An Extensible SAT-solver

Niklas Eén and Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract. In this article[1], we present a small, complete, and efficient SAT-solver in the style of conflict-driven learning, as exemplified by CHAFF. We aim to give sufficient details about implementation to enable the reader to construct his or her own solver in a very short time. This will allow *users* of SAT-solvers to make domain specific extensions or adaptions of current state-of-the-art SAT-techniques, to meet the needs of a particular application area. The presented solver is designed with this in mind, and includes among other things a mechanism for adding arbitrary boolean constraints. It also supports solving a series of related SAT-problems efficiently by an incremental SAT-interface.

1 Introduction

The use of SAT-solvers in various applications is on the march. As insight on how to efficiently encode problems into SAT is increasing, a growing number of problem domains are successfully being tackled by SAT-solvers. This is particularly true for the *electronic design automation* (EDA) industry [BC$^+$99, Lar92]. The success is further magnified by current state-of-the-art solvers being adapted to meet the specific characteristics of these problem domains [AR$^+$02, ES03].

However, modifying an existing solver, even with thorough knowledge of both the problem domain and of modern SAT-techniques, can be a time consuming journey into the inner workings of a ten-thousand-line software package. Likewise, writing a solver from scratch often means spending much time rediscovering the intricate details of a correct and efficient solver. The problem is that although the *techniques* used in a modern SAT-solver are well documented, the details necessary for an *implementation* have not been adequately presented before.

In the fall of 2002, the authors implemented the solvers SATZOO and SATNIK. In order to sufficiently understand the implementation tricks needed for a modern SAT-solver, it was necessary to consult the source-code of previous implementations.[2] We find that the material contained therein can be made more accessible, which is desirable for the SAT-community. Thus, the principal goal of this article is to bridge the gap between existing descriptions of SAT-techniques and their actual implementation.

[1] Extended version available at http://www.cs.chalmers.se/~een

[2] LIMMAT at http://www.inf.ethz.ch/personal/biere/projects/limmat
ZCHAFF at http://www.ee.princeton.edu/~chaff/zchaff

E. Giunchiglia and A. Tacchella (Eds.): SAT 2003, LNCS 2919, pp. 502–518, 2004.

We will do this by presenting the code of a minimal SAT-solver MiniSat, based on the ideas for conflict-driven backtracking [MS96], together with watched literals and dynamic variable ordering [MZ01]. The original C++ source code (downloadable from `http://www.cs.chalmers.se/~een`) for MiniSat is under 600 lines (not counting comments), and is the result of rethinking and simplifying the designs of Satzoo and Satnik without sacrificing efficiency. We will present all the relevant parts of the code in a manner that should be accessible to anyone acquainted with either C++ or Java.

The presented code includes an incremental SAT-interface, which allows for a series of related problems to be solved with potentially huge efficiency gains [ES03]. We also generalize the expressiveness of the SAT-problem formulation by providing a mechanism for defining arbitrary *constraints* over boolean variables.

From the documentation in this paper we hope it is possible for *you* to implement a fresh SAT-solver in your favorite language, or to grab the C++ version of MiniSat from the net and start modifying it to include new and interesting ideas.

2 Application Programming Interface

We start by presenting MiniSat's external interface, with which a user application can specify and solve SAT-problems. A basic knowledge about SAT is assumed (see for instance [MS96]). The types ***var***, ***lit***, and ***Vec*** for variables, literals, and vectors respectively are explained in more detail in section 4.

```
class Solver – Public interface
    var          newVar     ()
    bool         addClause  (Vec⟨lit⟩ literals)
    bool         add...     (...)
    bool         simplifyDB ()
    bool         solve      (Vec⟨lit⟩ assumptions)

    Vec⟨bool⟩ model         – If found, this vector has the model.
```

The "*add...*" method should be understood as a place-holder for additional constraints implemented in an extension of MiniSat.

For a standard SAT-problem, the interface is used in the following way: Variables are introduced by calling *newVar()*. From these variables, clauses are built and added by *addClause()*. Trivial conflicts, such as two unit clauses $\{x\}$ and $\{\overline{x}\}$ being added, can be detected by *addClause()*, in which case it returns False. From this point on, the solver state is undefined and must not be used further. If no such trivial conflict is detected during the clause insertion phase, *solve()* is called with an empty list of assumptions. It returns False if the problem is *unsatisfiable*, and True if it is *satisfiable*, in which case the model can be read from the public vector "model".

The *simplifyDB()* method can be used before calling *solve()* to simplify the set of problem constraints (often called the *constraint database*). In our imple-

mentation, *simplifyDB()* will first propagate all unit information, then remove all satisfied constraints. As for *addClause()*, the simplifier can sometimes detect a conflict, in which case FALSE is returned and the solver state is, again, undefined and must not be used further.

If the solver returns *satisfiable*, new constraints can be added repeatedly to the existing database and *solve()* run again. However, more interesting sequences of SAT-problems can be solved by the use of *unit assumptions*. When passing a non-empty list of assumptions to *solve()*, the solver temporarily assumes the literals to be true. After finding a model or a contradiction, these assumptions are undone, and the solver is returned to a usable state, even when *solve()* return FALSE, which now should be interpreted as *unsatisfiable under assumptions*.

For this to work, calling *simplifyDB()* before *solve()* is no longer optional. It is the mechanism for detecting conflicts independent of the assumptions – referred to as a *top-level* conflict from now on – which puts the solver in an undefined state. For an example of the use if unit assumptions, see [ES03].

