Deciding Separation Formulas with SAT [*]

Ofer Strichman Sanjit A. Seshia Randal E. Bryant

Computer Science, Carnegie Mellon University, Pittsburgh, PA
ofers|sanjit|bryant@cs.cmu.edu

Abstract. We show a reduction to propositional logic from a Boolean combination of inequalities of the form $v_i \geq v_j + c$ and $v_i > v_j + c$, where c is a constant and v_i, v_j are variables of type `real` or `integer`. Equalities and uninterpreted functions can be expressed in this logic as well. We discuss the advantages of using this reduction as compared to competing methods, and present experimental results that support our claims.

1 Introduction

Recent advances in SAT solving make it worthwhile to try and reduce hard decision problems, that were so far solved by designated algorithms, to the problem of deciding a propositional formula. Modern SAT solvers can frequently decide formulas with hundreds of thousands of variables in a short amount of time. They are used for solving a variety of problems such as AI planning, Automatic Test Pattern Generation (ATPG), Bounded Model Checking, and more. In this paper we show such a reduction to SAT from a theory of *separation predicates*[1], i.e., formulas that contain the standard Boolean connectives, as well as predicates of the form $v_i \rhd v_j + c$ where $\rhd \in \{>, \geq\}$, c is a constant, and v_i, v_j are variables of type `real` or `integer`. The other inequality signs as well as equalities can be expressed in this logic. Uninterpreted functions can be handled as well since they can be reduced to Boolean combinations of equalities[1].

Separation predicates are used in verification of timed systems, scheduling problems, and more. Hardware models with ordered data structures have inequalities as well. For example, if the model contains a queue of unbounded length, the test for $head \leq tail$ introduces inequalities. In fact, most inequalities in verification conditions, Pratt observed [9], are of this form. Furthermore, since theorem provers can decide mixed theories (by invoking an appropriate

[*] This research was supported in part by the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and the Gigascale Research Center under contract 98-DT-660. The second author is supported in part by a National Defense Science and Engineering Graduate Fellowship.

[1] The term *separation predicates* is adopted from Pratt[9], who considered 'separation theory', a more restricted case in which all the constraints are of the form $v_i \leq v_j + c$, and conjunction is the only Boolean operator allowed. This logic is also known as 'difference logic'.

decision procedure for each logic fragment[11]), restricting our attention to separation predicates does not mean that it is helpful only for pure combinations of these predicates. Rather it means that the new decision procedure can shorten the verification time of any formula that contains a significant number of these predicates.

The reduction to SAT we suggest is based on two steps. First, we encode the separation predicates as new Boolean variables. Second, we add constraints on these variables, based on an analysis of the transitivity of the original predicates. A similar framework was used by Bryant et al. to reduce equality predicates [4]. The current work can therefore be seen as a natural extension of their work to the more general segment of logic, namely a logic of separation predicates.

2 SAT vs. other decision procedures

There are many methods for deciding a formula consisting of a conjunction of separation predicates. For example, a known graph-based decision procedure for this type of formulas (frequently attributed to Bellman, 1957) works as follows: given a conjunction of separation predicates φ, it constructs a *constraints graph*, which is a directed graph $G(V, E)$ in which the set of nodes is equal to the set of variables in φ, and node v_i has a directed edge with 'weight' c to node v_j iff the constraint $v_i \leq v_j + c$ is in φ. It is not hard to see that φ is satisfiable iff there is no cycle in G with a negative accumulated weight. Thus, deciding φ is reduced to searching the graph for such cycles. Variations of this procedure were described, for example in [9], and are implemented in theorem provers such as Coq[2]. The Bellman-Ford algorithm [6] can find whether there is a negative cycle in such a graph in polynomial time, and is considered as the standard in solving these problems. It is used, for example, when computing Difference Decision Diagrams (DDD) [7]. DDD's are similar to BDDs, but instead of Boolean variables, their nodes are labeled with separation predicates. In order to compute whether each path in the DDD leads to '0' or '1', the Bellman-Ford procedure is invoked separately for each path.

Most theorem provers can decide the more general problem of linear arithmetic. Linear arithmetic permits predicates of the form $\sum_{i=1}^{n} a_i v_i \rhd a_{n+1}$ (the coefficients $a_1 \ldots a_{n+1}$ are constants). They usually apply variable elimination methods, most notably the Fourier-Motzkin technique [3], which is used in PVS, ICS , IMPS and others. Other approaches include the graph-theoretic analysis due to Shostak [10], the Simplex method, the Sup-Inf method, and more. All of these methods, however, need to be combined with case-splitting in order to handle disjunctions. Normally this is the bottleneck of the decision process, since the number of sub-problems that need to be solved is worst case exponential. One may think of case-splitting as a two steps algorithm: first, the formula is converted to Disjunctive Normal Form (DNF); second, each clause is solved separately. Thus, the complexity of this problem is dominated by the size of the generated DNF. For this reason modern theorem provers try to refrain from explicit case-splitting. They apply 'lazy' case-splitting (splitting only when encountering a disjunction) that only in the worst case generates all possible subformulas as described above. One exception to the need for case splitting in the presence of disjunctions is DDDs. DDDs do not require explicit case-splitting, in the sense that the DDD data structure allows term sharing. Yet the number of sub-problems that are solved can still be exponential.

