Journal on Satisfiability, Boolean Modeling and Computation 5 (2008) 217–242

Research Note

A Resolution Based SAT-solver Operating on Complete Assignments

Eugene Goldberg egold@cadence.com
Cadence Research Labs,
2150 Shattuck Ave., 10th floor, Berkeley,
California, 94704, USA

Abstract

Most successful systematic SAT-solvers are descendants of the DPLL procedure and so operate on partial assignments. Using partial assignments is explained by the "enumerative semantics" of the DPLL procedure. Current clause learning SAT-solvers, in a sense, have outgrown this semantics. Instead of enumerating the search space as the DPLL procedure does, they explicitly build a resolution proof. In this paper, we suggest a semantics that, in our opinion, is more suitable for clause learning SAT-solvers. The idea is to consider a set of complete assignments not just as a part of the search space but as an "encryption" of a resolution proof or a part thereof. Importantly, a set of points encrypting a resolution proof can be dramatically smaller than the entire search space. We introduce a resolution based SAT-solver with clause learning called *FI* (short for Find point Image of a proof) that is inspired by the new semantics. *FI* operates on *complete* assignments. We compare our naive implementation of *FI* with *Minisat* and *BerkMin*. Experiments show that *FI* is competitive with *Minisat* and *BerkMin* in terms of backtracks. In terms of performance, *FI* is slower than *Minisat* and *BerkMin* for small CNF formulas. On the other hand, even the current primitive implementation of *FI* is competitive with *Minisat* and *BerkMin* on large Bounded Model Checking formulas due to its superior decision making.

Keywords: *SAT-solver, resolution , decision-making, local search, complete assignment*

Submitted March 2007; revised April 2008; published June 2008

1. Introduction

The resolution proof system [1] has achieved outstanding popularity in practical applications. The best systematic SAT-solvers are descendants of the DPLL procedure [3] that can be simulated by so-called tree-like resolution (a special type of general resolution). The DPLL procedure operates on *partial* assignments. (An assignment is called partial if some of the formula variables are not assigned any value). The current partial assignment is extended until a clause is falsified. Then, the DPLL procedure backtracks to the last decision assignment and flips it.

The reason for using partial rather than *complete* assignments is that by rejecting a partial assignment the DPLL procedure may simultaneously reject an exponential number of complete assignments. (An assignment is called complete if every variable of the formula is assigned a value.) This greedy approach is a natural consequence of **enumerative**

semantics of DPLL. The DPLL procedure *partitions* the search space into non-overlapping subspaces corresponding to the leaves of the search tree. For this reason, one can view this procedure as *enumeration* of the complete assignments (or **points**) of the search space in a particular order. (From now on we will use the terms complete assignments and points interchangeably.) So, by enumerative semantics we mean the interpretation of DPLL as a method for enumerating the points of the search space.

In this paper, we are making a case for using *complete* assignments in resolution based SAT-solvers. This case is based on the following observations and results.

- Current resolution based SAT-solvers use clause learning. So instead of explicitly enumerating the search space as it is done in the DPLL procedure, they actually build a resolution proof. These SAT-solvers have outgrown the enumerative semantics of DPLL.

- Using complete assignments is justified by the proof encryption semantics we introduce in this paper. The proof encryption semantics can be viewed as a replacement of enumerative semantics for SAT-solvers with clause learning. In this semantics, a set of complete assignments is considered as an encryption of a resolution proof or a part thereof. Proof encryption semantics gives a more precise metric for measuring the quality of a set of points examined by a SAT-solver. Proof encryption semantics implies that it is more important for a SAT-solver to try to visit a particular set of points rather than attempt to enumerate all the points of the search space as fast as possible.

- We introduce a resolution based SAT-solver *FI* (Find point Image of a proof) with clause learning that operates on complete assignments. The set of points visited by *FI* forms an encryption of the proof *FI* builds. So *FI* can be viewed as a SAT-solver that builds a proof encryption along with the proof. Experimental results show that for large industrial formulas, even a very primitive implementation of *FI* is competitive with state-of-the-art SAT-solvers.

- *FI* can be viewed as a regular SAT-solver with clause learning whose choice of branching variables is *limited* to the variables of clauses falsified by the current complete assignment. Importantly, experiments show that good performance of *FI* can not be explained by great robustness of decision making based on computing conflict activity of literals/variables that was first introduced in [14]. So *FI* performance looks mysterious in the enumerative semantics. On the other hand, it has a natural explanation in proof encryption semantics.

- Complete assignments have more information about the formula than partial ones. For example, when a CNF formula F has a small unsatisfiable core F', any complete assignment $\boldsymbol{p}$ falsifies at least one clause of F'. If one restricts the choice of branching variables to those of clauses falsified by $\boldsymbol{p}$, then there is a high probability to pick a variable of F' as a branching variable. On the other hand, a SAT-solver operating on partial assignments may get stuck in the clauses of $F \setminus F'$ never reaching clauses of F'. The reason is that it has to choose the next branching variable out of *all* free variables of F, which significantly reduces the probability of picking a variable of F'.

- Greater informational capacity of complete assignments can be also leveraged when solving satisfiable formulas. When backtracking, the complete assignment maintained by *FI* remembers last assignments made in the subspace *FI* leaves. (A SAT-solver operating on partial assignments stores this information only implicitly through conflict clauses.) Besides, *FI* reports satisfiability as soon as the set of clauses falsified by the current complete assignment is empty. This may happen long before all clauses are satisfied by fixed assignments. On the contrary, a SAT-solver operating on partial assignments reports satisfiability only when all clauses of the formula are satisfied by fixed assignments.

Although, resolution based SAT-solvers like Grasp [19],SATO [20], Chaff [14],BerkMin [8], Siege and Minisat [4] are considered as descendants of DPLL, they are different from DPLL in one very important aspect. Namely, they use clause learning (first introduced in Grasp). For an unsatisfiable formula F, the DPLL procedure terminates if the right branch of the root node of the search tree is proved not to contain a satisfying assignment. So the DPLL procedure has to maintain a binary search tree. On the other hand, a clause learning SAT-solver terminates when an *empty clause* is derived. For that reason, clause learning SAT-solvers enjoy great flexibility in organizing search e.g. they can perform frequent restarts. (In addition to greater flexibility, adding learned clauses makes a SAT-solver more powerful [2].) One can say that clause learning procedures only *implicitly* enumerate the search space, while building a resolution proof is what these SAT-solvers do *explicitly*. (Even if a clause learning SAT-solver does not generate a resolution proof, such a proof can be extracted from the set of conflict clauses [9].) In a sense, current resolution based SAT-solvers have outgrown the enumerative semantics of DPLL. New decision making heuristics introduced by Chaff and then improved by BerkMin, Siege, Minisat and others can be viewed as the recognition of the fact that clause learning SAT-solvers build a resolution proof. Computation of literal/variable activity based on learned conflict clauses is nothing else but decision making driven by a resolution proof.

The idea of proof encryption semantics is based on the following observation Let F be a CNF formula and T be a set of complete assignments to the variables of F. Let $SAT(T,F)$ be a procedure that resolves *only* the clauses of F and their resolvents that are *specified* by the points of T (this procedure is described in Section 3 in more detail) Suppose $SAT(T,F)$ results in derivation of an empty clause, which means that $SAT(T,F)$ generated a resolution proof R that F is unsatisfiable. We will say that T is a **point image** of R. One can view a point image of R as its **encryption** in terms of complete assignments. Importantly, as we show in this paper, every resolution proof R that a CNF formula F is unsatisfiable has a point image T whose size is at most twice the size of R (in terms of resolution operations). One can view the interpretation of a set of points as an encryption of a resolution proof as an alternative to the enumerative semantics. (We will refer to this alternative semantics as **proof encryption semantics**). The fact that the size of a point image of a resolution proof can be dramatically smaller than the size of the entire search space, questions the applicability of enumerative semantics to generation of resolution proofs. The notion of a point image implies that it is much more important to visit a particular set of points (which comprises a very small subset of the entire search space) rather than enumerate all the points as fast as possible.

