Some Progress in Satisfiability Checking for Difference Logic*

Scott Cotton[1], Eugene Asarin[2], Oded Maler[1] and Peter Niebert[3]

[1] VERIMAG, 2 Av. de Vignate, 38610 Gières, France
[Scott.Cotton | Oded.Maler]@imag.fr
[2] LIAFA, Université Paris 7, 2 place Jussieu, 75251 Paris, France,
asarin@liafa.jussieu.fr
[3] Laboratoire d'Informatique Fondamentale, CMI, 39 rue Joliot-Curie
13453 Marseille Cedex 13, France, niebert@cmi.univ-mrs.fr

Abstract. In this paper we report a new SAT solver for *difference logic*, a propositional logic enriched with timing constraints. The main novelty of our solver is a tighter integration of the incremental analysis of numerical conflicts with the process of Boolean conflict analysis. This and other improvements lead to significant performance gains for some classes of problems.

1 Introduction

The development of increasingly stronger Boolean satisfiability (SAT) solvers such as [MS99,MMZ+01,GN02] made satisfiability checking an important ingredient in verification and synthesis of finite-state systems. Recently there is a growing interest in extending the scope of SAT-based methods to reason about systems admitting variables ranging over infinite domains such as integers and reals. To this end, new satisfiability checking methods should be developed for propositional logic extended with numerical constraints that are rich enough to capture the dynamics (transition relation) of the systems in question.

Difference logic, also known as *separation logic*, is one of the simplest extensions of propositional logic which has recently attracted a lot of attention. In addition to propositional variables, the atoms of this logic consist of inequalities of the form $x - y < c$ for real-valued variables x, y and an integer constant c. The popularity of this logic is due to the following: 1) It is rich enough to express bounded reachability for timed automata, feasibility of scheduling problems, existence of paths in digital circuits with bounded delays and other timing related problems; 2) The satisfiability of a conjunction of difference constraints can be reduced to the absence of negative cycles in finite weigthed graphs, a procedure more efficient than general linear (and, of course non-linear) constraints satisfaction.

In the last couple of years, several groups developed independently solvers for DL [ACG99,MNAM02,S02,NMA+02,F02,ACKS02,SSB02,WZP03] or for richer logics that contain it [ABC+02,MRS02,BDS+02]. These solvers use different approaches for

* This work was partially supported by the EC project IST-2001-35302 AMETIST (Advanced Methods for Timed Systems).

the crucial problem of managing the *interaction* between the *propositional* and *numerical* parts of the problem. In this work we introduce yet another solver, DLSAT, which is inspired by our previous solver MX-SOLVER reported in [MNAM02,NMA+02,M03] and also by some ideas in [SSB02]. The main novelty of this solver is in a more efficient algorithm for detecting numerical contradictions and in a tighter integration of this procedure with the conflict analysis and learning mechanisms used for the propositional part. We report some significant performance gains on some non toy problems. [4]

The rest of the paper is organized as follows. In Section 2 we define DL, and briefly present the process of Boolean SAT solving. In Section 3 we discuss the various approaches for combining propositional and numerical satisfiability and position our approach in this landscape. In Section 4 we discuss the process of discovering numerical contradictions using negative cycles detection in weighted graphs and present our procedure based on Goldberg's heuristic improvement of the Bellman-Ford algorithm [GR93]. Additional implementation details are described in Section 5 followed by experimental results and a discussion of future work.

2 Preliminaries

2.1 Difference Logic

Definition 1 (Difference Logic). *Let $\mathcal{P} = \{p_1, p_2, \ldots p_n\}$ be a set of propositional (Boolean) variables and $\mathcal{X} = \{x_1, x_2, \ldots x_m\}$ be a set of numerical variables. The set of atomic formulae of $DL(\mathcal{P}, \mathcal{X})$ consists of the propositions in $\mathcal{P}$ and numerical constraints of the following forms:*

$$x_i - x_j \leq c \quad \text{and} \quad x_i - x_j < c$$

with $c \in \mathbb{Z}$. The set $\mathcal{F}$ of all DL formulae is the smallest set containing the atomic formulae which is closed under negation and conjunction:

- $\varphi \in \mathcal{F}$ implies $\neg\varphi \in \mathcal{F}$.
- $\varphi \in \mathcal{F}$ and $\psi \in \mathcal{F}$ implies $\varphi \wedge \psi \in \mathcal{F}$.

Remaining Boolean connectives $\vee, \wedge, \rightarrow, \ldots$ may be defined in the usual ways in terms of conjunction and negation.

