Boolean Satisfiability with Transitivity Constraints*

Randal E. Bryant[1] and Miroslav N. Velev[2]

[1] Computer Science, Carnegie Mellon University, Pittsburgh, PA
Randy.Bryant@cs.cmu.edu
[2] Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
mvelev@ece.cmu.edu

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, symmetric, transitive relation over N elements. 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

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

* This research was supported by the Semiconductor Research Corporation, Contract 99-DC-684

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 85–98, 2000.
© Springer-Verlag Berlin Heidelberg 2000

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 encoding techniques 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.

Due to space limitations, this paper omits many technical details. More information, including formal proofs, is included in [BV00].

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 and contains an interlock to stall the instruction following a load by one cycle if it requires the loaded result. 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. 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 6 instruction types.

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

Circuit		Domain Variables	Propositional Variables	Equations
1×DLX-C		13	42	27
1×DLX-Ct		13	42	37
2×DLX-CA		25	58	118
2×DLX-CAt		25	58	137
2×DLX-CC		25	70	124
2×DLX-CCt		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.

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 do not require enforcing transitivity in order to verify them. In particular, the formula F_{sat} is unsatisfiable in all cases. This implies that the constrained satisfiability problems are unsatisfiable as well. 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 compares 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 $\mathtt{ESrc1} = \mathtt{MDest}$ to be $\mathtt{ESrc1} = \mathtt{MDest} \lor (\mathtt{ESrc1} = \mathtt{ESrc2} \land \mathtt{ESrc2} = \mathtt{MDest})$. 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-Ct, 2×DLX-CAt, and 2×DLX-CCt.

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 $\mathit{Trans}(\mathcal{E})$ (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, we will classify edge (i, j) 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, i.e., $1 \leq i_p \leq N$ for all p such that $1 \leq p \leq k$, and (i_p, i_{p+1}) is in $G(\mathcal{E})$ for all p such that $1 \leq p < k$. 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 to the variables in $\mathcal{E}$ violates transitivity if and only if some cycle in $G(\mathcal{E})$ contains exactly one 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 to the variables in $\mathcal{E}$ violates transitivity if and only if some* simple *cycle in $G(\mathcal{E})$ contains exactly one 0-edge.*

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 $1 \leq p < q \leq k$, that $p + 1 < q$, 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 to the variables in $\mathcal{E}$ violates transitivity if and only if some* chord-free *cycle in $G(\mathcal{E})$ contains exactly one 0-edge.*

For a set of relational variables $\mathcal{E}$, we define $F_{\text{trans}}(\mathcal{E})$ to be the conjunction of all transitivity constraints generated by enumerating the set of all chord-free cycles in the graph $G(\mathcal{E})$. Each length k cycle $[i_1, i_2, \ldots, i_k, i_1]$ yields k constraints. It is easily proved that an assignment to the relational variables will satisfy all of the transitivity constraints if and only if it satisfies $F_{\text{trans}}(\mathcal{E})$.

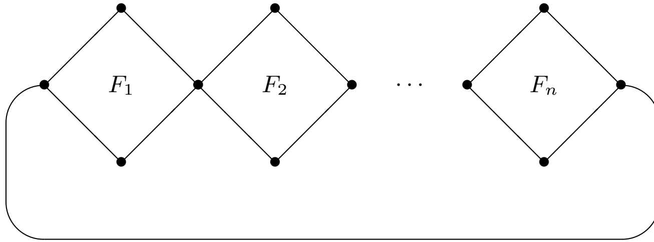

Fig. 1. Class of Graphs with Many Chord-Free Cycles. For a graph with n diamond-shaped faces, there are $2^n + n$ chord-free cycles.

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 $1 \leq p < q \leq k$, that $p + 1 < q$, 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.*

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.*

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 $1 \leq k \leq N$. As an initial case, we have $P_1 = \{[i] | 1 \leq i \leq n\}$, and we have $C = \emptyset$. At each step we consider all possible extensions to the paths in P_k to generate the set P_{k+1} and to add some cycles of length $k + 1$ to C.

As Figure 1 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.

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 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 2. 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

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-Ct		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-CAt		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-CCt		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.

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 only 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 $G(\mathcal{E}')$.

Proposition 6. *If $\mathcal{E}$ is the set of relational variables in F_{sat} and $\mathcal{E} \subseteq \mathcal{E}'$, then:*

$$F_{sat} \wedge F_{trans}(\mathcal{E}) \Leftrightarrow F_{sat} \wedge F_{trans}(\mathcal{E}').$$

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-Ct, 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 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

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 $C_{\text{trans}} \cup C_{\text{sat}}$ to find satisfying solutions to $F_{\text{trans}}(\mathcal{E}^+) \wedge F_{\text{sat}}$.