A straightforward way to use the notion of a point image of a proof is as follows. First, a set T of points that is a point image of a resolution proof (or at least has a high chance to be such) is generated. Second, the $SAT(T,F)$ procedure above is applied to generate a resolution proof. This approach has at least two problems. Firstly, it is not clear how to generate a good set of points T. Second, even if we find a way to generate T, the $SAT(T,F)$ procedure mentioned above is not practical because it may generate a very large number of redundant resolvents. (The reason for introducing $SAT(T,F)$ is to formalize the notion of proof encryption.)

In this paper, we describe a SAT-solver *FI* inspired by the proof encryption semantics. The main idea of *FI* is to use complete assignments to control decision making of a DPLL-like procedure. That is instead of constructing a set of points T and a proof R *separately*, *FI* builds them *together*. As we show later, the set of points visited by *FI* (after a natural extension) indeed *contains a point image* of the resolution proof *FI* builds.

Depending on semantics used, one can give two interpretations of *FI* that are *equivalent algorithmically*. In the first interpretation, *FI* is considered as a DPLL-like SAT-solver with clause learning that is allowed to make assignments *only* to variables of clauses that are falsified by a complete assignment $\boldsymbol{p}$. This complete assignment dynamically changes every time *FI* assigns a variable x_i a value that disagrees with the value of x_i in $\boldsymbol{p}$.

In the second interpretation, *FI* operates on complete assignments. In this interpretation, all variables are assigned and so there are no free variables. However, a value of variable x_i can be *fixed* so that all the points visited by *FI* after that have the same value of x_i until the latter gets unfixed.

The first interpretation is easier to understand from the operational point of view. In this interpretation, *FI* is just a "regular" clause learning SAT-solver in which the choice of variables to branch on is limited. The second interpretation better explains the semantics of *FI* which is visiting a set of points encrypting a resolution proof.

In the experiments described in this paper, we used a very simple implementation of *FI* that lacked many techniques employed by state-of-the art SAT-solvers (like fast BCP, efficient data structures, special treatment of binary clauses, removal of inactive conflict clauses and so on). We compared *FI* with SAT-solvers *BerkMin* and *Minisat*. The experiments showed that while *FI* was competitive with *BerkMin* and *Minisat* in terms of backtracks, it was slower on small and medium size formulas. However, when we run these SAT-solvers on large BMC formulas (up to a few millions of variables) the performance of *FI* was comparable with that of *BerkMin* and *Minisat*. In this case, the superior decision making of *FI* compensated for the inefficiency of implementation. Note that the good performance of *FI* can not be explained by enumerative semantics. In particular, we show that the SAT-solver different from *FI* only in that it picks a window of variables *randomly* (rather than uses the window specified by the set of clauses falsified by $\boldsymbol{p}$) works much worse.

This paper is structured as follows. Section 2 describes *FI*. Section 3 introduces proof encryption semantics. In Section 4, we argue that complete assignments contain more information about the formula than partial ones. Section 5 compares *FI* with local search and DPLL-based procedures. Experimental results are given in Section 6. We make some conclusions in Section 7.

2. Description of *FI*

In this section, we describe the SAT-solver *FI*. As mentioned in the introduction, there are two interpretations of *FI*. This section uses the first interpretation i.e. we consider *FI* as a regular SAT-solver with clause learning whose decision making is driven by a complete assignment $\boldsymbol{p}$. We will call assignments made by *FI* to decision and implied variables **fixed** to distinguish them from those assignments of $\boldsymbol{p}$ that still can be changed.

This section is structured as follows. In Subsection 2.1, we recall some basic definitions. Subsection 2.2 gives an example of how *FI* works. Subsection 2.3 gives the pseudo-code of the main procedure of *FI*. The decision making of *FI* is explained in Subsection 2.4. The other features of *FI* (BCP, conflict clause generation, restarts and initial point generation) are described in Subsections 2.5, 2.6. Some details of the implementation of *FI* we used in experiments will be described in Subsection 6.1.

2.1 Basic definitions

Let F be a CNF formula (i.e. conjunction of disjunctions of literals) over a set X of Boolean variables. The satisfiability problem (SAT) is to find a complete assignment $\boldsymbol{p}$ (called a **satisfying assignment**) to the variables of X such that $F(\boldsymbol{p}) = 1$ or to prove that such an assignment does not exist. If F has a satisfying assignment, F is called **satisfiable**. Otherwise, F is **unsatisfiable**. A disjunction of literals is further referred to as a **clause**. A clause with one literal is called **unit.** A complete assignment to variables of X will also be called a **point** of the Boolean space $B^{|X|}$ where B={0,1}. A complete assignment $\boldsymbol{p}$ **satisfies** clause C if $C(\boldsymbol{p})$=1. If $C(\boldsymbol{p})$=0, $\boldsymbol{p}$ is said to **falsify** C. We denote by $\boldsymbol{Vars(C)}$ and $\boldsymbol{Vars(F)}$ the set of variables of clause C and CNF formula F respectively.

2.2 Example

In this subsection, we show how *FI* works when testing the satisfiability of the CNF formula F below. Let F be the CNF formula consisting of the following eight clauses $C_1 = \overline{x_1} \vee x_2$, $C_2 = x_3 \vee x_4$, $C_3 = x_1 \vee x_4 \vee x_5$, $C_4 = \overline{x_2} \vee \overline{x_5}$, $C_5 = \overline{x_3} \vee x_5$, $C_6 = x_2 \vee \overline{x_4}$, $C_7 = \overline{x_5} \vee x_6$, $C_8 = x_1 \vee \overline{x_3} \vee \overline{x_5}$. First, *FI* generates a complete assignment $\boldsymbol{p}$. Assume that $\boldsymbol{p}$ is equal to (x_1=0, x_2=0, x_3=0, x_4=0, x_5=0, x_6=0). After generating $\boldsymbol{p}$, *FI* computes the set $M(\boldsymbol{p})$ of clauses falsified by $\boldsymbol{p}$. In our case $M(\boldsymbol{p}) = \{C_2, C_3\}$ because C_2 and C_3 are the only clauses of F falsified by $\boldsymbol{p}$.

Then *FI* starts making fixed assignments to variables of F. However, in contrast to a regular DPLL-like SAT-solver, not all variables are available for fixed assignments. Initially, only the variables of $Vars(M(\boldsymbol{p}))$=$\{x_1, x_3, x_4, x_5\}$ i.e. the variables of clauses C_2 and C_3 falsified by $\boldsymbol{p}$ can be used for decision making. Assume that *FI* picks variable x_1 and makes fixed assignment x_1=0. Since x_1 is equal to 0 in $\boldsymbol{p}$, the assignment x_1=0 *FI* is making agrees with $\boldsymbol{p}$ and so the latter remains the same. The set $M(\boldsymbol{p})$ does not change either. At this point, clause C_1 is satisfied (by this fixed assignment) and literal x_1 is removed from C_3 and C_8. Since no clause of F becomes unit, the Boolean Constraint Propagation (BCP) procedure does not make any new fixed assignments.

Suppose that the next fixed assignment *FI* makes is x_3=1. (Since x_1 is already assigned a fixed value, the only variables of $Vars(M(\boldsymbol{p}))$ that are available for branching are x_3, x_4, x_5.

Since this assignment disagrees with $\boldsymbol{p}$, *FI* flips the value of x_3 in $\boldsymbol{p}$, so the new current complete assignment $\boldsymbol{p}$ is (x_1=0, x_2=0, x_3=1, x_4=0, x_5=0, x_6=0). Since $\boldsymbol{p}$ has changed, $M(\boldsymbol{p})$ has to be recomputed. The new value of $M(\boldsymbol{p})$ is $\{C_3,C_5\}$. After making assignment x_3=1, clause C_2 is satisfied by this fixed assignment while literal $\overline{x_3}$ is removed from C_5 and C_8.

Since C_5 and C_8 became unit clauses, the BCP procedure makes new fixed assignments. Assume that unit clause C_8 is processed first. To satisfy C_8, *FI* makes fixed assignment x_5=0 that agrees with the current assignment $\boldsymbol{p}$ so the set $M(\boldsymbol{p})$ does not change. The fact that x_5=0 agrees with $\boldsymbol{p}$ means that the clause C_8 is satisfied by the current $\boldsymbol{p}$ and hence it is not in $M(\boldsymbol{p})$. So in contrast to a decision assignment that can be made only to a variable of $\mathit{Vars}(M(\boldsymbol{p}))$, implied assignments performed during BCP can be made to any variable of the formula whose value has not been fixed yet. After assigning x_5=0, clause C_5 becomes unsatisfiable. (Note that at this moment C_5 is falsified by the current assignment $\boldsymbol{p}$.) *FI* generates a conflict clause as described in Subsection 2.5. In this case, the conflict clause $C_9 = x_1 \vee \overline{x_3}$ is generated and added to F. Note that the conflict clause C is falsified by the current point $\boldsymbol{p}$. So it has to be added to $M(\boldsymbol{p})$.

