SAT Solving with Reference Points*

Stephan Kottler

Wilhelm–Schickard–Institute, University of Tübingen, Germany

Abstract. Many state-of-the-art SAT solvers use the VSIDS heuristic to make branching decisions based on the activity of variables or literals. In combination with rapid restarts and phase saving this yields a powerful decision heuristic in practice. However, there are approaches that motivate more in-depth reasoning to guide the search of the SAT solver. But more reasoning often requires more information and comes along with more complex data structures. This may sometimes even cause strong concepts to be inapplicable in practice.

In this paper we present a suitable data structure for the DMRP approach to overcome the problem above. Moreover, we show how DMRP can be combined with CDCL solving to be competitive to state-of-the-art solvers and to even improve on some families of industrial instances.

1 Introduction

Research in satisfiability checking (SAT) has managed to bridge the gap between theory and practice in many aspects. There are several kinds of real-world problems that are actually tackled by modelling those problems as SAT instances like hardware and software verification [21,10], planning [11], automotive product configuration [13] and haplotype inference in bioinformatics [15] (cf. [16]).

In the domain of SAT solving there are different schemes and even more variants of these schemes to decide whether there exists a satisfying assignment to the variables of a Boolean formula in CNF or if a formula cannot be satisfied by any assignment. Both experiments and applications show that there is no perfect SAT solving approach that is suited for all different categories and families of problem instances. However, conflict-driven solving has proven itself to be very successful on a wide range of benchmarks. In this paper we study the quite new DMRP algorithm (decision making with a reference point) [8,9] from a practical point of view. Moreover, a hybrid approach that combines DMRP and CDCL solving is presented which is also motivated by experimental evaluations.

The paper is organised as follows: In section 2 we sketch related work in the domain of CDCL and DMRP solving. Section 3 examines the DMRP approach from a practical point of view and we introduce a new implementation for this approach. In section 4 we motivate the combination of CDCL and DMRP to a new hybrid approach. In section 5 some experimental results are presented.

* This work was supported by DFG-SPP 1307, project "Structure-based Algorithm Engineering for SAT-Solving".

O. Strichman and S. Szeider (Eds.): SAT 2010, LNCS 6175, pp. 143–157, 2010.
© Springer-Verlag Berlin Heidelberg 2010

2 Related Work

In this chapter two different SAT solving approaches are sketched. The state-of-the-art conflict-driven solving and the quite recent DMRP approach that operates on complete assignments. By $\mathcal{V}(F)$ resp. $\Gamma(F)$ we state the set of variables resp. clauses of a formula F (we omit F if evident). A clause consists of literals l_i that are variables v or their negations $\overline{v}$. The polarity of a literal is *true* or $false$ respectively. $var(l)$ indicates the variable of literal l.

Conflict-Driven Solving. Conflict-driven solving with clause learning (CDCL) is a leading approach and is especially but not only successful for industrial problems. It is based on the GRASP algorithm [17] which extends the original DPLL branch-and-bound procedure [5,4] by the idea of learning from conflicting assignments. Moreover, conflicts are analysed to jump over parts of the search space that would cause further conflicts. There are several improvements to the original algorithm like the *two-watched-literal* data structure and the VSIDS (variable state independent decaying sum) variable selection heuristic [18]. In recent years further improvements have been achieved by developing different restart strategies like the concept of rapid and adaptive restarts [3,2] and so-called Luby restarts [14]. In combination with phase-saving [19] frequent restarts constitute a strong concept especially for industrial SAT instances.

Decision Making with a Reference Point. DMRP is a new SAT solving approach that was proposed by Goldberg in [8,9]. Even though DMRP uses Boolean constraint propagation (BCP) with backtracking and learning from conflicting assignments it is not a simple variant of CDCL. In difference to CDCL solvers DMRP additionally holds a complete assignment (a so-called reference point). The algorithm aims for modifying the current reference point $\mathcal{P}$ to $\mathcal{P}'$ in order to satisfy a clause under consideration. Furthermore, it is crucial that all clauses being satisfied by $\mathcal{P}$ remain satisfied by the modified reference point $\mathcal{P}'$.

Algorithm 1 gives an overview of the DMRP approach, though this notation varies in some ways from the original notation in [9]. One invocation of the DMRP subsolver (line 7 of Algorithm 1) takes a clause and a reference point as arguments. It may either compute a modification to the reference point or it may learn the empty clause or else it times out. The latter case causes the surrounding algorithm to call the DMRP subsolver with another unsatisfied clause.