An alternative interface would be for *solve()* to return one of three values: *satisfiable, unsatisfiable,* or *unsatisfiable under assumptions*. This is indeed a less error-prone interface as there is no longer a pre-condition on the use of *solve()*. The current interface, however, represents the smallest modification of a non-incremental SAT-solver.

3 Overview of the SAT-solver

This article will treat the popular style of SAT-solvers based on the DPLL algorithm [DLL62], backtracking by conflict analysis and clause recording (also referred to as *learning*) [MS96], and boolean constraint propagation (BCP) using *watched literals* [MZ01]. We will refer to this style of solver as a *conflict-driven SAT-solver*. The components of such a solver, and indeed a more general constraint solver, can be conceptually divided into three categories:

- **Representation.** Somehow the SAT-instance must be represented by internal data structures, as must any derived information.
- **Inference**. Brute force search is seldom good enough on its own. A solver also needs some mechanism for computing and propagating the direct implications of the current state of information.
- **Search**. Inference is almost always combined with search to make the solver complete. The search can be viewed as another way of deriving information.

A standard conflict-driven SAT-solver can represent *clauses* (with two literals or more) and *assignments*. Although the assignments can be viewed as unit-clauses, they are treated specially, and are best viewed as a separate type of information.

The only inference mechanism used by a standard solver is *unit propagation*. As soon as a clause becomes *unit* under the current assignment (all literals except one are false), the remaining unbound literal is asserted, possibly making more clauses unit. The process continues until no more information can be propagated.

The search procedure of a modern solver is the most complex part. Heuristically, variables are picked and assigned values (*assumptions* are made), until the propagation detects a *conflict* (all literals of a clause have become false). At that point, a so called *conflict clause* is constructed and added to the SAT problem. Assumptions are then canceled by backtracking until the conflict clause becomes unit, at which point it is propagated and the search process continues.

MiniSat is extensible with arbitrary boolean constraints. This will affect the *representation*, which must be able to store these constraints; the *inference*, which must be able to derive unit information from these constraints; and the *search*, which must be able to analyze and generate conflict clauses from the constraints. The mechanism we suggest for managing general constraints is very lightweight, and by making the dependencies between the SAT-algorithm and the constraints implementation explicit, it adds to the clarity of the solver.

Propagation. The propagation procedure of MiniSat is largely inspired by that of Chaff [MZ01]. For each literal, a list of constraints is kept. These are the constraints that *may* propagate unit information (variable assignments) if the literal becomes True. For clauses, no unit information can be propagated until all literals except one have become False. Two unbound literals p and q of the clause are therefore selected, and references to the clause are added to the lists of $\overline{p}$ and $\overline{q}$ respectively. The literals are said to be *watched* and the lists of constraints are referred to as *watcher lists*. As soon as a watched literal becomes True, the constraint is invoked to see if information may be propagated, or to select new unbound literals to be watched.

An effect of using watches for clauses is that on backtracking, no adjustment to the watcher lists need to be done. Backtracking is therefore cheap. However, for other constraint types, this is not necessarily a good approach. MiniSat therefore supports the optional use of *undo lists* for those constraints; storing what constraints need to be updated when backtracking unbinds a variable.

Learning. The learning procedure of MiniSat follows the ideas of Marques-Silva and Sakallah in [MS96]. The process starts when a constraint becomes conflicting (impossible to satisfy) under the current assignment. The conflicting constraint is then asked for a set of variable assignments that make it contradictory. For a clause, this would be all the literals of the clause (which are False under a conflict). Each of the variable assignments returned must be either an *assumption* of the search procedure, or the result of some *propagation* of a constraint. The propagating constraints are in turn asked for the set of variable assignments that made the propagation occur, continuing the analysis backwards. The procedure is repeated until some termination condition is met, resulting in a set of variable assignments that implies the conflict. A clause prohibiting that particular assignment is added to the clause database. This *learnt* (conflict) clause will always be implied by the original problem constraints.

Learnt clauses serve two purposes: they drive the backtracking and they speed up future conflicts by "caching" the reason for the conflict. Each clause

will prevent only a constant number of inferences, but as the recorded clauses start to build on each other and participate in the unit propagation, the accumulated effect of learning can be massive. However, as the set of learnt clauses increase, propagation is slowed down. Therefore, the number of learnt clauses is periodically reduced, keeping only the clauses that seem useful by some heuristic.

Search. The search procedure of a conflict-driven SAT-solver is somewhat implicit. Although a recursive definition of the procedure might be more elegant, it is typically described (and implemented) iteratively. The procedure will start by selecting an unassigned variable x (called the *decision variable*) and assume a value for it, say TRUE. The consequences of $x = \text{TRUE}$ will then be propagated, possibly resulting in more variable assignments. All variables assigned as a consequence of x is said to be from the same *decision level*, counting from 1 for the first assumption made and so forth. Assignments made before the first assumption (decision level 0) are called *top-level*.