After generating a conflict clause, *FI* backtracks to the decision assignment that is closest to the last one and is responsible for the conflict. In our case, *FI* backtracks to assignment x_1=0. The last decision assignment x_3=1 and all implied assignments made after x_3=1 are erased. When backtracking, the current complete assignment $\boldsymbol{p}$ *does not change* i.e. it remains exactly the same it was at the moment of the conflict. After backtracking, the conflict clause $C_9 = x_1 \vee \overline{x_3}$ becomes unit due to the decision assignment x_1=0 that is still fixed. So the BCP procedure is invoked. At this time $M(\boldsymbol{p})$ is equal to $\{C_3,C_5,C_9\}$.

The fixed assignment x_3=0 deduced from C_9, does not agree with the current complete assignment $\boldsymbol{p}$= (x_1=0, x_2=0, x_3=1, x_4=0, x_5=0, x_6=0). So *FI* flips the value of x_3 in $\boldsymbol{p}$ and recomputes $M(\boldsymbol{p})$. The new value of $M(\boldsymbol{p})$ is $\{C_2,C_3\}$. After assigning x_3=0, the clause C_2 becomes unit and the fixed assignment x_4=1 is deduced from it. Since x_4=1 disagrees with the current complete assignment $\boldsymbol{p}$**,** the value of x_4 is flipped in $\boldsymbol{p}$. So $\boldsymbol{p}$ is now equal to (x_1=0, x_2=0, x_3=0, x_4=1, x_5=0, x_6=0). The new value of $M(\boldsymbol{p})$ is $\{C_6\}$. At this point, the clause C_6 is unit. After making the assignment x_2=1, deduced from C_6, and recomputing $\boldsymbol{p}$ and $M(\boldsymbol{p})$ we obtain $\boldsymbol{p}$= (x_1=0, x_2=1, x_3=0, x_4=1, x_5=0, x_6=0) and $M(\boldsymbol{p}) = \emptyset$. This means that F is satisfiable and $\boldsymbol{p}$ is a satisfying assignment. Note that the current set of fixed assignments (x_1=0, x_3=0, x_4=1, x_2=1) does not satisfy clauses $C_4 = \overline{x_2} \vee \overline{x_5}$, $C_7 = \overline{x_5} \vee x_6$ (but they are satisfied by the assignment $x_5 = 0$ of $\boldsymbol{p}$ that is not fixed.) This is an important feature of *FI*. It may recognize the satisfiability of a CNF formula much earlier than a SAT-solver operating on partial assignments (because such a SAT-solver has to keep making fixed assignments until each clause of the formula is satisfied by a fixed assignment).

2.3 Main procedure

The pseudo-code of *FI* (without restarts) is shown in Figure 1. An initial complete assignment is generated by the procedure *generate_initial_assignment*. Then the set $M(\boldsymbol{p})$ of clauses falsified by $\boldsymbol{p}$ is computed . After that, *FI* follows the well-known procedure of [19] used by the state-of-the-art resolution based SAT-solvers. The only difference is that *FI* maintains two additional entities: a complete assignment $\boldsymbol{p}$ and the set $M(\boldsymbol{p})$ of clauses falsified by $\boldsymbol{p}$. When *FI* makes an assignment to a variable x_i (either decision or implied one) it checks if the chosen value of x_i is equal to the value of x_i in $\boldsymbol{p}$. If these values are equal, then no re-computation of $\boldsymbol{p}$ or $M(\boldsymbol{p})$ occurs. Otherwise, a new complete assignment $\boldsymbol{p'}$ is produced from $\boldsymbol{p}$ by flipping the value of x_i and the set $M(\boldsymbol{p'})$ is computed (by re-computation of $M(\boldsymbol{p})$). If an unsatisfiable clause is found, *FI* backtracks *without changing* the current point $\boldsymbol{p}$.

The current complete assignment $\boldsymbol{p}$ and the set $M(\boldsymbol{p})$ are used in *FI* solely for the purpose of decision making. Namely, the next variable to be assigned is picked *only* among variables of $Vars(M(\boldsymbol{p}))$ i.e among the variables of the clauses currently falsified by $\boldsymbol{p}$. Note that *FI* reports that F is satisfiable as soon as the set $M(\boldsymbol{p})$ becomes empty. As we saw in the example above, this may happen before the current set of fixed assignments (decision and implied) satisfies all the clauses.

FI is *sound* because it only adds clauses implied by the original formula F. *FI* is also *complete* because it never derives a conflict clause that is implied by another clause of the current formula (see below). In particular, *FI* never derives a conflict clause that is *identical* to another clause of the formula. In other words, *FI* adds to the formula only new clauses. So if F is unsatisfiable, *FI* will eventually derive an empty clause (because the total number of unique clauses of n variables is finite and equal to 3^n.) Otherwise, it will find a satisfying assignment. The reason is that every partial assignment leading to a conflict is also unique (repetition of such a partial assignment is prohibited by the corresponding conflict clause). On the other hand, the number of partial assignment is finite.

The reason why *FI* derives only new clauses is as follows. (The same argument applies to other SAT-solvers like Zchaff, BerkMin under the assumption that they never delete derived clauses.) *FI* uses *complete* BCP. That is at the time a new fixed decision assignment is made (after the previous BCP is over), each clause of the current formula is either satisfied by a fixed assignment or has at least two literals that are not assigned (by fixed assignments). On the other hand, a conflict clause C derived by *FI* contains exactly one literal assigned in the last BCP (i.e. BCP that has led to the conflict) and all literals of C are set to 0 by fixed assignments.

Clause C can not be implied by a clause C' of the current formula that was satisfied by a fixed assignment before the last BCP. The reason is that C' has to contain a literal that is not in C. Clause C can not be implied by a clause C'' that was not satisfied before the last BCP. If C'' became satisfied in the last BCP then C'' has at least one literal that is not in C. If C'' became falsified by the last BCP it has at least two literals set to 0 in this BCP while C has only one such literal. So again C'' has a literal that is not in C. Finally, if a literal of C'' is left unassigned this literal is not in C (because all literals of C are assigned).

2.4 Decision making of FI

The decision making of *FI* is based on computing conflict activity of literals first introduced in Chaff [14] and then developed in BerkMin [8], Siege, Minisat [4] and other SAT-solvers. The activity of literals is computed similar to BerkMin (but in contrast to *FI*, BerkMin computes the activity of variables rather than literals). Let $lit(x_i)$ be a literal of variable x_i. If $lit(x_i)$ occurs in k clauses of the current formula that were involved in the last conflict,

```
$FI(F)$
  {$\boldsymbol{p}$=$generate\_initial\_point(F)$;
  $M(\boldsymbol{p})$=$find\_falsified\_clauses(F,\boldsymbol{p})$;
  assgn = $nil$;
  while ($true$)
    {if ($BCP(F,\boldsymbol{p},M(\boldsymbol{p}),assgn)$ == $conflict$)
      {$C$=$generate\_conflict\_clause(F)$;
      if ($empty(C)$ return(UNSAT);
      else $backtrack(F,\boldsymbol{p},M(\boldsymbol{p}))$;}
    else // no conflict yet
    if ($M(\boldsymbol{p})$ =$\emptyset$) return (SAT);
    else $assgn$= $assignment\_to\_fix(F, M(\boldsymbol{p}))$; } }
```

Figure 1. Pseudo-code of FI

then the activity of $lit(x_i)$ is incremented by k. After a fixed number of conflicts, the activity of literals is divided by a small constant as it was first done in Chaff [14].

```
// $Vars(M(\boldsymbol{p}),n)$ denotes the set of variables of last $n$ clauses of $M(\boldsymbol{p})$.
$assignment\_to\_fix1(F,M(\boldsymbol{p}))$
  { $act\_lit$ = $most\_active(Vars(M(\boldsymbol{p}),n)$;
  return($satisfying\_assgn(act\_lit)$); }


$assignment\_to\_fix2(F, M(\boldsymbol{p}))$
  {$C$ = $find\_derived\_clause(F,M(\boldsymbol{p}))$;
  if ($C$ == $nil$) return($assignment\_to\_fix1(F,M(\boldsymbol{p}))$); // no conflict clauses in $M(\boldsymbol{p})$
  $act\_lit$ = $most\_active(Vars(C))$;
  return($satisfying\_assgn(act\_lit)$); }
```

Figure 2. Pseudo-code of the two decision-making procedures of FI

In the experiments described in this paper we used two decision making procedures. Their pseudo-code is shown in Figure 2. The first procedure (*assignment_to_fix1*) just finds the assignment satisfying the most active literal among the variables of clauses from $M(\boldsymbol{p})$ (obviously, only variables whose values are not fixed yet are considered). The number of clauses in $M(\boldsymbol{p})$ may be large which may slow down decision making. To solve this problem we consider only *the last* n clauses of $M(\boldsymbol{p})$ i.e. the last n entries of $M(\boldsymbol{p})$. In the experiments n was set to 32.