3 A Closer Look at DMRP

Taking the set of clauses that are not satisfied by a current assignment as basis for branching decisions requires the solver to know this set of clauses. This could be realized analogously to how it is implemented in many local search approaches [20,7] where the solver keeps track of clauses that change their state from 'satisfied' to 'not satisfied' and vice versa whenever the value of a variable changes. However, for any variable v this implies the solver to know all clauses where v

Algorithm 1. Sketch of the DMRP approach

Require Formula F in CNF with $\mathcal{V}, \Gamma$ the set of variables and clauses, a reference point $\mathcal{P}$ and any two timeout criteria T_1, T_2

Function `solveDMRP`$(F, \mathcal{P}, T_1, T_2)$
  $\mathcal{M} \leftarrow \{C \in \Gamma(F) \mid C$ not satisfied by $\mathcal{P}\}$;
  while $\neg T_1$ **do**
    if $\mathcal{M} = \emptyset$ **then return** 'Satisfiable';
6   $C \leftarrow$ remove any clause from $\mathcal{M}$;
7   $res \leftarrow$ `dmrpTryModifyPoint`$(F \setminus \mathcal{M}, C, \mathcal{P}, T_2)$;
    if $res =$ *'Unsatisfiable'* **then return** 'Unsatisfiable';
    else if $res =$ *'Timeout'* **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{C\}$;
    else
      $\mathcal{P} \leftarrow$ `modify`$(\mathcal{P}, res)$; `/* adapt ref. point */`
      $\mathcal{M} \leftarrow \{C \in \Gamma \mid C$ not satisfied by $\mathcal{P}\}$;

Require (Sub)formula F', a clause C that shall be satisfied by modification of the current point $\mathcal{P}$, and a timeout criteria T

14 **Function** `dmrpTryModifyPoint`$(F', C, \mathcal{P}, T)$
  $\mathcal{D} \leftarrow \{C\}$ $\mathcal{P}_t \leftarrow \mathcal{P}$;
  while $\neg T$ **do**
    if $\mathcal{D} = \emptyset$ **then return** $\mathcal{P}_t$; `/* found valid modification of` $\mathcal{P}$ `*/`
    $C \leftarrow$ choose any clause from $\mathcal{D}$;
    $l \leftarrow l \in C \mid \mathcal{P}_t \setminus \{\bar{l}\} \cup \{l\}$ satisfies maximal number of clauses in $\mathcal{D}$;
    $< res, \mathcal{P}_t > \leftarrow$ `boolean-constraint-propagation`$(F', l := true, \mathcal{P}_t)$;
    while $res =$ *'Conflict'* **do**
22    $lemma \leftarrow$ `analyze-conflict`(F', res) ;
      if $lemma = \emptyset$ **then return** 'Unsatisfiable';
24    $\mathcal{P}_t \leftarrow$ `backtrack-reset-point`$(F', lemma)$;
25    $< res, \mathcal{P}_t > \leftarrow$ `learn-and-propagate`$(F', lemma, \mathcal{P}_t)$;
    $\mathcal{D} \leftarrow \{C \in \Gamma(F') \mid C$ not satisfied by $\mathcal{P}_t\}$;
  return 'Timeout'

resp. $\overline{v}$ occurs in. Since the *two watched literals* scheme was introduced [18] most CDCL based solvers do not maintain complete *occurrence-lists* of variables.

In this section we present a data structure that allows for a fast computation of the most frequently required information in the DMRP approach by simultaneously avoiding the maintenance of complete occurrence-lists.

3.1 Different States of Variables

In CDCL solvers each variable v can actually have three values: $val(v) \in \{true, false, unknown\}$. In general, any variable whose value is known has either been chosen as decision variable or its value was implied by BCP. To undo decisions and their implications both types of assignments (decisions and implications) are placed on a stack (often called trail) in the order they are assigned [6,3].

In the DMRP algorithm we introduce two different kinds of values expressed by the functions *pval* and *tval*: The DMRP algorithm maintains a reference

point $\mathcal{P}$ which is an assignment to all the variables in the formula. Hence, for any variable v in the formula the reference point $\mathcal{P}$ either contains v or its negation $\overline{v}$. For a variable v we refer to its value in $\mathcal{P}$ by $pval(v) \in \{true, false\}$. The second kind of value $tval(v)$ is used to state a temporary modification of $pval(v)$. The default of $tval(v) \in \{true, false, ref\}$ is ref which indicates that the corresponding variable is not affected by the current temporary modification of $\mathcal{P}$ and hence the value given by $pval(v)$ is valid. During the search for a modification of $\mathcal{P}$ to $\mathcal{P}'$ (line 14 of Algorithm 1) that reduces the set of unsatisfied clauses $\mathcal{M}$ to $\mathcal{M}' \subset \mathcal{M}$ the temporary value $tval(v) \neq ref$ hides $pval(v)$ for any variable v. For any literal l with polarity b the function $pval(l)$ (resp. $tval(l)$) is true iff $pval(var(l)) = b$ (resp. $tval(var(l)) = b$) and it is $tval(l) = ref$ iff $tval(var(l)) = ref$.

3.2 Clauses Satisfied by the Reference Point

In addition to standard SAT solving the algorithm has to maintain a reference point $\mathcal{P}$. Obviously, if all clauses Γ are satisfied by $\mathcal{P}$ the algorithm has found a model for the formula. Hence, for the remaining section we assume the set of clauses $\mathcal{M}$ that contains all clauses not satisfied by $\mathcal{P}$ to be non-empty.

After any initialisation of $\mathcal{P}$ the set $\mathcal{M}$ can be computed by simply traversing Γ. However, whilst the algorithm tries to modify $\mathcal{P}$ in order to satisfy more clauses of $\mathcal{M}$ we have to keep track of those clauses in $\Gamma \setminus \mathcal{M}$ that become temporarily unsatisfied by a temporarily modified reference point. These clauses are put onto a stack $\mathcal{D}$ which is described further below. The first matter is how to compute the clauses that become unsatisfied by a modification of the reference point.