All assignments will be stored on a stack in the order they were made; from now on referred to as the *trail*. The trail is divided into decision levels and is used to undo information during backtracking. The decision phase will continue until either all variables have been assigned, in which case we have a model, or a conflict has occurred. On conflicts, the learning procedure will be invoked and a conflict clause produced. The trail will be used to undo decisions, one level at a time, until precisely one of the literals of the learnt clause becomes unbound (they are all FALSE at the point of conflict). By construction, the conflict clause cannot go directly from conflicting to a clause with two or more unbound literals. If the clause is unit for several decision levels, it is advantageous to chose the lowest level (referred to as *backjumping* or *non-chronological backtracking* [MS96]).

```
loop
    propagate()    – propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide()    – pick a new variable and assign it
    else
        analyze()    – analyze conflict and add a conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()    – undo assignments until conflict clause is unit
```

Activity heuristics. One important technique introduced by CHAFF [MZ01] is a dynamic variable ordering based on activity (referred to as the VSIDS heuristic). The original heuristic imposes an order on *literals*, but borrowing from SATZOO, we make no distinction between p and $\overline{p}$ in MINISAT.

Each variable has an *activity* attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased. We refer to this as *bumping*. After the conflict, the activity of all the variables in the system are multiplied by a constant less than 1, thus *decaying* the activity of variables over time.

Activity is also used for clauses. When a learnt clause takes part in the conflict analysis, its activity is bumped. Inactive clauses are periodically removed.

Constraint removal. The constraint database is divided into two parts: the *problem constraints* and the *learnt clauses*. The set of learnt clauses is periodically reduced to increase the performance of propagation. This may result in a larger search space, as learnt clauses are used to crop future branches of the search tree. The balance between the two forces is delicate, and there are SAT-instances for which a big learnt clause set is advantageous, and others where a small set is better. MiniSat's default heuristic starts with a small set and gradually increases the size.

Problem constraints can also be removed if they are satisfied at the top-level. The API method *simplifyDB()* is responsible for this. The procedure is particularly important for incremental SAT-problems.

Top-level solver. The pseudo-code for the search procedure presented above suffices for a simple conflict-driven SAT-solver, but a solver *strategy* can improve the performance. A typical strategy applied by modern conflict-driven SAT-solvers is the use of *restarts* to prevent from getting stuck in a futile part of the search tree. In MiniSat we also vary the number of learnt clauses kept at a given time. Furthermore, the *solve()* method of the API supports incremental assumptions, not handled by the above pseudo-code.

4 Implementation

The following conventions are used in the code. Atomic types start with a lower-case letter and are passed by value. Composite types start with a capital letter and are passed by reference. Blocks are marked by indentation level. The bottom symbol $\bot$ always mean *undefined*; False is used to denote the boolean false.

We will use, but not specify an implementation of, the following abstract data types: ***Vec***$\langle \boldsymbol{T} \rangle$ an extensible vector of type $\boldsymbol{T}$; ***lit*** the type of literals containing a special literal $\bot_{lit}$; ***lbool*** for the lifted boolean domain containing elements $\text{True}_\bot$, $\text{False}_\bot$, and $\bot$; ***Queue***$\langle \boldsymbol{T} \rangle$ a queue of type $\boldsymbol{T}$. We also use ***var*** as a type synonym for ***int*** (for implicit documentation) with the special constant $\bot_{var}$. The literal data type has an *index*() method which converts a literal to a "small" integer suitable for array indexing.

4.1 The Solver State

A number of things need to be stored in the solver state. *Figure 2* shows the complete set of member variables of the solver type of MiniSat. A number

of trivial, one-line functions will be assumed to exist, such as *nVars()* for the number of variables etc. The interface of *VarOrder* is given in *Figure 1*, and is further explained in section 4.6. Note that the state does *not* contain a boolean "conflict" to remember if a top-level conflict has been reached. Instead we impose as an invariant that the solver must never be in a conflicting state.

4.2 Constraints

MiniSat can handle arbitrary constraints over boolean variables through the abstraction presented in *Figure 3*. Each constraint type needs to implement methods for constructing, removing, propagating and calculating reasons. In addition, methods for simplifying the constraint and updating the constraint on backtrack can be specified. The contracts of these methods are as follows:

Constructor. The constructor may only be called at the top-level. It must create and add the constraint to appropriate watcher lists after enqueuing any unit information derivable under the current top-level assignment. Should a conflict arise, this must be communicated to the caller.

Remove. The remove method supplants the destructor by receiving the solver state as a parameter. It should dispose the constraint and remove it from the watcher lists.

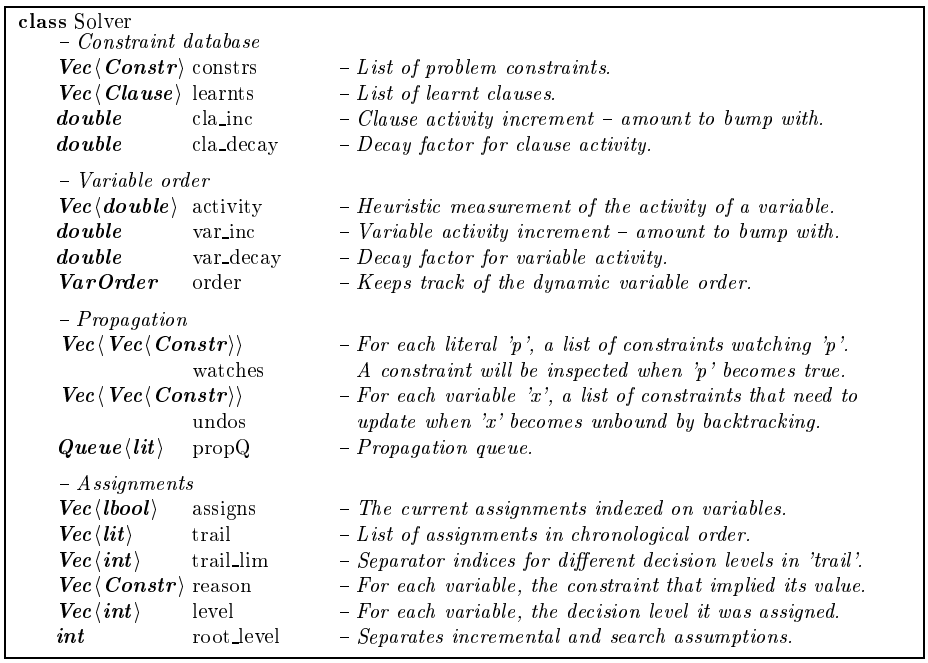
Propagate. The propagate method is called if the constraint is found in a watcher list during propagation of unit information p. The constraint is removed from the list and is required to insert itself into a new or the same watcher list. Any unit information derivable as a consequence of p should be enqueued. If successful, True is returned; if a conflict is detected, False is returned. The constraint may add itself to the undo list of $var(p)$ if it needs to be updated when p becomes unbound.

