

# Generalizing DPLL to Richer Logics

Kenneth L. McMillan<sup>1</sup> and Andreas Kuehlmann<sup>1</sup> and Mooly Sagiv<sup>2</sup>

<sup>1</sup> Cadence Research Labs

<sup>2</sup> Tel Aviv University

**Abstract.** The DPLL approach to the Boolean satisfiability problem (SAT) is a combination of search for a satisfying assignment and logical deduction, in which each process guides the other. We show that this approach can be generalized to a richer class of theories. In particular, we present an alternative to lazy SMT solvers, in which DPLL is used only to find propositionally satisfying assignments, whose feasibility is checked by a separate theory solver. Here, DPLL is applied directly to the theory. We search in the space of theory structures (for example, numerical assignments) rather than propositional assignments. This makes it possible to use conflict in model search to guide deduction in the theory, much in the way that it guides propositional resolution in DPLL. Some experiments using linear rational arithmetic demonstrate the potential advantages of the approach.

## 1 Introduction

Many modern Boolean satisfiability (SAT) solvers use an approach known as DPLL that is in some sense a fusion of the exhaustive resolution approach of Davis and Putnam [8] and the backtracking search approach of Davis Logeman and Loveland [7]. In DPLL, these two approaches are tightly coupled in a way that helps to focus the search on relevant decisions and the resolution on relevant deductions. When the search reaches a conflict (a point from which no further progress can be made) a new clause is deduced by resolution. This new clause guides the search away from the conflict and allows it to continue. By focusing deduction only on the points where model search fails, we avoid an explosion of irrelevant deductions. We also gain a measure of the relevance of variables which allows us to focus search decisions.

In this paper, we provide a highly abstract view of this process that generalizes to richer languages, such as linear arithmetic. We start with a theory (a set of sentences in a given logic for which we wish to find a model) and a logical structure which is proposed as a model of the theory. For example, in the case of SAT, a theory is a set of clauses over some propositional atoms, and a structure is a Boolean assignment to those atoms. Two processes then proceed alternately:

- A *model search* process attempts to mutate the structure in a way that will make it, in a well-defined sense, closer to being a model.
- A *refutation* process attempts to deduce new sentences that will make the structure, in a well-defined sense, closer to being refuted.

The refutation process is only triggered when the model search process can no longer make progress (is in conflict). Thus, failure of model search guides deduction. Conversely, by adding a new sentence that must be satisfied, deduction guides model search. We will refer to this general proof search strategy as *general DPLL*, or *GDPLL*.

**Related work** The prevalent method of applying DPLL to non-Boolean theories is the lazy SMT solver [2]. In this approach, a Boolean SAT solver is combined with a *theory solver*. The SAT solver sees atomic formulas of the theory as propositional variables, and attempts to construct a propositionally satisfying assignment. The theory solver refutes these attempts by adding clauses that are tautologies of the theory. In the simplest case, these are “blocking clauses”, built from existing atoms, and representing partial truth assignments that are infeasible in the theory. A more sophisticated solver may add clauses containing new atoms, perhaps representing instances of proof rules that can be used to refute a given assignment [10]. This is referred to as “theory learning”.

A potential weakness of this approach is that the theory solver has no effective means of choosing between the many possible refutations of a given truth assignment. Nonetheless, as we will see, this choice can have an exponential effect on performance of the solver. Theory deduction in lazy SMT solvers occurs only in response to satisfying propositional assignments (or partial assignments) rather than as a response to a conflict in the model search process. Thus, there is very limited relevance feedback for deductions in the theory, as compared to propositional deduction. In the general DPLL approach, we search not in the Boolean space, but in the space of structures of the theory. These may be, for example, numeric assignments or first-order structures. This allows the general DPLL mechanism to choose relevant deductions in the theory based on conflict in model search, just as DPLL does in the propositional case.

After submitting this work, we were made aware of unpublished work of Scott Cotton along similar lines [6]. This independent work also contains the basic ideas of searching in the space of numerical structures, using theory deduction to resolve conflicts.

**Outline** In the next section, we introduce the concepts of GDPLL, using the example of quantifier-free linear rational arithmetic (QFLRA). We present some experiments with a prototype implementation. Section 3 then presents the general framework.

## 2 Linear Rational Arithmetic

The abstract framework of GDPLL will be more easily understood if we begin with a concrete instance of it. We will take as our logical language the formulas of quantifier-free linear rational arithmetic (QFLRA) in clausal form. A sentence (or clause) in this logic is a disjunction of linear constraints, each of which is of the form  $c_0 \bowtie c_1x_1 + \dots + c_nx_n$ , where  $c_i$  is a rational constant,  $x_i$  is a rational variable, and  $\bowtie$  is either  $\leq$  or  $<$ . We don’t require negation, because  $\neg(x \leq y)$  can be expressed as  $y < x$ . For clarity, we will often write constraints with variables

on both sides of  $\bowtie$ , but it should be understood that this is a shorthand for a constraint in the normalized form above.

A *structure* for QFLRA is just an assignment of rational values to the variables. A structure is a *model* of a set of clauses  $S$  if it satisfies every clause in  $S$  according to the usual semantics of arithmetic.

A variable  $x_i$  *occurs* in a constraint if its coefficient  $c_i$  in that constraint is non-zero, and it occurs in a clause if it occurs in any constraint of the clause. A constraint or clause in which no variable occurs is equivalent to either TRUE or FALSE. We will not distinguish between a clause equivalent to FALSE and the constant FALSE. We also drop redundant literals without comment.

A variable is *lower-bounded* by a constraint if its coefficient is positive, and *upper-bounded* if it is negative. It is lower-bounded (upper-bounded) by a clause if lower-bounded (upper-bounded) by any constraint in the clause.