The second procedure (*assignment_to_fix2*) is *BerkMin*-like. First it looks for the conflict clause C of $M(\boldsymbol{p})$ that was derived the most recently. If all conflict clauses are satisfied by $\boldsymbol{p}$, it calls the first decision making procedure i.e *assignment_to_fix1*. Otherwise, it finds the assignment satisfying the most active literal among variables of C whose values have not been fixed yet.

2.5 BCP procedure and conflict clause generation

The BCP procedure of *FI* is almost identical to that of a generic DPLL based SAT-solver. The only difference is that after making a fixed assignment that disagrees with $\boldsymbol{p}$, the latter changes and the set $M(\boldsymbol{p})$ is recomputed. The pseudo-code of a generic BCP procedure (it does not use the speed-up scheme introduced in [20] and further improved in [14]) is shown in Figure 3.

```
BCP(F,p,M(p),assgn)
 { initialize(assgn_queue,assgn); // initialize assignment queue with assgn
 // assignment loop
 while (assgn_queue is not empty)
   { next_assgn = extract_assignment(assgn_queue); // extract next assignment
   // let next_var denote the variable to be assigned by next_assgn
   for every clause C having variable next_var
     {if (next_assgn satisfies C) mark_as_satisfied(C);
     else if (unit(C)) // if next_assgn  falsifies a literal of C and it became unit
     add_new_assgn (C,assgn_queue)} // add the assignment satisfying C to the queue
   if (next_assgn agrees with p) continue;
   else // next_assgn does not agree with p
     {flip_value(next_var, p); // flip the assignment of next_var in p
     recompute(next_var,M(p)); }}} // recompute the set of falsified clauses
```

Figure 3. Pseudo-code of the BCP procedure of *FI*

In contrast to the decision making of *FI*, its BCP procedure is *global*. As described above, in its decision making, *FI* fixes *only* assignments of variables that occur in clauses of $M(\boldsymbol{p})$. In the BCP procedure, *FI* keeps track of all the unit clauses regardless of whether they are falsified or satisfied by $\boldsymbol{p}$ and so regardless of their presence in $M(\boldsymbol{p})$. (In terms of fixed assignments, a clause is *unit*, if all its literals but one are set to 0 by fixed assignments.)

Let $\boldsymbol{p'}$ be the point obtained from $\boldsymbol{p}$ by flipping the value of *next_var* (when *next_assgn* does not agree with $\boldsymbol{p}$). The set $M(\boldsymbol{p'})$ is obtained by re-computation of the set $M(\boldsymbol{p})$ in two steps. First, all the clauses of $M(\boldsymbol{p})$ having variable *next_var* are removed from $M(\boldsymbol{p})$. Second, all the clauses of F that have the literal of *next_var* falsified by the new assignment to this variable are examined. Every clause of this set that is falsified by the new complete assignment is added to $M(\boldsymbol{p})$.

FI employs a traditional conflict analysis and first-UIP conflict clause generation whose description can be found in [19, 21].

2.6 Initial point generation and restarts

Currently *FI* uses the following procedure for generation of initial point $\boldsymbol{p}$. This procedure makes an assignment to a variable x_i of the formula and runs BCP procedure to derive all the implied assignments. If implied assignments to a variable contradict each other, one of them is picked randomly. (The contradiction of implied assignments means that the set of assignments made so far can not be extended to a satisfying assignment. In case of a conflict, the DPLL procedure backtracks, but *FI*'s goal here is to build a complete assignment. For this reason, *FI* continues making assignments even if a conflict occurs.) This goes on until all variables are assigned. This procedure may vary in how variable x_i and its assignment are chosen. Variable x_i can be chosen randomly or according to a particular order. An assignment to x_i can be also picked randomly or according to some heuristic.

We use BCP when generating an initial point $\boldsymbol{p}$ to minimize the size of $M(\boldsymbol{p})$. The reason is twofold. First, by reducing the size of $M(\boldsymbol{p})$ we make all operations involving $M(\boldsymbol{p})$ more efficient. Second, by minimizing the size of $M(\boldsymbol{p})$ we make the decision making of *FI* more precise. This part is explained in more detail in Subsection 4.1.

FI uses occasional restarts as suggested in [10]. (I.e. once in a while, *FI* abandons the current search tree to start a new one.) After a restart, *FI* inherits the last complete assignment $\boldsymbol{p}$ obtained before abandoning the previous search tree. In experiments, *FI* performed a restart every 150 conflicts.

3. Proof encryption semantics

As mentioned in the introduction, one can give two interpretations of *FI*. In the previous section we described *FI* in terms of the first interpretation. It views *FI* as a regular SAT-solver with clause learning that maintains a complete assignment to direct its decision-making. The advantage of this interpretation is that *FI*'s operation is very easy to understand. The problem is that this interpretation is based on enumerative semantics. From the viewpoint of this semantics it is very hard to understand why *FI* should work at all. Indeed, the set of variables that can be assigned a fixed value is specified by a complete assignment i.e. by a minuscule piece of the search space. In this section, we describe and justify an alternative interpretation of *FI*. In this interpretation, *FI* does not enumerate the search space, but rather visits a set of points that encrypts a resolution proof. The latter is built simultaneously with its encryption.

This section is structured as follows. In subsection 3.1, we recall some basic definitions of the resolution proof system. Subsection 3.2 introduces procedure $SAT(T,F)$ that checks if a set of points T encrypts a resolution proof that F is unsatisfiable. In Subsection 3.3, we show that after a natural extension, the set of points visited by *FI* contains a point image of the proof built by *FI*. This result substantiates interpretation of *FI* in terms of proof encryption semantics. Subsection 3.4 shows that the $SAT(T,F)$ procedure (and more generally, proof encryption semantics) can be used to measure the quality of a set of points visited by a SAT-solver.

3.1 Resolution proofs

Here we recall basic definitions of the resolution proof system [1]. Let C_1 and C_2 be two clauses that have opposite literals of a variable x_i. Then the clause consisting of all the literals of C_1,C_2 except those of x_i is called the **resolvent** of C_1,C_2. (For example if $C_1 = x_1 \vee x_3 \vee x_5$, $C_2 = x_2 \vee \overline{x_3} \vee x_7$, the resolvent of C_1 and C_2 is the clause $x_1 \vee x_5 \vee x_2 \vee x_7$.) The resolvent of C_1,C_2 is said to be obtained by the **resolution operation** in variable x_i.

The resolvent of C_1,C_2 is implied by $C_1 \wedge C_2$. So, if an empty clause is derived from clauses of F, then F implies an empty clause and so F is unsatisfiable. Hence, the resolution system is sound. It is also complete, that is, given an unsatisfiable CNF formula F, one can always generate a sequence of resolution operations resulting in producing an empty clause. This sequence of operations is called a **resolution proof**. The resolution proof system is very important from a practical point of view because the best SAT-solvers for solving "industrial" formulas are based on resolution.

3.2 Procedure $SAT(T,F)$

In this subsection, we introduce a procedure $SAT(T,F)$. This procedure is not *practical* (for the reason explained below) and we introduce it just to formalize the notion of a set of points T encrypting a proof. Namely, T encrypts a resolution proof that F is unsatisfiable, if $SAT(T,F)$ terminates with the answer '*unsatisfiable*'.