In experimenting with a number of Boolean satisfiability checkers, we have found that FGRASP [MS99] gives the most consistent results. 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 $1\times$DLX-C, $2\times$DLX-CA, and

Circuit		C_{sat} Satisfiable?	Secs.	$C_{trans} \cup C_{sat}$ Satisfiable?	Secs.	Ratio
1×DLX-C		N	3	N	4	1.4
1×DLX-Ct		Y	1	N	9	N.A.
2×DLX-CA		N	176	N	1,275	7.2
2×DLX-CAt		Y	3	N	896	N.A.
2×DLX-CC		N	5,035	N	9,932	2.0
2×DLX-CCt		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.

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-Ct, 2×DLX-CAt, and 2×DLX-CCt, 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 $C_{trans} \cup C_{sat}$ 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 $F_{trans} \wedge F_{sat}$, 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 $F_{trans}(\mathcal{E})$ can be too large to represent and manipulate. In our experiments, we use the CUDD OBDD package with variable reordering by sifting.

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

We prove that for some sets $\mathcal{E}$, the OBDD representation of $F_{trans}(\mathcal{E})$ 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 $F_{trans}(\mathcal{E})$ 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

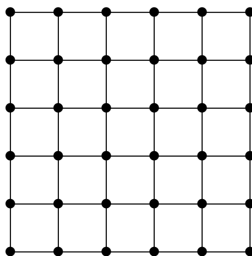

Fig. 2. Mesh Graph M_6

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 $F_{\mathrm{trans}}(\mathcal{E})$ 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 2 shows the graph for $n = 6$. Define $\mathcal{E}_{n\times n}$ to be a set of relational variables corresponding to the edges in M_n. $F_{\mathrm{trans}}(\mathcal{E}_{n\times n})$ is then an encoding of the transitivity constraints for these variables.

Theorem 1. *Any OBDD representation of* $F_{\mathit{trans}}(\mathcal{E}_{n\times n})$ *must have* $\Omega(2^{n/4})$ *vertices.*

A complete proof of this theorem is given in [BV00]. We give only a brief sketch here. Being a planar graph, the edges partition the plane into *faces*. The proof first involves a combinatorial argument showing that for any partitioning of the edges into sets A and B, we can identify a set of at least $(n-3)/4$ edge-independent, "split faces," where a split face has some of its edge variables in set A and others in set B. The proof of this property is similar to a proof by Leighton [Lei92, Theorem 1.21] that M_n has a bisection bandwidth of at least n, i.e., one must remove at least n vertices to split the graph into two parts of equal size.

Given this property, for any ordering of the OBDD variables, we can construct a family of $2^{(n-3)/4}$ assignments to the variables in the first half of the ordering that must lead to distinct vertices in the OBDD. That is, the OBDD must encode information about each split face for the variables in the first half of the ordering so that it can correctly deduce the function value given the variables in the last half of the ordering.

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, the lower bound is weaker, because our proof requires that we find a set of vertex-independent, split faces.

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-Ct 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-CAt (178 variables) or 2×DLX-CCt (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 we did not require additional effort to find a counterexample.

For circuits that do require enforcing transitivity constraints, we have found this approach impractical. For example, in verifying 1×DLX-Ct by this means, we generated 253,216 implicants, requiring a total of 35 seconds of CPU time (vs. 0.1 seconds for 1×DLX-C). For benchmarks 2×DLX-CAt and 2×DLX-CCt, 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 primes 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

Circuit		Verts.	Direct			Dense			Sparse		
			Edges	Cycles	Clauses	Edges	Cycles	Clauses	Edges	Cycles	Clauses
1×DLX-Ct		9	18	14	45	36	84	252	20	19	57
2×DLX-CAt		17	44	101	395	136	680	2,040	49	57	171
2×DLX-CCt		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}.

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 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.

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-Ct		530	344	1	2
2×DLX-CA		1	1	1	11
2×DLX-CAt		22,491	10,656	1	109
2×DLX-CC		1	1	1	29
2×DLX-CCt		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 $\hat{\mathcal{E}}$.

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-Ct, 2×DLX-CAt, and 2×DLX-CCt 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 $F_{\text{trans}}(\hat{\mathcal{E}}^{+}) \wedge F_{\text{sat}}$ 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.

[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`.

[BV00] R. E. Bryant, and M. N. Velev, "Boolean satisfiability with transitivity constraints," Technical report CMU-CS-00-101, Carnegie Mellon University, 2000. Available as:
`http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-00-101.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.