An *interval* is a connected subset of the rational line  $\mathcal{Q}$ . We will express intervals using the standard notation, as a pair of possibly infinite lower and upper bounds with a square bracket indicating a closed end and a round bracket an open end. Thus, for example,  $(-\infty, 5]$  is the set  $\{r \in \mathcal{Q} \mid r \leq 5\}$ .

Given a structure  $A$  and a clause  $s$ , we will say the *feasible set* for variable  $x_i$ , denoted  $\text{feas}(A, s, x_i)$ , is the set of rational numbers  $r$  such that  $A(r/x_i)$  satisfies  $s$ . For example, given the assignment  $(a = 0, b = 0, c = 0)$  and the clause  $(a + 5 \leq b)$ , the feasible set for  $b$  is the rational interval  $[5, \infty)$ . The feasible set for variable  $c$  is the empty set, since we cannot make the constraint true by plugging in any value for  $c$ .

## 2.1 Algorithm

The algorithm in Figure 1 searches for a model of a set of clauses by assigning feasible values to variables in some chosen order. This is similar to DPLL except that the values are rational rather than Boolean. When we reach a situation in which some variable has no feasible value, we are in *conflict*. We resolve the conflict by performing a deduction step. This introduces a new clause that guides us away from the conflict. Again, this is similar to DPLL, except that we will require a richer proof calculus. As we discuss the algorithm, we will use a running example, in which the set  $S$  consists of the clauses  $(a < b)$ ,  $(a < c)$ ,  $(b < d \vee c < d)$  and  $(d < a)$ . Our initial structure  $A_0$  will be  $(a = 0, b = 0, c = 0, d = 0)$ .

Let  $\sqsubset$  be a fixed total order on the variables. In our example, we will put the variables in alphabetical order, so that  $a \sqsubset b \sqsubset c \sqsubset d$ . Relative to this order, a variable  $x_i$  *dominates* a clause  $s$  if  $x_i$  is the maximal variable occurring in  $s$ , according to  $\sqsubset$ . For example, the variable  $c$  dominates the clause  $(a < c)$ . The subset of  $S$  dominated by  $x_i$  will be denoted  $\text{domin}(S, x_i)$ . In our example,  $\text{domin}(S, c) = \{(a < c)\}$ .

The *ordered* feasible set for  $x_i$ , denoted  $\text{feas}_{\sqsubset}(A, S, x_i)$ , is the *intersection* of the feasible sets for  $x_i$  over the clauses dominated by  $x_i$ , that is,

$$\text{feas}_{\sqsubset}(A, S, x_i) = \cap \{\text{feas}(A, s, x_i) \mid s \in \text{domin}(S, x_i)\}$$

Algorithm: GDPLLQFLRA  
Input: a set  $S$  of clauses of QFLRA  
Output: a satisfying assignment for  $S$ , or UNSAT

```

Let  $\sqsubset$  be some total order on  $x_1 \dots x_n$ 
Let  $A$  be a structure of QFLRA
repeat
  if  $A \models S$  return  $A$ 
  if  $\text{FALSE} \in S$  return UNSAT
  let  $x_i$  be the least variable w.r.t.  $\sqsubset$  such that  $A(x_i) \notin \text{feas}_{\sqsubset}(A, S, x_i)$ 
  if  $\text{feas}_{\sqsubset}(A, S, x_i)$  is not empty
    choose  $r \in \text{feas}_{\sqsubset}(A, S, x_i)$  and set  $A \leftarrow A\langle r/x_i \rangle$ 
  else if there exists strictly compatible  $(s, s')$  in  $\text{minim}_{\sqsubset}(A, S, x_i)$ 
    set  $S \leftarrow S \cup \{\text{shad}_{x_i}(s, s')\}$ 
  else abort

```

**Fig. 1.** Decision procedure for QFLRA

In our example,  $\text{feas}_{\sqsubset}(A_0, S, c) = (0, \infty)$  and  $\text{feas}_{\sqsubset}(A_0, S, d) = \emptyset$  (since no value of  $d$  can satisfy both  $(b < d \vee c < d)$  and  $d < a$ ). We will say that a variable  $x_i$  is *correct* if  $A(x_i) \in \text{feas}_{\sqsubset}(A, S, x_i)$  and *incorrect* otherwise.

The algorithm of Figure 1 executes a loop so long as  $A$  is not a model of clause set  $S$  and  $S$  does not contain the clause FALSE (in which case there is no model). In this loop, we choose the least incorrect variable  $x_i$ . In our example, this variable is  $b$ , since its feasible set is  $(0, \infty)$  and its initial value is  $A_0(b) = 0$ . If the feasible set of  $x_i$  is not empty, we correct  $x_i$  by assigning it some value in its feasible set. This step corresponds to making a “decision” in DPLL. In our example, we can choose any value for  $b$  greater than zero, say,  $A_1(b) = 1$ . Notice that the mutation of  $x_i$  cannot, by definition, modify the feasible sets of any variables less than  $x_i$ . Therefore, by this mutation, we are in some sense making progress towards a satisfying assignment. In our example, we would proceed to set  $A_2(c) = 1$ . At this point, the least incorrect variable is  $d$ , whose feasible set is empty. We say that  $d$  is in conflict.

When we reach a variable  $x_i$  whose feasible set is empty, we resolve the conflict by choosing two clauses  $s_1$  and  $s_2$  from the set dominated by  $x_i$  and combining them to deduce a new clause. The feasible set for  $x_i$  defined by this new clause is the intersection of the feasible sets defined by  $s_1$  and  $s_2$ . This is precisely what happens in DPLL when we use a resolution step to learn a “conflict clause”. That is, for some Boolean variable  $x_i$ , we have one clause whose feasible set for  $x_i$  is  $\{\text{TRUE}\}$  (it implies  $x_i$  in the current assignment) and one whose feasible set is  $\{\text{FALSE}\}$  (it implies  $\neg x_i$  in the current assignment). We resolve these two clauses to obtain a learned clause whose feasible set is the empty set (it is false in the current assignment).