A $(\mathcal{P}, \mathcal{X})$-*valuation* consists of two functions (overloaded with the name v) $v : \mathcal{P} \rightarrow \{\mathrm{T}, \mathrm{F}\}$ and $v : \mathcal{X} \rightarrow \mathbb{R}$. The valuation v is extended to all $DL(\mathcal{P}, \mathcal{X})$ formulae by letting

$$v(x_i - x_j \leq c) = \mathrm{T} \text{ iff } v(x_i) - v(x_j) \leq c$$

and applying the obvious rules for the Boolean connectives. A partial valuation v is a valuation defined over a subset of the variables. We denote by $\varphi[v]$ the formula obtained from φ by substituting $v(p)$ and $v(x)$ (when they are defined) in p and x, respectively. A formula φ is satisfied by a valuation v iff $v(\varphi) = \mathrm{T}$ (we denote it also by $v \models \varphi$). A formula φ is satisfiable if it has a satisfying valuation. The satisfiability problem for $DL(\mathcal{P}, \mathcal{X})$ is NP-complete.

Like most SAT solvers we work with formulae in conjunctive normal form:

[4] I.e. no Fisher's protocol.

Definition 2 (CNF). *A Boolean literal is a formula of the form p or $\neg p$ with $p \in \mathcal{P}$. A numerical literal is a formula of the form $x - y \leq c$ or $x - y < c$. A clause is a disjunction $C = L_1 \vee L_2 \vee \ldots L_l$ of literals. A DL formula is in CNF if it is a conjunction $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ of clauses.*

As in propositional logic (see [T70]), efficient translations from arbitrary formulae to CNF can be done using auxiliary Boolean variables.

2.2 Basics of SAT Solving

In order to be self-contained we sketch briefly the principles underlying contemporary SAT solvers as applied to Boolean satisfiability, a special case of DL with $\mathcal{X} = \emptyset$. The DPLL-based procedure for SAT can be seen as an intelligent search in the space of valuations, accompanied by formula simplification and learning. The search is conducted by iteratively generating partial valuations and extending them. Extension is performed with simplification rules which are applied after substituting the valuation in the variables. Literals are removed that evaluate to F. Clauses are removed that evaluate to T, and variables are inferred that appear in unit clauses. The result of simplification of a formula $\varphi[v]$ can thus be either T, F or an extended valuation v'. The search for a satisfying assignment should be continued from $\varphi[v']$.

Learning is a process, where after some $\varphi[v]$ simplifies to F, a subset v_{con} of v is identified as a sufficient cause for unsatisfiability, and its negation $\neg v_{con}$ is added as an additional clause to φ in order to prune the search tree and prevent exploration of partial assignments that extend v_{con}. The whole procedure is sketched below where the formula φ is a global variable and the partial valuation v is an argument to the recursive procedure $Solve$, initially called with $v = \emptyset$:

procedure *Solve*(v)

$(v', \varphi') :=$ *Simplify*$(\varphi[v])$
if $\varphi' = \mathrm{T}$
  return(yes)
else if $\varphi' = \mathrm{F}$
  $v_{con} :=$ *Conflict*(φ, v)
  $\varphi := \varphi \wedge \neg v_{con}$
  return(no)
else
  pick a variable p not appearing in v
  if *Solve*$(v \cup \{p = \mathrm{T}\})$
    return(yes)
  else
    return(*Solve*$(v \cup \{p = \mathrm{F}\})$)

Modern SAT solvers employ a myriad of optimizations and of course utilize an iterative version of this procedure. Since SAT solvers spend most of their time simplifying the formula [Zha95], a notable optimization is *two literal watching* [MMZ+01,ZH96], which reduces to two the number of literals the procedure must scan in a clause while

identifying whether a clause is solved, empty, or unit. Additionally, various variable ordering heuristics are employed, most of which are based on the frequency of variable occurrences in the formula. Finally, non chronological backtracking is often employed upon analyzing a conflict, in which the procedure either jumps back to the smallest assignment which leaves the conflicting clause unit. This process prevents the solver from searching a larger number unsatisfiable subtrees that it would otherwise.

Perhaps the most interesting point (and one which can yield a great degree of variance on the performance) is exactly what clause(s) a solver decides to learn upon coming across a conflict. All the different approaches make use of an *implication graph* in which literals are vertices, and those which are deduced via unit resolution are implied by the false literals in the unit clause. One of the more successful methods of examining this graph backtracks through the implications until it finds a *unique implication point*, or a literal l which lies between the literals in the empty clause and the guessed literals in such a way that every path from the guessed literals to those in the empty clause passes through l.

Taken together, these methods, heuristics, and optimizations have pushed the performance of Boolean SAT solvers up by a few orders of magnitude. On the other hand, the extension of these techniques to formulae with numeric atoms has not seen such improvements. In the next section, we will review some methods of leveraging and extending these techniqes for solving SAT for DL.