Similar to the concept of watched literals [18] for each clause C in $\Gamma \setminus \mathcal{M}$ we choose one literal $l \in \mathcal{P}$ to take on responsibility for C regarding its satisfiability by the current reference point $\mathcal{P}$. By definition for any clause in $\Gamma \setminus \mathcal{M}$ at least one such literal $l \in C$ has to exist with $pval(l) = true$. We say a literal l is responsible for a set of clauses $R(l)$. Whenever the value of a variable $v := var(l)$ changes from $tval(v) = ref$ to $\neg pval(v)$ all clauses in $R(l)$ have to be traversed. For each clause $C \in R(l)$ a new literal from the current (modified) reference point has to be found that takes on responsibility for C.

Note that - in addition to the responsibilities regarding the reference point - there are also two literals per clause that watch this clause in the sense of the usual two-watched-literal scheme [18]. This is necessary to notice whenever a temporary modification ($tval$) generates a unit clause or completely unsatisfies a clause. Let the set of clauses that are watched by a literal l be $W(l)$. We examine this in more detail now.

Whenever the value of a variable is changed whilst searching for a modified reference point $\mathcal{P}'$ ($tval$ is changed) we have to take care of $W(l)$ and $R(l)$ of the corresponding literal l that became false under $tval$. When examining $W(l)$ the usual three cases may happen for any affected clause C that is watched by l and any other literal $l_w \in C$. For these cases only the values of $tval$ act a part:

W.1 Another literal $l_j \in \{C \setminus l_w\}$ with $tval(l_j) \neq false$ can watch C.
W.2 There is no other literal in $\{C \setminus l_w\}$ that is not false. Hence $tval(l_w)$ has to be set to $true$ to satisfy C.
W.3 If in the second case above $tval(l_w)$ is already set to $false$ a conflicting assignment is generated and the algorithm jumps back to resolve the conflict.

The following update is done after the list $W(l)$ was examined successfully:

R.1 If $tval(l)$ equals $pval(l)$ nothing has to be done.
R.2 $tval(l)$ differs from $pval(l)$ and another literal in C can be found to take responsibility for C. This might be any literal $l_j \in C$ for which it is $tval(l_j) = ref$ and $pval(l_j) = true$. In that case C is removed from $R(l)$ and put into $R(l_j)$. Or we might find a literal with $tval(l_j) = true$. In that case C remains in $R(l)$ since $tval(l_j)$ was obviously assigned before the current modification of l in the reference point.
R.3 $tval(l)$ differs from $pval(l)$ but no other literal $\in C$ satisfies C under the current temporary point. In that case C is put on the stack $\mathcal{D}$ that keeps track of all clauses that are not satisfied by the current temporary reference point. Note that since $W(l)$ was examined first there are at least two literals $l_i, l_j \in C$ for which $tval(l_i) = tval(l_j) = ref$ and $pval(l_i) = pval(l_j) = false$. If this did not hold one of the cases W.2, W.3 or R.2 would apply.

Note that this implementation (sketched in Algorithm 2) allows for backtracking without any updates of the sets $R(l)$ of any literal l. The responsibility list[1] $R(l)$ only has to be examined when $tval$ of a variable changes from ref to $true$ or $false$ not for the opposite case. Moreover, the data structure is sound in the sense that no clause that becomes unsatisfied by $\mathcal{P}$ will be missed.

3.3 Keeping Track of Temporarily Unsatisfied Clauses

While trying to modify a reference point $\mathcal{P}$ to $\mathcal{P}'$ to reduce the set $\mathcal{M}$ of clauses that are unsatisfied by $\mathcal{P}$ to $\mathcal{M}' \subset \mathcal{M}$ a data structure $\mathcal{D}$ is used to store those clauses that are unsatisfied by any temporary reference point $\mathcal{P}_t$. In the subsection above we described when clauses are added to $\mathcal{D}$. The data structure $\mathcal{D}$ has to meet three main demands:

- Clauses that are not satisfied by the current point $\mathcal{P}_t$ have to be found in reasonable time without having to traverse the clause's literals at each look-up in the data structure.
- Backjumping over parts of the temporary modification (due to a conflict - see case W.3) has to be very fast with least possible overhead to update $\mathcal{D}$.

[1] For any literal l the list $R(l)$ is only meaningful if $pval(l) = true$. To save memory, responsibility lists can be associated with variables in practice.