In the case of QFLRA, the deduction step is a bit more subtle, again because the variables are rational rather than Boolean. To resolve a conflict, we will use a deduction rule we call the *shadow rule*. If a pair of clauses  $(s_1, s_2)$  is *compatible*,

in a sense we will define shortly, this rule produces a new clause whose feasible set for  $x_i$  is the intersection of the feasible sets for  $s_1$  and  $s_2$ .

**The shadow rule** The shadow rule is shown in Figure 2. To express transitivity of inequality with both strict and non-strict inequalities, we use the notation  $\bowtie\bowtie'$ , where  $\bowtie$  and  $\bowtie'$  are  $<$  or  $\leq$ . This stands for  $\leq$  if both  $\bowtie$  and  $\bowtie'$  are  $\leq$  and  $<$  otherwise. Thus if  $x \bowtie y$  and  $y \bowtie' z$ , then  $x \bowtie\bowtie' z$ , for all four choices of strict and non-strict comparisons. Given this, and using distributivity of conjunction over disjunction, the shadow rule is easily shown to be sound. The side conditions of the rule are not needed for soundness. Rather, they restrict deduction so that, from any ordered pair of premises, for any given variable  $x_i$ , exactly one conclusion can be deduced. To derive this conclusion from a pair  $(s, s')$  of clauses, we gather all the lower-bounding constraints from  $s$ , and rewrite then in the form  $l_j \bowtie_j x_i$ . Similarly, we rewrite all the upper-bounding constraints of  $s'$  in the form  $x_i \bowtie'_{j'} u_{j'}$ . In our example, applying the shadow to the clauses  $(b < d \vee c < d)$  and  $(d < a)$ , on variable  $d$ , yields  $(b < a \vee c < a)$ .

We will denote the clause resulting from application of the shadow rule  $\text{shad}_{x_i}(s, s')$ .

$$\frac{C \vee l_1 \bowtie_1 x_i \vee \cdots \vee l_m \bowtie_m x_i \quad C' \vee x_i \bowtie'_1 u_1 \vee \cdots \vee x_i \bowtie'_{m'} u_{m'}}{C \vee C' \vee \bigvee_{j=1\dots m} \bigvee_{j'=1\dots m'} (l_j \bowtie_j \bowtie'_{j'} u_{j'})} \quad \begin{array}{l} x_i \text{ not occurs in } l_{1\dots m}, u_{1\dots m'} \\ C \text{ does not lower-bound } x_i \\ C' \text{ does not upper-bound } x_i \end{array}$$

**Fig. 2.** The shadow rule

In our search algorithm, we apply the shadow rule only to pairs of clauses  $(s, s')$  that are *compatible* at  $A$ . To define this notion, we begin with a corresponding definition on intervals. Given two intervals  $I$  and  $I'$ , let  $\text{glue}(I, I')$  be the interval that takes its lower bound from  $I$  and its upper bound from  $I'$ . For example,  $\text{glue}((-\infty, 5), [-1, 3]) = (-\infty, 3]$ . Note that this operator is not commutative, and is not the convex hull operator. An ordered pair of intervals  $I, I'$  is *compatible* if  $\text{glue}(I, I') = I \cup I'$ . Examples of compatible pairs are  $(-\infty, 0), [0, \infty)$  and  $(3, 5], (4, 7]$ . Incompatible examples are  $(-\infty, 0), (0, \infty)$  and  $(4, 5], (3, 7]$ .

Now, for any structure  $A$ , clause  $s$  and variable  $x_i$ , the feasible set  $\text{feas}(A, s, x_i)$  is the complement of an interval that we will call the *forbidden interval*, denoted  $\text{fbd}(A, s, x_i)$ . For example, for structure  $(a = 0, b = 0, c = 0)$  and clause  $(c + 5 < a \vee b \leq c)$  the forbidden interval for  $c$  is  $[-5, 0)$ . In effect, the shadow rule applies the glue operator to the forbidden intervals of two clauses. We say a pair of clauses is compatible when their forbidden intervals are compatible:

**Definition 1.** *Given structure  $A$  and clauses  $s, s'$  dominated by variable  $x_i$ , the pair  $(s, s')$  is compatible when the pair of intervals  $\text{fbd}(A, s, x_i), \text{fbd}(A, s', x_i)$  is compatible. It is strictly compatible when, further,  $\text{fbd}(A, s, x_i) \neq \text{fbd}(A, s', x_i)$ .*

Since taking the union of the forbidden intervals corresponds to taking the intersection of the feasible sets, we have the following theorem:

**Theorem 1.** *For any QFLRA structure  $A$  and compatible pair of clauses  $(s, s')$  of QFLRA, and any variable  $x_i$ :*

$$\text{feas}(A, \text{shad}_{x_i}(s, s'), x_i) = \text{feas}(A, s, x_i) \cap \text{feas}(A, s', x_i)$$

*Proof sketch.* Let  $I$  and  $I'$  be the forbidden intervals of  $s$  and  $s'$  respectively. We can show that the forbidden interval of  $\text{shad}_{x_i}(s, s')$  is  $\text{glue}(I, I')$ . Since  $(s, s')$  is compatible, this is  $I \cup I'$ , which gives us the theorem by DeMorgans's laws.  $\square$

Our search procedure places one additional restriction on deductions. We say a clause in  $s \in S$  is *minimal* for a given variable  $x_i$ , if there is no other clause  $t \in S$  such that  $\text{feas}(A, S, t) \subset \text{feas}(A, S, s)$ . We denote the set of such minimal clauses  $\text{minim}_{\sqsubset}(A, S, x_i)$ . By restricting deduction to minimal clauses, we avoid repetition of deductions.

