Boolean Satisfiability with Transitivity Constraints

Randal E. Bryant
Miroslav N. Velev
Carnegie Mellon University

June, 2000

CMU-CS-00-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Supported, in part, by the Semiconductor Research Corporation under contract 00-DC-684

Keywords: Formal verification, Boolean satisfiability, Decision procedures

Abstract

We consider a variant of the Boolean satisfiability problem where a subset $\mathcal{E}$ of the propositional variables appearing in formula F_{sat} encode a symmetric, transitive, binary relation over N elements. Each of these *relational* variables, $e_{i,j}$, for $1 \leq i < j \leq N$, expresses whether or not the relation holds between elements i and j. The task is to either find a satisfying assignment to F_{sat} that also satisfies all transitivity constraints over the relational variables (e.g., $e_{1,2} \wedge e_{2,3} \Rightarrow e_{1,3}$), or to prove that no such assignment exists. Solving this satisfiability problem is the final and most difficult step in our decision procedure for a logic of equality with uninterpreted functions. This procedure forms the core of our tool for verifying pipelined microprocessors.

To use a conventional Boolean satisfiability checker, we augment the set of clauses expressing F_{sat} with clauses expressing the transitivity constraints. We consider methods to reduce the number of such clauses based on the sparse structure of the relational variables.

To use Ordered Binary Decision Diagrams (OBDDs), we show that for some sets $\mathcal{E}$, the OBDD representation of the transitivity constraints has exponential size for all possible variable orderings. By considering only those relational variables that occur in the OBDD representation of F_{sat}, our experiments show that we can readily construct an OBDD representation of the relevant transitivity constraints and thus solve the constrained satisfiability problem.

1 Introduction

Consider the following variant of the Boolean satisfiability problem. We are given a Boolean formula F_{sat} over a set of variables $\mathcal{V}$. A subset $\mathcal{E} \subseteq \mathcal{V}$ symbolically encodes a binary relation over N elements that is reflexive, symmetric, and transitive. Each of these *relational* variables, $e_{i,j}$, where $1 \leq i < j \leq N$, expresses whether or not the relation holds between elements i and j. Typically, $\mathcal{E}$ will be "sparse," containing much fewer than the $N(N-1)/2$ possible variables. Note that when $e_{i,j} \notin \mathcal{E}$ for some value of i and of j, this does not imply that the relation does not hold between elements i and j. It simply indicates that F_{sat} does not directly depend on the relation between elements i and j.

A *transitivity constraint* is a formula of the form

$$e_{[i_1,i_2]} \wedge e_{[i_2,i_3]} \wedge \cdots \wedge e_{[i_{k-1},i_k]} \quad \Rightarrow \quad e_{[i_1,i_k]} \tag{1}$$

where $e_{[i,j]}$ equals $e_{i,j}$ when $i < j$ and equals $e_{j,i}$ when $i > j$. Let $Trans(\mathcal{E})$ denote the set of all transitivity constraints that can be formed from the relational variables. Our task is to find an assignment $\chi\colon \mathcal{V} \rightarrow \{0,1\}$ that satisfies F_{sat}, as well as every constraint in $Trans(\mathcal{E})$. Goel, *et al.* [GSZAS98] have shown this problem is NP-hard, even when F_{sat} is given as an Ordered Binary Decision Diagram (OBDD) [Bry86]. Normally, Boolean satisfiability is trivial given an OBDD representation of a formula.

We are motivated to solve this problem as part of a tool for verifying pipelined microprocessors [VB99]. Our tool abstracts the operation of the datapath as a set of uninterpreted functions and uninterpreted predicates operating on symbolic data. We prove that a pipelined processor has behavior matching that of an unpipelined reference model using the symbolic flushing technique developed by Burch and Dill [BD94]. The major computational task is to decide the validity of a formula F_{ver} in a logic of equality with uninterpreted functions [BGV99a, BGV99b]. Our decision procedure transforms F_{ver} first by replacing all function application terms with terms over a set of domain variables $\{v_i | 1 \leq i \leq N\}$. Similarly, all predicate applications are replaced by formulas over a set of newly-generated propositional variables. The result is a formula F^*_{ver} containing equations of the form $v_i = v_j$, where $1 \leq i < j \leq N$. Each of these equations is then encoded by introducing a relational variable $e_{i,j}$, similar to the method proposed by Goel, *et al.* [GSZAS98]. The result of the translation is a propositional formula $encf(F^*_{\text{ver}})$ expressing the verification condition over both the relational variables and the propositional variables appearing in F^*_{ver}. Let F_{sat} denote $\neg encf(F^*_{\text{ver}})$, the complement of the formula expressing the translated verification condition. To capture the transitivity of equality, e.g., that $v_i = v_j \wedge v_j = v_k \Rightarrow v_i = v_k$, we have transitivity constraints of the form $e_{[i,j]} \wedge e_{[j,k]} \Rightarrow e_{[i,k]}$. Finding a satisfying assignment to F_{sat} that also satisfies the transitivity constraints will give us a counterexample to the original verification condition F_{ver}. On the other hand, if we can prove that there are no such assignments, then we have proved that F_{ver} is universally valid.

We consider three methods to generate a Boolean formula F_{trans} that encodes the transitivity constraints. The *direct* method enumerates the set of *chord-free* cycles in the undirected graph having an edge (i, j) for each relational variable $e_{i,j} \in \mathcal{E}$. This method avoids introducing additional relational variables but can lead to a formula of exponential size. The *dense* method uses relational variables $e_{i,j}$ for all possible values of i and j such that $1 \leq i < j \leq N$. We can then axiomatize transitivity by forming constraints of the form $e_{[i,j]} \wedge e_{[j,k]} \Rightarrow e_{[i,k]}$ for all distinct values of i, j, and k. This will yield a formula that is cubic in N. The *sparse* method augments $\mathcal{E}$ with additional relational variables to form a set of variables $\mathcal{E}^+$, such that the resulting graph is *chordal* [Rose70]. We then only require transitivity constraints of the form $e_{[i,j]} \wedge e_{[j,k]} \Rightarrow e_{[i,k]}$ such that $e_{[i,j]}, e_{[j,k]}, e_{[i,k]} \in \mathcal{E}^+$. The sparse method is guaranteed to generate a smaller formula than the dense method.

To use a conventional Boolean Satisfiability (SAT) procedure to solve our constrained satisfiability problem, we run the checker over a set of clauses encoding both F_{sat} and F_{trans}. The latest version of the FGRASP SAT checker [M99] was able to complete all of our benchmarks, although the run times increase significantly when transitivity constraints are enforced.

When using Ordered Binary Decision Diagrams to evaluate satisfiability, we could generate OBDD representations of F_{sat} and F_{trans} and use the APPLY algorithm to compute an OBDD representation of their conjunction. From this OBDD, finding satisfying solutions would be trivial. We show that this approach will not be feasible in general, because the OBDD representation of F_{trans} can be intractable. That is, for some sets of relational variables, the OBDD representation of the transitivity constraint formula F_{trans} will be of exponential size regardless of the variable ordering. The NP-completeness result of Goel, *et al.* shows that the OBDD representation of F_{trans} may be of exponential size using the ordering previously selected for representing F_{sat} as an OBDD. This leaves open the possibility that there could be some other variable ordering that would yield efficient OBDD representations of both F_{sat} and F_{trans}. Our result shows that transitivity constraints can be intrinsically intractable to represent with OBDDs, independent of the structure of F_{sat}.

We present experimental results on the complexity of constructing OBDDs for the transitivity constraints that arise in actual microprocessor verification. Our results show that the OBDDs can indeed be quite large. We consider two techniques to avoid constructing the OBDD representation of all transitivity constraints. The first of these, proposed by Goel, *et al.* [GSZAS98], generates implicants (cubes) of F_{sat} and rejects those that violate the transitivity constraints. Although this method suffices for small benchmarks, we find that the number of implicants generated for our larger benchmarks grows unacceptably large. The second method determines which relational variables actually occur in the OBDD representation of F_{sat}. We can then apply one of our three techniques for encoding the transitivity constraints in order to generate a Boolean formula for the transitivity constraints over this reduced set of relational variables. The OBDD representation of this formula is generally tractable, even for the larger benchmarks.