Algorithm 2. Update data structure when the value of a variable changed

```
Require A variable v ∈ 𝒱 where tval(v) was changed from ref to
b ∈ {true, false}. Let l_c be the literal of v with tval(l_c) = false.
Function onChangeOfVariableTVal(v)
    forall C ∈ W(l_c) do
        l_w ← other watcher of C (l_w ≠ l_c) ;
        if tval(l_w) = true then continue;
        if ∃ l_n ∈ C : tval(l_n) ≠ false ∧ l_n ≠ l_w then
            W(l_c) ← W(l_c) \ {C};   W(l_n) ← W(l_n) ∪ {C}
        else if tval(l_w) = false then return 'conflict';
        else tval(l_w) ← true; ;      /* usual two watched literal scheme */
    if tval(v) = pval(v) then return 'ok' ;                          /* R.1 */
    forall C ∈ R(¬l_c) do
        if ∃ l_0 ∈ C : tval(l_0) = ref ∧ pval(l_0) = true then
            R(¬l_c) ← R(¬l_c) \ {C}; R(l_0) ← R(l_0) ∪ {C} ;        /* R.2.1 */
        else if ∃ l_0 ∈ C : tval(l_0) = true then continue ;        /* R.2.2 */
        else push C at 𝒟 ;                                            /* R.3 */
```

– When a temporarily unsatisfied clause C is chosen from $\mathcal{D}$ by the decision procedure the set $\Lambda_C = \{l \in C \ : tval(l) = ref\}$ contains those literals whose value in $\mathcal{P}_t$ may possibly be modified to satisfy C (as stated in case R.3 above it is $|\Lambda_C| \geq 2$). Our data structure has to support finding that literal of Λ_C which satisfies the most clauses in $\mathcal{D}$.

Fast Backjumping. The main data structure is depicted in Figure 1. Since the second issue above is fundamental we use a stack to realise $\mathcal{D}$ which has one entry pL for each decision level. Each entry itself basically points to a set of clauses L that became unsatisfied by the current point $\mathcal{P}_t$ at this level. In addition each L has a flag that indicates if the referred clauses still have to be considered to belong to $\mathcal{D}$. This allows for very fast backjumping: For each level we jump back the according flag in L is set to false, the set of clauses in L is deleted and pL is popped from the stack. This means a negligible overhead compared to backjumping in CDCL solving. Note, that an entry pL which is removed from the stack does not destroy the corresponding set L which allows other data still to refer to L. These invalid references may be updated lazily later on. Another important advantage of this implementation will become evident further below.

Finding clauses not satisfied by $\mathcal{P}_t$. To find those clauses in $\mathcal{D}$ that still have to be satisfied by further modifications of the current point $\mathcal{P}_t$ the clause sets L that are referred by entries pL of the stack have to be traversed. Let this procedure be called *findUnsat($\mathcal{D}$)*. We do not remove any satisfied clause C from any set L that is still flagged to be valid since this would require to put such clauses back into L whenever the satisfying modification to C is undone. Instead, we additionally cache a literal for each clause in L as a kind of representative.

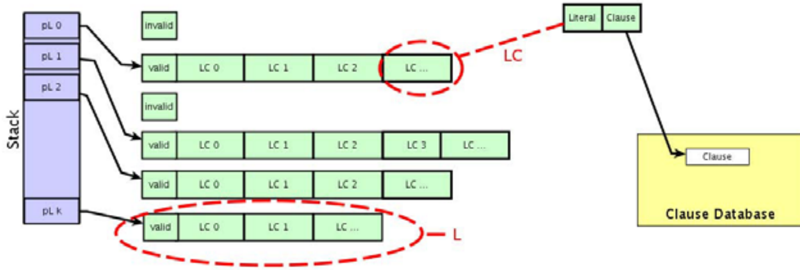

Fig. 1. Basic data structure to represent $\mathcal{D}$

Thus any entry in L (besides the flag) is an element LC which itself consists of a clause C and one representative literal $l \in C$. When a clause C in a set L is found to be satisfied by a temporary point $\mathcal{P}_t$ the representative literal l is set to that one which actually satisfies this clause ($tval(l) = true$).

When searching for unsatisfied clauses in $\mathcal{D}$ we first check the state of the representative literal before the entire clause is checked. On the one hand this guarantees that a satisfied clause is not checked twice unless a modification makes this necessary. On the other hand significant changes of $\mathcal{P}_t$ to the satisfiability state of any clause are not missed. The latter issue would require extra maintenance if only Boolean flags were used to mark satisfied clauses. To find clauses in $\mathcal{D}$ that are still unsatisfied by the current point $\mathcal{P}_t$ we traverse the stack from its top[2] to its bottom. This prefers the most recently added clauses. The first clause that is found to be unsatisfied by $\mathcal{P}_t$ is taken as basis for the next branching decision.

Computation of the MakeCount of variables. Given a clause C^* that is unsatisfied under the current point $\mathcal{P}_t$ the algorithm has to find that literal of $\Lambda_{C^*} = \{l \in C^* \ : tval(l) = ref\}$ which satisfies most clauses in $\mathcal{D}$ (or optionally $\mathcal{D} \cup \mathcal{M}$) when its value in $\mathcal{P}_t$ is changed. To compute this so-called *MakeCount* of a variable we use another data structure that interplays with the above one. This data structure allows for lazy computation of the MakeCount of a variable and is basically organised as follows:

Each variable v that is not yet affected by the temporary modification of the reference point $\mathcal{P}_t$ ($tval(v) = ref$) is associated with a list Ω_v of elements M. Each element represents a clause in $\mathcal{D}$ that can be satisfied by flipping the current value of v in the point $\mathcal{P}_t$. Due to the laziness of the data structure it might be that an element M is out of date. More precisely (see Figure 2) each element M (representing a clause C) in a list of variable v consists of two fields:

The first field references the set L of clauses in which C is contained. The second field is an index into L that indicates the particular clause C (i.e. the according element LC) that can be satisfied by flipping the value of v in $\mathcal{P}_t$.