With these provisos, our algorithm is a decision procedure for QFLRA. Partial correctness is trivial, since a result can only be returned in case  $A$  is a satisfying assignment, or we have deduced FALSE. It remains only to argue that the procedure terminates (without aborting). We postpone this until we have introduced the general framework.

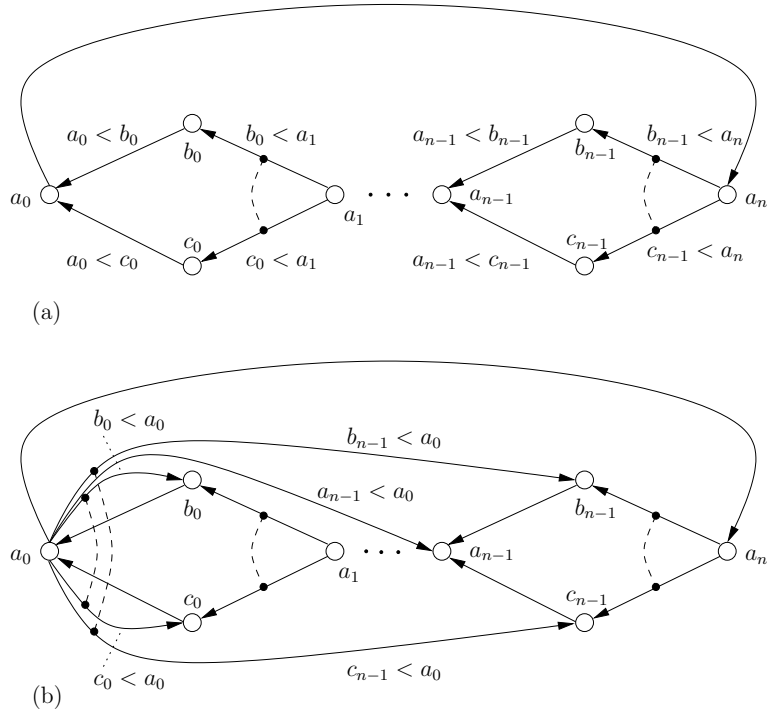
To complete our example, recall our structure  $A_2$  is  $(a = 0, b = 1, c = 1, d = 0)$  and we have just deduced  $(b < a \vee c < a)$ . Now  $c$  is the least incorrect variable. The clause  $(a < c)$  has forbidden interval  $(-\infty, 0]$  while  $(b < a \vee c < a)$  has  $[0, \infty)$ . These are strictly compatible, so we apply the shadow rule to obtain  $(b < a \vee 0 < 0)$ . We can ignore the second constraint, since it equivalent to FALSE. Now we find that the feasible set for  $b$  is empty, and we apply the shadow rule to  $(a < b)$  and  $(b < a)$ , obtaining  $(0 < 0)$ . Since this is equivalent to FALSE, we report UNSAT.

## 2.2 Discussion

In our instantiation of GDPLL for linear arithmetic, we can observe several significant differences with respect to the lazy SMT approach. First, notice that GDPLL searches in the space of assignments to rational variables, while lazy SMT searches in the space of Boolean assignments to atoms. Second, theory deduction in GDPLL is performed only in response to conflict in the model search, just as in propositional DPLL. In lazy SMT, theory deduction occurs in response to *satisfying* Boolean assignments (or partial assignments). Third, GDPLL applies *proof rules* of the theory, while lazy SMT uses only *axioms* generated by the theory solver. While the lazy theory solver generates only tautologies of the theory, our procedure never deduces tautologies. Every clause we generate derives from original problem clauses and is false in some structure (the one generating the conflict). In lazy SMT, Boolean and theory reasoning are deliberately separated. The theory solver cannot resolve problem clauses, because it never sees them. In GDPLL, there is no such separation.

From a heuristic point of view, we will argue that the Boolean assignments generated by the SAT solver in lazy SMT provide inadequate guidance to the theory solver as to which of many possible refutations is preferred. In principle, for any deduction performed using our shadow rule, a theory solver could provide a sufficient set of axioms to allow the SAT solver to reproduce the conclusion by a sequence of resolution steps. In practice, lazy SMT lacks the heuristic guidance needed to produce this result.

A simple example will serve to illustrate this point. We consider a class of “diamond formulas”, that are hard for most existing solvers based on lazy SMT [15]. Our running example above is a diamond formula for  $N = 1$ . Figure 3 depicts the general case. In the graph, an arrow represents a  $<$  constraint between variables, and a dashed line between arrows indicates disjunction of constraints.



**Fig. 3.** Diamond example for general  $N$ : (a) problem graph, (b) clauses deduced by GDPLL, by repeating these three steps: (1)  $(b_{i-1} < a_i \vee c_{i-1} < a_i), (a_i < a_0) \vdash (b_{i-1} < a_0 \vee c_{i-1} < a_0)$ , followed by (2)  $(b_{i-1} < a_0 \vee c_{i-1} < a_0), (a_{i-1} < c_{i-1}) \vdash (b_{i-1} < a_0 \vee a_{i-1} < a_0)$  and (3)  $(b_{i-1} < a_0 \vee a_{i-1} < a_0), (a_{i-1} < b_{i-1}) \vdash (a_{i-1} < a_0)$ .