2 Benchmarks

Our benchmarks [VB99] are based on applying our verifier to a set of high-level microprocessor designs. Each is based on the DLX RISC processor described by Hennessy and Patterson [HP96]:

1×DLX-C: is a single-issue, five-stage pipeline capable of fetching up to one new instruction every clock cycle. It implements six instruction types: register-register, register-immediate,

Circuit		Domain Variables	Propositional Variables	Equations
1×DLX-C		13	42	27
1×DLX-C-t		13	42	37
2×DLX-CA		25	58	118
2×DLX-CA-t		25	58	137
2×DLX-CC		25	70	124
2×DLX-CC-t		25	70	143
Buggy	min.	22	56	89
2×DLX-CC	avg.	25	69	124
	max.	25	77	132

Table 1: **Microprocessor Verification Benchmarks.** Benchmarks with suffix "t" were modified to require enforcing transitivity.

load, store, branch, and jump. The pipeline stages are: Fetch, Decode, Execute, Memory, and Write-Back. An interlock causes the instruction following a load to stall one cycle if it requires the loaded result. Branches and jumps are predicted as not-taken, with up to 3 instructions squashed when there is a misprediction. This example is comparable to the DLX example first verified by Burch and Dill [BD94].

2×DLX-CA: has a complete first pipeline, capable of executing the six instruction types, and a second pipeline capable of executing arithmetic instructions. Between 0 and 2 new instructions are issued on each cycle, depending on their types and source registers, as well as the types and destination registers of the preceding instructions. This example is comparable to one verified by Burch [Bur96].

2×DLX-CC: has two complete pipelines, i.e., each can execute any of the six instruction types. There are four load interlocks—between a load in Execute in either pipeline and an instruction in Decode in either pipeline. On each cycle, between 0 and 2 instructions can be issued.

In all of these examples, the domain variables v_i, with $1 \leq i \leq N$, in F^*_{ver} encode register identifiers. As described in [BGV99a, BGV99b], we can encode the symbolic terms representing program data and addresses as distinct values, avoiding the need to have equations among these variables. Equations arise in modeling the read and write operations of the register file, the bypass logic implementing data forwarding, the load interlocks, and the pipeline issue logic.

Our original processor benchmarks can be verified without enforcing any transitivity constraints. The unconstrained formula F_{sat} is unsatisfiable in every case. We are nonetheless motivated to study the problem of constrained satisfiability for two reasons. First, other processor designs might rely on transitivity, e.g., due to more sophisticated issue logic. Second, to aid designers in debugging their pipelines, it is essential that we generate counterexamples that satisfy all transitivity constraints. Otherwise the designer will be unable to determine whether the counterexample represents a true bug or a weakness of our verifier.

To create more challenging benchmarks, we generated variants of the circuits that require enforcing transitivity in the verification. For example, the normal forwarding logic in the Execute stage of 1×DLX-C must determine whether to forward the result from the Memory stage instruction as either one or both operand(s) for the Execute stage instruction. It does this by comparing the two source registers `ESrc1` and `ESrc2` of the instruction in the Execute stage to the destination register `MDest` of the instruction in the memory stage. In the modified circuit, we changed the bypass condition $\texttt{ESrc1} = \texttt{MDest}$ to be $\texttt{ESrc1} = \texttt{MDest} \vee (\texttt{ESrc1} = \texttt{ESrc2} \wedge \texttt{ESrc2} = \texttt{MDest})$. Given transitivity, these two expressions are equivalent. For each pipeline, we introduced four such modifications to the forwarding logic, with different combinations of source and destination registers. These modified circuits are named 1×DLX-C-t, 2×DLX-CA-t, and 2×DLX-CC-t.

To study the problem of counterexample generation for buggy circuits, we generated 105 variants of 2×DLX-CC, each containing a small modification to the control logic. Of these, 5 were found to be functionally correct, e.g., because the modification caused the processor to stall unnecessarily, yielding a total of 100 benchmark circuits for counterexample generation.

Table 1 gives some statistics for the benchmarks. The number of domain variables N ranges between 13 and 25, while the number of equations ranges between 27 and 143. The verification condition formulas F^*_{ver} also contain between 42 and 77 propositional variables expressing the operation of the control logic. These variables plus the relational variables comprise the set of variables $\mathcal{V}$ in the propositional formula F_{sat}. The circuits with modifications that require enforcing transitivity yield formulas containing up to 19 additional equations. The final three lines summarize the complexity of the 100 buggy variants of 2×DLX-CC. We apply a number of simplifications during the generation of formula F_{sat}, and hence small changes in the circuit can yield significant variations in the formula complexity.

3 Graph Formulation

Our definition of $Trans(\mathcal{E})$ (Equation 1) places no restrictions on the length or form of the transitivity constraints, and hence there can be an infinite number. We show that we can construct a graph representation of the relational variables and identify a reduced set of transitivity constraints that, when satisfied, guarantees that all possible transitivity constraints are satisfied. By introducing more relational variables, we can alter this graph structure, further reducing the number of transitivity constraints that must be considered.

For variable set $\mathcal{E}$, define the undirected graph $G(\mathcal{E})$ as containing a vertex i for $1 \leq i \leq N$, and an edge (i, j) for each variable $e_{i,j} \in \mathcal{E}$. For an assignment χ of Boolean values to the relational variables, define the labeled graph $G(\mathcal{E}, \chi)$ to be the graph $G(\mathcal{E})$ with each edge (i, j) labeled as a *1-edge* when $\chi(e_{i,j}) = 1$, and as a *0-edge* when $\chi(e_{i,j}) = 0$.

A *path* is a sequence of vertices $[i_1, i_2, \ldots, i_k]$ having edges between successive elements. That is, each element i_p of the sequence ($1 \leq p \leq k$) denotes a vertex: $1 \leq i_p \leq N$, while each successive pair of elements i_p and i_{p+1} ($1 \leq p < k$) forms an edge (i_p, i_{p+1}) We consider each edge (i_p, i_{p+1}) for $1 \leq p < k$ to also be part of the path. A *cycle* is a path of the form $[i_1, i_2, \ldots, i_k, i_1]$.

Proposition 1 *An assignment χ to the variables in $\mathcal{E}$ violates transitivity if and only if some cycle in $G(\mathcal{E}, \chi)$ contains exactly one 0-edge.*

Proof: If. Suppose there is such a cycle. Letting i_1 be the vertex at one end of the 0-edge, we can trace around the cycle, giving a sequence of vertices $[i_1, i_2, \ldots, i_k]$, where i_k is the vertex at the other end of the 0-edge. The assignment has $\chi(e_{[i_j,i_{j+1}]}) = 1$ for $1 \leq j < k$, and $\chi(e_{[i_1,i_k]} = 0)$, and hence it violates Equation 1.

Only If. Suppose the assignment violates a transitivity constraint given by Equation 1. Then, we construct a cycle $[i_1, i_2, \ldots, i_k, i_1]$ of vertices such that only edge (i_k, i_1) is a 0-edge. □

A path $[i_1, i_2, \ldots, i_k]$ is said to be *acyclic* when $i_p \neq i_q$ for all $1 \leq p < q \leq k$. A cycle $[i_1, i_2, \ldots, i_k, i_1]$ is said to be *simple* when its prefix $[i_1, i_2, \ldots, i_k]$ is acyclic.