[2] An index into $\mathcal{D}$ can be cached such that search only starts from the top of $\mathcal{D}$ if a conflict occurred at the previous decision.

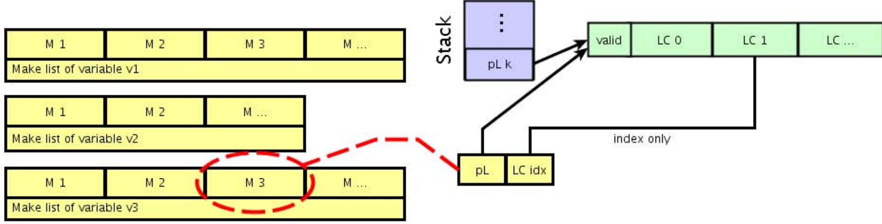

Fig. 2. Computation of the *MakeCount* of variables

Whenever case $R.3$ from above applies for a clause C a pointer to C and any representing literal are wrapped into an object LC. This data is appended to the set of clauses L which is referenced from the topmost entry on the stack $\mathcal{D}$. At this point we also add an entry M to the Make-lists (Ω_v) of all variables v of $\Lambda_C = \{l \in C : tval(l) = ref\}$. We take a closer look at the different cases when computing the valid MakeCount from the possibly out-of-date information:

– It might be that an element M refers to a clause that has already been removed from $\mathcal{D}$. In that case the flag of the structure L referenced by M has been invalidated during backtracking. Hence, this case can be realised immediately and M can also be deleted from the list.
– We can assume now that the clause C indirectly referenced by M is still contained in $\mathcal{D}$. Let us assume for now that C is already satisfied by a further modification of the point $\mathcal{P}_t$. Recall, that what we actually get from M is a reference to LC - the clause C and a representing literal l of C. We can distinguish between two cases:
  - C might have been already considered by the procedure *findUnsat($\mathcal{D}$)* to find unsatisfied clauses in $\mathcal{D}$ as described above. In that case *findUnsat()* has changed the representative literal l such that by checking the value ($tval$) of l in $\mathcal{P}_t$ we know that C is satisfied and we are done.
  - If C was not considered by *findUnsat($\mathcal{D}$)* yet, the satisfiability state of the clause has to be computed by checking its literals. Given that C is satisfiable under $\mathcal{P}_t$ a literal that satisfies C will be found and will be made the representative literal for this clause in LC. This allows for fast detection of the satisfiability of C later on and will relieve *findUnsat($\mathcal{D}$)* from checking all literals of C. The representing literal guarantees that for each temporary point $\mathcal{P}_t$ there is at most one traversal through all literals of a clause to recognise that this clause is satisfied by $\mathcal{P}_t$.
– In case C is not satisfied by the current point $\mathcal{P}_t$ this is recognised by a check of all literals in C.

The realisation of the set $\mathcal{D}$ and the data structure to compute MakeCounts of variables follows the concept of lazy data structures and avoids to store complete occurrence lists for literals. MakeCount lists do not require any update operation on backjumping. Even though indices in M become undefined when the referred

set L is cleared during backjumping, this is not problematic since an index is only used after L is asserted to be still valid by its flag.

The size of a list Ω_v of a variable v gives an upper bound $\widehat{\Omega_v}$ on the valid MakeCount of v. Hence, to determine the variable with maximum MakeCount of a clause, the variables are traversed in descending order regarding $\widehat{\Omega}$. This allows for early termination when $\widehat{\Omega}$ becomes smaller than the actual valid maximum MakeCount.

3.4 Learning

Two aspects that have to be considered for the realisation of the DMRP approach are related to learning (Algorithm 1, line 22) as mentioned in case W.3:

Whenever a unit clause $C = (l)$ is learned the algorithm jumps back to decision level 0, assigns l to be true and propagates all implications of this assignment. This also requires a modification of the current reference point $\mathcal{P}$ to $\mathcal{P}'$ with a difference to previously described modifications: The set of clauses $\mathcal{M}'$ unsatisfied by $\mathcal{P}'$ does not necessarily have to be a subset of $\mathcal{M}$ – the set of clauses unsatisfied by the previous reference point $\mathcal{P}$.

Secondly, for any learned lemma that is generated when a conflict is analysed the data structure has to be updated properly. We use the following property.

Property: Any lemma generated by the function *analyze-conflict* in line 22 of Algorithm 1 contains at least one literal l with $l \in \mathcal{P}$ $(pval(l) = true)$ regarding the current valid reference point $\mathcal{P}$. In other words: No generated lemma extends[3] the current set $\mathcal{M}$.