Simplify. At the top-level, a constraint may be given the opportunity to simplify its representation (returns True) or state that the constraint is satisfied under the current assignment (returns False). A constraint must *not* be simplifiable to produce unit information or to be conflicting; in that case the propagation has not been correctly defined.

```
class VarOrder – Public interface
    VarOrder ( Vec⟨lbool⟩ ref_to_assigns, Vec⟨double⟩ ref_to_activity)

    void newVar()          – Called when a new variable is created.
    void update(var x)     – Called when a variable has increased in activity.
    void updateAll()       – Called when all variables have been assigned new activities.
    void undo(var x)       – Called when variable is unbound (and may be selected again).
    var  select()          – Called to select a new, unassigned variable.
```

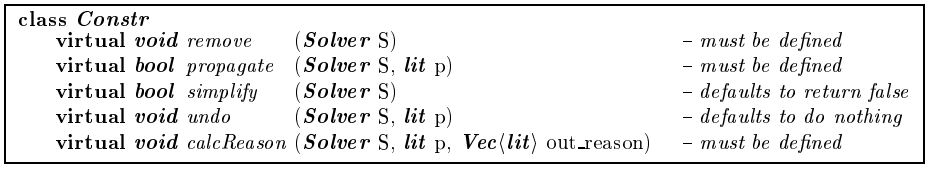
Fig. 1. Assisting ADT for the dynamic variable ordering of the solver. The constructor takes references to the assignment vector and the activity vector of the solver. The method *select()* will return the unassigned variable with the highest activity.



class Solver

– Constraint database		
Vec⟨***Constr***⟩	constrs	*– List of problem constraints.*
Vec⟨***Clause***⟩	learnts	*– List of learnt clauses.*
double	cla_inc	*– Clause activity increment – amount to bump with.*
double	cla_decay	*– Decay factor for clause activity.*
– Variable order		
Vec⟨***double***⟩	activity	*– Heuristic measurement of the activity of a variable.*
double	var_inc	*– Variable activity increment – amount to bump with.*
double	var_decay	*– Decay factor for variable activity.*
VarOrder	order	*– Keeps track of the dynamic variable order.*
– Propagation		
Vec⟨***Vec***⟨***Constr***⟩⟩ watches		*– For each literal 'p', a list of constraints watching 'p'. A constraint will be inspected when 'p' becomes true.*
Vec⟨***Vec***⟨***Constr***⟩⟩ undos		*– For each variable 'x', a list of constraints that need to update when 'x' becomes unbound by backtracking.*
Queue⟨***lit***⟩	propQ	*– Propagation queue.*
– Assignments		
Vec⟨***lbool***⟩	assigns	*– The current assignments indexed on variables.*
Vec⟨***lit***⟩	trail	*– List of assignments in chronological order.*
Vec⟨***int***⟩	trail_lim	*– Separator indices for different decision levels in 'trail'.*
Vec⟨***Constr***⟩	reason	*– For each variable, the constraint that implied its value.*
Vec⟨***int***⟩	level	*– For each variable, the decision level it was assigned.*
int	root_level	*– Separates incremental and search assumptions.*

Fig. 2. Internal state of the solver.



class ***Constr***

virtual ***void*** *remove*	(***Solver*** S)	*– must be defined*
virtual ***bool*** *propagate*	(***Solver*** S, ***lit*** p)	*– must be defined*
virtual ***bool*** *simplify*	(***Solver*** S)	*– defaults to return false*
virtual ***void*** *undo*	(***Solver*** S, ***lit*** p)	*– defaults to do nothing*
virtual ***void*** *calcReason*	(***Solver*** S, ***lit*** p, ***Vec***⟨***lit***⟩ out_reason)	*– must be defined*

Fig. 3. Abstract base class for constraints.

Undo. During backtracking, this method is called if the constraint added itself to the undo list of $var(p)$ in *propagate()*. The current variable assignments are guaranteed to be identical to that of the moment before *propagate()* was called.

Calculate Reason. This method is given a literal p and an empty vector. The constraint is the *reason* for p being true, that is, during propagation, the current constraint enqueued p. The received vector is extended to include a set of assignments (represented as literals) implying p. The current variable assignments are guaranteed to be identical to that of the moment before the constraint propagated p. The literal p is also allowed to be the special constant $\perp_{lit}$ in which case the reason for the clause being *conflicting* should be returned through the vector.