Reducing the problem to deciding a propositional formula (SAT) obviously does not avoid the potential exponential blow-up. The various branching algorithms used in SAT solvers can also be seen as case-splitting. But there is a difference between applying case-splitting to formulas and splitting the domain. While the former requires an invocation of a (theory-specific) procedure for deciding each case considered, the second is an instantiation of the formula with a finite number of assignments. Thus, the latter amounts to checking whether all clauses are satisfied under one of these assignments.

This difference, we now argue, is the reason for the major performance gap between CNF - SAT solvers and alternative decision procedures that have the same theoretical complexity. We will demonstrate the implications of this difference by considering three important mechanisms in decision procedures: *pruning*, *learning* and *guidance*. In the discussion that follows, we refer to the techniques applied in the Chaff [8] SAT solver. Most modern SAT solvers work according to similar principles.

- *Pruning.* Instantiation in SAT solvers is done by following a binary decision tree, where each decision corresponds to choosing a variable and assigning it a Boolean value. This method makes it very easy to apply pruning: once it discovers a contradictory partial assignment a, it backtracks, and consequently all assignments that contain a are pruned. It is not clear whether an equivalent or other pruning techniques can be applied in case-splitting over formulas, other than stopping when a clause is evaluated to true (or false, if we check validity).
- *Learning.* Every time a conflict (an unsatisfied clause) is encountered by Chaff, the partial assignment that led to this conflict is recorded, with the aim of preventing the same partial assignment from being repeated. In other words, all assignments that contain a 'bad' sub-assignment that was encountered in the past are pruned. Learning is applied in different ways in other decision procedures as well. For example, PVS records sub-goals it has proven and adds them as an antecedent to yet unproven sub-goals, with the hope it will simplify their proofs. In regard to separation theory, we are not aware of a specific learning mechanism, but it's not hard to think of one. Our argument in this case is therefore not that learning is harder or impossible in other decision procedures - rather that by reducing problems to SAT, one benefits from the existing learning techniques that were already developed and implemented over the years.
- *Guidance.* By 'guidance' we mean prioritizing the internal steps of the decision procedure. For example, consider the formula $\varphi_1 \vee \varphi_2$, where φ_1 is unsatisfiable and hard to solve, and φ_2 is satisfiable but easy to solve. If the clauses are solved from left to right, solving the above formula will take

longer than solving $\varphi_2 \vee \varphi_1$. We experimented with several such formulas in both ICS and PVS, and found that changing the order of expressions can have a significant impact on performance, which means that guidance is indeed problematic in the general case.

The success of guidance depends on the ability to efficiently estimate how hard it is to process each sub formula and/or to what extent it will simplify the rest of the proof. Both of these measures are easy to estimate in CNF-SAT solving, and hard to estimate when processing more general sub formulas. Guidance in SAT is done when choosing the next variable and Boolean value in each level in the decision tree. There are many heuristics for making this choice. For example: choose the variable and assignment that satisfies the largest number of clauses. Thus, the hardness of what will remain to prove after each decision is estimated by the number of unsatisfied clauses.

Not only that these mechanisms are harder to integrate in the alternative procedures, they become almost impossible to implement in the presence of mixed theories (what can be learned from solving a sub-goal with e.g. bit-vectors that will speed up another sub-goal with linear arithmetic, even if both refer to the same variables?). This is why reducing mixed theories to a common theory like propositional logic makes it easier to enjoy the potential speedup gained by these techniques. Many decidable theories that are frequently encountered in verification have known efficient reductions to propositional formulas. Therefore a similar reduction from separation predicates broadens the logic that can be decided by solving a single SAT instance.

3 A graph theoretic approach

Let φ be a formula consisting of the standard propositional connectives and predicates of the form $v_i \rhd v_j + c$ and $v_i \rhd c$, where c is a constant, and v_i, v_j are variables of type `real` (we treat integer variables in Section 5). We decide φ in three steps, as described below. A summary of the procedure and an example will be given in Section 3.4.

3.1 Normalizing φ

As a first step, we normalize φ.