Proof encryption is discussed more thoroughly in [7] using the notion of a stable set of points [6]. In particular, in [7], we explain why one needs two points to encrypt a resolution operation. (If C is the resolvent of clauses C' and C'', then to prove $C' \wedge C'' \rightarrow C$ it is sufficient to build a stable set of two points.) Besides, in [7], we describe how one can encrypt a resolution proof in such a way that the latter can be efficiently recovered. The procedure $SAT(T,F)$ described below is an adaptation of procedure $SAT(T,F,L)$ of [7] for the purpose of this paper. The full discussion of this topic is beyond the scope of this paper.

The pseudo-code of $SAT(T,F)$ is given in Figure 4. First, $SAT(T,F)$ checks if a point $\boldsymbol{p}$ of T satisfies F. If such a point exists, then $SAT(T,F)$ returns '*satisfiable*'. Then in the while loop, $SAT(T,F)$ generates resolvents specified by T in topological order i.e. level by level. Clauses of the original formula F have level 0. If C is the resolvent of clauses C' and C'', then $level(C) = max(level(C'),level(C'') + 1$. The $SAT(T,F)$ procedure generates the resolvent C of clauses C', C'' of the current formula F, *only* if T contains points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ such that the following two conditions hold:

- $C'(\boldsymbol{p'}) = 0$ and $C''(\boldsymbol{p''})$=0.
- $C(\boldsymbol{p'}) = 0$ and $C(\boldsymbol{p''})$=0.

The points $\boldsymbol{p'}$ and $\boldsymbol{p''}$ are called a **point image of the resolution operation** over clauses C' and C''. In other words, $SAT(T,F)$ generates a resolvent C only if T contains a point image of the corresponding resolution operation. We also assume that the resolvent C is generated only if it is *not implied* by some existing clause of F. In other words, we assume that $SAT(T,F)$ generates only new clauses. Let F_1 denote the set of resolvents generated at the current topological level. If $F_1 = \emptyset$ (i.e. no new resolvents were generated), $SAT(T,F)$ returns '*unknown*'. This outcome means that T contains an insufficient number of points.

If F_1 contains an empty clause, $SAT(T,F)$ returns '*unsatisfiable*'. Otherwise, the clauses of F_1 are added to F and $SAT(T,F)$ starts building a new topological level of the proof.

```
// $T$ is a set of points, $F$ is a CNF formula
$SAT(T,F)$
  {if ($F(\boldsymbol{p})$=1 for a point $\boldsymbol{p}$ of $T$) return('satisfiable');
  while (true)
    {$F_1 = generate\_next\_level\_resolvents(T,F)$;
    if ($F_1 == \emptyset$) return('unknown');
    if ($F_1$ contains an empty clause) return('unsatisfiable');
    $F = F \cup F_1$; } }
```

Figure 4. Pseudo-code of $SAT(T,F)$

Let R be a resolution proof that a CNF formula F is unsatisfiable. Let T be a set of points that has the following property. For any resolvent C of R obtained from parent clauses C' and C'' there are two points $\boldsymbol{p'}$, $\boldsymbol{p''}$ of T forming a point image of the resolution operation over C' and C''. Then the set T is called a **point image of resolution proof *R***. From definitions of $SAT(T,F)$ and point image it follows that $SAT(T,F)$ returns '*unsatisfiable*' if and only if the set T contains a point image of a resolution proof that F is unsatisfiable. One can view a point image T of a proof R as an **encryption** of R, while $SAT(T,F)$ can be viewed as a procedure for recovering R from T.

It is not hard to show that a proof R has a point image whose size is at most twice the size of the proof R (in terms of resolution operations). Indeed, if C is a resolvent of R obtained from clauses C' and C'', one can always find points $\boldsymbol{p'}$, $\boldsymbol{p''}$ forming a point image of this resolution operation. The union of point images of all resolution operations of R is a point image of R. Obviously, the size of such a point image is at most twice the size of R.

Note that one can not guarantee that $SAT(T,F)$ will recover R from a point image T in polynomial time (which makes $SAT(T,F)$ impractical). Even though $SAT(T,F)$ imposes restrictions on generated resolvents, their number still can be prohibitively large. However, since we do not use $SAT(T,F)$ for real computation it does not matter much. Our goal here is twofold. First, we want to show how one can encrypt a resolution proof using complete assignments. Second, we use $SAT(T,F)$ to measure the quality of a set of points visited by a SAT-solver (see Subsection 3.4). So, in a sense, we only use the fact that for some sets of points, $SAT(T,F)$ results in generating an empty clause, while for others it stops before generating such a clause.

3.3 *FI* builds a point image of the proof it generates

Let F be an unsatisfiable formula and R be a resolution proof found by *FI* when solving F. Let T be the set of points visited by *FI*. In this subsection, we show that after a natural extension of the set T, it becomes a point image of R. In Subsection 3.3.1, we describe this extension and in subsection 3.3.2 we prove that the extended set T is a point image of R. Importantly, the fact that the set of visited points forms an image of the proof *FI* builds does not mean that *FI* does anything “special”. This fact just implies that the proof encryption semantics *agrees* with a resolution based SAT-solver using clause learning.

3.3.1 Extension of the set of visited points

Denote by T the set of points visited by *FI* when generating a proof R. We need to extend T because during BCP *FI* may fix assignments of variables of clauses that are not in $M(\boldsymbol{p})$. This extension of T is *natural* in the sense that the only reason why *FI* does not visit “explicitly” a point added by the extension is that this point can not be a satisfying assignment. Let C be a unit clause (i.e all the literals of C but one are set to 0 by fixed assignments). Suppose, the assignment satisfying C *agrees* with the current complete assignment $\boldsymbol{p}$. (So C is not $M(\boldsymbol{p})$.) After this assignment is fixed by *FI*, the clause C may get involved in a conflict without being falsified by any point of T. However, the necessary condition for T to be a point image of R is that every clause of R is falsified by a point of T. So, obviously, T is not an image of R.

Suppose we extend T by the set of points that falsify all clauses that became unit during a run of BCP and from which an (implied) assignment was derived. This extension can be built in the following way. If, during BCP a unit clause C is satisfied by fixing an assignment to a variable x_i and this assignment agrees with the current point $\boldsymbol{p}$**,** we add to T the point $\boldsymbol{p'}$ obtained from $\boldsymbol{p}$ by flipping the value of x_i. Obviously, $\boldsymbol{p'}$ falsifies C. Note that each added point is at Hamming distance one with an existing point of the current set T. The size of the extended set is no more than n times the size of the original set T.

3.3.2 Extended set of points is an image of the proof built by *FI*

Let *FI* encounter a conflict and C_{cnfl} be the conflict clause produced in this conflict. The conflict clause generation based on the first-UIP scheme (that is used by *FI*) is well described in [21]. According to this scheme, C_{cnfl} is generated by resolving clauses from which assignments were deduced at the conflict decision level and the clause that became unsatisfiable. Let C_{cnfl} be obtained by resolving clauses C_1,..,C_k of the current CNF formula F. Here C_1,..,C_{k-1} are clauses from which assignments were deduced during BCP at the conflict level and C_k is the unsatisfiable clause. (Note that during BCP at the conflict level, assignments may have been deduced from clauses other than C_1,..,C_{k-1}. But we are interested only in the clauses *involved* in the conflict.) Assume that C_1,..,C_{k-1} are numbered in the order in which they were processed by BCP (that is if $i < j$, then an assignment was deduced from C_i *before* deduction from C_j).

In the first-UIP scheme, C_{cnfl} is obtained from C_1,..,C_k by resolving them in the “reverse order”. That is, first, C_k is resolved with C_{k-1}. Then C_{k-2} is resolved with the resolvent of C_k and C_{k-1} and so on.

Proposition. The extended set of points T is a point image of the proof generated by *FI*.

Proof. We need to show that for each of k-1 resolution operations performed to generate C_{cnfl}, the extended set T contains two points forming a point image of this resolution operation. This can be proved by induction in the number of resolutions. Note that by definition of the conflict situation, the clause C_k (i.e. the clause that became unsatisfiable) had to be falsified by the current complete assignment (denote it by $\boldsymbol{p}_{cnfl}$).