3 Approaches to DL SAT Solving

In the last couple of years several approaches for checking satisfiability of DL were introduced. The approach of [WZP03] is restricted to integer solutions (discrete time semantics). The clock variables are interpreted as integer variables encoded in binary, and the whole problem is transformed into a Boolean SAT problem on the bits of the numbers, which can then be submitted to one's favorite solver. This approach has been tried already in the context of BDD-based verification of timed automata [ABK$^+$97] and its disadvantage is that the arithmetical content of numerical constraints is lost when they are coded in binary. The rest of the approaches known to us can be classified into the following three categories:

3.1 The Lazy Approach

This approach, used for example in [ABC$^+$02], consists of transforming a formula φ into a purely Boolean formula φ' by replacing every numerical constraint of the form $x - y < c$ by a new propositional variable p_{xyc}. The formula φ' is "easier" to satisfy, because the new variable are not interpreted and the implications between them are not visible to the Boolean solver. Consequently if φ' is found to be unsatisfiable we can conclude that so is φ. On the other hand, whenever a satisfying assignment v' is found, an additional feasibility check should be performed to see whether v' can be transformed into an assignment v for φ. This is done by constructing a conjunction of all numerical constraints $x-y < c$ such that $p_{xyc} = \mathrm{T}$ and $\neg(x-y < c)$ such that $p_{xyc} = \mathrm{F}$ in v'. This conjunction is then submitted to a numerical solver and if it is feasible then

φ is satisfiable; otherwise v' is declared unsatisfying and the enumeration of satisfying assignments for φ' continues. Additionally, learning clauses from the feasibility checks and adding them to φ' is usually performed.

This approach is very easy to implement as, with the exception of learning, the Boolean and numerical parts are kept separate. This is also the reason for its practical weakness for problems with many numerical implications as there is a limited flow of information between these parts. Typically the algorithm can check many assignments to φ' until it finds one which can be transformed to an assignment for φ or until it concludes that no such assignment exists.

3.2 The Preprocessing Approach

This approach, first suggested in [SSB02] is the opposite one. Although it is also based on introducing propositional variables for the numerical constraints, it computes all the intrinsic dependencies between the numeric variables, encoding the dependencies as Boolean constraints and adding these constraints to φ'. The advantage of this approach is that, by construction, each assignment to the augmented formula can be transformed into a satisfying assignment for the original formula. There are two major shortcomings: 1) As noted in [SSB02], some classes of sets of numerical constraints lead to an exponential blow-up in the size of the formula when their implications are added; 2) This procedure may need to compute dependencies between numerical constraints mentioned in φ that, due to the structure of φ, will never have to be considered simultaneously.

3.3 Incremental Approaches

This class consists of approaches that try to check feasibility constraints somewhere between the above cited approaches. This is the case of our previous solver MX-SOLVER [NMA$^+$02] and also that of [MRS02] where it was called "lemmas on demand". In [NMA$^+$02] the following strategy is used: a conjunction of numerical constraints that need to be satisfied under the current assignment is maintained, represented as a DBM (Difference Bound Matrix). Whenever a Boolean variable p_{xyc} is assigned to T or F, its corresponding numerical constraint is added to the DBM, which is tested for feasibility using the Floyd-Warshall algorithm (more on the structure of difference constraints in the next section). If the set of constraints is found infeasible at a partial assignment v' , there is no need to explore extension of v'. This approach has an obvious advantage over the lazy approach which needs to wait until a satisfying assignment for φ' is found and an advantage over the preprocessing approach by not checking feasibility of all combinations of constraints, only those that need to be satisfied simultaneously in some explored branch of the search tree. However, in the absence of a learning mechanism, this expensive procedure is invoked too often.

3.4 Our Approach

The solver described in the present paper is another variant of the incremental approach with the following features:

1. An integrated solver that treats DL formulae and CNF Boolean formulae with two literal watching, conflict analysis (first unique implication point), as well as Boolean and numeric variable ordering heuristics.
2. Transformation to CNF is done using the more efficient construction of Wilson [W90] rather than the classical Tseitin translation [T70].
3. Equal treatment of Boolean and numerical literals in terms of branching (unlike [NMA$^+$02] where branching was applied only to Booleans).
4. Optimizations for reducing the number of feasibility checks: We omit feasibility checks upon assigning a truth value to a new numerical constraint in two cases: when all the clauses where the new constraint appears are already solved (simplified to true), and when the constraint involves a "new" numeric variable not mentioned in the constraints assigned so far.
5. A more efficient algorithm for checking feasibility of a conjunction of difference constraints. We detect cycles using a depth-first variant of the Bellman-Ford-Moore algorithm [GR93] which has much better average case complexity in practice.
6. Integration of difference constraint feasibility checks with the conflict analysis mechanism. Inconsistent sets of difference constraints are analyzed with respect to their implication graph, and the procedure learns a "reason" for the inconsistency.

In the next section we discuss numerical feasibility checks in general and present the algorithm that we use.

4 The Fine Structure of Sets of Difference Constraints

4.1 Feasibility

While conjunctions of difference constraints are a special case of linear inequalities, their structure is much simpler. We can describe the satisfiability problem of conjunctions of difference constraints using transitivity:

$$x - y < c_1 \wedge y - z < c_2 \Rightarrow x - z < c_1 + c_2$$

If a conjunction of difference constraints implies that $x - x < 0$ for some x, then it is infeasible, otherwise it is is feasible. It is sometimes illustrative to consider the case where each constant is 0, as all the difference constraints then take the form $x < y$. In particular in this case it is easy to see that any cycle $x < y < \ldots < z < x$ is false. In the more general case, we have that $(x_1 - x_2 < c_1) \wedge (x_2 - x_3 < c_2) \wedge \ldots \wedge (x_n - x_1 < c_n)$ is false just in case $\Sigma_{i=1}^{n} c_i \leq 0$.