1. Rewrite $v_i \rhd c$ as $v_i \rhd v_0 + c$.[2]
2. Rewrite equalities as conjunction of inequalities.
3. Transform φ to Negation Normal Form (NNF), i.e., negations are allowed only over atomic predicates, and eliminate negations by reversing inequality signs[3].

[2] $v_0 \notin \varphi$ can be thought of as a special variable that always has a coefficient '0' (an idea adopted from [10]).

[3] This step is only required for the integer case, described in Section 5.

4. Rewrite '$<$' and '$\leq$' predicates as '$>$' and '$\geq$', e.g., rewrite $v_i < v_j + c$ as $v_j > v_i - c$.

The normalized formula has no negations, and all predicates are of the form $v_i > v_j + c$ or $v_i \geq v_j + c$

3.2 Boolean encoding and basic graph construction

After normalizing φ, our decision procedure abstracts all predicates by replacing them with new Boolean variables. By doing so, the implicit transitivity constraints of these predicates are lost. We use a graph theoretic approach to represent this 'lost transitivity' and, in the next step, to derive a set of constraints that compensate for this loss.

Let $G_\varphi(V, E)$ be a weighted directed multigraph, where every edge $e \in E$ is a 4-tuple (v_i, v_j, c, x) defined as follows: v_i is the source node, v_j is the target node, c is the weight, and $x \in \{>, \geq\}$ is the type of the edge. We will denote by $s(e), t(e), w(e)$ and $x(e)$ the source, target, weight, and type of an edge e, respectively. We will also define the dual edge of e, denoted $\hat{e}$, as follows:

1. if $e = (i, j, c, >)$, then $\hat{e} = (j, i, -c, \geq)$.
2. if $e = (i, j, c, \geq)$, then $\hat{e} = (j, i, -c, >)$.

Informally, $\hat{e}$ represents the complement constraint of e. Thus, $\hat{\hat{e}} = e$.

We encode φ and construct G_φ as follows:

1. *Boolean encoding and basic graph construction*
   (a) Add a node for each variable in φ.
   (b) Replace each predicate of the form $v_i \rhd v_j + c$ with a Boolean variable $e^{c,\rhd}_{i,j}$, and add $(v_i, v_j, c, \rhd)$ to E.
2. *Add dual edges.*
   For each edge $e \in E$, $E := E \cup \hat{e}$.

The dual edges are needed only if φ was not transformed to NNF in step 3 of Section 3.1. In the rest of this section we assume that this is the case.

We denote the encoded Boolean formula by φ'. Since every edge in G_φ is associated with a Boolean variable in φ' (while its dual is associated with the negation of this variable), we will refer to edges and their associated variables interchangeably when the meaning is clear from the context.

3.3 Identifying the transitivity constraints

The transitivity constraints imposed by separation predicates can be inferred from previous work on this logic [9, 10]. Before we state these constraints formally, we demonstrate them on a simple cycle of size 2. Let $p1 : v_1 \rhd_1 v_2 + c_1$ and $p2 : v_2 \rhd_2 v_1 + c_2$ be two predicates in φ. It is easy to see that if $c_1 + c_2 > 0$ then $p1 \wedge p2$ is unsatisfiable. Additionally, if $c1 + c2 = 0$ and at least one of $\rhd_1, \rhd_2$ is equal to '$>$', then $p1 \wedge p2$ is unsatisfiable as well. The constraints on the other direction can be inferred by applying the above constraints to the duals of $p1$ and $p2$: if $c1 + c2 < 0$, or if $c1 + c2 = 0$ and at least one of $\rhd_1, \rhd_2$ is equal to '$<$', then $\neg p1 \wedge \neg p2$ is unsatisfiable.

We continue by formalizing and generalizing these constraints.

Definition 1. *A* directed path *of length m from v_i to v_j is a list of edges $e_1 ... e_m$ s.t. $s(e_1) = v_i$, $t(e_m) = v_j$ and $\forall_{i=1}^{m-1} t(e_i) = s(e_{i+1})$. A directed path is called* simple *if no node is repeated in the path.*

We will use capital letters to denote directed paths, and extend the notations $s(e)$, $t(e)$ and $w(e)$ to paths, as follows. Let $T = e_1 ... e_m$ be a directed path. Then $s(T) = s(e_1)$, $t(T) = t(e_m)$ and $w(T) = \sum_{i=1}^{m} w(e_i)$. $x(T)$ is defined as follows:

$$x(T) = \begin{cases} \geq & \text{if } \forall_{i=1}^{m} x(e_i) = \text{`}\geq\text{'} \\ > & \text{if } \forall_{i=1}^{m} x(e_i) = \text{`}>\text{'} \\ \sim & \text{otherwise} \end{cases}$$

We also extend the notation for dual edges to paths: if T is a directed path, then $\hat{T}$ is the directed path made of the dual edges of T.