The code for the *Clause* constraint is presented in *Figure 4*. It is also used for learnt clauses, which are unique in that they can be added to the clause database

```
class Clause : public Constr
    bool      learnt
    float     activity
    Vec⟨lit⟩ lits

    – Constructor – creates a new clause and adds it to watcher lists:
    static bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
        "Implementation in Figure 5"

    – Learnt clauses only:
    bool locked(Solver S)
        return S.reason[var(lits[0])] == this

    – Constraint interface:
    void remove(Solver S)
        removeElem(this, S.watches[index(¬lits[0])])
        removeElem(this, S.watches[index(¬lits[1])])
        delete this

    bool simplify(Solver S)                    – only called at top-level with empty prop. queue
        int j = 0
        for (int i = 0; i < lits.size(); i++)
            if (S.value(lits[i]) == TRUE⊥)
                return TRUE
            else if (S.value(lits[i]) == ⊥)
                lits[j++] = lits[i]            – false literals are not copied (only occur for i ≥ 2)
        lits.shrink(lits.size() − j)
        return FALSE

    bool propagate(Solver S, lit p)
        – Make sure the false literal is lits[1]:
        if (lits[0] == ¬p)
            lits[0] = lits[1], lits[1] = ¬p

        – If 0th watch is true, then clause is already satisfied.
        if (S.value(lits[0]) == TRUE⊥)
            S.watches[index(p)].push(this)                  – re-insert clause into watcher list
            return TRUE

        – Look for a new literal to watch:
        for (int i = 2; i < size(); i++)
            if (S.value(lits[i]) != FALSE⊥)
                lits[1] = lits[i], lits[i] = ¬p
                S.watches[index(¬lits[1])].push(this)       – insert clause into watcher list
                return TRUE

        – Clause is unit under assignment:
        S.watches[index(p)].push(this)
        return S.enqueue(lits[0], this)                                – enqueue for propagation

    void calcReason(Solver S, lit p, vec⟨lit⟩ out_reason)
        – invariant: (p == ⊥) or (p == lits[0])
        for (int i = ((p == ⊥) ? 0 : 1); i < size(); i++)
            out_reason.push(¬lits[i])                   – invariant: S.value(lits[i]) == FALSE⊥
        if (learnt) S.claBumpActivity(this)
```

Fig. 4. Implementation of the *Clause* constraint.

```
bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
    out_clause = NULL

    - Normalize clause:
    if (!learnt)
        if ("any literal in ps is true")      return TRUE
        if ("both p and ¬p occurs in ps") return TRUE
        "remove all false literals from ps"
        "remove all duplicates from ps"

    if (ps.size() == 0)
        return FALSE
    else if (ps.size() == 1)
        return S.enqueue(ps[0])                                  - unit facts are enqueued
    else
        - Allocate clause:
        Clause c = new Clause
        ps.moveTo(c.lits)
        c.learnt   = learnt
        c.activity = 0                                           - only relevant for learnt clauses

        if (learnt)
            - Pick a second literal to watch:
            "Let max_i be the index of the literal with highest decision level"
            c.lits[1] = ps[max_i], c.lits[max_i] = ps[1]

            - Bumping:
            S.claBumpActivity(c)              - newly learnt clauses should be considered active
            for (int i = 0; i < ps.size(); i++)
                S.varBumpActivity(ps[i])          - variables in conflict clauses are bumped

        - Add clause to watcher lists:
        S.watches[index(¬c.lits[0])].push(c)
        S.watches[index(¬c.lits[1])].push(c)
        out_clause = c

        return TRUE
```

Fig. 5. Constructor function for clauses. Returns FALSE if top-level conflict is detected. 'out_clause' may be set to NULL if the new clause is already satisfied under the current top-level assignment. **Post-condition:** 'ps' is cleared. For learnt clauses, all literals will be false except 'lits[0]' (by design of *analyze()*). For the propagation to work, the second watch must be put on the literal which will first be unbound by backtracking.

while the solver is not at top-level. This makes the constructor code a bit more complicated than it would be for a normal constraint.

Implementing the *addClause()* method of the solver API is just a matter of calling *Clause_new()* and pushing the new constraint on the "constrs" vector, storing the list of problem constraints. The *newVar()* method of the API has a trivial implementation.

```
Constr Solver.propagate()
  while (propQ.size() > 0)
      lit p = propQ.dequeue()
      - 'p' is the enqueued fact to propagate
      Vec⟨Constr⟩ tmp
      watches[index(p)].moveTo(tmp)
      - 'tmp' contains the watcher list for 'p'

      for (int i = 0; i < tmp.size(); i++)
          if (!tmp[i].propagate(this, p))
              - Constraint is conflicting;
              - copy remaining watches to
              - 'watches[p]' and return
              - constraint:
              int j = i+1
              for (; j < tmp.size(); j++)
                watches[index(p)].push(tmp[j])
              return tmp[i]
  return Null
```

```
bool Solver.enqueue(lit p, Constr from)
  if (value(p) != ⊥)
      if (value(p) == False⊥)
          - Conflicting enqueued assignment:
          propQ.clear()
          return False
      else
          - Existing consistent assignment;
          -   don't enqueue:
          return True
  else
      - New fact, store it:
      assigns [var(p)] = lbool(!sign(p))
      level   [var(p)] = decisionLevel()
      reason  [var(p)] = from
      trail.push(p)
      propQ.insert(p)
      return True
```

Fig. 6. ***propagate():*** Propagates all enqueued facts. If a conflict arises, the *conflicting* clause is returned, otherwise Null. ***enqueue():*** Puts a new fact on the propagation queue, and immediately updates the variable's value in the assignment vector. If a conflict arises, False is returned and the propagation queue is cleared. 'from' contains a reference to the constraint from which 'p' was propagated (defaults to Null if omitted).