Proposition 2 *An assignment χ to the variables in $\mathcal{E}$ violates transitivity if and only if some* simple *cycle in $G(\mathcal{E}, \chi)$ contains exactly one 0-edge.*

Proof: The "if" portion of this proof is covered by Proposition 1. The "only if" portion is proved by induction on the number of variables in the antecedent of the transitivity constraint (Equation 1.) That is, assume a transitivity constraint containing k variables in the antecedent is violated and that all other violated constraints have at least k variables in their antecedents. If there are no values p and q such that $1 \leq p < q \leq k$ and $i_p = i_q$, then the cycle $[i_1, i_2, \ldots i_k, i_1]$ is simple. If such values p and q exist, then we can form a transitivity constraint:

$$e_{[i_1,i_2]} \wedge \cdots \wedge e_{[i_{p-1},i_p]} \wedge e_{[i_q,i_{q+1}]} \wedge \cdots \wedge e_{[i_{k-1},i_k]} \quad \Rightarrow \quad e_{[i_1,i_k]}$$

This transitivity constraint contains fewer than k variables in the antecedent, but it is also violated. This contradicts our assumption that there is no violated transitivity constraint with fewer than k variables in the antecedent. □

Define a *chord* of a simple cycle to be an edge that connects two vertices that are not adjacent in the cycle. More precisely, for a simple cycle $[i_1, i_2, \ldots, i_k, i_1]$, a chord is an edge (i_p, i_q) in $G(\mathcal{E})$ such that $1 \leq p < q \leq k$, with $p + 1 < q$, and either $p \neq 1$ or $q \neq k$. A cycle is said to be *chord-free* if it is simple and has no chords.

Proposition 3 *An assignment χ to the variables in $\mathcal{E}$ violates transitivity if and only if some* chord-free *cycle in $G(\mathcal{E}, \chi)$ contains exactly one 0-edge.*

Proof: The "if" portion of this proof is covered by Proposition 1. The "only if" portion is proved by induction on the number of variables in the antecedent of the transitivity constraint (Equation 1.) Assume a transitivity constraint with k variables is violated, and that no transitivity constraint with fewer variables in the antecedent is violated. If there are no values of p and q such that there is a variable $e_{[i_p,i_q]} \in \mathcal{E}$ with $p+1 < q$ and either $p \neq 1$ or $q \neq k$, then the corresponding cycle is chord-free. If such values of p and q exist, then consider the two cases illustrated in Figure 1, where 0-edges are shown as dashed lines, 1-edges are shown as solid lines, and the wavy lines

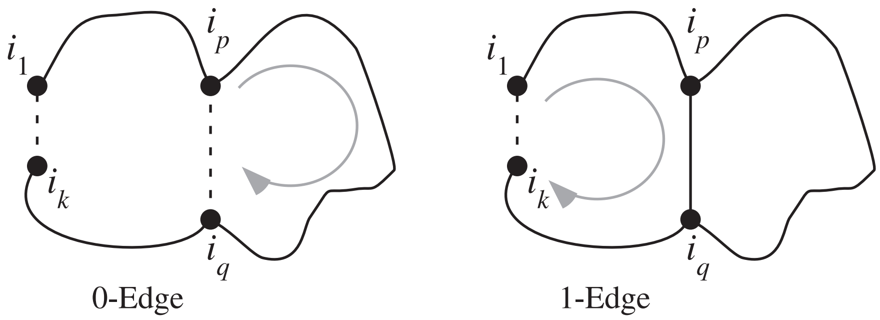


Figure 1: **Case Analysis for Proposition 3.** 0-Edges are shown as dashed lines. When a cycle representing a transitivity violation contains a chord, we can find a smaller cycle that also represents a transitivity violation.

represent sequences of 1-edges. Case 1: Edge (i_p, i_q) is a 0-edge (shown on the left). Then the transitivity constraint:

$$e_{[i_p,i_{p+1}]} \wedge \cdots \wedge e_{[i_{q-1},i_q]} \quad \Rightarrow \quad e_{[i_p,i_q]}$$

is violated and has fewer than k variables in its antecedent. Case 2: Edge (i_p, i_q) is a 1-edge (shown on the right). Then the transitivity constraint:

$$e_{[i_1,i_2]} \wedge \cdots \wedge e_{[i_{p-1},i_p]} \wedge e_{[i_p,i_q]} \wedge e_{[i_q,i_{q+1}]} \wedge \cdots \wedge e_{[i_{k-1},i_k]} \quad \Rightarrow \quad e_{[i_1,i_k]}$$

is violated and has fewer than k variables. Both cases contradict our assumption that there is no violated transitivity constraint with fewer than k variables in the antecedent. □

Each length k cycle $[i_1, i_2, \ldots, i_k, i_1]$ yields k constraints, given by the following clauses. Each clause is derived by expressing Equation 1 as a disjunction.

$$\begin{array}{l}
\neg e_{[i_1,i_2]} \vee \cdots \vee \neg e_{[i_{k-1},i_k]} \vee e_{[i_k,i_1]} \\
\neg e_{[i_2,i_3]} \vee \cdots \vee \neg e_{[i_{k-1},i_k]} \vee \neg e_{[i_k,i_1]} \vee e_{[i_1,i_2]} \\
\quad \cdots \\
\neg e_{[i_k,i_1]} \vee \neg e_{[i_1,i_2]} \vee \cdots \vee \neg e_{[i_{k-2},i_{k-1}]} \vee e_{[i_{k-1},i_k]}
\end{array} \qquad (2)$$

For a set of relational variables $\mathcal{E}$, we define $F_{\text{trans}}(\mathcal{E})$ to be the conjunction of all transitivity constraints for all chord-free cycles in the graph $G(\mathcal{E})$.

Theorem 1 *An assignment to the relational variables $\mathcal{E}$ will satisfy all of the transitivity constraints given by Equation 1 if and only if it satisfies $F_{\textit{trans}}(\mathcal{E})$.*

This theorem follows directly from Proposition 3 and the encoding given by Equation 2.

3.1 Enumerating Chord-Free Cycles

To enumerate the chord-free cycles of a graph, we exploit the following properties. An acyclic path $[i_1, i_2, \ldots, i_k]$ is said to have a chord when there is an edge (i_p, i_q) in $G(\mathcal{E})$ such that $1 \leq p < q \leq k$

with $p + 1 < q$, and either $p \neq 1$ or $q \neq k$. We classify a chord-free path as *terminal* when (i_k, i_1) is in $G(\mathcal{E})$, and as *extensible* otherwise.

Proposition 4 *A path $[i_1, i_2, \ldots, i_k]$ is chord-free and terminal if and only if the cycle $[i_1, i_2, \ldots, i_k, i_1]$ is chord-free.*

This follows by noting that the conditions imposed on a chord-free path are identical to those for a chord-free cycle, except that the latter includes a closing edge (i_k, i_1).

A *proper prefix* of path $[i_1, i_2, \ldots, i_k]$ is a path $[i_1, i_2, \ldots, i_j]$ such that $1 \leq j < k$.

Proposition 5 *Every proper prefix of a chord-free path is chord-free and extensible.*

Clearly, any prefix of a chord-free path is also chord-free. If some prefix $[i_1, i_2, \ldots, i_j]$ with $j < k$ were terminal, then any attempt to add the edge (i_j, i_{j+1}) would yield either a simple cycle (when $i_{j+1} = i_1$), some other cycle (when $i_{j+1} = i_p$ for some $1 < p < j$), or a path having (i_1, i_j) as a chord.

Given these properties, we can enumerate the set of all chord-free paths by breadth first expansion. As we enumerate these paths, we also generate C, the set of all chord-free cycles. Define P_k to be the set of all extensible, chord-free paths having k vertices, for $1 \leq k \leq N$.

Initially we have $P_1 = \{[i] | 1 \leq i \leq n\}$, and $C = \emptyset$. Given set P_k, we generate set P_{k+1} and add some cycles of length $k + 1$ to C. For each path $[i_1, i_2, \ldots, i_k] \in P_k$, we consider the path $[i_1, i_2, \ldots, i_k, i_{k+1}]$ for each edge (i_k, i_{k+1}) in $G(\mathcal{E})$. When $i_{k+1} = i_p$ for some $1 \leq p < k$, we classify the path as cyclic. When there is an edge (i_{k+1}, i_p) in $G(\mathcal{E})$ for some $1 < p < k$, we classify the path as having a chord. When there is an edge (i_{k+1}, i_1) in $G(\mathcal{E})$, we add the cycle $[i_1, i_2, \ldots, i_k, i_{k+1}, i_1]$ to C. Otherwise, we add the path to P_{k+1}.