Definition 2. *A* Transitive Sub-Graph (TSG) *$\mathcal{A} = T \cup B$ is a sub-graph comprised of two directed paths T and B, $T \neq B$, starting and ending in the same nodes, i.e., $s(T) = s(B)$ and $t(T) = t(B)$. $\mathcal{A}$ is called* simple *if both B and T are simple and the only nodes shared by T and B are $s(T) (= s(B))$ and $t(T) (= t(B))$.*

The transitivity requirements of a directed cycle[4] $\mathcal{C}$ and a TSG $\mathcal{A}$ are presented in Fig. 1. These requirements can be inferred from previous work on this logic, and will not be formally proved here.

	$x(\mathcal{C})$	*Rules*
l_1 :	'$\geq$'	**R1, R2**
l_2 :	'$>$'	**R3, R4**
l_3 :	*else*	**R2, R3**

R1 : if $w(\mathcal{C}) > 0$, $\bigwedge_{e_i \in \mathcal{C}} e_i = 0$

R2 : if $w(\mathcal{C}) \leq 0$, $\bigvee_{e_i \in \mathcal{C}} e_i = 1$

R3 : if $w(\mathcal{C}) \geq 0$, $\bigwedge_{e_i \in \mathcal{C}} e_i = 0$

R4 : if $w(\mathcal{C}) < 0$, $\bigvee_{e_i \in \mathcal{C}} e_i = 1$

(a) *Cycles*

	$x(T)$	$x(B)$	*Rules*
l'_1 :	'$\geq$'	'$>$'	**R1', R2'**
l'_2 :	'$>$'	'$\geq$'	**R3', R4'**
l'_3 :	*else*		**R2', R3'**

R1' : if $w(T) > w(B)$, $\bigwedge_{e_i \in T} e_i \rightarrow \bigvee_{e_j \in B} e_j$

R2' : if $w(T) \leq w(B)$, $\bigwedge_{e_i \in B} e_i \rightarrow \bigvee_{e_j \in T} e_j$

R3' : if $w(T) \geq w(B)$, $\bigwedge_{e_j \in T} e_j \rightarrow \bigvee_{e_i \in B} e_i$

R4' : if $w(T) < w(B)$, $\bigwedge_{e_i \in B} e_i \rightarrow \bigvee_{e_j \in T} e_j$

(b) *Transitive sub-graphs*

Fig. 1. Transitivity requirements of cycles (a) and transitive sub-graphs (b)

[4] By a 'directed cycle' we mean a closed directed path in which each sub-cycle is iterated once. It is obvious that iterations over cycles do not add transitivity constraints.

Both sets of rules have redundancy due to the dual edges. For example, each cycle $\mathcal{C}$ has a dual cycle $\hat{\mathcal{C}}$ with an opposite direction and $w(\mathcal{C}) = -w(\hat{\mathcal{C}})$. Applying the four rules to both cycles will yield exactly the same constraints. We can therefore consider cycles in one direction only. Alternatively, we can ignore **R3** and **R4**, since the first two rules yield the same result when applied to the dual cycle. Nevertheless we continue with the set of four rules for ease of presentation.

Definition 3. *A cycle $\mathcal{C}$ (alternatively, a* TSG *$\mathcal{A}$) is* satisfied by assignment α, *denoted $\alpha \models \mathcal{C}$, if α satisfies its corresponding constraints as defined in Fig. 1.*

We will denote by $\alpha(e)$ the Boolean value assigned to e by an assignment α. We will use the notation $\alpha \not\models_i \mathcal{C}$, $1 \leq i \leq 4$, to express the fact that rule **R***i* is applied to $\mathcal{C}$ and is not satisfied by α.

Proposition 1. *Let $\mathcal{A} = T \cup B$ and $\mathcal{C} = T \cup \hat{B}$ be a* TSG *and a directed cycle in G_φ, respectively. Then $\alpha \models \mathcal{A}$ iff $\alpha \models C$.*

(Proofs for all propositions in this article can be found in the full version of this article [13]).

Example 1. We demonstrate the duality between TSG's and cycles with a cycle $\mathcal{C}$ where $x(\mathcal{C})$ ='>' and $w(\mathcal{C}) > 0$ (Fig. 2(a)). Assume α assigns 1 to all of $\mathcal{C}$ edges, i.e., $\alpha(\mathcal{C}) = 1$. Consequently, $\alpha \not\models_3 \mathcal{C}$.