One can easily express the negation of a difference constraint as a difference constraint: $\neg(x - y < c) \iff (x - y \geq c) \iff (y - x \leq -c)$. However, this situation requires that the logic allow for both strict and non strict constraints. As a result, we need to extend the notion of feasibility to accommodate strict and non strict constraints. Letting $\prec \in \{<, \leq\}$, the infeasibility condition becomes $(x_1 - x_2 \prec_1 c_1) \wedge (x_2 - x_3 \prec_2 c_2) \wedge \ldots \wedge (x_n - x_1 \prec_n c_n)$ is false just in case $\Sigma_{i=1}^{n} c_i < 0$ or $\Sigma_{i=1}^{n} c_i = 0$ and at least one $\prec_i$ is strict.

With such mixed constraints, it is convenient to speak of *bounds*, or pairs $(\prec, c)$, representing either the interval $(-\infty, c)$ or the interval $(-\infty, c]$. We refer to the set of

bounds as B. Additionally, we define an order $<_B$ on bounds with $(\prec, c) <_B (\prec', c')$ whenever $c < c'$ or when $c = c'$, $\prec$ is $<$ and $\prec'$ is $\leq$. Finally, we define addition for bounds with $(\prec, c) + (\prec', c') = (\prec'', c + c')$ with $\prec''$ strict just in case either $\prec$ or $\prec'$ are strict.

A natural data structure for describing sets of difference constraints is a *bound weighted graph* in which the vertices are variables and edges represent constraints between variables:

Definition 3 (Constraint Graph). *The constraint graph of a set (or conjunction) of difference constraints Γ is a graph $G = (V, E, \xi)$ with one vertex per numeric variable occurring in some difference constraint in Γ, edges $E = \{(x, y) : (x - y \prec c) \in \Gamma$ for some $(\prec, c) \in B\}$, and a function $\xi : E \to B$ defined by $(x, y) \mapsto min\{(\prec, c) : (x - y \prec c) \in \Gamma\}$.*

A well known data structure for storing a constraint graph is a Difference Bound Matrix (DBM), in which a $|V| \times |V|$ matrix stores the implied relationship between each pair of variables, and each cell x, y without an associated edge in the constraint graph is initialized to $(<, \infty)$.

4.2 Finding Shortest Paths and Negative Cycles

Under any constraint graph representation, a feasibility check reduces to the detection of a negative cycle, or any cycle where the edge bounds sum to a value less than $(\leq, 0)$. Normally negative cycles are detected as a side effect of a shortest path algorithm. Timed automata verification tools apply the Floyd-Warshall all pairs shortest path algorithm to normalize the set of constraints and the detection of negative cycles (which imply that the DBM is normalized to the empty set) is obtained as a side effect of the algorithm. Unlike the case of reachability computation for timed automata, where the DBMs are rather small, in DL solving, one can easily obtain sets with hundreds of numeric variables and a more efficient algorithm is needed. We will use a single source shortest path algorithm to do the feasibility checks.

Shortest path algorithms, either single-source or all-pairs, function by iteratively approximating the minimum distance between vertices, where a distance from x to y is taken to be the sum of the edge bounds on a path from x to y. In the context of a single source shortest path algorithm, the distance estimates are represented by a function $\delta : V \to B$ indicating a bound on the distance from a distinguished source vertex. Distance functions $(\delta_0, \delta_1, \ldots)$ are successively approximated by taking any edge (s, t) such that $\delta_i(s) + \xi(s, t) < \delta_i(t)$ and letting $\delta_{i+1}(t) = \min_s(\delta_i(s) + \xi(s, t))$. Following [GR93] we can use such a sequence of functions to filter out all the positive cycles whose edge bounds sum to a value greater than $(\leq, 0)$:

Definition 4. Given a distance function $\delta : V \to B$, an edge (s, t) such that $\delta(s) + \xi(s, t) \leq \delta(t)$ is called *admissible*. The *admissible subgraph* of a constraint graph G, written G_a, is the subgraph of G containing all of its admissible edges.

Proposition 1. Given a constraint graph G and a series of distance estimating functions $(\delta_0, \delta_1, \ldots)$, G has a negative or zero weight cycle if and only if G_a has a cycle under some distance estimate δ_k.

Proof. It is well known that the distance estimation will converge to a fixed point if and only if the graph has no negative cycle [CLRS01]. If it does not converge, then there is a stage k for which all $\delta_k(v) = \delta_{k+1}(v)$ for all $v \in V$ except some V', some of which are in negative cycles. Let $x_0 \rightarrow x_1 \ldots x_n \rightarrow x_0$ be such a cycle. Then $\delta(x_i) + \xi(x_i, x_{i+1}) < \delta(x_{i+1})$ for some [5] i and $\exists v' \in V' \,.\, v' \in \{x_0, \ldots, x_n\}$. For sufficiently large k, the remaining edges in the cycle must be admissible, for otherwise the process would converge. Hence G_a will contain the cycle.