After generating all of these paths, we can use the set C to generate the set of all chord-free cycles. For each terminal, chord-free cycle having k vertices, there will be $2k$ members of C—each of the k edges of the cycle can serve as the closing edge, and a cycle can traverse the closing edge in either direction. To generate the set of clauses given by Equation 2, we simply need to choose one element of C for each closing edge, e.g., by considering only cycles $[i_1, \ldots, i_k, i_1]$ for which $i_1 < i_k$.

As Figure 2 indicates, there can be an exponential number of chord-free cycles in a graph. In particular, this figure illustrates a family of graphs with $3n + 1$ vertices. Consider the cycles passing through the n diamond-shaped faces as well as the edge along the bottom. For each diamond-shaped face F_i, a cycle can pass through either the upper vertex or the lower vertex. Thus there are 2^n such cycles. In addition, the edges forming the perimeter of each face F_i create a chord-free cycle, giving a total of $2^n + n$ chord-free cycles.

The columns labeled "Direct" in Table 2 show results for enumerating the chord-free cycles for our benchmarks. For each correct microprocessor, we have two graphs: one for which transitivity constraints played no role in the verification, and one (indicated with a "t" at the end of the name) modified to require enforcing transitivity constraints. We summarize the results for the transitivity

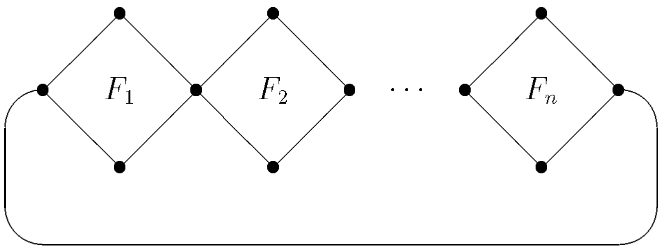


Figure 2: **Class of Graphs with Many Chord-Free Cycles.** For a graph with n diamond-shaped faces, there are $2^n + n$ chord-free cycles.

Circuit		Direct			Dense			Sparse		
		Edges	Cycles	Clauses	Edges	Cycles	Clauses	Edges	Cycles	Clauses
1×DLX-C		27	90	360	78	286	858	33	40	120
1×DLX-C-t		37	95	348	78	286	858	42	68	204
2×DLX-CA		118	2,393	9,572	300	2,300	6,900	172	697	2,091
2×DLX-CA-t		137	1,974	7,944	300	2,300	6,900	178	695	2,085
2×DLX-CC		124	2,567	10,268	300	2,300	6,900	182	746	2,238
2×DLX-CC-t		143	2,136	8,364	300	2,300	6,900	193	858	2,574
Full	min.	89	1,446	6,360	231	1,540	4,620	132	430	1,290
Buggy	avg.	124	2,562	10,270	300	2,300	6,900	182	750	2,244
2×DLX-CC	max.	132	3,216	12,864	299	2,292	6,877	196	885	2,655
M_4		24	24	192	120	560	1,680	42	44	132
M_5		40	229	3,056	300	2,300	6,900	77	98	294
M_6		60	3,436	61,528	630	7,140	21,420	131	208	624
M_7		84	65,772	1,472,184	1,176	18,424	55,272	206	408	1,224
M_8		112	1,743,247	48,559,844	2,016	41,664	124,992	294	662	1,986

Table 2: **Cycles in Original and Augmented Benchmark Graphs.** Results are given for the three different methods of encoding transitivity constraints.

constraints in our 100 buggy variants of 2×DLX-CC in terms of the minimum, the average, and the maximum of each measurement. We also show results for five synthetic benchmarks consisting of $n \times n$ planar meshes M_n, with n ranging from 4 to 8, where the mesh for $n = 6$ is illustrated in Figure 3. For all of the circuit benchmarks, the number of cycles, although large, appears to be manageable. Moreover, the cycles have at most 4 edges. The synthetic benchmarks, on the other hand, demonstrate the exponential growth predicted as worst case behavior. The number of cycles grows quickly as the meshes grow larger. Furthermore, the cycles can be much longer, causing the number of clauses to grow even more rapidly.

3.2 Adding More Relational Variables

Enumerating the transitivity constraints based on the variables in $\mathcal{E}$ runs the risk of generating a Boolean formula of exponential size. We can guarantee polynomial growth by considering a larger set of relational variables. In general, let $\mathcal{E}'$ be some set of relational variables such that $\mathcal{E} \subseteq \mathcal{E}'$, and let $F_{\text{trans}}(\mathcal{E}')$ be the transitivity constraint formula generated by enumerating the chord-free cycles in the graph $G(\mathcal{E}')$.

Theorem 2 *If $\mathcal{E}$ is the set of relational variables in F_{sat} and $\mathcal{E} \subseteq \mathcal{E}'$, then the formula $F_{sat} \wedge F_{trans}(\mathcal{E})$ is satisfiable if and only if $F_{sat} \wedge F_{trans}(\mathcal{E}')$ is satisfiable.*

We introduce a series of lemmas to prove this theorem. For a propositional formula F over a set of variables $\mathcal{A}$ and an assignment $\chi\colon \mathcal{A} \rightarrow \{0,1\}$, define the *valuation* of F under χ, denoted $[F]_{\chi}$, to be the result of evaluating formula F according to assignment χ. We first prove that we can extend any assignment over a set of relational variables to one over a superset of these variables yielding identical valuations for both transistivity constraint formulas.

Lemma 1 *For any sets of relational variables $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1 \subseteq \mathcal{E}_2$, and for any assignment $\chi_1\colon \mathcal{E}_1 \rightarrow \{0,1\}$, such that $[F_{trans}(\mathcal{E}_1)]_{\chi_1} = 1$, there is an assignment $\chi_2\colon \mathcal{E}_2 \rightarrow \{0,1\}$ such that $[F_{trans}(\mathcal{E}_2)]_{\chi_2} = 1$*

Proof: We consider the case where $\mathcal{E}_2 = \mathcal{E}_1 \cup \{e_{i,j}\}$. The general statement of the proposition then holds by induction on $|\mathcal{E}_2| - |\mathcal{E}_1|$.

Define assignment χ_2 to be:

$$\chi_2(e) \quad = \quad \begin{cases} \chi_1(e), & e \neq e_{i,j} \\ 1, & \text{Graph } G(\mathcal{E}_1, \chi) \text{ has a path of 1-edges from node } i \text{ to node } j. \\ 0, & \text{otherwise} \end{cases}$$

We consider two cases:

1. If $\chi_2(e_{i,j}) = 0$, then any cycle in $G(\mathcal{E}_2, \chi_2)$ through $e_{i,j}$ must contain a 0-edge other than $e_{i,j}$. Hence adding this edge does not introduce any transitivity violations.

2. If $\chi_2(e_{i,j}) = 1$, then there must be some path P_1 of 1-edges between nodes i and j in $G(\mathcal{E}_1, \chi_1)$. In order for the introduction of 1-edge $e_{i,j}$ to create a transitivity violation, there must also be some path P_2 between nodes i and j in $G(\mathcal{E}_1, \chi_1)$ containing exactly one 0-edge. But then we could concatenate paths P_1 and P_2 to form a cycle in $G(\mathcal{E}_1, \chi_1)$ containing exactly one 0-edge, implying that $[F_{\text{trans}}(\mathcal{E}_1)]_{\chi_1} = 0$. We conclude therefore that adding 1-edge $e_{i,j}$ does not introduce any transitivity violations.

□

Lemma 2 *For $\mathcal{E}_1 \subseteq \mathcal{E}_2$ and for any assignment $\chi_2\colon \mathcal{E}_2 \rightarrow \{0,1\}$, such that $[F_{trans}(\mathcal{E}_2)]_{\chi_2} = 1$, we also have $[F_{trans}(\mathcal{E}_1)]_{\chi_2} = 1$*

Proof: We note that any cycle in $G(\mathcal{E}_1, \chi_2)$ must be present in $G(\mathcal{E}_2, \chi_2)$ and have the same edge labeling. Thus, if $G(\mathcal{E}_2, \chi_2)$ has no cycle with a single 0-edge, then neither does $G(\mathcal{E}_1, \chi_2)$. $\square$

We now return to the proof of Theorem 2.