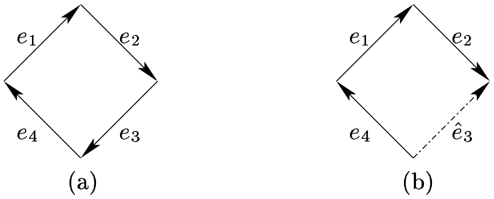

Fig. 2. A cycle (a) and a possible dual transitive sub-graph (b). Solid edges represent strict inequality (>) while dashed edges represent weak inequalities ('≥').

We construct $\mathcal{A}$ from $\mathcal{C}$ by substituting e.g., e_3 with its dual (Fig. 2(b)). $\mathcal{A}$ is a TSG made of the two directed paths $T = e_4, e_1, e_2$ and $B = \hat{e_3}$, that satisfy $x(T)$ ='>', $x(B)$ = '≥' and $w(T) > w(B)$ (because $w(B) = -w(e_3)$). According to Fig. 1(b), we apply **R3'** and **R4'**. But since $\alpha(\hat{e_3}) = \neg\alpha(e_3) = 0$, **R3'** is not satisfied. Thus, $\alpha \not\models_{3'} \mathcal{A}$. □

Proposition 1 implies that it is sufficient to concentrate on either TSG's or cycles. In the rest of this paper we will concentrate on cycles, since their symmetry makes them easier to handle.

The following proposition will allow us to concentrate only on *simple* cycles.

Proposition 2. *Let $\mathcal{C}$ be a non simple cycle in G_φ, and let α be an assignment to $\mathcal{C}$ edges. If $\alpha \not\models \mathcal{C}$ then there exists a sub-graph of $\mathcal{C}$ that forms a simple cycle $\mathcal{C}'$ s.t. $\alpha \not\models \mathcal{C}'$.*

Thus, our decision procedure adds constraints to φ' for every simple cycle in G_φ according to Fig. 1(a).

3.4 A decision procedure and its complexity

To summarize this section, our decision procedure consists of three stages:

1. Normalizing φ. After this step the formula contains only the '$>$' and '$\geq$' signs.
2. Deriving φ' from φ by encoding φ's predicates with new Boolean variables. Each predicate adds an edge and its dual to the inequality graph G_φ, as explained in Section 3.2
3. Adding transitivity constraints for every simple cycle in $G\varphi$ according to Fig. 1(a).

Example 2. Consider the formula $\varphi : x > y - 1 \vee \neg(z > y - 2 \wedge x \geq z)$. After step 2 we have $\varphi' : e^{-1,>}_{x,y} \vee \neg(e^{-2,>}_{z,y} \wedge \neg e^{0,>}_{z,x})$ (for simplicity we only refer to strict inequality predicates in φ', while the weak inequality predicates are referred to by a negation of their duals). Together with the dual edges, G_φ contains one cycle with weight 1 consisting of the vertices x, y, z, and the dual of this cycle. Considering the former, according to **R3** we add to φ' the constraint $\neg e^{-1,>}_{x,y} \vee \neg(\neg e^{-2,>}_{z,y}) \vee \neg e^{0,>}_{z,x}$. The constraint on the dual cycle is equivalent and is therefore not computed. □

This example demonstrates that the suggested procedure may generate redundant constraints (yet none of them makes the procedure incomplete). There is no reason to consider cycles that their edges are not conjoined in the DNF of φ. In [12] we prove this observation and explain how the above procedure can be combined with *conjunctions matrices* in order to avoid redundant constraints. The conjunction matrix of φ is a $|E| \times |E|$ matrix, computable in polynomial time, that state for each pair of predicates in φ whether they would appear in the same clause if the formula was transformed to DNF. This information is sufficient for concluding whether a given cycle ever appears in a DNF clause. Only if the answer is yes, we add the associated constraint. We refer the reader to the above reference for more details on this improvement (note that the experiments in Section 6 did not include this optimization).

Complexity. The complexity of enumerating the constraints for all simple cycles is linear in the number of cycles. There may be an exponential number of such cycles. Thus, while the number of variables is fixed, the number of constraints can be exponential (yet bounded by $2^{|E|}$). SAT is exponential in the number of variables and linear in the number of constraints. Therefore the complexity of the SAT checking stage in our procedure is tightly bounded by $O((2^{|E|})^2) = O(2^{2|E|})$, which is similar to the complexity of the Bellman-Ford procedure combined with case-splitting. The only argument in favor of our method is that in practice SAT solvers are less sensitive to the number of variables, and are more affected by the connectivity between them. The experiments detailed in Section 6 proves that this observation applies at least to the set of examples we tried. The SAT phase was never the bottleneck in our experiments; rather it was the generation of the formula.