Our GDPLL procedure can show unsatisfiability of these formulas in a number of steps linear in  $N$ , the number of diamonds. Figure 3 shows the deduction sequence for any  $N$  for the variable order  $a_0, b_0, c_0, a_1, b_1, c_1, \dots$ . It turns out that this result is not particularly sensitive to variable order. For example, the order  $a_0, \dots, a_n, b_0, \dots, b_{n-1}, c_0, \dots, c_{n-1}$  also yields a linear number of steps.

Notice how deterministic this procedure is. Once the variable order is chosen, the sequence of deductions is fixed. The conflicts in model search tell us precisely which deductions to perform.

Now consider the lazy SMT approach. The difficulty with diamond formulas for lazy SMT is that a sequence of  $N$  diamonds contains  $2^N$  infeasible cycles. Suppose, for example, that the theory solver can only generate blocking clauses. Each such clause eliminates exactly one of the infeasible cycles, so  $2^N$  calls to the theory solver are needed. This is true even if theory propagation [11] is used. This problem can be solved in principle by having the theory solver introduce new atoms into the SAT solver. For example, new equality literals are commonly introduced when using Nelson-Oppen style theory combinations [4]. This can also be done through *theory learning*, which introduces tautologies containing fresh atoms [10]. A lazy SMT solver could handle the diamonds efficiently if the theory solver were to inject, for example, all the atoms of form  $a_i < a_0$ ,  $b_i < a_0$  and  $c_i < a_0$  into the SAT solver. In practice, though, we must choose these atoms without generating an explosion of irrelevant ones. A Boolean assignment, by itself, does not give us enough information to make this choice. In fact, as we will observe shortly, all of the applicable solvers from the 2008 SMT competition [1] are exponential on diamond formulas.

This problem has sparked some research effort. For example, in the Dynamic Predicate Learning method [16], Wang, Gupta and Ganai apply transitivity to difference logic predicates (a subset of LRA). They use frequency of occurrence in blocking clauses as a selection criterion, with a fixed threshold of 200 occurrences, and other *ad hoc* criteria. While this heuristic is sufficient to handle diamond formulas, the method applies only to difference logic, and is not shown to be effective in general (for other than diamond formulas). Despite some progress [13], the problem of effective theory learning in QFLRA still appears to be open.

We note that there may be many other methods that can handle the diamond formulas efficiently. We use this example only to illustrate the heuristic difficulties faced by a lazy SMT solver, and to show how searching in the space of numerical assignments avoids these difficulties.

We also note that the “shadow rule” we used for QFLRA is only one of many possibilities. For example, the LASCA proof calculus [14] has three proof rules that deal with inequalities. By combining these rules, we can generate many conclusions whose feasible set is the intersection of the feasible sets of the premises. Thus, while we say that deduction in GDPLL is *guided* by conflict, it is not always fully determined, as it was in the example above.

Finally, there are some related proof methods that should be mentioned. Exhaustive methods, such as Fourier-Motzkin elimination use no heuristic guidance whatever, and simply perform all possible deductions. Eager variable elimination [9] uses resolution steps to simplify the formula in a pre-processing step. This method can also be used with GDPLL. There are methods that use models generated by theory solvers to guide the lazy SMT approach [4, 12]. They do not address the issues discussed above. The method DPLL( $\perp$ ) makes some of the Boolean structure of the problem visible to the theory solver, allowing it to

produce joins of facts deducible along different search paths [3]. This method attacks the problem of lack of guidance in theory learning, while maintaining the basic lazy SMT framework. There are also eager SMT approaches [2] that encode the formula into an equisatisfiable Boolean SAT formula. This has the advantage of fully exploiting DPLL principles, but the disadvantage that the encodings can be extremely large.

### 2.3 Experiments

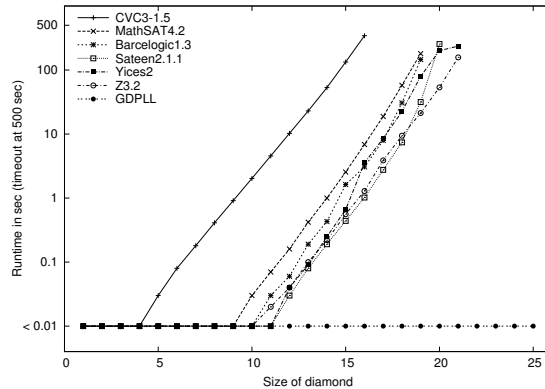
To evaluate the performance potential of GDPLL and to illustrate the difference between our search approach and existing SMT solvers, we performed two experiments using a prototype implementation of the linear arithmetic decision procedure using GDPLL. All experiments were performed on a server with 32 GB of main memory, using one CPU of a 4-processor Opteron machine, running at 2.587 GHz with 1 MB of cache.

For our experiments, we use a very simple static variable ordering heuristic. Boolean SAT solvers generally base their initial variable ordering on the number of occurrences of each variable. However, in the case of QFLRA we can also exploit the numerical relationships between variables. We construct a directed graph in which the vertices are variables, and an edge  $(x, y)$  occurs when there is a unit constraint, such as  $x < y$ , in which is upper-bounding for  $x$  and lower-bounding for  $y$  (thus, increasing  $x$  may force us to increase  $y$ ). We use a DFS-based heuristic that orders the nodes of this graph, attempting to reduce the number of back edges in the graph. Intuitively, this reduces the number of times that a variable will have to be updated because its predecessor is updated.

In the first experiment we confirmed our conjecture of the solver’s performance for the diamond example described in Section 2.2. For this, we run the GDPLL solver on diamond structures shown in Figure 3 for sizes  $N$  ranging from 1 to 25. We also run six solvers that participated in the 2008 SMT competition and can handle rational difference logic (QF\_RDL). We applied a timeout of 500 seconds to all runs. Figure 4 provides the results. As shown, the GDPLL implementation solves all the diamond problems in a negligible amount of time whereas all other solvers demonstrate an exponential run time in  $N$ .