Proof: Suppose that $F_{\text{sat}} \wedge F_{\text{trans}}(\mathcal{E})$ is satisfiable, i.e., there is some assignment χ such that $[F_{\text{sat}}]_{\chi} = [F_{\text{trans}}(\mathcal{E})]_{\chi} = 1$. Then by Lemma 1 we can find an assignment χ' such that $[F_{\text{trans}}(\mathcal{E}')]_{\chi'} = 1$. Furthermore, since the construction of χ' by Lemma 1 preserves the values assigned to all variables in $\mathcal{E}$, and these are the only relational variables occurring in F_{sat}, we can conclude that $[F_{\text{sat}}]_{\chi'} = 1$. Therefore $F_{\text{sat}} \wedge F_{\text{trans}}(\mathcal{E}')$ is satisfiable.

Suppose on the other hand that $F_{\text{sat}} \wedge F_{\text{trans}}(\mathcal{E}')$ is satisfiable, i.e., there is some assignment χ' such that $[F_{\text{sat}}]_{\chi'} = [F_{\text{trans}}(\mathcal{E}')]_{\chi'} = 1$. Then by Lemma 2 we also have $[F_{\text{trans}}(\mathcal{E})]_{\chi'} = 1$, and hence $F_{\text{sat}} \wedge F_{\text{trans}}(\mathcal{E})$ is satisfiable. $\square$

Our goal then is to add as few relational variables as possible in order to reduce the size of the transitivity formula. We will continue to use our path enumeration algorithm to generate the transitivity formula.

3.3 Dense Enumeration

For the *dense* enumeration method, let $\mathcal{E}_N$ denote the set of variables $e_{i,j}$ for all values of i and j such that $1 \leq i < j \leq N$. Graph $G(\mathcal{E}_N)$ is a complete, undirected graph. In this graph, any cycle of length greater than three must have a chord. Hence our algorithm will enumerate transitivity constraints of the form $e_{[i,j]} \wedge e_{[j,k]} \Rightarrow e_{[i,k]}$, for all distinct values of i, j, and k. The graph has $N(N-1)$ edges and $N(N-1)(N-2)/6$ chord-free cycles, yielding a total of $N(N-1)(N-2)/2 = O(N^3)$ transitivity constraints.

The columns labeled "Dense" in Table 2 show the complexity of this method for the benchmark circuits. For the smaller graphs $1\times$DLX-C, $1\times$DLX-C-t, M_4 and M_5, this method yields more clauses than direct enumeration of the cycles in the original graph. For the larger graphs, however, it yields fewer clauses. The advantage of the dense method is most evident for the mesh graphs, where the cubic complexity is far superior to exponential.

3.4 Sparse Enumeration

We can improve on both of these methods by exploiting the sparse structure of $G(\mathcal{E})$. Like the dense method, we want to introduce additional relational variables to give a set of variables $\mathcal{E}^+$ such that the resulting graph $G(\mathcal{E}^+)$ becomes *chordal* [Rose70]. That is, the graph has the property that every cycle of length greater than three has a chord.

Chordal graphs have been studied extensively in the context of sparse Gaussian elimination. In fact, the problem of finding a minimum set of additional variables to add to our set is identical to the problem of finding an elimination ordering for Gaussian elimination that minimizes the amount of fill-in. Although this problem is NP-complete [Yan81], there are good heuristic solutions. In

Circuit		C_{sat} Satisfiable?	Secs.	$C_{\text{trans}} \cup C_{\text{sat}}$ Satisfiable?	Secs.	Ratio
1×DLX-C		N	3	N	4	1.4
1×DLX-C-t		Y	1	N	9	N.A.
2×DLX-CA		N	176	N	1,275	7.2
2×DLX-CA-t		Y	3	N	896	N.A.
2×DLX-CC		N	5,035	N	9,932	2.0
2×DLX-CC-t		Y	4	N	15,003	N.A.
Full	min.	Y	1	Y	1	0.2
Buggy	avg.	Y	125	Y	1,517	2.3
2×DLX-CC	max.	Y	2,186	Y	43,817	69.4

Table 3: **Performance of** FGRASP **on Benchmark Circuits.** Results are given both without and with transitivity constraints.

particular, our implementation proceeds as a series of elimination steps. On each step, we remove some vertex i from the graph. For every pair of distinct, uneliminated vertices j and k such that the graph contains edges (i, j) and (i, k), we add an edge (j, k) if it does not already exist. The original graph plus all of the added edges then forms a chordal graph. To choose which vertex to eliminate on a given step, our implementation uses the simple heuristic of choosing the vertex with minimum degree. If more than one vertex has minimum degree, we choose one that minimizes the number of new edges added.

The columns in Table 2 labeled “Sparse” show the effect of making the benchmark graphs chordal by this method. Observe that this method gives superior results to either of the other two methods. In our implementation we have therefore used the sparse method to generate all of the transitivity constraint formulas.

4 SAT-Based Decision Procedures

Most Boolean satisfiability (SAT) checkers take as input a formula expressed in clausal form. Each clause is a set of *literals*, where a literal is either a variable or its complement. A clause denotes the disjunction of its literals. The task of the checker is to either find an assignment to the variables that satisfies all of the clauses or to determine that no such assignment exists. We can solve the constrained satisfiability problem using a conventional SAT checker by generating a set of clauses C_{trans} representing $F_{\text{trans}}(\mathcal{E}^+)$ and a set of clauses C_{sat} representing the formula F_{sat}. We then run the checker on the combined clause set $C_{\text{sat}} \cup C_{\text{trans}}$ to find satisfying solutions to $F_{\text{sat}} \wedge F_{\text{trans}}(\mathcal{E}^+)$.

In experimenting with a number of Boolean satisfiability checkers, we have found that FGRASP [MS99] has the best overall performance. The most recent version can be directed to periodically restart the search using a randomly-generated variable assignment [M99]. This is the first SAT checker we have tested that can complete all of our benchmarks. All of our experiments were conducted on a 336 MHz Sun UltraSPARC II with 1.2GB of primary memory.

As indicated by Table 3, we ran FGRASP on clause sets C_{sat} and $C_{\text{trans}} \cup C_{\text{sat}}$, i.e., both without and with transitivity constraints. For benchmarks 1×DLX-C, 2×DLX-CA, and 2×DLX-CC, the formula F_{sat} is unsatisfiable. As can be seen, including transitivity constraints increases the run time significantly. For benchmarks 1×DLX-C-t, 2×DLX-CA-t, and 2×DLX-CC-t, the formula F_{sat} is satisfiable, but only because transitivity is not enforced. When we add the clauses for F_{trans}, the formula becomes unsatisfiable. For the buggy circuits, the run times for C_{sat} range from under 1 second to over 36 minutes. The run times for $C_{\text{trans}} \cup C_{\text{sat}}$ range from less than one second to over 12 hours. In some cases, adding transitivity constraints actually decreased the CPU time (by as much as a factor of 5), but in most cases the CPU time increased (by as much as a factor of 69). On average (using the geometric mean) adding transitivity constraints increased the CPU time by a factor of 2.3. We therefore conclude that satisfiability checking with transitivity constraints is more difficult than conventional satisfiability checking, but the added complexity is not overwhelming.

5 OBDD-Based Decision Procedures

A simple-minded approach to solving satisfiability with transitivity constraints using OBDDs would be to generate separate OBDD representations of F_{trans} and F_{sat}. We could then use the APPLY operation to generate an OBDD for $F_{\text{trans}} \wedge F_{\text{sat}}$, and then either find a satisfying assignment or determine that the function is unsatisfiable. We show that for some sets of relational variables $\mathcal{E}$, the OBDD representation of $F_{\text{trans}}(\mathcal{E})$ can be too large to represent and manipulate. In our experiments, we use the CUDD OBDD package with dynamic variable reordering by sifting.