Basis. The fact that T contains a complete assignment $\boldsymbol{p}_{cnfl}$ falsifying C_k and that the variables of $Vars(C_k)$ have fixed assignments is the basis of our inductive proof.

Denote by R_m the clause equal to the unsatisfiable clause C_k for m=1 or to the resolvent of clause $C_{k-(m-1)}$ and clause R_{m-1} (obtained by resolving clauses C_k,C_{k-1},..,$C_{k-(m-2)}$) for $m > 1$.

The inductive statement of the proof is as follows. We assume that R_m is falsified by $\boldsymbol{p}_{cnfl}$ and all the variables of $Vars(R_m)$ have assignments fixed *before* an assignment was deduced from clause $C_{k-(m-1)}$.

Using this assumption we will show that

1. The inductive statement holds for the next value of m.

2. T contains a point $\boldsymbol{p}_m$ falsifying the clause C_{k-m} such that $\boldsymbol{p}_{cnfl}$ and $\boldsymbol{p}_m$ form a point image of the resolution operation over R_m and C_{k-m} producing clause R_{m+1}.

Proof of the first condition. Denote by $Ded\_var(C_{k-m})$ the variable whose value was deduced from C_{k-m} during BCP. Note that before a value of $Ded\_var(C_{k-m})$ was deduced from C_{k-m}, the literals of the other variables of C_{k-m} were set to 0 by *fixed* assignments. So these literals are set to 0 by $\boldsymbol{p}_{cnfl}$ too. Then, taking into account that $\boldsymbol{p}_{cnfl}$ falsifies R_m we conclude that the clause R_{m+1} is falsified by $\boldsymbol{p}_{cnfl}$. Note that the assignments to the variables of $Vars(C_{k-m}) \setminus \{Ded\_var(C_{k-m})\}$ were fixed *before* deducing a value of $Ded\_var(C_{k-m})$. So all the literals of R_{m+1} were fixed at 0 before derivation of $Ded\_var(C_{k-m})$.

Proof of the second condition. Let $\boldsymbol{p}$ be the point that was the current complete assignment of FI at the time an assignment was deduced from clause C_{k-m}. Denote by $\boldsymbol{p}_m$ the point of T defined as follows. If C_{k-m} was falsified by $\boldsymbol{p}$, then $\boldsymbol{p}_m = \boldsymbol{p}$. If C_{k-m} was satisfied by $\boldsymbol{p}$, then $\boldsymbol{p}_m$=$\boldsymbol{p'}$ where $\boldsymbol{p'}$ is obtained from $\boldsymbol{p}$ by the extension of T described above (i.e. by flipping the value of variable $Ded\_var(C_{k-m})$). In either case, $C_{k-m}(\boldsymbol{p}_m)$=0.

Now we need to show that both $\boldsymbol{p}_m$ and $\boldsymbol{p}_{cnfl}$ falsify the resolvent R_{m+1} of C_{k-m} and R_m (and so $\boldsymbol{p}_m$ and $\boldsymbol{p}_{cnfl}$ form a point image of the resolution operation over C_{k-m} and R_m.) We already showed above that $\boldsymbol{p}_{cnfl}$ falsifies R_{m+1}. Now we show that $\boldsymbol{p}_m$ sets to 0 all the literals of R_m (except maybe the literal of $Ded\_var(C_{k-m})$) and hence, taking into account that $C_{k-m}(\boldsymbol{p}_m)$=0, the point $\boldsymbol{p}_m$ falsifies R_{m+1}. Let $lit(x_i)$ be a literal of R_m (where x_i is different from $Ded\_var(C_{k-m})$). Assume the contrary i.e. that the value of x_i in $\boldsymbol{p}_m$ sets $lit(x_i)$ to 1 and so $\boldsymbol{p}_m$ satisfies the resolvent R_m. This assignment to x_i can not be fixed because then $\boldsymbol{p}_{cnfl}$ would have the same value of x_i and so it would satisfy $\boldsymbol{R}_m$. So the opposite value of x_i must have been deduced later. This value of x_i could not have been deduced from C_{k-m} (because x_i is different from $Ded\_var(C_{k-m})$) or $C_{k-(m-1)}$ (or from any clause $C_{k-(m-p)}$, $p > 1$) because all assignments to the variables of R_m were fixed before derivation from $C_{k-(m-1)}$). This implies that there was one more derivation between C_{k-m} and $C_{k-(m-1)}$. But this contradicts our assumption that the sequence C_1,...,C_k lists all the clauses involved in the conflict in the order in which assignments were deduced from them by BCP.

3.4 Using proof encryption semantics to measure quality of sets of points

One of the main problems of the enumerative semantics is as follows. Let T^i and T^j be two sets of points which have been proved not to contain a satisfying assignment. Enumerative semantics essentially implies that if $|T^i| > |T^j|$ then T^i is more "valuable" than T^j. This makes perfect sense if the search space is partitioned into non-overlapping subsets T^i. However this is not true, for example, for SAT-solvers with clause learning where subspaces T^i corresponding to leaf nodes may overlap. In such a case, the most important quality of subspaces T^i is not their size but their synergy i.e. how well they fit each other to produce an empty clause as a proof of unsatisfiability.

The proof encryption semantics gives a way to measure the quality of a set of visited points. We illustrate this claim by two examples below.

Suppose that a CNF formula F is equal to $F' \wedge \{x_i\}$ where x_i is just a unit clause. From the viewpoint of enumerative semantics setting x_i to 1 (that would be done by a DPLL-like SAT-solver to satisfy the unit clause x_i) rules out half the search space. So one has to conclude that 50% of work has been done. On the other hand, obviously, the complexity of proving that $F(x_i{=}1)$ is unsatisfiable may be arbitrary high.

From the viewpoint of the proof encryption semantics, the situation is different. Let R be a "reasonable" resolution proof that F is unsatisfiable. By a reasonable proof we mean one that first derives clause $\overline{x_i}$ from F' and then resolve x_i and $\overline{x_i}$ to produce an empty clause. (This is exactly the kind of a proof a state-of-the-art SAT-solver with clause learning would build). If one constructs a point image of such a proof, *only one point* of this image has to falsify x_i (namely a point of the image of the resolution operation over clauses x_i and $\overline{x_i}$.) So in the proof encryption semantics, the work done by ruling out the subspace $x_i{=}0$ amounts to gaining only one point of the image of a future proof.

Let us consider another example. Suppose an unsatisfiable CNF formula F contains a clause $C = x_i \vee x_m$. Suppose that C is irredundant i.e. removing C from F makes the latter satisfiable. Then C has to be a part of any resolution proof R that F is unsatisfiable. Let us T' be the set of complete assignments falsifying C (i.e. T' consists of all complete assignments with $x_i = 0$ and $x_m = 0$). Although T' makes up a quarter of the search space, it does not contain a point image of a resolution proof (because T' can not contain point images of operations resolving clauses in x_i or x_m). On the other hand, there might be a dramatically smaller set of points T'' that contains a point image of a proof.

4. Complete assignments contain more information about the formula than partial assignments

In this section, we compare partial and complete assignments at a different angle. A partial assignment specifies more points of the search space, but it contains much less information about formula F (because a partial assignment "sees" only a fraction of variables of F). On the contrary, a complete assignment specifies only one point $\boldsymbol{p}$ of the search space but contains more information about F. In Subsection 4.1 we show that this advantage of a complete assignment can be used for identifying small unsatisfiable cores. Subsection 4.2 demonstrates how greater informational capacity of complete assignments can be used for solving satisfiable formulas.

4.1 Identification of unsatisfiable cores

Let F be a CNF formula that can be represented as $F_1 \wedge F_2$ where F_1 is a large CNF formula (either satisfiable or unsatisfiable) that is very hard to solve and F_2 is a small unsatisfiable CNF formula. Assume for the sake of simplicity, that the variables of F_1 and F_2 do not overlap. Since F_1 is much larger than F_2, first conflict clauses derived by a SAT-solver will depend on variables of F_1. Indeed, suppose, for example, that $|\mathit{Vars}(F_2)|$=10 and $|\mathit{Vars}(F_1)|$=10^6. Suppose that to get a conflict in F_2 one has to assign at least 5 variables of F_2. Then the probability of producing a conflict in F_2 is roughly speaking $(10^{-6})^5$ i.e. 10^{-30}. (Of course we assume here that the clauses derived from conflicts in F_1 are not too long. Otherwise we would need to take into account that a SAT-solver has to make many assignments in F_1 before a conflict occurs. This increases the chances of producing a conflict in F_2.) So with great probability conflicts will be generated in the formula F_1. Then the activity of variables of F_1 will be growing (we assume that our SAT-solver uses *Zchaff*-like decision making). This will force the SAT-solver to keep branching on variables of F_1 until an assignment satisfying F_1 is produced or F_1 is proved unsatisfiable.