Thus, the more interesting question is whether the cycle enumeration phase is easier than case splitting, as both are exponential in $|E|$. The answer is that normally there are significantly more clauses to derive and check than there are cycles to enumerate. There are two reasons for this: first, the same cycles can be repeated in many clauses; second, in satisfiable formulas many clauses do not contain a cycle at all.

4 Compact representation of transitivity constraints

Explicit enumeration of cycles will result in 2^n constraints in the case of Fig. 3(a), regardless of the weights on the edges. In many cases this worst case can be avoided by adding more edges to the graph. The general idea is to project the information that is contained in a directed path (i.e., the accumulated weight and type of edges in the path) to a single edge. If there are two or more paths that bear the same information, the representation will be more compact. In Section 4.2 we will elaborate on the implication of this change on the complexity of the procedure.

4.1 From cycles to triangles

The main tool that we will use for deriving the compact representation is *chordal graphs.* Chordal graphs (a.k.a. triangulated graphs) are normally defined in the context of undirected, unweighted graphs. A chordal graph in that context is a graph in which all cycles of size 4 or more contain an internal chord (an edge between non adjacent vertices). Chordal graphs were used in [4] to represent transitivity constraints (of equality, in their case) in a concise way. We will use them for the same purpose. Yet, there are several aspects in which G_φ is different from the graph considered in the standard definition: G_φ is a directed multigraph with two types of edges, the edges are weighted and each one of them has a dual.

Definition 4. *Let $\mathcal{C}$ be a simple cycle in G_φ. Let v_i and v_j be two non adjacent nodes in $\mathcal{C}$. We denote the path from v_i to v_j by $T_{i,j}$. A chord e from v_i to v_j is called $T_{i,j}$-*accumulating *if it satisfies these two requirements:*

1. *$w(e) = w(T_{i,j})$*
2. *$x(e) =$ '$\geq$' if $x(T_{i,j}) =$ '$\geq$' or if $x(T_{i,j}) =$ '$\sim$' and $x(T_{j,i}) =$ '$>$'. Otherwise $x(e) =$ '$>$'.*

This definition refers to the case of one path between i and j, and can be easily extended if there is more than one such path. Note that the definition of $x(e)$ relies on $x(T_{j,i})$, which is based on the edges of the 'other side' of the cycle. Since there can be more than one path $T_{j,i}$, and each one can have different types of edges, making the graph chordal may require the addition of two edges between i and j, corresponding to the two types of inequality signs. As will be shown in Section 4.2, our decision procedure refrains from explicitly checking all the paths $T_{j,i}$. Rather it adds these two edges automatically when $x(T_{i,j})$ ='$\sim$'.

Definition 4 gives rise to the following observation:

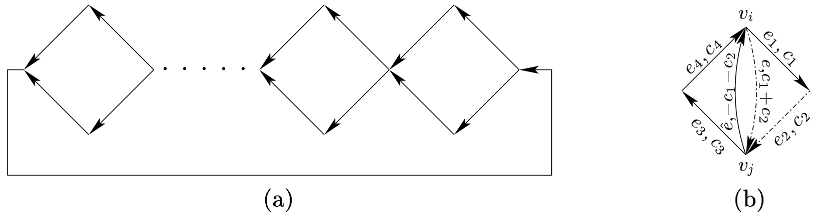

Fig. 3. (a) In a closed n-diamonds shape there are 2^n simple cycles. (b) The edge e accumulates the path $T_{i,j} = (e_1, e_2)$.

Proposition 3. *Let e be a $T_{i,j}$-accumulating chord in a simple cycle $\mathcal{C}$, and let $\mathcal{C}' = (\mathcal{C} \cup e) \setminus T_{i,j}$. The following equivalencies hold: $x(\mathcal{C}) = x(\mathcal{C}')$ and $w(\mathcal{C}) = w(\mathcal{C}')$.*

Example 3. In Fig. 3(b), each edge is marked with its identifier e_i and weight c_i. By Definition 4, e is a $T_{i,j}$-accumulating chord. Let $\mathcal{C}' = (\mathcal{C} \cup e) \setminus T_{i,j} = (e, e_3, e_4)$. Then as observed in Proposition 3, $x(\mathcal{C}') = x(\mathcal{C})$ ='$\sim$' and $w(\mathcal{C}') = w(\mathcal{C}) = \Sigma_{i=1}^{4} c_i$. □

Definition 5. *G_φ is called* chordal *if all simple cycles in G_φ of size greater or equal to 4 contain an accumulating chord.*

We leave the question of how to make G_φ chordal to the next section. We first prove the following proposition:

Proposition 4. *Let $\mathcal{C}$ be a simple cycle in a chordal graph G_φ, and let α be an assignment to the edges of $\mathcal{C}$. If $\alpha \not\models \mathcal{C}$ then there exists a simple cycle $\mathcal{C}'$ of size 3 in G_φ s.t. $\alpha \not\models \mathcal{C}'$.*

4.2 The enhanced decision procedure and its complexity

Based on the above results, we change the basic decision procedure of Section 3. We add a stage for making the graph chordal, and restrict the constraints addition phase to cycles of size 3 or less:

1. In the graph construction stage of Section 3.2, we add a third step for making the graph chordal:
   3. *Make the graph chordal.*
      While $V \neq \emptyset$
      (a) Choose an unmarked vertex $i \in V$ and mark it.
      (b) For each pair of edges $(j, i, c_1, x_1), (i, k, c_2, x_2) \in E$, where j and k are unmarked vertices and $j \neq k$:
         - Add $(j, k, c_1 + c_2, x_1)$ and its dual to E.
         - If $x1 \neq x2$, add $(j, k, c_1 + c_2, x_2)$ and its dual to E.
2. Rather than enumerating constraints for all simple cycles, as explained in Section 3.3, we only concentrate on cycles of size 2 and 3.

Various heuristics can be used for deciding the order in which vertices are chosen in step 3(a). Our implementation follows a greedy criterion: it removes the vertex that results in the minimum number of added edges.

Proposition 5. *The graph G_φ, as constructed in step 3, is chordal.*

Although Definition 5 requires accumulating chords in cycles larger than 3, the above procedure adds accumulating chords in triangles as well (i.e., one of the edges of the triangle accumulates the other two). It can be shown that it is sufficient to constrain these cycles (rather than all cycles of size 3) and cycles of size 2. With this improvement, the number of constraints becomes linear in the number of added edges. We skip the formalization and proof of this improvement due to space restrictions.

We now have all the necessary components for proving the soundness and the completeness of this procedure:

Proposition 6. *φ is satisfiable if and only if φ' is satisfiable.*

Complexity. In the worst case the process of making the graph chordal can add an exponential number of edges. Together with the complexity of SAT, it makes the procedure double exponential. However, in many cases it can reduce the complexity: consider, for example, a graph similar to the one in Fig. 3(a), where all the diamonds are 'balanced', i.e., the accumulated weight of the top and bottom paths of each diamond are equal (for example, in the frequently encountered case where all weights are equal to '0'). In this case the number of added edges is linear in n. Thus, in this case the size of the formula and the complexity of generating it is smaller than in the explicit enumeration method of Section 3.

5 Integer domains

In our discussion so far we assumed that all variables in the formula are of type `real`. We now extend our analysis to *integer separation predicates*, i.e., predicates of the form $v_i \rhd v_j + c$, where v_i and v_j are declared as integers (predicates involving both types of variables are assumed to be forbidden). We add a preprocessing stage right after φ is normalized:

1. Replace all integer separation predicates of the form $v_i \rhd v_j + c$ where c is not an integer with $v_i \geq v_j + \lceil c \rceil$.
2. For each integer predicate of the form $v_i > v_j + c$, add to φ the constraint $v_i > v_j + c \rightarrow v_i \geq v_j + c + 1$

The procedure now continues as before, assuming all variables are of type `real`.

Example 4. Consider the unsatisfiable formula $\varphi : x > y + 1.2 \wedge y > x - 2$ where x and y are integers. After the preprocessing step $\varphi : x \geq y + 2 \wedge y > x - 2 \wedge (y > x - 2 \rightarrow y \geq x - 1)$. □

The following proposition justifies the preprocessing stage:

Proposition 7. *Let φ^I be a normalized combination of integer separation predicates, and let φ^R be the result of applying the preprocessing stage to φ^I. Then φ^I is satisfiable iff φ^R is satisfiable.*

6 Experimental results

To test whether checking the encoded propositional formula φ' is indeed easier than checking the original formula φ, we generated a number of sample formulas and checked them before and after the encoding. We checked the original formulas with the ICS theorem prover, and checked the encoded formula φ' with the SAT solver Chaff [8].

First, we generated formulas that have the 'diamond' structure of Fig. 3(a), with D conjoined diamonds. Although artificial examples like this one are not necessarily realistic, they are useful for checking the decision procedure under controlled conditions. Each diamond had the following properties: the top and bottom paths have S conjoined edges each; the top and bottom paths are disjointed; the edges in the top path represent strict inequalities, while the edges in the bottom path represent weak inequalities. Thus, there are 2^D simple conjoined cycles, each of size $(D \cdot S + 1)$.