Suppose the process converges at δ_k and G contains a cycle $x_0 \rightarrow x_1 \ldots x_n \rightarrow x_0$ whose edge bounds sum to $(\leq, 0)$. Since the process converges, $\delta(x_i) + \xi(x_i, x_{i+1}) \geq \delta(x_{i+1})$. But if $\delta(x_i) + \xi(x_i, x_{i+1}) > \delta(x_{i+1})$ then there must be some $j \neq i$ such that $\delta(x_j) + \xi(x_j, x_{j+1}) < \delta(x_{j+1})$ since the edge bounds sum to $(\leq, 0)$. However, this implies the process has not converged and we have arrived at a contradiction. We can then conclude that each edge in the cycle must satisfy $\delta(x_i) + \xi(x_i, x_{i+1}) \leq \delta(x_{i+1})$ and so the cycle is in G_a.

In the other direction, if there is a cycle in G_a then each edge in the cycle must satisfy $\delta(s) + \xi(s, t) \leq \delta(t)$. Hence $\Sigma_i \delta(x_i) + \xi(x_i, x_{i+1}) \leq \Sigma_i \delta(x_i)$, and so $\Sigma_i \xi(x_i, x_{i+1}) \leq (\leq, 0)$. ⌟

The problem of detecting negative cycles can be reduced to checking for cycles in G_a which contain an edge with a strict bound (in the form $(<, c)$) or an edge (s, t) such that $\delta(s) + \xi(s, t) < \delta(t)$. This in turn can be accomplished via a depth first search which keeps track of the location of such edges.

This algorithm provides a significant advantage over that of Floyd-Warshall used in [NMA+02]. On a graph with n nodes and m edges, Floyd-Warshall takes n^3 time for a full canonicalization and n^2 time for a single incremental update on *every* execution. Bellman Ford's runtime is bounded by nm, and in this setting typically runs in m time. Additionally, all the formulae associated with problems known to us have $m \ll 10n$. If we assume the graph is sparse with $m = 10n$, then our typical run requires less than $10n$ steps, our worst case run time is $10n^2$, a significant improvement over n^3 full canonicalization or even the n^2 incremental canonicalization on each run.

5 Implementation

The solver, including parsing and CNF translation, is implemented in 3500 lines of C++. Here we discuss some of the features of the solver.

5.1 Numeric Conflict Analysis

To explain numeric conflict analysis let us emphasize that such conflicts may appear in a richer set of circumstances than ordinary Boolean conflicts which always appear during simplification of the formula. Numerical conflicts can, in addition, appear when guessing or flipping a truth value for a numeric constraint. Our observations indicated that numeric conflicts arising outside of the simplification process occurred with less

[5] We take $i + 1$ modulo n.

frequency on the harder satisfiable problems, presumably because the number of clauses was sufficient to induce more simplification.

When the solver arrives at an inconsistent set of difference constraints, two types of analysis and learning are performed. First, a small such inconsistent set of constraints is identified and its negation is encoded as a clause. Second, if the numeric conflict appears as a result of simplification, the implication graph of the literals in the conflict clause is analyzed in order to find a reason for the inconsistency.

However the actual frequency with which the implication graphs of numericly induced conflict clauses were analyzed varied widely from never (for example in the scheduling problem FT06 described in 6.1) to roughly half (for example in the circuit problems described in 6.3). Nonetheless this process allows the solver to learn reasons for numeric conflicts which include settings of strictly Boolean variables (and vice versa). The analysis mechanism itself is a straightforward implementation of Chaff style [MMZ$^+$01] first unique implication point (1UIP) cut of the implication graph. As with Boolean SAT solving, experiments with other mechanisms such as a decision-only cut or performing analysis on all conflicts did not perform as well as the 1UIP scheme.

The introduction of such learning into the DPLL algorithm requires a minor but essential change to the conflict resolution mechanism as it occurs in the Boolean case. This can be explained as follows. The backtracking mechanism is responsible for finding a point in the search tree to jump back to. Once this point in found, the assignment for the variable v associated with this node in the search tree is flipped and the alternative subtree is explored. In the presence of numerical constraints, this assignment can of course introduce a numeric conflict. In the Boolean case, this assignment will not by itself introduce an empty clause, for otherwise $\neg v$ would have been deduced by unit resolution at a previous decision level, or higher in the search tree. Thus the conflict resolution process in the numeric case needs to be sensitive to conflicts which arise *as a result of resolving conflicts*. This suggests that conflict resolution can be made recursive for this case. However, our solver simply adds a clause representing the negation of such a numeric conflict to the clause database and continues backtracking, without analyzing the implication graph associated with the numeric conflict. As this case was rare in practice, this seems like a reasonable course of action.