Proof: We prove this property by the construction of learned lemmata. The surrounding function *dmrpTryModifyPoint* (lines 7, 14 of Algorithm 1) only considers clauses in $\Gamma \setminus \mathcal{M} \cup C$ where a modification of $\mathcal{P}$ is wanted that additionally satisfies C. Since C is always the base for the decision at the first decision level any temporary point $\mathcal{P}_t \neq \mathcal{P}$ will always satisfy C during one execution of *dmrpTryModifyPoint*. Hence, C can never be an assign-reason to an assignment of a variable, since assign-reasons are clauses that become unit during the search. Thus, all assign-reasons are clauses from the set $\Gamma \setminus \mathcal{M}$ that, by definition, are satisfied by at least one literal $\in \mathcal{P}$.

Running into a conflict means that for a clause C_0 (conflicting clause) all literals are set to false. The lemma λ_* is generated by recursively resolving out variables (that were no decisions) from the conflicting clause by using the according assign-reasons. C_0 can be seen as the first version (λ_0) of the generated lemma λ_*. Given that $C_0 \in \Gamma \setminus \mathcal{M}$ one of the literals of λ_0 is in $\mathcal{P}$. Let $l*$ be one literal $\in \lambda_i$ with $l* \in \mathcal{P}$ $(pval(l*) = true)$. If any literal $l' \neq l*$ is resolved out from λ_i the resolvent λ_{i+1} still contains literal $l*$. If on the other hand $l*$ is resolved out by the use of its assign reason C^*, clause C^* has to contain literal $\overline{l*}$. Since $C^* \in \Gamma \setminus \mathcal{M}$ it also has to contain a literal $l^\star \in \mathcal{P}$. Moreover, with $l* \in \mathcal{P}$ it is $l^\star \neq \overline{l*}$ and the new resolvent λ_{i+1} contains literal $l^\star \in \mathcal{P}$. By induction the final lemma contains at least one literal that is in $\mathcal{P}$. $\square$

[3] Note the difference that generated lemmata always extend the formula.

For any generated lemma λ_* we chose that literal $l \in \mathcal{P} \cap \lambda_*$ which was assigned at the highest decision level d (most recently). By the above property at least one such literal l has to exist. Literal l takes on responsibility for λ_*: $R(l) \leftarrow R(l) \cup \{\lambda_*\}$. The functions in lines 24 and 25 of Algorithm 1 determine a new point $\mathcal{P}_t$. If $l \notin \mathcal{P}_t$ the lemma λ_* is also appended to the list L that is referred by the stack $\mathcal{D}$ for decision level d and considered for the MakeCounts as described in the previous section 3.3. These two actions guarantee a proper update of the entire data structure and no more special treatments are needed.

4 Combining DMRP and CDCL to a Hybrid Solver

In Algorithm 1 we assume the initial reference point to be given from outside. In the original paper [9] reference points are chosen at random and are then tried to modify by a call to function *solveDMRP*. In case no result can be computed within a certain amount of time (i.e. number of conflicts) *solveDMRP* will be invoked with a new initial point. This is similar to local search restarts but with the difference that the DMRP algorithm itself carefully reasons on how to modify a reference point. However, the choices of initial points are crucial for the algorithm as presented in section 5.

As mentioned in section 2 CDCL solvers perform restarts quite frequently. At a restart activity values of variables or literals are kept and also a subset of the learned clauses is carried along for the next start. However, the current partial assignment (all literals in the trail) is almost completely rejected, even though phase saving keeps some information. This motivates a hybrid approach that reasonably alternates the CDCL and the DMRP algorithms. The DMRP approach offers a suitable possibility to take a closer look at the drawback of a partial assignment before it is rejected. It may focus on the not yet satisfied clauses.

Our recent implementation that is shown in Algorithm 3 combines both approaches by the use of the Luby et al. restart strategy [14] which proved itself successful in both theory and practice. The Luby strategy assumes that the algorithm does not have any external information and does not know when it is best to perform a restart. In that case the available computation time is shared almost equally among different restart strategies [14]. The function *maxConflictCount* in Algorithm 3 returns the number of conflicts for the next run due to the Luby strategy. That is the product of a constant factor f and the next number of the sequence $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, \ldots)$ (see [14] for details).

The function *chooseAlgo* decides on which algorithm to use for the next run. On average we achieved the best results when running the DMRP algorithm exactly for the smallest conflict limit (when $cl = f$).

Since the DMRP algorithm requires a reference point i.e. an assignment to all variables the last partial assignment of the CDCL solver has to be extended to a complete assignment ($extendPartialAssignmToRefPoint$). This is done by continuing the previous CDCL search with the last partial assignment. However, within this execution only binary clauses are considered during search until all

Algorithm 3. The hybrid approach

```
Require Formula F in CNF with V, Γ the set of variables and clauses
Function solveHybrid(F)
    last ← 'CDCL'    res ← 'Unknown' ;
    while res = 'Unknown' do
        cl ← maxConflictCount() ;                    /* Use Luby strategy */
        algo ← chooseAlgo(cl) ;                      /* Apply CDCL or DMRP ? */
        if algo = 'DMRP' then
            if last = 'CDCL' then
                < res, P >← extendPartialAssignmToRefPoint() ;
                if res = 'Unsatisfiable' then return res;
            res ← solveDMRP(F, P, cl, cl);
        else res ← solveCDCL(F, cl);
        last ← algo ;
    return res;
```

variables are assigned a value. This assignment constitutes the initial reference point for the DMRP algorithm. In this phase the solver may also realise that the formula is unsatisfiable. For the case the partial assignment is empty (at algorithm start) this function simply computes a reference point that satisfies all binary clauses. Taking care of binary clauses at first is also motivated by the work in [22] and [1] where the idea to primarily focus on binary clauses has also improved solving for some families of instances. This also guarantees an additional invariant for our data structure that a binary clause can neither be contained in the set $\mathcal{M}$ nor in the delta stack $\mathcal{D}$ (resp. its elements).