4.3 Propagation

Given the mechanism for adding constraints, we now move on to describe the propagation of unit information on these constraints.

The propagation routine keeps a set of literals (unit information) that is to be propagated. We call this the *propagation queue*. When a literal is inserted into the queue, the corresponding variable is immediately assigned. For each literal in the queue, the watcher list of that literal determines the constraints that may be affected by the assignment. Through the interface described in the previous section, each constraint is asked by a call to its *propagate()* method if more unit information can be inferred, which will then be enqueued. The process continues until either the queue is empty or a conflict is found.

An implementation of this procedure is displayed in *Figure 6*. It starts by dequeuing a literal and clearing the watcher list for that literal by moving it to "tmp". The propagate method is then called for each constraint of "tmp". This will re-insert watches into new lists. Should a conflict be detected during the traversal, the remaining watches will be copied back to the original watcher list.

The method for enqueuing unit information is relatively straightforward. Note that the same fact can be enqueued several times, as it may be propagated from different constraints, but it will only be put on the queue once.

4.4 Learning

We describe the basic conflict-analysis algorithm by an example. Assume the database contains the clause $\{x, y, z\}$ which just became unsatisfied during propagation. This is our conflict. We call $\overline{x} \wedge \overline{y} \wedge \overline{z}$ the reason set of the conflict. Now x is false because $\overline{x}$ was propagated from some constraint. We ask that constraint to give us the reason for propagating $\overline{x}$ (the *calcReason()* method). It will respond with another conjunction of literals, say $u \wedge v$. These were the variable assignment that implied $\overline{x}$. The constraint may in fact have been the clause $\{\overline{u}, \overline{v}, \overline{x}\}$. From this little analysis we know that $u \wedge v \wedge \overline{y} \wedge \overline{z}$ must also lead to a conflict. We may prohibit this conflict by adding the clause $\{\overline{u}, \overline{v}, y, z\}$ to the clause database. This would be an example of a *learnt* conflict clause.

In the example, we picked only one literal and analyzed it one step. The process of expanding literals with their reason sets can be continued, in the extreme case until all the literals of the conflict set are decision variables. Different learning schemes based on this process have been proposed. Experimentally the *First UIP* heuristic has been shown effective [ZM01]. We will not give the definition here, but just state the algorithm: In a breadth-first manner, continue to expand literals of the current decision level, until there is just one left.

In the code for *analyze()*, displayed in *Figure 7*, we make use of the fact that a breadth-first traversal can be achieved by inspecting the trail backwards. Particularly, the variables of the reason set of p is always before p in the trail. In the algorithm we initialize p to $\bot_{lit}$, which makes *calcReason()* return the reason for the conflict. Besides returning a conflict clause, *analyze()* sets the backtracking level, which is the lowest decision level the conflict clause is unit.

4.5 Search

The search method in *Figure 8* works basically as the pseudo-code presented in section 3 but with the following additions:

Restarts. The first argument of the search method is "nof_conflicts". The search for a model or a contradiction will only be conducted for this many conflicts. If failing to solve the SAT-problem within the bound, all assumptions will be canceled and $\bot$ returned. The surrounding solver strategy will then restart the search.

Reduce. The second argument, "nof_learnts", sets an upper limit on the number of learnt clauses that are kept. Once this number is reached, *reduceDB()* is called. Clauses that are currently the reason for a variable assignment are said to be *locked* and cannot be removed by *reduceDB()*. For this reason, the limit is extended by the number of assigned variables, which approximates the number of locked clauses.

Parameters. The third argument groups some tuning constants. In the current version of MiniSat, it only contains the decay factors for variables and clauses.

Root-level. To support incremental SAT, the concept of a *root-level* is introduced. The root-level acts a bit as a new top-level. Above the root-level are the incremental assumptions passed to *solve()* (if any). The search procedure is not allowed to backtrack above the root-level, as this would change the incremental assumptions. If we reach a conflict at root-level, the search will return False.

```
void Solver.analyze( Constr confl, Vec⟨lit⟩ out_learnt, Int out_btlevel)
    Vec⟨bool⟩ seen(nVars(), FALSE)
    int       counter  = 0
    lit       p        = ⊥_lit
    Vec⟨lit⟩  p_reason

    out_learnt.push()                                  – leave room for the asserting literal
    out_btlevel = 0
    do
        p_reason.clear()
        confl.calcReason(this, p, p_reason)            – invariant here: confl != NULL

        – TRACE REASON FOR P:
        for (int j = 0; j < p_reason.size(); j++)
            lit q = p_reason[j]
            if (!seen[var(q)])
                seen[var(q)] = TRUE
                if (level[var(q)] == decisionLevel())
                    counter++
                else if (level[var(q)] > 0)            – exclude variables from decision level 0
                    out_learnt.push(¬q)
                    out_btlevel = max(out_btlevel, level[var(q)])

        – SELECT NEXT LITERAL TO LOOK AT:
        do
            p     = trail.last()
            confl = reason[var(p)]
            undoOne()
        while (!seen[var(p)])
        counter−−
    while (counter > 0)
    out_learnt[0] = ¬p

void Solver.record( Vec⟨lit⟩ clause)
    Clause c
    Clause_new(this, clause, TRUE, c)
    enqueue(clause[0], c)  – cannot fail
    if (c != NULL) learnts.push(c)
```