In the second experiment we evaluated the solver’s performance for a simplified circuit timing analysis problem. For this we generated timing constraints from 142 ISCAS benchmarks using simple gate delays. We also modeled the fact that the delay of a CMOS gate increases if the two latest inputs arrive at approximately the same time. As an example, suppose the arrival times of a two-input gate are denoted by  $t_1, t_2$  and  $t_o$  for the two inputs and for the output, respectively. The following inequalities model the described timing behavior of that gate:

$$\begin{aligned} &(t_o \geq t_1 + \delta_{small}) \wedge \\ &(t_o \geq t_2 + \delta_{small}) \wedge \\ &((t_1 > t_2) \vee (t_o \geq t_1 + \delta_{large})) \wedge \\ &((t_2 > t_1) \vee (t_o \geq t_2 + \delta_{large})) \end{aligned}$$



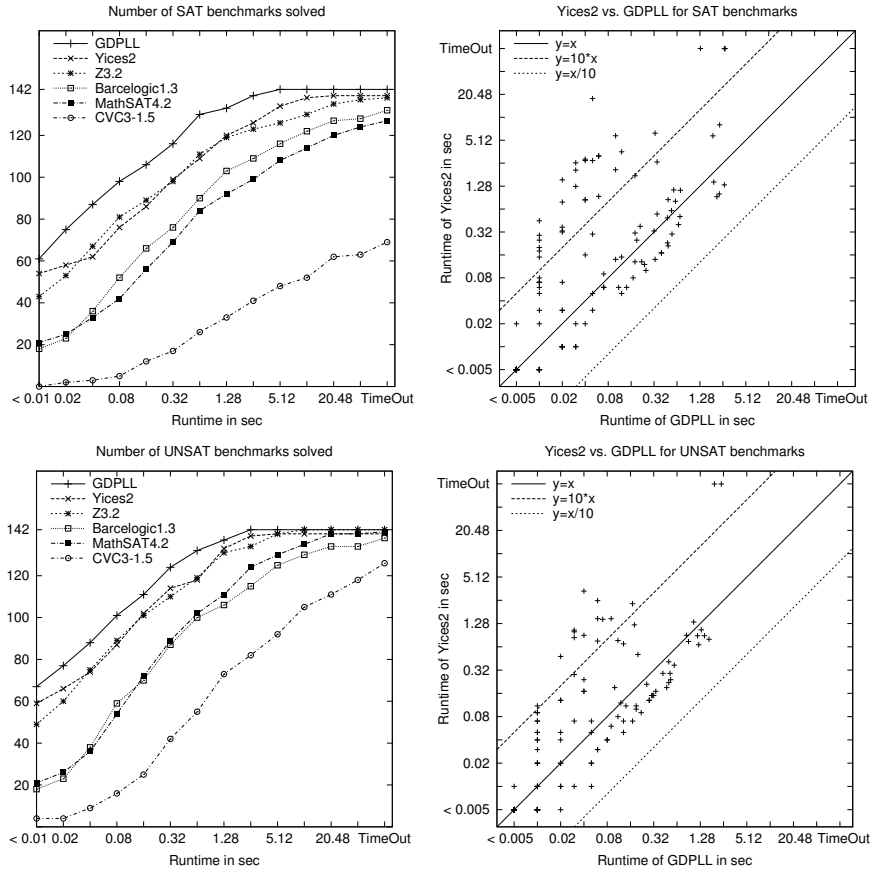
**Fig. 4.** Run time comparison for the diamond structure for our solver “GDPLL” and 6 solvers from SMT-COMP’08.

For each benchmark circuit we generate the timing constraints for all gates and added for every output an additional constraint, requiring that its arrival time be less or equal than its precomputed arrival time for the SAT case and strictly less for the UNSAT case. Figure 5 shows the run time comparison for the SAT and UNSAT cases. For the SMT-COMP’08 solvers, we excluded the Sateen solver because it reported “unknown” for all problems. For the other solvers, we recorded all correct results produced within the given time limit of 82.9 sec. As shown, GDPLL outperforms all solvers by a significant margin. In the right part of the figure, we show a scatter plot comparing the run times of our implementation with Yices2, the best performing solver on this benchmark among the SMT-COMP’08 participants. The points lying below the diagonal show that the constant factors of our preliminary implementation are not as good as the highly tuned Yices2. However, in some cases, Yices2 is two orders of magnitude slower; in five cases Yices2 was not able to solve the problem within the time limit of 82.9 sec, though our solver handled all cases in less than 2.8s. We conjecture that in these cases, a large number of timing paths in these circuits had to be individually refuted by blocking clauses.

We do not claim based on these results that GDPLL is superior in any way to other methods. In particular, we were not able to perform comparisons on the standard SMTLIB benchmarks because our implementation currently does not support Boolean variables. Moreover, the question of variable ordering heuristics needs to be addressed in a more general way. Nonetheless, we do feel that these results demonstrate that GDPLL has different performance characteristics from lazy SMT, and that the use of model search to drive theory deduction can be an effective heuristic.

### 3 General DPLL

We now introduce a framework for GDPLL that abstracts away from particular logics. This framework can be used as a proof search and model search strategy



**Fig. 5.** Solver statistics for synthesized timing analysis problems for 142 ISCAS circuit comparing GDPLL with 5 solvers from SMT-COMP'08 and detailed comparison of GDPLL with Yices2 (Timeout at 81.9 sec.)

for a range of logics and proof calculi, provided certain general conditions are met.