5.2 Reducing Feasibility Checks

In an incremental setting, feasibility checks occur with high frequency. In problems dominated by numeric constraints this becomes a bottleneck. We employ two optimizations for reducing the number of required feasibility checks. First, we do not trigger feasibility checks when branching on difference constraints which do not solve any clauses. Thus at the end of the solving process, if the problem is satisfiable we will have checked that the difference constraints in a prime implicant of the problem are feasible. Second, we observe that a feasible set of difference constraints cannot be made infeasible by adding a single constraint which mentions an otherwise unconstrainted numeric variable. Our observations showed that these optimizations reduced the number of feasibility checks by one third on hard problems dominated by numeric constraints.

6 Experimental Results

In this section we report the performance results of DLSAT on several classes of benchmark problems.

6.1 Job Shop Scheduling

The problem of finding optimal schedules for the job shop problem [JM99] is a hard combinatorial optimization problem whose constraints express very naturally in DL. The optimization is converted into a decision (satisfiability) problem of the form "is there a schedule whose length is smaller than d?" As observed in [NMA$^+$02], when d is much larger or much smaller than the length of the optimal schedule for the problem, the solver finds the negative (resp. positive) answer quickly, but as we approach the optimum things become harder.

d / Solver	50	51	52	53	54	55	56	57	58	59	60
ICS	1.88	2.95	3.41	21.90	38.00	174.00	85.00	68.00	95.00	69.00	0.11
MX-SOLVER	0.14	0.14	0.14	1.79	7.67	21.47	1.31	0.20	0.92	1.88	0.21
DLSAT	0.09	0.10	0.12	0.24	0.29	0.69	0.86	0.50	0.69	0.36	0.37

Table 1. Comparison of DLSAT with ICS and MX-SOLVER on the FT06 problem whose optimal schedule is of length 55.

d / Solver	950	975	980	990	1000	1100	1234	1300	1390	1395	1400	1600	1800	3000
MX-SOLVER	4.6	4.7	4.9	1709.0	t/o	t/o	t/o	t/o	t/o	17.2	17.2	10.4	10.8	11.7
DLSAT	0.5	0.5	0.6	0.9	0.8	30.2	t/o	62.0	47.3	6.1	12.1	0.6	0.5	0.6

Table 2. Comparison of DLSAT with MX-SOLVER on the hard ABZ5 problem whose optimal schedule is of length 1234. Time is given in seconds and t/o means more than 5 minutes.

Table 1 compares the performance results of DLSAT with those of MX-SOLVER [NMA$^+$02] and ICS [FORS01] on the FT06 job shop problem with 6 machines and 6 jobs. The optimal schedule is of length 55 and DLSAT finds it within 0.69 seconds compared to 21.47 with MX-SOLVER and almost 3 minutes with ICS. This problem consists of 132 clauses, 222 difference constraints and 37 numeric variables. It is not surprising that ICS does not perform as well as MX-SOLVER or DLSAT because ICS uses a method for combining decision procedures and can hence solve more complex problems. Nevertheless we keep it as a point of reference for more general techniques. Table 2 compares DLSAT and MX-SOLVER on the hard ABZ5 problem [ABZ88] with

10 machines and 10 jobs whose optimum is 1234. Bounding execution time to 5 minutes for each query, MX-Solver can deduce that the optimum is somewhere in the interval $(990, 1395]$ while DLSAT can conclude that it is in $(1100, 1300]$. Yet none of them can find the exact optimum. The problem is made up of 560 clauses over 1010 difference constraints and 101 numeric variables.

Note that we restrict the comparison to MX-SOLVER because on this type of problems, dominated by numerical constraints, it already had a much better performance than solvers that use the lazy or preprocessing approaches.

6.2 Diamond Problems

The diamond problems were introduced by Strichman as benchmarks for the preprocessing approach. These are sets of difference constraints whose graphical representation is a series of diamond-like shapes with an additional back-edge from the last node to the first (see Figure 1). These graphs are parametrized by the length d of each side of the diamond and the number n of diamonds. The corresponding DL formula is generated to require the existence of a cycle in the diamond. Such graphs have $2nd + 1$ edges and 2^n cycles, hence they pose a challenge to numerical feasibility checks.

The problems are partitioned into three classes, unsatisfiable, satisfiable and tightly satisfiable, i.e. with only one satisfying assignment. As Table 3 shows, DLSAT is much superior to the preprocessing approach of [SSB02] on satisfiable instances, but much inferior on unsatisfiable ones. On the satisfiable problems, DLSAT only considers combinations of difference constraints necessary to get a satisfiable answer, and hence is able to reduce the required processing of the constraints to a level far below that of the preprocessing approach. On the other hand, the approach of [SSB02] acts relatively independently of the satisfiability of the problem, since it spends by far most of its time coding the DL formula into an equivalent Boolean one, rather than solving the resulting Boolean formula. Hence its performance remains relatively constant accross the satisfiable and unsatisfiable diamond problems. Also, as this coding process examines all the possible constraints in a single step rather than piecewise (as in DLSAT), it is able to globally analyze the structure of the set of all constraints and compress the representation of the resulting Boolean formula by chordalizing the graph which represents all the possible constraints. The advantage of this global analysis is more prominent in the unsatisfiable problems because there is no satisfying subset to which DLSAT can restrict its attention.