Fig. 7. Analyze a conflict and produce a reason clause. **Pre-conditions:** (1) 'out_learnt' is assumed to be cleared. (2) Current decision level must be greater than root level. **Post-conditions:** (1) 'out_learnt[0]' is the asserting literal at level 'out_btlevel'. **Effect:** Will undo part of the trail, but not beyond last decision level. ***record():*** records a clause and drives backtracking; 'clause[0]' must contain the asserting literal.

A problem with the approach presented here is conflict clauses that are unit. For these, *analyze()* will always return a backtrack level of 0 (top-level). As unit clauses are treated specially, they are never added to the clause database. Instead they are enqueued as facts to be propagated (see the code of *Clause_new()*). There would be no problem if this was done at top-level. However, the search procedure will only undo until root-level, which means that the unit fact will be enqueued there. Once *search()* has solved the current SAT-problem, the surrounding solver strategy will undo any incremental assumption and put the solver back at the top-level. By this the unit clause will be forgotten, and the next incremental SAT problem will have to infer it again. There are many possible solutions to this problem. In practice we have not seen any performance difference in our applications [ES03, CS03].

Simplify. Provided the root-level is 0 (no assumptions were passed to *solve()*) the search will return to the top-level every time a unit clause is learnt. At that point it is legal to call *simplifyDB()* to simplify the problem constraints according to the top-level assignment. If a stronger simplifier than presented here is implemented, a contradiction may be found, in which case the search should be aborted. As our simplifier is not stronger than normal propagation, it can never reach a contradiction, so we ignore the return value of *simplify()*.

4.6 Activity Heuristics and Constraint Removal

In the *VarOrder* data type of MiniSat, the list of variables is kept sorted on activity at all time. The search will always accurately choose the most active variable. The original suggestion for the VSIDS dynamic variable ordering was to sort periodically. MiniSat implements variable decay by bumping with larger and larger numbers. Only when the limit of what is representable by a floating point number is reached need activities be scaled down.

Activity for conflict clauses are also maintained. The method for reducing the set of learnt clauses based on this activity, as well as the top-level simplification procedure can be found in *Figure 9*.

4.7 Top-Level Solver

The method implementing MiniSat's top-level strategy can be found in *Figure 8*. It is responsible for making the incremental assumptions and setting the root level. Furthermore, it completes the simple backtracking search with restarts, which are performed less and less frequently. After each restart, the number of allowed learnt clauses is increased.

5 Conclusions and Related Work

By this paper, we have provided a minimal reference implementation of a modern conflict-driven SAT-solver. We have tested MiniSat against zChaff and Berkmin 5.61 on 177 SAT-instances. These instances were used to tune Satzoo for the *SAT 2003 Competition*. As Satzoo solved more instances and series of problems, ranging over all three categories (*industrial, handmade,* and *random*), than any other solver in the competition, we feel that this is a representative test-set. No extra tuning was done in MiniSat; it was just run once with the constants presented in the code. At a time-out of 10 minutes, MiniSat solved 158 instances, while zChaff solved 147 instances and Berkmin 157 instances.

Another approach to incremental SAT and non-clausal constraints was presented by Aloul, Ramani, Markov, and Sakallah in their work on Satire and PBS [WKS01, AR$^+$02]. Our implementation differs in that it has a simpler notion of incrementality, and that it contains an interface for non-clausal constraints.

Finally, a set of reference implementations of modern SAT-techniques is present in the OpenSAT project. However, the project aim for completeness rather than minimal exposition, as we have chosen in this paper.

```
lbool Solver.search(int nof_conflicts,
      int nof_learnts, SearchParams params)
  int conflictC = 0
  var_decay = 1 / params.var_decay
  cla_decay = 1 / params.cla_decay
  model.clear()

  loop
    Constr confl = propagate()
    if (confl != Null)
        - Conflict
        conflictC++
        if (decisionLevel() == root_level)
            return False⊥
        Vec⟨lit⟩ learnt_clause
        int      backtr_level
        analyze(confl, learnt_clause, backtr_level)
        cancelUntil(max(backtr_level,root_level))
        record(learnt_clause)
        decayActivities()
    else
        - No conflict
        if (decisionLevel() == 0)
            - Simplify the set of problem clauses:
            simplifyDB()  - our simplifier cannot
                            return false here
        if (learnts.size()-nAssigns()≥nof_learnts)
            - Reduce the set of learnt clauses:
            reduceDB()
        if (nAssigns() == nVars())
            - Model found:
            model.growTo(nVars())
            for (int i = 0; i < nVars(); i++)
              model[i] = (value(i) == True⊥)
            cancelUntil(root_level)
            return True⊥
        else if (conflictC ≥ nof_conflicts)
            - Reached bound on num. of conflicts:
            cancelUntil(root_level) - force restart
            return ⊥
        else
            - New variable decision:
            lit p = lit(order.select())
            - may have heuristic for polarity here
            assume(p)       - cannot return false
```

```
bool Solver.solve(Vec⟨lit⟩ assumps)
  SearchParams params(0.95, 0.999)
  double nof_conflicts = 100
  double nof_learnts   = nConstraints()/3
  lbool  status        = ⊥

  - Push incremental assumptions:
  for (int i = 0; i < assumps.size(); i++)
      if (!assume(assumps[i])
      ||  propagate() != Null)
          cancelUntil(0)
          return False
  root_level = decisionLevel()

  - Solve:
  while (status == ⊥)
      status = search((int)nof_conflicts,
      (int)nof_learnts, params)
      nof_conflicts *= 1.5
      nof_learnts   *= 1.1

  cancelUntil(0)
  return status == True⊥
```

```
void Solver.undoOne()
  lit    p = trail.last()
  var    x = var(p)
  assigns [x] = ⊥
  reason  [x] = Null
  level   [x] = -1
  order.undo(x)
  trail.pop()

  while (undos[x].size() > 0)
      undos[x].last().undo(this, p)
      undos[x].pop()
```

```
bool Solver.assume(lit p)
  trail_lim.push(trail.size())
  return enqueue(p)
```

```
void Solver.cancel()
  int c = trail.size() - trail_lim.last()
  for (; c != 0; c--)
      undoOne()
  trail_lim.pop()
```

```
void Solver.cancelUntil(int level)
  while (decisionLevel() > level) cancel()
```