Let  $\mathcal{L}$  be a logical language, that is, a set of sentences. A subset of  $\mathcal{L}$  is a *theory*. We assume a set of logical *structures*  $\mathcal{A}$ . The semantics of  $\mathcal{L}$  is defined by the relation  $\models$ , and we write  $A \models s$ , when  $A \in \mathcal{A}$  is a model of  $s \in \mathcal{L}$  (alternatively,  $A$  satisfies  $s$ , or  $s$  is true in  $A$ ). We allow the use of non-standard structures and semantics, provided  $A \models s$  implies that  $s$  is satisfiable in the standard semantics. In particular, we may use a three-valued semantics, so that possibly  $A \not\models a$  and  $A \not\models \neg a$ . We assume that  $\mathcal{L}$  contains a sentence FALSE that has no models (in the standard semantics).

We think of GDPLL as a search process. The state of our search is a pair  $(S, A)$ , where  $S$  is a set of sentences and  $A$  a structure. The search goal is to reach a state in which  $A \models S$ . As in any search, we require a *heuristic*, that is, some measure of progress toward reaching the goal. We will define this in terms

of a partial order  $\succ$  on search states. That is, if  $(S, A) \succ (S', A')$ , then in some heuristic sense, state  $(S', A')$  is closer to the goal.

Our moves in the search space are defined by a *mutation relation*  $\triangleright$  and a *deduction relation*  $\vdash$ . The former is a relation on structures. If  $A \triangleright A'$ , we can move from  $A$  to  $A'$ . The latter allows us to add sentences to  $S$ . That is, if  $S \vdash s$ , where  $s$  is a sentence, then we may add  $s$  to  $S$ . Of course, we require that  $\vdash$  is sound, in the sense that  $S \vdash s$  implies  $S \models s$ . In our search process, we only perform mutations in the downward direction of  $\succ$ . To move from  $(S, A)$  to  $(S, A')$ , where  $A \triangleright A'$ , we require  $(S, A) \succ (S, A')$ . When we reach a “local minimum”, in which no mutation can move us downward, we say we are in conflict. We escape the local minimum by deducing a new sentence  $s$ , such that addition of  $s$  moves us *upward* in  $\succ$ . Intuitively, we have learned new information about the problem which has refined our heuristic.

This general algorithm is depicted in Figure 6. It starts with an initial theory  $S$  and an arbitrary initial structure  $A$ . It repeatedly performs either a model search step (1) or a deduction step (2). In a model search step, we mutate  $A$  to some  $A'$ , moving downward in  $\succ$ . In a deduction step, we deduce a new sentence  $s$ , moving upward in  $\succ$ . This process continues until either  $A$  is a model of  $S$ , or the sentence FALSE is deduced.

while  $A \not\models S$  and  $\text{FALSE} \notin S$  do  
 $\left\{ \begin{array}{l} \text{1) set } A \leftarrow A', \text{ where } A \triangleright A' \text{ and } (S, A) \succ (S, A') \\ \text{or} \\ \text{2) set } S \leftarrow S', \text{ where } S' = S \cup \{s\}, S \vdash s \text{ and } (S, A) \prec (S', A) \end{array} \right\}$

**Fig. 6.** The GDPLL algorithm

**Correctness** We would like to argue that this is a decision procedure for  $\mathcal{L}$ . Partial correctness is trivial, since the procedure only terminates with a model of  $S$  or a deduction of FALSE. It remains only to guarantee termination. For this, we require three key conditions:

- Progress: there are no infinite chains  $(S, A_1) \succ (S, A_2) \succ \dots$ .
- Boundedness: the deductive closure of any finite  $S$  under  $\vdash$  is finite.
- Deadlock freedom: in any state  $(S, A)$ , such that  $A \not\models S$  and  $\text{FALSE} \notin S$ , either a model search step or a deduction step is possible.

**Theorem 2.** *Given progress, boundedness and deadlock freedom, the GDPLL algorithm terminates.*

*Proof sketch.* The progress condition guarantees that we cannot execute model search steps infinitely. If we execute model search steps only, we eventually reach a state where no model search step is possible. By deadlock freedom, a deduction step must be possible in this state. Thus, if we do not terminate,

we must execute a deduction step infinitely often. However, this is a contradiction, since each deduction step must add a sentence to  $S$  and  $S$  is contained in the finite deductive closure of the initial  $S$ .  $\square$

Note that it is also possible to modify the heuristic order  $\succ$  during execution of GDPLL, provided we do this only a bounded number of times between deductions. This can model dynamic ordering heuristics in DPLL. Also, note that this general framework allows many heuristically useless instantiations. In the extreme,  $\triangleright$  could allow mutation to any structure, and  $\vdash$  could deduce FALSE in one step from every inconsistent  $S$ . The key in practice is to define a search space in which mutation and deduction are tractable and not overly non-deterministic.

**Instantiation for QFLRA** To illustrate these concepts, consider instantiating the general algorithm for QFLRA. In this case, a mutation changes the value of one rational variable, and the deduction relation is defined by the shadow rule. Our heuristic  $\succ$  puts a priority on correcting the least incorrect variable according to the fixed variable order  $\sqsubset$ . Given two states  $(S, A)$  and  $(S', A')$  whose least incorrect variables are  $x_i$  and  $x_{i'}$  respectively, we say that  $(S, A) \succ (S', A')$  when  $x_i \sqsubset x_{i'}$ . That is, we make progress when we increase the least incorrect variable. In case of a tie, when  $x_i = x_{i'}$ , we will say  $(S, A) \succ (S', A')$  iff  $\text{domin}(S, x_i) \subset \text{domin}(S', x_i)$ . Intuitively, more constraints on  $x_i$  make the problem harder to solve (though there are actually many equivalent ways we could define this).

Now consider the three properties needed for termination. Progress is easily guaranteed, since a mutation must increase the least incorrect variable and this cannot be done infinitely.