Let us consider solving the formula above by *FI*. No matter what the current complete assignment $\boldsymbol{p}$ is, it falsifies a clause of F_2 (because the latter is unsatisfiable). Then, although the choice of branching variables reduces to $\mathit{Vars}(M(\boldsymbol{p}))$ (i.e. to the variables of clauses falsified by $\boldsymbol{p}$), a variable of F_2 is *always available* for branching and so good choices remain intact. Moreover, since the set $\mathit{Vars}(M(\boldsymbol{p}))$ is much smaller than $\mathit{Vars}(F)$, finding a variable of F_2 becomes dramatically easier.

One may think that the case when a CNF formula has a small unsatisfiable core is relatively rare. However, even if the original CNF formula F is irredundant (i.e. no clause of F can be removed without making F satisfiable), it becomes redundant after adding conflict clauses and/or making decision assignments. This may lead to appearance of small unsatisfiable cores. So the problem of finding such cores is ubiquitous in resolution based SAT solving.

Interestingly, the reasoning above implies that to make identification of an unsatisfiable core easier one should keep the size of the set $M(\boldsymbol{p})$ small. One can view variables $\mathit{Vars}(F_2)$ of the small unsatisfiable core F_2 as "signal" and variables $\mathit{Vars}(F_1)$ of F_1 as "noise". No matter how *FI* picks $\boldsymbol{p}$, a signal variable will be in $\mathit{Vars}(M(\boldsymbol{p}))$. On the other hand, by minimizing the size of $M(\boldsymbol{p})$, *FI* may considerably reduce the number of noise variables and so improve the signal/noise ratio in $\mathit{Vars}(M(\boldsymbol{p}))$. In particular, when generating an initial point $\boldsymbol{p}$ (Subsection 2.6) *FI* tries to minimize the size of $M(\boldsymbol{p})$.

4.2 Storing information about previous assignments when solving satisfiable formulas

Greater informational "capacity" of complete assignments can be also leveraged when solving satisfiable formulas. A DPLL-like SAT-solver operating on complete assignments loses information about assignments made before backtracking (although some part of this information is represented implicitly in the learned clauses). Besides, to find a satisfying assignment such a SAT-solver has to keep extending the current *partial* assignment until all clauses are satisfied. On the contrary, when backtracking, *FI* preserves more information due to maintaining a complete assignment (that remembers the last assignments made before backtracking). Besides, *FI* does not have to satisfy all the clauses by fixed assignments. It stops as soon as the *current complete assignment* $\boldsymbol{p}$ satisfies all the clauses (at this time, only a small fraction of assignments of $\boldsymbol{p}$ may be fixed).

5. Background

In this section, we compare *FI* with other SAT-solvers and give some background information. As mentioned above, *FI* has two different interpretations. In the first interpretation, *FI* should be compared with other DPLL-like SAT-solvers like *Zchaff* [14], BerkMin [8], *Siege*, *Minisat* [4]. (In this interpretation, *FI* is a DPLL-like SAT-solver with clause learning whose choice of branching variables is limited to $\mathit{Vars}(M(\boldsymbol{p}))$). The main difference of *FI* from these SAT-solvers is in decision making. The decision making driven by conflict activity of literals (introduced by *Zchaff*) is very slow for large formulas if one has to examine all free (i.e. unassigned) variables. A solution of *BerkMin* was to branch on variables of the most recently derived conflict clause that was still unsatisfied. This considerably reduced the number of variables to examine. (A similar solution was independently introduced in *Siege*.) The flaw of this approach is that if the current CNF formula does not have any unsatisfied conflict clauses, BerkMin still has to pick an assignment out of the entire set of free variables. (As we show in Section 6 this drawback slows down the performance of BerkMin for large BMC formulas.)

Minisat took a slightly different approach. The activity of variables is computed in *Minisat* similarly to *BerkMin*. Like *BerkMin* it is more aggressive than *Zchaff* in giving preference to recently derived clauses. However, in contrast to *BerkMin* and *Siege*, *Minisat* still looks for the most active variable among all the free ones. It does this efficiently by maintaining a *heap* of variable activities. *FI* suggests a new way to speed up decision making. It reduces the choice of available branching variables to those that are falsified by the current complete assignment.

In the second interpretation, *FI* is a resolution based algorithm with clause learning that operates on complete assignments. From this point of view *FI* should be compared with algorithms operating on complete assignments pioneered in [16, 17]. One can classify these algorithms into three groups: incomplete stochastic local search algorithms, hybrid algorithms combining DPLL and local search and local search algorithms that can solve unsatisfiable formulas. The first group comprises of algorithms like *gsat* [16],*walksat* [17], *unitwalk* [12]. Although *FI* bears some similarity to *walksat* (they both maintain the set $M(\boldsymbol{p})$ of clauses falsified by the current complete assignment), the main difference of *FI* from the algorithms of this group is that it is complete. So *FI* can escape local minima that may trap a local search algorithm. On the other hand, being a DPLL-like procedure, *FI* does not work well for classes of formulas like satisfiable random formulas where local search algorithms show strong performance.

The second group consists of algorithms like [13, 11]. In [13], in every node of the DPLL procedure, a local search procedure is invoked to identify the next variable to branch on. In [13], an important observation was made that local search can be used for identifying an unsatisfiable core of the initial formula. The approach of [13] was also tried in [11] with the following modification. Before running a local search procedure at a node of the search tree, dependencies between variables of the current formula were computed. The difference of *FI* from the algorithms of this group is twofold. First, *FI* uses clause learning and decision making based on computing conflict activity of literals and so it can be applied to solving very large CNF formulas. Second, algorithms of this group consider local search procedures and DPLL procedure as separate entities. On the contrary, in *FI* these two kinds of procedures are merged together into a single DPLL-like procedure operating on complete assignments.

There are very few local search procedures that can be applied to solving unsatisfiable formulas. A complete local search algorithm augmented by clause generation was introduced in [5]. Clauses were generated in the algorithm of [5] in "a mechanical way" just to escape local minima. Our experiments showed that the SAT-solver of [5] generates an enormous number of new clauses and so fails to prove the unsatisfiability of even very small CNF formulas. In [18], the idea of proving unsatisfiability by local search was introduced. The suggestion was to encrypt proofs in some way and then look for proof encryptions by a stochastic local search procedure. This idea was implemented in [15] but the suggested algorithm performed well only on a fraction of CNF formulas. Since *FI* is based on proof encryption semantics one can view it as an implementation of the elegant idea of [18]. However, in contrast to [18], *FI* is a *complete* algorithm that builds a proof encryption *along* with the proof itself. Such a strategy makes sense because, intuitively, an encryption that is good for one formula may not work well for another.

6. Experimental Results

In this section, we give results of some experiments with an implementation of *FI*. Experiments were run on Intel's Xeon CPU (3.06GHz) under Linux. The main objective of the experiments was to show that although *FI* was very limited in the choice of branching variables, it was still competitive with state-of-the-art SAT-solvers in the number of backtracks (conflicts). So we used a very simple implementation that lacked the techniques commonly employed to speed up a SAT-solver. Besides, we tried to keep our implementation of *FI* as simple as possible (to facilitate changing the code of *FI*). Nevertheless we give some results of *FI* performance showing that even our primitive implementation is competitive with highly optimized SAT-solvers for large Bounded Model Checking (BMC) formulas. In our experiments, we used the second decision-making heuristic of Subsection 2.4 for the formulas of Tables 1, 7 (it was slightly modified for the formulas of Table 7 as described below). For the rest of the formulas we used the first decision-making heuristic of subsection 2.4 with n=32 (i.e. *FI* took into account only last 32 entries of the set $M(\boldsymbol{p})$). In all the experiments, a restart was performed every 150 conflicts.