Some Adaptions for the Hybrid Approach

In addition to standard CDCL solving each clause of the formula is assigned an activity value initialised to zero at the beginning. Whenever a clause is involved in a conflict (i.e. it is used for resolution during the generation of a lemma) its activity value is increased. In some solvers (for instance [6]) this technique is common for learned clauses to clear the clause database of inactive learned clauses periodically. Our hybrid solver maintains an activity value for every clause.

This activity value of a clause is taken into account when the next clause from set $\mathcal{M}$ has to be chosen (line 6 of Algorithm 1) to be handled by the function *dmrpTryModifyPoint*. We always choose the clause with the highest activity value for the next attempt to modify the current reference point. However, if the call to *dmrpTryModifyPoint* times out or for two subsequent calls to *solveDMRP* the next clause with the second highest activity value is chosen.

In contrast to the original DMRP algorithm the conflict limit (timeout) for the function *solveDMRP* depends on the success of its subroutine *dmrpTryModifyPoint* in line 7 of Algorithm 1. If the current reference point could be improved the initial conflict limit is reset.

The solver also differs in the computation of the MakeCount of a variable. For the MakeCount one can count only the clauses currently in $\mathcal{D}$ to get the most local improvement or all clauses in $\mathcal{D} \cup \mathcal{M}$ can be considered to make decisions more globally. For variables that have the same MakeCount ties can be broken in favour of different issues which is explained in more detail in the next section.

5 Experiments and Evaluation

For the evaluation presented in this section we have run our solver for all industrial (application) instances of the SAT competitions resp. SAT Races of the years 2006 - 2009 that add up to 564 non-trivial[4] instances. Each instance is preprocessed in advance and the timeout for the solvers was set to 1200 seconds. As a reference and also to check results we have run our CDCL solver using the Luby restart strategy (without DMRP) and MiniSAT 2.0 [6].

Figure 3 shows the results of different configurations of the hybrid approach. Furthermore, there are results that show performance of a pure DMRP solver. The presented configurations differ in the following issues that are related to decision making: As mentioned above the MakeCount may consider all clauses in $\mathcal{D} \cup \mathcal{M}$ (global) or only clauses in $\mathcal{D}$ (local). If two variables have the same MakeCount ties are broken in favour of the variable v that:

(Act) has the highest activity value similar to the VSIDS heuristic [18].
(BC) has the smallest set $R(v)$. This can be seen as a simple approximation of the *BreakCount* of the variable. In difference to the MakeCount the BreakCount of a variable v states the number of clauses that become unsatisfied by a flip of the value of variable v.
(DC) was chosen least often for DMRP decisions. This avoids flipping always the same variables back and forth in different calls to *solveDMRP*.

The left plot of Figure 3 shows clearly that pure DMRP solving could not compete with CDCL solving. Both pure DMRP configurations (global and local MakeCount) solve around 224 of 564 instances within 1200 seconds. Initial reference points are always chosen at random. Timeouts for the analysis of one reference point (one call to *solveDMRP*) are changed according to the Luby sequence. Modifying the strategy on how to choose initial reference points showed quite some impact. Our assumption was that DMRP requires a better guidance on where to start search and how to choose its initial reference points. That motivates our hybrid approach where DMRP gets its initial reference points indirectly from the CDCL solver. As the plots show this clearly improves the performance of the solver.

A previous version of our hybrid approach [12] has taken part in the SAT competition 2009. That version mainly differs from the presented one regarding the restart strategy and the choice of when to perform DMRP resp. CDCL solving. It also implemented a more extensive solving of particular subsets of clauses which

[4] Instances that are not solved by preprocessing.

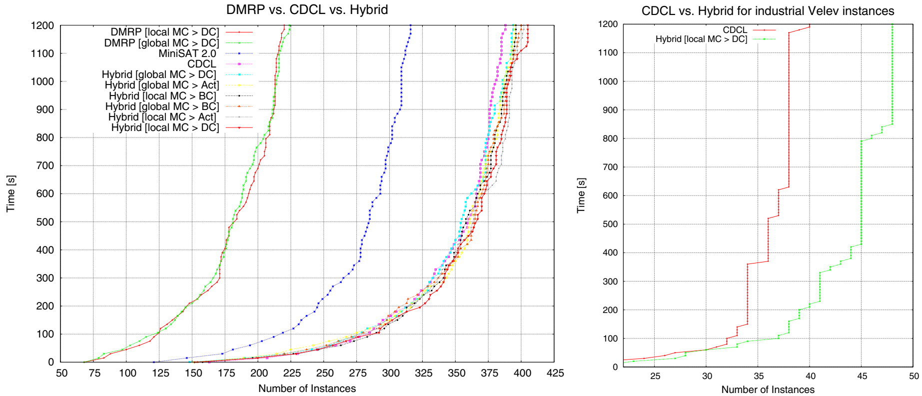

Fig. 3. The left plot compares DMRP, CDCL and our hybrid approach on 564 industrial benchmarks. The right plot compares CDCL and the hybrid approach on 51 instances from hardware verification. A point (x, y) states that x instances were solved within y sec. by that solver. Legends are ordered regarding the number of solved instances after 1200 seconds. Using local (resp. global) MakeCount and smaller decision count (resp. Activity or BreakCount) to break ties is indicated by [local MC > DC].