5.1 Lower Bound on the OBDD Representation of $F_{\text{trans}}(\mathcal{E})$

We prove that for some sets $\mathcal{E}$, the OBDD representation of $F_{\text{trans}}(\mathcal{E})$ may be of exponential size for all possible variable orderings. As mentioned earlier, the NP-completeness result proved by Goel, *et al.* [GSZAS98] has implications for the complexity of representing $F_{\text{trans}}(\mathcal{E})$ as an OBDD. They showed that given an OBDD G_{sat} representing formula F_{sat}, the task of finding a satisfying assignment of F_{sat} that also satisfies the transitivity constraints in $Trans(\mathcal{E})$ is NP-complete in the size of G_{sat}. By this, assuming $P \neq NP$, we can infer that the OBDD representation of $F_{\text{trans}}(\mathcal{E})$ may be of exponential size when using the same variable ordering as is used in G_{sat}. Our result extends this lower bound to arbitrary variable orderings and is independent of the P vs. NP problem.

Let M_n denote a planar mesh consisting of a square array of $n \times n$ vertices. For example, Figure 3 shows the graph for $n = 6$. Being a planar graph, the edges partition the plane into *faces*. As shown in Figure 3 we label these $F_{i,j}$ for $1 \leq i, j \leq n-1$. There are a total of $(n-1)^2$ such faces. One can see that the set of edges forming the border of each face forms a chord-free cycle of M_n. As shown in Table 2, many other cycles are also chord-free, e.g., the perimeter of any rectangular region having height and width greater than 1, but we will consider only the cycles

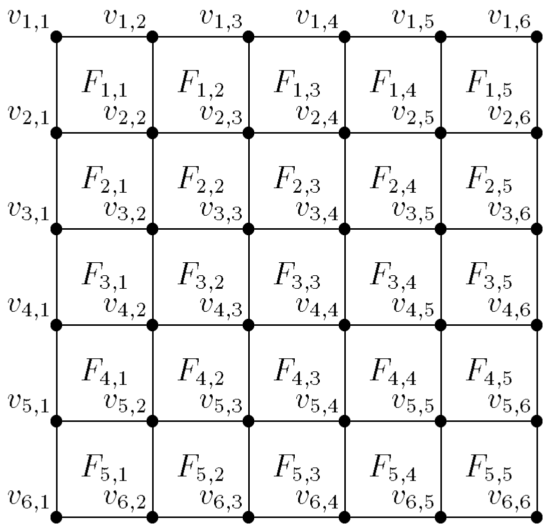


Figure 3: **Mesh Graph M_6.**

corresponding to single faces.

Define $\mathcal{E}_{n\times n}$ to be a set of relational variables corresponding to the edges in M_n. $F_{\text{trans}}(\mathcal{E}_{n\times n})$ is then an encoding of the transitivity constraints for these variables.

Theorem 3 *Any OBDD representation of $F_{\mathit{trans}}(\mathcal{E}_{n\times n})$ must have $\Omega(2^{n/4})$ vertices.*

To prove this theorem, consider any ordering of the variables representing the edges in M_n. Let A denote those in the first half of the ordering, and B denote those in the second half. We can then classify each face according to the four edges forming its border:

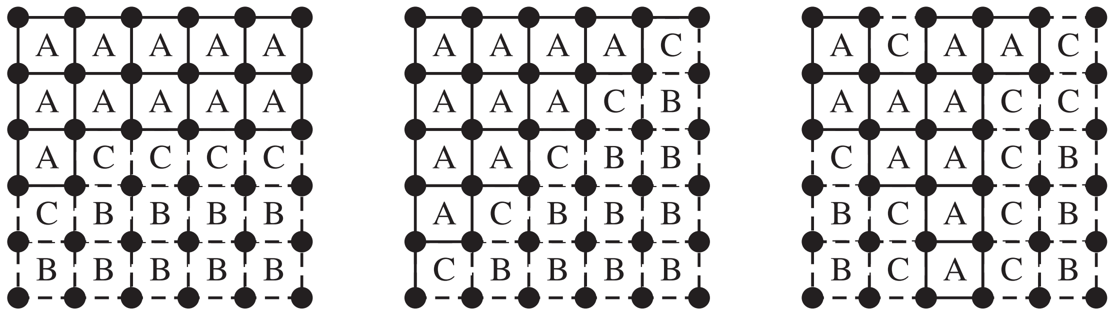
A: All are in A.

B: All are in B.

C: Some are in A, while others are in B. These are called "split" faces.

Observe that we cannot have a type A face adjacent to a type B face, since their shared edge cannot be in both A and B. Therefore there must be split faces separating any region of type A faces from any region of type B faces.

For example, Figure 4 shows three possible partitionings of the edges of M_6 and the resulting classification of the faces. If we let a, b, and c denote the number of faces of each respective type, we see that we always have $c \geq 5 = n - 1$. In particular, a minimum value for c is achieved when the partitioning of the edges corresponds to a partitioning of the graph into a region of type A faces and a region of type B faces, each having nearly equal size, with the split faces forming the boundary between the two regions.



$a = 11, b = 9, c = 5$ $\quad$ $a = 10, b = 10, c = 5$ $\quad$ $a = 10, b = 5, c = 10$

Figure 4: **Partitioning Edges into Sets A (solid) and B (dashed).** Each face can then be classified as type A (all solid), B (all dashed), or C (mixed).

Lemma 3 *For any partitioning of the edges of mesh graph M_n into equally-sized sets A and B, there must be at least $(n-3)/2$ split faces.*

Note that this lower bound is somewhat weak—it seems clear that we must have $c \geq n-1$. However, this weaker bound will suffice to prove an exponential lower bound on the OBDD size.

Proof: Our proof is an adaptation of a proof by Leighton [Lei92, Theorem 1.21] that M_n has a bisection bandwidth of at least n. That is, one would have to remove at least n edges to split the graph into two parts of equal size.

Observe that M_n has n^2 vertices and $2n(n-1)$ edges. These edges are split so that $n(n-1)$ are in A and $n(n-1)$ are in B.

Let M_n^D denote the planar dual of M_n. That is, it contains a vertex $u_{i,j}$ for each face $F_{i,j}$ of M_n, and edges between pairs of vertices such that the corresponding faces in M_n have a common edge. In fact, one can readily see that this graph is isomorphic to M_{n-1}.

Partition the vertices of M_n^D into sets U_a, U_b, and U_c according to the types of their corresponding faces. Let a, b, and c denote the number of elements in each of these sets. Each face of M_n has four bordering edges, and each edge is the border of at most two faces. Thus, as an upper bound on a, we must have $4a \leq 2n(n-1)$, giving $a \leq n(n-1)/2$, and similarly for b. In addition, since a face of type A cannot be adjacent in M_n to one of type B, no vertex in U_a can be adjacent in M_n^D to one in U_b.

Consider the complete, directed, bipartite graph having as edges the set $(U_a \times U_b) \cup (U_b \times U_a)$, i.e., a total of $2ab$ edges. Given the bounds: $a + b = (n-1)^2 - c$, $a \leq n(n-1)/2$, and $b \leq n(n-1)/2$, the minimum value of $2ab$ is achieved when either $a = n(n-1)/2$ and $b = (n-1)^2 - (n-1)n/2 - c = (n-1)(n-2)/2 - c$, or vice-versa, giving a lower bound:

$$2ab \quad \geq \quad 2[n(n-1)/2] \cdot [(n-1)(n-2)/2 - c]$$

$$= \quad n(n-1)^2(n-2)/2 \ - \ cn(n-1)$$

We can embed this bipartite graph in M_n^D by forming a path from vertex $u_{i,j}$ to vertex $u_{i',j'}$, where either $u_{i,j} \in U_a$ and $u_{i',j'} \in U_b$, or vice-versa. By convention, we will use the path that first follows vertical edges to $u_{i',j}$ and then follows horizontal edges to $u_{i',j'}$. We must have at least one vertex in U_c along each such path, and therefore removing the vertices in U_c would cut all $2ab$ paths.

