 or 0 arbitrarily but under the condition that the out-edges of any vertex are labeled one 1 and one 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. However, it might have redundancy and can be reduced. But based on the fact that each vertex appearing in the pairs has exactly one out-edge, these vertices will not be inferenced.

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 consistent 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 consistent 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 consistent requirement, then there is a path goes 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 $e_{i,n+1} = 1, e_{i,n+2} = 1$ but $e_{n+1,n+2} = 0$, which is contradictory with the fact that the assignment obeys the consistent 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 $e_{12}, e_{13}, \ldots, e_{(n-1)n}$ 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. 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 can not 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 [5]. 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 [4] 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 [6], 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 [10]. 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 [3]; 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 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 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 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 *Satisfiable* 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; the e_{ij}-encoded function for the output is nonempty, but has no consistent minterms.

It is noteworthy that for the finite instantiation approach, the default BDD variable ordering would always result in memory overflows; dynamic variable re-ordering [12] had to be enabled for the process to complete. Even so, the example PIPE5.V would exhaust available memory. For the equality based approach, variables are allocated dynamically, and added to the end of the order; no variable re-ordering was needed.

We observed that the number of BDD variables needed for the e_{ij} encoded function approach was never more than twice the number of inputs and hence substantially smaller than for the finite instantiation approach, which always requires $n \cdot \lceil \log(n) \rceil$ Boolean variables (where n is the number of inputs). This is surprising, since the e_{ij} encoded approach may need as many as $n \cdot (n-1)/2$ Boolean variables. However, not all inputs are compared in the design; input comparisons are "sparse". We create variables on demand, resulting in the saving.

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 [3]; this is not surprising given the large overheads 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 point out an enhancement which we believe should make the BDD based approach highly competitive with the existing formula-based approaches.

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

Table 1. Comparing Symbolic Procedures for Equality.

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 [11], enabling the verification of million gate circuits. We are developing a specification language for designs with UIFs, a data structure for representing the same, and a set of routines for restructuring and verifying the design; this will be made available to the general public.

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.

References

1. Wilhelm Ackermann. *Solvable Cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. C. Barrett, D. Dill, and Jeremy Levitt. Validity Checking for Combinations of Theories with Equality. In *Proc. of the Formal Methods in CAD Conf.*, November 1996.
3. J. Burch and D. Dill. Automatic Verification of Microprocessor Control. In *Proc. of the Computer Aided Verification Conf.*, July 1994.
4. 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 *Proc. of the Computer Aided Verification Conf.*, July 1997.
5. T. H. Cormen, C. E. Leiserson, and R. H. Rivest. *Introduction to Algorithms.* MIT Press, 1989.
6. R. K. Brayton et al. VIS: A system for Verification and Synthesis. In *Proc. of the Computer Aided Verification Conf.*, July 1996.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and Co., 1979.
8. R. Hojati, A. Isles, D. Kirkpatrick, and R. Brayton. Verification Using Finite Instantiations and Uninterpreted Functions. In *Proc. of the Formal Methods in CAD Conf.*, November 1996.
9. R. Hojati, A. Kuehlmann, S. German, and R. Brayton. Validity Checking in the Theory of Equality Using Finite Instantiations. In *Proc. Intl. Workshop on Logic Synthesis*, May 1997.
10. Robert B. Jones, David Dill, and Jerry R. Burch. Efficient Validity Checking for Processor Validation. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 2–6, 1995.
11. Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In *Proc. of the Design Automation Conf.*, June 1997.
12. R. Rudell. Dynamic Variable Ordering for Binary Decision Diagrams. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 42–47, November 1993.
13. R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.
14. Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.