Boundedness is a property of the shadow rule. We will refer to an instance of the shadow rule as *ordered*, with respect to our total order  $\sqsubset$  on variables, if the same variable  $x_i$  dominates both premises. Note that our algorithm only applies the shadow rule to clauses  $s$  and  $s'$  dominated by  $x_i$ , so we use only ordered instances.

**Lemma 1.** *For any total order  $\sqsubset$  on variables, the ordered shadow rule is bounded.*

*Proof sketch.* Consider the set of constraints  $C$  occurring in a finite set  $S$  of clauses. The ordered shadow rule can only create a new constraint by applying transitivity, eliminating the highest variable. Since there can be no infinite chains of such deductions, the set of constraints generated from  $S$  is finite. Since every deduced clause is a disjunction over this finite set, the deductive closure of  $S$  is finite.  $\square$

To show deadlock-freedom, we rely on the following property of intervals:

**Lemma 2.** *For any finite set  $\mathcal{I}$  of intervals, such that  $\cup \mathcal{I} = \mathcal{Q}$ , either  $\mathcal{Q} \in \mathcal{I}$ , or  $\mathcal{I}$  contains a strictly compatible pair.*

If we cannot execute a model search step, it is because the feasible set of some variable  $x_i$  is empty. This means that the forbidden intervals of  $\text{minim}_{\sqsubset}(A, S, x_i)$

cover the real line. By the above lemma, these intervals must contain a strictly compatible pair  $(s, s')$ . By applying the shadow rule to the corresponding clauses, we obtain a new clause  $t$  with strictly smaller feasible set than  $s, s'$ . If the feasible set of  $t$  for  $x_i$  is empty, then  $A \not\models t$  and  $x_i$  does not occur in  $t$ . Thus we have reduced the least incorrect variable. Otherwise, minimality of  $s$  and  $s'$  implies that  $t$  is not already present in  $S$ , thus we increase  $\text{domin}(S, x_i)$ . In either case we move up in the heuristic order  $\succ$ , making this a legal deduction step.

It follows that this instantiation of GDPLL terminates. Since every step of our original algorithm GDPLLQFLRA can be simulated by this instantiation, we can show:

**Theorem 3.** *GDPLLQFLRA terminates (without aborting) for all inputs.*

Finally, we note that, unlike in the Boolean case, we cannot change the variable order arbitrarily during execution of the solver because the unordered shadow rule is not bounded. In fact, it is easy to construct cases in which regularly changing the order leads to non-termination. For this reason, care will be needed in developing dynamic variable ordering strategies for QFLRA without sacrificing termination.

## 4 Conclusion

An essential feature of DPLL algorithms for SAT is the interaction of model search and proof. In DPLL, we focus a very prolific deduction system (resolution) by using conflict in model search as a guide. The resulting deduced clauses in turn guide the model search away from the conflict. We have attempted to generalize the DPLL procedure in a way that preserves this essential feature, while extending it to non-propositional theories. We have seen that GDPLL applies to linear arithmetic, allowing us to search in the space of numerical assignments instead of Boolean assignments. It allows us to use the DPLL approach to focus arithmetic deduction as well as propositional deduction. This may provide a significant advantage over the lazy SMT approach, which lacks heuristic guidance in learning theory facts. We saw, for example, that the ability to use model search to guide deduction allows us to avoid an exponential explosion of Boolean combinations that the lazy SMT solvers we tested incurred for diamond formulas. We noted that GDPLL is not simply a special case of theory learning in lazy SMT. In GDPLL, deduction occurs only in response to conflicts, while in lazy SMT, theory deduction occurs in response to satisfying assignments. In lazy SMT, the theory solver generates only tautologies, while in GDPLL, tautologies are never deduced.

We would like to stress that our purpose is not to create a more efficient arithmetic solver, but rather a general framework to allow the DPLL approach to guide reasoning in a broad range of theories. For example, we have implemented versions of GDPLL for quantifier-free equality with uninterpreted functions and quantified first-order logic without equality. There remains considerable work to be done to create practical solvers, however. For example, the linear arithmetic

approach we describe uses a static variable order. It is likely to be more efficient to order the variables dynamically as is done in a SAT solver. Variable ordering heuristics in the style of VSIDS are needed. It is also theoretically possible to make an integer linear arithmetic (QFLIA) solver in this style, for example using GDPLL to guide Cooper’s method [5]. However, it is unclear whether such a solver would be efficient, and there seem to be many possible approaches to integer arithmetic. There remains also the question of combining theories, for example combining arithmetic with uninterpreted functions. The GDPLL model also suggests a wide space of alternative SAT implementations, including possible parallel SAT. We have not explored these possibilities.

## References

1. C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 4th annual satisfiability modulo theories competition (SMT-COMP 2008). To appear.
2. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, chapter 8. IOS Press, 2009.
3. N. Bjørner, B. Dutertre, and L. de Moura. Accelerating lemma learning using joins - DPPL( $\sqcup$ ). In *LPAR*, 2008.
4. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *CAV*, pages 335–349, 2005.
5. D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence 7*, pages 91–99, 1972.
6. S. Cotton. Algebraic satisfiability solving. Personal communication, 2009.
7. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
8. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
9. N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.
10. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003.
11. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.
12. D. Goldwasser, O. Strichman, and S. Fine. A theory-based decision heuristic for DPLL(T). In *FMCAD*, pages 1–8, 2008.
13. P. Koppensteiner and H. Veith. A novel SAT procedure for linear real arithmetic. In *PDPAR*, 2005.
14. K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In *CSL*, pages 223–237, 2007.
15. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with sat. In *CAV*, pages 209–222, 2002.
16. C. Wang, A. Gupta, and M. K. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *DAC*, pages 235–240, 2006.