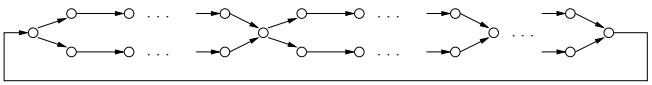

Fig. 1. A constraints graph having the form of diamond concatenation.

n	sat		one-sat		unsat	
	DLSAT	SEP	DLSAT	SEP	DLSAT	SEP
5	0.01	0.17	0.01	0.17	15.10	0.17
10	0.03	0.16	0.01	0.17	t/o	0.17
20	0.03	0.51	0.03	0.50	t/o	0.50
20	0.03	0.51	0.03	0.50	t/o	0.50
30	0.04	1.21	0.06	1.23	t/o	1.25
40	0.06	2.60	0.10	2.60	t/o	2.60
50	0.20	5.21	0.14	5.10	t/o	5.30
100	0.20	45.30	0.50	44.20	t/o	47.60
200	0.70	t/o	2.40	t/o	t/o	t/o
500	4.90	t/o	21.90	t/o	t/o	t/o

Table 3. A comparison of the performance DLSAT with the approach of [SSB02] (column SEP) on benchmark diamond problems with $d = 5$ (t/o means more than 4 minutes).

6.3 Circuit Timing Analysis

The last set of benchmarks is concerned with bounded model checking of timed automata that model digital circuits using the bi-bounded delay model [BS94,MP95]. We use models of n-bit adders constructed from gates where changes are propagated from inputs to outputs within $t \in [l, u]$ time and would like to find the maximal stabilization time of the circuit. We assume that the circuit starts from a stable state and the inputs change at time zero and then remain constant, consequently there is a finite number of transitions in the circuit. We submit to the solver queries, parametrized by d and k, of the form "is there a run of the automaton with k transitions which remains in an unstable state after d time?". The parameter k defines the number of unfolding of the transition relation of the automaton and hence the size of the DL formula. An upper-bound on d can be computed by methods of static timing analysis (summing the delays along the longest path in the circuit), and also each k gives an upper bound on the metric length of runs with k steps. The reader may look at [BBM04] for more details on the problem definition and at [NMA$^+$02] for the formulation of bounded reachability of timed automata in DL. Readers familiar with SAT based methods for circuit verification should bear in mind that we are dealing here with a much richer model of the circuits, with one clock variable per gate.

Table 4 shows the execution time for different queries for a 3-bit adder with 10 gates while Table 5 shows similar results for a 4-bit adder with 16 gates. The results constitute and enormous progress for DL SAT solving (MX-SOLVER could not treat any of these problems) but still they are very far from coping with the size of real problems. The DL formula corresponding to 12 unfoldings of the 4-bit adder (before conversion to CNF) has 624 Boolean variables, 222 numeric variables, and 1381 difference constraints. After conversion to CNF the number of Boolean variables increases to 19463, and the formula has 31516 clauses.

	k=4	k=5	k=6	k=7	k=8	k=9	k=10
d=10	2.10	4.02	8.85	19.38	45.09	51.18	186.01
d=15	2.43	4.03	8.86	15.86	39.81	116.00	376.10

Table 4. The time (in seconds) to answer (k, d) queries for a 3-bit adder (10 gates).

	k=8	k=9	k=10	k=11	k=12
d=25	1:29	3:42	9:23	28:33	18:52
d=35	1:24	3:38	9:22	28:04	17:58

Table 5. The time (in minutes) to answer (k, d) queries for a 4-bit adder (16 gates).

7 Conclusions

This work represents a step forward in DL SAT solving and hence a step in the same direction for exporting bounded model checking for timed systems. For most of the problem classes we considered, our new solver performs much better than other solvers, and we attribute this to the efficient numerical feasibility checks and their integration with learning. Another direction that we explored but not report on the current version is that of *aggressive learning* where the idea is to use the result of numerical feasibility checks to encode (in a compact manner) *all* the negative cycles in the constraint graph and learn their negation. For this purpose we have derived a compact CNF formula to represent the set of all sub-graphs which have no negative cycles. Unfortunately, this formula requires auxiliary variables which could not be shared across feasibility checks. Consequently it had a negative effect on the performance. Some more experimentation and fine tuning are needed in order to assess this technique.

Another future direction of attack is a more efficient encoding of bounded reachability properties for timed automata and circuits. The size of the DL formulae after conversion to CNF is very big, even for small problem and this may be improved by more sophisticated encoding scheme (e.g. the asynchronous time approach mentioned in [NMA+02]) or by extending the solver to work with non CNF formulae.