Fig. 8. ***search():*** assumes and propagates until a conflict is found, from which a conflict clause is learnt and backtracking performed until search can continue. **Pre-condition:** root_level == *decisionLevel*(). ***solve():*** **Pre-condition:** If assumptions are used, *simplifyDB*() must be called right before using this method. If not, a top-level conflict (resulting in a non-usable internal state) cannot be distinguished from a conflict under assumptions. ***assume():*** returns False if immediate conflict. **Pre-condition:** propagation queue is empty. ***undoOne():*** unbinds the last variable on the trail. ***cancel():*** reverts to the state before last *push*(). **Pre-condition:** propagation queue is empty. ***cancelUntil():*** cancels several levels of assumptions.

```
void Solver.reduceDB()
    int     i, j
    double lim = cla_inc / learnts.size()

    sortOnActivity(learnts)
    for (i=j=0; i < learnts.size()/2; i++)
        if (!learnts[i].locked(this))
            learnts[i].remove(this)
        else
            learnts[j++] = learnts[i]
    for (; i < learnts.size(); i++)
        if (!learnts[i].locked(this)
        && learnts[i].activity() < lim)
            learnts[i].remove(this)
        else
            learnts[j++] = learnts[i]
    learnts.shrink(i − j)
```

```
bool Solver.simplifyDB()
    if (propagate() != Null)
        return False

    for (int type = 0; type < 2; type++)
        Vec⟨Constr⟩ cs = type ?
                ( Vec⟨Constr⟩)learnts : constrs
        int j = 0
        for (int i = 0; i < cs.size(); i++)
            if (cs[i].simplify(this))
                cs[i].remove(this)
            else
                cs[j++] = cs[i]
        cs.shrink(cs.size()−j)
    return True
```

Fig. 9. ***reduceDB():*** Remove half of the learnt clauses minus some locked clauses. A locked clause is a clauses that is reason to a current assignment. Clauses below a threshold activity are also be removed. ***simplifyDB():*** Top-level simplify of constraint database. Will remove any satisfied constraint and simplify remaining constraints under current (partial) assignment. If a top-level conflict is found, False is returned. **Pre-condition:** Decision level must be zero. **Post-condition:** Propagation queue is empty.

References

[AR+02] F. Aloul, A. Ramani, I. Markov, K. Sakallah. **"Generic ILP vs. Specialized 0-1 ILP: an Update"** in *International Conference on Computer Aided Design (ICCAD)*, 2002.

[BC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu. **"Symbolic Model Checking using SAT procedures instead of BDDs"** in *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[CS03] K. Claessen, N. Sörensson. **"New Techniques that Improve MACE-style Finite Model Finding"** in *CADE-19, Workshop W4. Model Computation – Principles, Algorithms, Applications*, 2003.

[DLL62] M. Davis, M. Logman, D. Loveland. **"A machine program for theorem proving"** in *Communications of the ACM*, vol 5, 1962.

[ES03] N. Eén, N. Sörensson. **"Temporal Induction by Incremental SAT Solving"** in *Proc. of First International Workshop on Bounded Model Checking*, 2003.

[Lar92] T. Larrabee. **"Test Pattern Generation Using Boolean Satisfiability"** in *IEEE Transactions on Computer-Aided Design*, vol. 11-1, 1992.

[MS96] J.P. Marques-Silva, K.A. Sakallah. **"GRASP – A New Search Algorithm for Satisfiability"** in *ICCAD. IEEE Computer Society Press*, 1996

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. **"Chaff: Engineering an Efficient SAT Solver"** in *Proc. of the 38th Design Automation Conference*, 2001.

[ZM01] L. Zhang, C.F. Madigan, M.W. Moskewicz, S. Malik. **"Efficient Conflict Driven Learning in Boolean Satisfiability Solver"** in *Proc. of the International Conference on Computer Aided Design (ICCAD)*, 2001.

[WKS01] J. Whittemore, J. Kim, K. Sakallah. **"SATIRE: A New Incremental Satisfiability Engine"** in *Proc. 38th Conf. on Design Automation*, ACM Press 2001.