is only done for binary clauses in this improved approach. However, the results indicate that the older version did not utilize the DMRP approach in a sufficient way. Compared to MiniSAT 2.0 our hybrid approach also performs much better. Admittedly, this is not only due to the hybridisation with DMRP. This version of MiniSAT does neither use the Luby restart strategy nor phase saving. However, the hybrid approach also clearly outperforms our CDCL implementation with Luby restarts and phase saving.

The hybrid configuration where the MakeCount is computed locally outperformes the other configurations. It is interesting to notice that using the activity of variables to break ties does not achieve the best results. It turns out that it is better to prefer variables that were flipped least often at the current call of *solveDMRP*.

The right plot of Figure 3 compares pure CDCL with the hybrid approach on the 51 "Velev" instances of last years SAT competitions. For these instances that stem from the domain of hardware verification the hybrid approach clearly outperforms pure CDCL by solving 8 more instances.

Even though the hybrid implementation beats our pure CDCL solver on the entire benchmark set it turns out that for the most instances solved by the hybrid solver the answer was given by the CDCL part. Only about 6% were finally solved by the DMRP subsolver. Moreover, the improvement due to the hybridisation was mainly for unsatisfiable instances (17 more unsat results).

Our conjecture about this phenomenon is that DMRP generates some important lemmata: When the CDCL solver reaches the (current) maximum number of conflicts it delivers work to the DMRP solver. DMRP starts with an extension R of the last partial assignment P of the CDCL solver and hence focuses on a

nearby part of the search space. When analysing this part it purposely examines clauses $\mathcal{M}$ that are not satisfied by R. In CDCL these clauses in $\mathcal{M}$ could likely become conflicting clauses if decisions were made similar to the values in R. Up to a certain point phase saving would do this after a normal CDCL restart. However, DMRP immediately considers clauses in $\mathcal{M}$ for search and resolution.

6 Conclusion

In this paper we have presented a data structure to implement the DMRP approach in an efficient way. Similar to the two-watched-literals scheme we choose one literal for each clause. The literal takes on responsibility so that a clause which is satisfied by a reference point is also satisfied by a modification of the point. Moreover, we present a way how to determine that variable which satisfies the most previously unsatisfied clauses when its value is flipped (MakeCount). Based on this implementation we motivate a hybrid SAT solver that combines CDCL and DMRP solving. Experiments have shown that our hybrid approach is competetive to the highly optimised state-of-the-art CDCL solvers and that the maintenance of complete assignments may definitely turn to account.

References

1. Bacchus, F.: Exploring the computational tradeoff of more reasoning and less searching. In: SAT 2002, pp. 7–16 (2002)
2. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
3. Biere, A.: Picosat essentials. JSAT 4, 75–97 (2008)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. ACM Commun. 5(7), 394–397 (1962)
5. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Fukunaga, A.S.: Efficient Implementations of SAT Local Search. In: SAT (2004)
8. Goldberg, E.: Determinization of resolution by an algorithm operating on complete assignments. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 90–95. Springer, Heidelberg (2006)
9. Goldberg, E.: A decision-making procedure for resolution-based SAT-solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 119–132. Springer, Heidelberg (2008)
10. Ivancic, F., Yang, Z., Ganai, M., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theoretical Computer Science 404(3) (2008)
11. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Proceedings of the Tenth European Conference on Artificial Intelligence ECAI 1992, pp. 359–363 (1992)
12. Kottler, S.: Solver descriptions for the SAT competition (2009), `satcompetition.org`

13. Küchlin, W., Sinz, C.: Proving consistency assertions for automotive product data management. J. Automated Reasoning 24(1-2), 145–163 (2000)
14. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. In: ISTCS, pp. 128–133 (1993)
15. Lynce, I., Marques-Silva, J.: SAT in bioinformatics: Making the case with haplotype inference. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 136–141. Springer, Heidelberg (2006)
16. Marques-Silva, J.P.: Practical Applications of Boolean Satisfiability. In: Workshop on Discrete Event Systems, WODES 2008 (2008)
17. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Comput. 48(5), 506–521 (1999)
18. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC (2001)
19. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
20. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Tenth National Conference on Artificial Intelligence (1992)
21. Velev, M.N.: Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In: DATE 2002 (2002)
22. Zheng, L., Stuckey, P.J.: Improving SAT using 2SAT. In: Proceedings of the 25th Australasian Computer Science Conference, pp. 331–340. E (2002)