Acknowledgments.
Our understanding of SAT benefited from discussions with K. Sakallah and O. Strichman. We thank Moez Mahfoudh for his thesis through which we all took our first lessons in the domain.

References

[ABZ88] J. Adams, E. Balas and D. Zawack, The Shifting Bottleneck Procedure for Job Shop Scheduling, *Management Science* 34, 391-401, 1988.

[ACG99] A. Armando, C. Castellini and E. Giunchiglia, SAT-based Procedures for Temporal Reasoning, *Proc. ECP'99*, LNCS, Springer, 1999.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse, Data Structures for the Verification of Timed Automata, *Proc. Hybrid and Real-Time Systems*, 346-360, LNCS 1201, Springer, 1997.

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics and R. Sebastiani, A SAT-Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions, in *Proc. CADE'02*, 193-208, LNCS 2392, Springer, 2002.

[ACKS02] G. Audemard, A. Cimatti, A. Kornilowics and R. Sebastiani, Bounded Model Checking for Timed Systems, Technical report ITC-0201-05, IRST, Trento, 2002.

[BDS+02] C. W. Barrett, D. L. Dill, A. Stump, Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT, in *Proc CAV'02*, 236-249.

[BBM04] R. Ben Salah, M. Bozga and O. Maler, On Timing Analysis of Combinational Circuits, *Proc. FORMATS'03*, 2004.

[BS94] J.A. Brzozowski and C-J.H. Seger, *Asynchronous Circuits*, Springer, 1994.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 2001.

[F02] Martin Fränzle, Take It NP-Easy: Bounded Model Construction for Duration Calculus *Proc. FTRTFT'02*, of 226-243, LNCS 2469, Springer-Verlag, 2002.

[FORS01] J-C. Filliâtre, S. Owre, H. Rueß and N. Shankar, ICS: Integrated Canonizer and Solver, *Proc. CAV'01*, 246-250, 2001.

[GN02] E. Goldberg and Y. Novikov: BerkMin, a Fast and Robust SAT-solver, *Proc. DATE '02*, 142-149, 2002.

[GR93] A. V. Goldberg and T. Radzik, A Heuristic Improvement of the Bellman-Ford Algorithm, In *Applied Mathematics Letters* 6, 1993.

[H00] J. Hooker, *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction*, Wiley, 2000

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey, *Journal of Logic Programming*, 19/20, 503-581, 1994.

[JM99] A.S. Jain and S. Meeran, Deterministic Job-Shop Scheduling: Past, Present and Future, *European Journal of Operational Research* 113, 390-434, 1999.

[M03] M. Mahfoudh, *On Satisfaiblity Checking for Difference Logic*, PhD Thesis, Univeristé Joseph Fourier, Grenoble, 2003.

[MNAM02] M. Mahfoudh, P. Niebert, E. Asarin and O. Maler, A Satisfiability Checker for Difference Logic, *Proc. SAT'2002*, 2002.

[MP95] O. Maler and A. Pnueli, Timing Analysis of Asynchronous Circuits using Timed Automata, *Proc. CHARME'95*, 189-205, LNCS 987, Springer, 1995.

[MS99] J.P. Marques-Silva and K.A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, *IEEE Transactions on Computers* 48, 506-21, 1999.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an Efficient SAT Solver, *Proc. DAC 2001*, 2001.

[MRS02] L. de Moura, H. Rueß and M. Sorea, Lazy Theorem Proving for Bounded Model Checking over Infinite Domains, *Proc. CADE'02*, 437-453, LNCS 2392, Springer, 2002.

[NMA+02] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain and O. Maler, Verification of Timed Automata via Satisfiability Checking, *Proc. FTRTFT'02*, 225-244, LNCS 2469, Springer, 2002.

[S02] M. Sorea, Bounded Model Checking for Timed Automata, *Proc. MTCS'02*, 2002.

[SSB02] O. Strichman, S.A. Seshia, and R.E. Bryant, Deciding Separation Formulas with SAT, in *Proc. CAV'2002*, Springer, 2002.

[Tar72] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 146-160, 1972.

[T70] G. Tseitin, On the Complexity of Derivation in Propositional Calculus, in *Studies in Constructive Mathematics and Mathematical Logic* 2, 115-125, Consultants Bureau, New York, 1970.

[W90] J.M. Wilson, Compact normal forms in propositional logic and integer programming formulations, *Computers and Operation Research*, 309-314, 1990.

[WZP03] B. Wozna, A. Zbrzezny and W. Penczek, Checking Reachability Properties for Timed Automata via SAT, *Fundamenta Informaticae* 55, 223-241, 2003.

[Zha95] G. Zhang. The Davis-Putnam Resolution Procedure, In *Advances in Logic Programming and Automated Reasoning*, volume 2. Ablex Publishing Corporation, 1995.

[ZH96] H. Zhang and M. Stickel: An Efficient Algorithm for Unit Propogation, In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics. 1996.