Example 5. The formula below represents the diamond structure that we used in our benchmark for $S = 2$. For better readability, we use the notation of edges rather than the one for their associated Boolean variables. We denote by $t_i^j (b_i^j)$ the j^{th} node in the top (bottom) path of the i^{th} diamond. Also, for simplicity we chose a uniform weight c, which in practice varied as we explain below.
$\bigwedge_{i=1}^{D}((v_i, t_i^1, c, >) \wedge (t_i^1, v_{i+1}, c, >) \vee (v_i, b_i^1, c, \geq) \wedge (b_i^1, v_{i+1}, c, \geq)) \wedge (v_{i+1}, v_1, c, >)$
□

By adjusting the weights of each edge, we were able to control the difficulty of the problem: first, we guaranteed that there is only one satisfying assignment to the formula, which makes it more difficult to solve (e.g., in Example 5, if we assign $c = -1$ for all top edges, and $c = (D - 1)$ for all bottom edges, and $c = S \cdot D - 1$ for the last, closing edge, only the path through the top edges is satisfiable); second, the weights on the bottom and top paths are uniform (yet the diamonds are not balanced), which, it can be shown, causes a quadratic growth in the number of added edges and constraints. This, in fact, turned out to be the bottleneck of our procedure. As illustrated in the table, Chaff solved all SAT instances in negligible time, while the procedure for generating the CNF formula (titled 'CNF') became less and less efficient. However, in all cases except the last one, the combined run time of our procedure was faster than the three theorem provers we experimented with. The table in Fig. 4 includes results for 7 cases. The results clearly demonstrate the easiness of solving the propositional encoding in comparison with the original formula. As a more realistic test, we experimented with formulas that are generated in hardware verification problems. To generate these formulas we used the UCLID verification tool [5]. These hardware models include a load-store unit from an industrial microprocessor, an out-of-order execution unit, and a cache coherence protocol. The formulas were generated by symbolically simulating the models for several steps starting from an initial state, and checking a safety property at the end of each step. Fig. 5(a) summarizes these results. Finally, we also solved formulas generated during symbolic model checking of timed systems. These examples are derived from a railroad crossing gate controller that is commonly used in the timed systems literature. Fig. 5(b) shows the results for these formulas.

Topology			Separation		
D	S	ICS	CNF	Chaff	Total
3	2	< 1	< 1	< 1	< 1
4	2	5.9	< 1	< 1	< 1
5	2	95.1	< 1	< 1	< 1
7	4	*	< 1	< 1	< 1
100	5	*	32	< 1	33
250	5	*	754	1.6	756
500	5	*	*		

Fig. 4. Results in seconds, when applied to a diamond-shaped graphs with D diamonds, each of size S. '*' denotes run time exceeding 10^4 sec.

Acknowledgments We thank S. German for giving us the cache-protocol example, and S. Lahiri for helping with the experimens. The first author also wishes to thank D. Kroening for his guidance through the maze of algorithms that various theorem provers use.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The

			Separation		
Model	Steps	ICS	CNF	Chaff	Total
Load-Store unit	1	< 1	< 1	< 1	< 1
	2	87.1	< 1	< 1	< 1
	3	*	90	1	91
Out-of-order unit	2	< 1	< 1	< 1	< 1
	3	*	2.9	< 1	3
Cache protocol	1	< 1	< 1	< 1	< 1
	2	1.8	< 1	< 1	< 1

(a)

		Separation		
Model	ICS	CNF	Chaff	Total
RailRoad-2	52	< 1	< 1	< 1
RailRoad-12	15.2	< 1	< 1	< 1
RailRoad-13	189	< 1	< 1	< 1
RailRoad-14	49.6	< 1	< 1	< 1

(b)

Fig. 5. Results in seconds, when applied to formulas generated by symbolically simulating several hardware designs (a) and symbolic model checking of timed systems(b).

Coq Proof Assistant Reference Manual – Version V6.1. Technical Report RT-0203, INRIA, August 1997. revised version distributed with Coq.

3. A.J.C. Bik and H.A.G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.
4. R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, 2001.
5. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV'02)*, July 2002. This volume.
6. T. Cormen, C. Leiserson, and L. Rivest. *Introduction to Algorithms.* MIT press.
7. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Conference on Computer Science Logic*, volume 1683 of *LNCS*, pages 111–125, 1999.
8. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. Design Automation Conference (DAC'01)*, 2001.
9. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute os Technology, 1977. Cambridge, Mass.
10. R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, October 1981.
11. R. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.
12. O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.
13. O. Strichman, S.A.Seshia, and R.E.Bryant. Reducing separation formulas to propositional logic. Technical Report CMU-CS-02-132, Carnegie Mellon University, 2002.