For each vertex $u_{i,j} \in U_c$, we can bound the total number of paths passing through it by separately considering paths that enter from the bottom, the top, the left, and the right. For those entering from the bottom, there are at most $n-i-1$ source vertices and $i(n-1)$ destination vertices, giving at most $i(n-i-1)(n-1)$ paths. This quantity is maximized for $i = (n-1)/2$, giving an upper bound of $(n-1)^3/4$. A similar argument shows that there are at most $(n-1)^3/4$ paths entering from the top of any vertex. For the paths entering from the left, there are at most $(j-1)(n-1)$ source vertices and $(n-j)$ destinations, giving at most $(j-1)(n-j)(n-1)$ paths. This quantity is maximized when $j = (n-1)/2$, giving an upper bound of $(n-1)^3/4$. This bound also holds for those paths entering from the right. Thus, removing a single vertex would cut at most $(n-1)^3$ paths.

Combining the lower bound on the number of paths $2ab$, the upper bound on the number of paths cut by removing a single vertex, and the fact that we are removing c vertices, we have:

$$\begin{aligned}
c(n-1)^3 &\geq n(n-1)^2(n-2)/2 - cn(n-1) \\
c(n-1)^3 + cn &\geq n(n-1)(n-2)/2 \\
c(n^2-n+1) &\geq n(n-1)(n-2)/2
\end{aligned}$$

We can rewrite $n(n-1)(n-2)$ as $(n^2-n+1)(n-3)+n^2-2n+3$. Observing that $n^2-2n+3 > 0$ for all values of n, we have:

$$\begin{aligned}
c(n^2-n+1) &\geq (n^2-n+1)(n-3)/2 + (n^2-2n+3)/2 \\
&\geq (n^2-n+1)(n-3)/2 \\
c &\geq (n-3)/2
\end{aligned}$$

□

A set of faces is said to be *edge independent* when no two members of the set share an edge.

Lemma 4 *For any partitioning of the edges of mesh graph M_n into equally-sized sets A and B, there must be an edge-independent set of split faces containing at least $(n-3)/4$ elements.*

Proof: Classify the *parity* of face $F_{i,j}$ as "even" when $i+j$ is even, and as "odd" otherwise. Observe that no two faces of the same parity can have a common edge. Divide the set of split faces into two subsets: those with even parity and those with odd. Both of these subsets are edge independent, and one of them must have at least 1/2 of the elements of the set of all split faces. □

We can now complete the proof of Theorem 3 *Proof:* Suppose there is an edge-independent set of k split faces. For each split face, choose one edge in A and one edge in B bordering that face.

For each value $\vec{y} \in \{0,1\}^k$, define assignment $\alpha_{\vec{y}}$ (respectively, $\beta_{\vec{y}}$), to the variables representing edges in A (resp., B) as follows. For an edge e that is not part of any of the k split faces, define $\alpha_{\vec{y}}(e) = 0$ (resp., $\beta_{\vec{y}}(e) = 0$). For an edge e that is part of a split face, but it was not one of the ones chosen specially, let $\alpha_{\vec{y}} = 1$ (resp., $\beta_{\vec{y}}(e) = 1$). For an edge e that is the chosen variable in face i, let $\alpha_{\vec{y}}(e) = y_i$ (resp., $\beta_{\vec{y}}(e) = y_i$). This will give us an assignment $\alpha_{\vec{y}} \cdot \beta_{\vec{y}}$ to all of the variables that evaluates to 1. That is, for each independent, split face F_i, we will have two 1-edges when $y_i = 0$ and four 1-edges when $y_i = 1$. All other cycles in the graph will have at least two 0-edges.

On the other hand, for any $\vec{y}, \vec{z} \in \{0,1\}^k$ such that $\vec{y} \neq \vec{z}$ the assignment $\alpha_{\vec{y}} \cdot \beta_{\vec{z}}$ will cause an evaluation to 0, because for any face i where $y_i \neq z_i$, all but one edge will be assigned value 1. Thus, the set of assignments $\{\alpha_{\vec{y}} | \vec{y} \in \{0,1\}^k\}$ forms an OBDD fooling set, as defined in [Bry91], implying that the OBDD must have at least $2^k \geq 2^{(n-3)/4} = \Omega(2^{n/4})$ vertices. □

We have seen that adding relational variables can reduce the number of cycles and therefore simplify the transitivity constraint formula. This raises the question of how adding relational variables affects the BDD representation of the transitivity constraints. Unfortunately, the exponential lower bound still holds.

Corollary 1 *For any set of relational variables $\mathcal{E}$ such that $\mathcal{E}_{n\times n} \subseteq \mathcal{E}$, any OBDD representation of $F_{\mathit{trans}}(\mathcal{E})$ must contain $\Omega(2^{n/8})$ vertices.*

The extra edges in $\mathcal{E}$ introduce complications, because they create cycles containing edges from different faces. As a result, our lower bound is weaker.

Define a set of faces as *vertex independent* if no two members share a vertex.

Lemma 5 *For any partitioning of the edges of mesh graph M_n into equal-sized sets A and B, there must be a vertex-independent set of split faces containing at least $(n-3)/8$ elements.*

Proof: Partition the set of split faces into four sets: EE, EO, OE, and OO, where face $F_{i,j}$ is assigned to a set according to the values of i and j:

EE: Both i and j are even.

EO: i is even and j is odd.

OE: i is odd and j is even.

OO: Both i and j are odd.

Each of these sets is vertex independent. At least one of the sets must contain at least $1/4$ of the elements. Since there are at least $(n-3)/2$ split faces, one of the sets must contain at least $(n-3)/8$ vertex-independent split faces. □

We can now prove Corollary 1.

Proof: For any ordering of the variables in $\mathcal{E}$, partition them into two sets A and B such that those in A come before those in B, and such the number of variables that are in $\mathcal{E}_{n\times n}$ are equally split between A and B. Suppose there is a vertex-independent set of k split faces. For each value $\vec{y} \in \{0,1\}^k$, we define assignments $\alpha_{\vec{y}}$ to the variables in A and $\beta_{\vec{y}}$ to the variables in B. These assignments are defined as they are in the proof of Theorem 3 with the addition that each variable $e_{i,j}$ in $\mathcal{E} - \mathcal{E}_{n \times n}$ is assigned value 0. Consider the set of assignments $\alpha_{\vec{y}} \cdot \beta_{\vec{z}}$ for all values $\vec{y}, \vec{z} \in \{0,1\}^k$. The only cycles in $G(\mathcal{E}, \alpha_{\vec{y}} \cdot \beta_{\vec{z}})$ that can have less than two 0-edges will be those corresponding to the perimeters of split faces. As in the proof of Theorem 3, the set $\{\alpha_{\vec{y}} | \vec{y} \in \{0,1\}^k\}$ forms an OBDD fooling set, as defined in [Bry91], implying that the OBDD must have at least $2^k \geq 2^{(n-3)/8} = \Omega(2^{n/8})$ vertices. □

Our lower bounds are fairly weak, but this is more a reflection of the difficulty of proving lower bounds. We have found in practice that the OBDD representations of the transitivity constraint functions arising from benchmarks tend to be large relative to those encountered during the evaluation of F_{sat}. For example, although the OBDD representation of $F_{\text{trans}}(\mathcal{E}^+)$ for benchmark 1×DLX-C-t is just 2,692 nodes (a function over 42 variables), we have been unable to construct the OBDD representations of this function for either 2×DLX-CA-t (178 variables) or 2×DLX-CC-t (193 variables) despite running for over 24 hours.

5.2 Enumerating and Eliminating Violations

Goel, *et al.* [GSZAS98] proposed a method that generates implicants (cubes) of the function F_{sat} from its OBDD representation. Each implicant is examined and discarded if it violates a transitivity constraint. In our experiments, we have found this approach works well for the normal, correctly-designed pipelines (i.e., circuits 1×DLX-C, 2×DLX-CA, and 2×DLX-CC) since the formula F_{sat} is unsatisfiable and hence has no implicants. For all 100 of our buggy circuits, the first implicant generated contained no transitivity violation and hence was a valid counterexample.

For circuits that do require enforcing transitivity constraints, we have found this approach impractical. For example, in verifying 1×DLX-C-t by this means, we generated 253,216 implicants, requiring a total of 35 seconds of CPU time (vs. 0.2 seconds for 1×DLX-C). For benchmarks 2×DLX-CA-t and 2×DLX-CC-t, our program ran for over 24 hours without having generated all of the implicants. By contrast, circuits 2×DLX-CA and 2×DLX-CC can be verified in 11 and 29 seconds, respectively. Our implementation could be improved by making sure that we generate only implicants that are irredundant and prime. In general, however, we believe that a verifier that generates individual implicants will not be very robust. The complex control logic for a pipeline can lead to formulas F_{sat} containing very large numbers of implicants, even when transitivity plays only a minor role in the correctness of the design.

5.3 Enforcing a Reduced Set of Transitivity Constraints

One advantage of OBDDs over other representations of Boolean functions is that we can readily determine the *true support* of the function, i.e., the set of variables on which the function depends. This leads to a strategy of computing an OBDD representation of F_{sat} and intersecting its support with $\mathcal{E}$ to give a set $\hat{\mathcal{E}}$ of relational variables that could potentially lead to transitivity violations. We then augment these variables to make the graph chordal, yielding a set of variables $\hat{\mathcal{E}}^+$ and

Circuit		Verts.	Direct			Dense			Sparse		
			Edges	Cycles	Clauses	Edges	Cycles	Clauses	Edges	Cycles	Clauses
1×DLX-C-t		9	18	14	45	36	84	252	20	19	57
2×DLX-CA-t		17	44	101	395	136	680	2,040	49	57	171
2×DLX-CC-t		17	46	108	417	136	680	2,040	52	66	198
Reduced	min.	3	2	0	0	3	1	3	2	0	0
Buggy	avg.	12	17	19	75	73	303	910	21	14	42
2×DLX-CC	max.	19	52	378	1,512	171	969	2,907	68	140	420

Table 4: **Graphs for Reduced Transitivity Constraints.** Results are given for the three different methods of encoding transitivity constraints based on the variables in the true support of F_{sat}.

Circuit		OBDD Nodes			CPU
		F_{sat}	$F_{\text{trans}}(\hat{\mathcal{E}}^+)$	$F_{\text{sat}} \wedge F_{\text{trans}}(\hat{\mathcal{E}}^+)$	Secs.
1×DLX-C		1	1	1	0.2
1×DLX-C-t		530	344	1	2
2×DLX-CA		1	1	1	11
2×DLX-CA-t		22,491	10,656	1	109
2×DLX-CC		1	1	1	29
2×DLX-CC-t		17,079	7,168	1	441
Reduced	min.	20	1	20	7
Buggy	avg.	3,173	1,483	25,057	107
2×DLX-CC	max.	15,784	93,937	438,870	2,466

Table 5: **OBDD-based Verification.** Transitivity constraints were generated for a reduced set of variables $\hat{\mathcal{E}}$.

generate an OBDD representation of $F_{\text{trans}}(\hat{\mathcal{E}}^+)$. We compute $F_{\text{sat}} \wedge F_{\text{trans}}(\hat{\mathcal{E}}^+)$ and, if it is satisfiable, generate a counterexample.

Table 4 shows the complexity of the graphs generated by this method for our benchmark circuits. Comparing these with the full graphs shown in Table 2, we see that we typically reduce the number of relational vertices (i.e., edges) by a factor of 3 for the benchmarks modified to require transitivity and by an even greater factor for the buggy circuit benchmarks. The resulting graphs are also very sparse. For example, we can see that both the direct and sparse methods of encoding transitivity constraints greatly outperform the dense method.

Table 5 shows the complexity of applying the OBDD-based method to all of our benchmarks. The original circuits 1×DLX-C, 2×DLX-CA, and 2×DLX-CC yielded formulas F_{sat} that were unsatisfiable, and hence no transitivity constraints were required. The 3 modified circuits 1×DLX-C-t, 2×DLX-CA-t, and 2×DLX-CC-t are more interesting. The reduction in the number of relational variables makes it feasible to generate an OBDD representation of the transitivity constraints. Compared to benchmarks 1×DLX-C, 2×DLX-CA, and 2×DLX-CC, we see there is a significant, although tolerable, increase in the computational requirement to verify the modified circuits. This can be attributed to both the more complex control logic and to the need to apply the transitivity constraints.

For the 100 buggy variants of 2×DLX-CC, F_{sat} depends on up to 52 relational variables, with an average of 17. This yielded OBDDs for $F_{\text{trans}}(\hat{\mathcal{E}}^+)$ ranging up to 93,937 nodes, with an average of 1,483. The OBDDs for $F_{\text{sat}} \wedge F_{\text{trans}}(\hat{\mathcal{E}}^+)$ ranged up to 438,870 nodes (average 25,057), showing that adding transitivity constraints does significantly increase the complexity of the OBDD representation. However, this is just one OBDD at the end of a sequence of OBDD operations. In the worst case, imposing transitivity constraints increased the total CPU time by a factor of 2, but on average it only increased by 2%. The memory required to generate F_{sat} ranged from 9.8 to 50.9 MB (average 15.5), but even in the worst case the total memory requirement increased by only 2%.

6 Conclusion

By formulating a graphical interpretation of the relational variables, we have shown that we can generate a set of clauses expressing the transitivity constraints that exploits the sparse structure of the relation. Adding relational variables to make the graph chordal eliminates the theoretical possibility of there being an exponential number of clauses and also works well in practice. A conventional SAT checker can then solve constrained satisfiability problems, although the run times increase significantly compared to unconstrained satisfiability. Our best results were obtained using OBDDs. By considering only the relational variables in the true support of F_{sat}, we can enforce transitivity constraints with only a small increase in CPU time.

References

[Bry86] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[Bry91] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February, 1991), pp. 205–213.

[BGV99a] R. E. Bryant, S. German, and M. N. Velev, "Exploiting positive equality in a logic of equality with uninterpreted functions," *Computer-Aided Verification (CAV '99)*, N. Halbwachs, and D. Peled, *eds.*, LNCS 1633, Springer-Verlag, July, 1999, pp. 470–482.

[BGV99b] R. E. Bryant, S. German, and M. N. Velev, "Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic," Technical report CMU-CS-99-115, Carnegie Mellon University, 1999. Available as: `http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-99-115.ps`.

[BD94] J. R. Burch, and D. L. Dill, "Automated verification of pipelined microprocessor control," *Computer-Aided Verification (CAV '94)*, D. L. Dill, *ed.*, LNCS 818, Springer-Verlag, June, 1994, pp. 68–80.

[Bur96] J. R. Burch, "Techniques for verifying superscalar microprocessors," *33rd Design Automation Conference (DAC '96)*, June, 1996, pp. 552–557.

[GSZAS98] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, "BDD based procedures for a theory of equality with uninterpreted functions," *Computer-Aided Verification (CAV '98)*, A. J. Hu and M. Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June, 1998, pp. 244–255.

[HP96] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition Morgan-Kaufmann, San Francisco, 1996.

[Lei92] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kaufmann, 1992.

[MS99] J. P. Marques-Silva, and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, Vol. 48, No. 5 (May, 1999), pp. 506–521.

[M99] J. P. Marques-Silva, "The impact of branching heuristics in propositional satisfiability algorithms," *9th Portugese Conference on Artificial Intelligence*, September, 1999.

[Rose70] D. Rose, "Triangulated graphs and the elimination process," *Journal of Mathematical Analysis and Applications*, Vol. 32 (1970), pp. 597–609.

[VB99] M. N. Velev, and R. E. Bryant, "Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions," *Correct Hardware Design and Verification Methods (CHARME '99)*, L. Pierre, and T. Kropf, *eds.*, LNCS 1703, Springer-Verlag, September, 1999, pp. 37–53.

[Yan81] M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM Journal of Algebraic and Discrete Mathematics*, Vol. 2 (1981), pp. 77–79.