This section is structured as follows. In Subsection 6.1 we briefly describe the implementation we used in experiments. Subsection 6.2 compares *FI* with *Zchaff*, *Minisat* and *BerkMin*. In Subsection 6.3, we show that *FI* can not be efficiently simulated by a SAT-solver that reduces choices by picking the next branching variable from a random window of variables. Finally, Subsection 6.4 gives some results on testing the satisfiability of formulas with small unsatisfiable cores.

6.1 A brief description of the implementation we used in experiments

FI was implemented in C++ using the gcc compiler (version 3.2.2). We used the STL library for data structures like dynamic arrays. As mentioned above, our implementation of *FI* was very simple. It did not have advanced features like fast BCP, efficient formula representation, special treatment of binary clauses, removal of inactive conflict clauses and so on. In our implementation of the BCP procedure, to check if a clause was unit *FI* just counted the number of unassigned literals (as it was done, for example, in Grasp [19]). Instead of "watching" literals, for every literal $lit(x_i)$, *FI* maintained the list of clauses of the current formulas having $lit(x_i)$. So when $lit(x_i)$ was set to 0, *FI* just needed to examine the clauses of the corresponding list to see if new unit clauses appeared. To avoid examining satisfied clauses, when $lit(x_i)$ was set to 1, all the clauses with $lit(x_i)$ unsatisfied so far were marked as satisfied. The clauses satisfied at a particular decision level were recorded so that they could be easily unmarked when backtracking.

For more efficient processing of the set $M(\boldsymbol{p})$ of clauses falsified by the current complete assignment $\boldsymbol{p}$, our implementation of *FI* maintained the following three data structures. An array *weight* contained the information about variable occurrences in clauses of $M(\boldsymbol{p})$. If $weight[i] = 0$, then no clause of $M(\boldsymbol{p})$ contains x_i. An array *Vars* contained the set of all variables that were present in $M(\boldsymbol{p})$ (i.e. all variables x_i with $weight[i] > 0$). Finally, *FI* maintained an array *falsified_clauses*, where *falsified_clauses*$[i]$ gave the clauses of $M(\boldsymbol{p})$ having variable x_i. All three data structures were updated every time, the set $M(\boldsymbol{p})$ changed. The array *weights* allowed one to efficiently check if x_i was in a clause of $M(\boldsymbol{p})$. The array *Vars* was used to efficiently enumerate all variables of clauses falsifying $\boldsymbol{p}$. The array *falsified_clauses* was used for updating $M(\boldsymbol{p})$. When the value of variable x_i was flipped, the clauses to be removed from $M(\boldsymbol{p})$ were extracted from *falsified_clauses*$[i]$.

6.2 Comparison with other SAT-solvers

In this subsection, we compare our implementation of *FI* with *BerkMin* [8], version 561 and *Minisat* [4], version 1.13. To put *FI* performance in perspective, we also give the results of *Zchaff* (version Z2001.2.17 of year 2001). Table 1 compares SAT-solvers for some known benchmark formulas of small and medium size (up to 60 thousand variables). The second column gives the number of formulas in a benchmark set. For every SAT-solver the runtime (in seconds) and number of conflicts is reported. Results of Table 1 show that *FI* was competitive with *BerkMin* and *Minisat* in terms of conflicts (with the exception of the hole formulas where *FI* had considerably fewer backtracks). *FI* was better than *Zchaff* in terms of backtracks (except for the hole formulas where *Zchaff* had slightly fewer backtracks). As for performance, *BerkMin* and *Minisat* were much faster than *FI* due to their advanced optimization techniques. *Zchaff* was also faster than *FI* and only when it had a significantly larger number of backtracks (*bmc* and *ii* benchmarks) it was slower than *FI*.

In Table 2, we compare these four SAT-solvers on a set of large BMC formulas (up to a few million variables). These formulas describe various properties of more than a dozen of customer designs. This set consists of 79 formulas (28 satisfiable and 51 unsatisfiable). The formulas were generated by the SMV model-checker [22]. For all the four SAT-solvers, Table 2 gives the number of conflicts (in thousands) and runtime (in hours). (*Zchaff* and

Table 1. Comparison of *FI* with other SAT-solvers on some known benchmarks

Name	#inst	*Zchaff-2001*		*BerkMin561*		*Minisat*		*FI*	
		#cnfls	time (s)	#cnfls	time (s)	#cnfls	time	#cnfls	time
blocksworld	7	15,325	9.5	**4,239**	**1.6**	4,732	1.7	7,211	25.3
miters	13	124,987	7.9	31,998	**1.8**	48,118	2.7	**28,386**	9.8
bmc	13	309,652	523.1	53,989	54.0	44,195	**29.8**	**41,846**	210.4
planning	6	74,639	49.2	31,238	13.4	**17,153**	**6.0**	22,786	159.1
ii	41	75,859	53.4	6,505	0.5	4,088	**0.4**	**905**	1.8
hole	5	**37,452**	**11.3**	224,803	32.3	1,538,350	33.6	48,383	159.0

BerkMin were not able to finish 5 and 3 formulas respectively with the timeout limit of 10 hours.)

In terms of performance, *FI* was considerably faster than *Zchaff* and *BerkMin*. *FI* was faster than *Minisat* on satisfiable formulas and about two times slower for unsatisfiable formulas. These surprising results can be explained as follows. Using fast BCP is crucial only when a CNF formula has a lot of long clauses. These long clauses may be either present in the initial formula or may appear as conflict clauses. When solving a large BMC formula that initially consists of clauses with no more than three literals, fast BCP becomes beneficial only if the SAT-solver has to generate a lot of conflict clauses. However, if the number of conflicts is relatively small (like 10-20 thousands of conflict clauses for a formula of a few million clauses) the speed-up provided by fast BCP is not so significant. In this situation, efficient and "precise" decision making becomes very important. However, the decision making of all three SAT-solvers that we compared with *FI* (we briefly described their decision-making in the previous section) has drawbacks.

Zchaff takes a large amount of time to find next branching variable. For smaller formulas this drawback did not show but for large formulas of Table 2 it became a big problem. The decision making of *BerkMin* is much faster than Zchaff's when the current formula has unsatisfied conflict clauses. However, for large BMC formulas that can be solved without generating too many clauses, *BerkMin* often had to pick next branching variable out of all free variables. (Version 561 of *BerkMin* does use a method to reduce time taken by the decision-making procedure, but it was designed for small or medium size formulas.)

Minisat spends much less time on decision making. The problem, however, is in the quality of decision making of *Minisat*. In general, *Minisat* had to make many more decisions per conflict than *FI*, which increased the amount of work done during BCP. (The same applies to *Zchaff* and *BerkMin*.)

Tables 3 and 4 give more detailed comparison of *Minisat* and *FI* on BMC formulas used in Table 2. Their performance on a sample of these formulas is given in Table 3. The names of satisfiable formulas are marked in the first column of Table 3 with an asterisk. For either SAT-solver, Table 3 gives the number of decision assignments (in thousands), implied assignments (in millions), conflicts (in thousands) and runtime (in seconds). Results for the first three formulas (*always.100*, *mcbdm.40*, *cfet.22*) show that *FI* was faster because it made fewer implied and decision assignments. So even though *Minisat* is much more optimized than *FI*, the amount of extra work *Minisat* had to do slowed it down. For example, for the formula *cfet.22*, *FI* made more backtracks (31,000 versus 14,000) but considerably fewer decisions (368,000 versus 1,763,000) and implied assignments ($23{\times}10^6$ versus $98{\times}10^6$).

For satisfiable formulas, in addition to making more implied and decision assignments, in many cases *Minisat* had many more backtracks. The reason for the good performance of *FI* on satisfiable formulas was already mentioned in the introduction and in Section 2. *FI* reports satisfiability as soon as the set $M(\boldsymbol{p})$ becomes empty. This may happen even if only a fraction of variables have fixed values while the rest of the variables are assigned a value only in $\boldsymbol{p}$. In this case, only a fraction of clauses of the original formula are satisfied by *fixed* assignments. On the contrary, a traditional SAT-solver operating on partial assignments has to go on making assignments until all the clauses are satisfied by fixed assignments. Needless to say, that such a SAT-solver may still encounter a lot of conflicts before a satisfying assignment is found. For example, for a formula of Table 2 of 700 thousand variables, *FI* reported satisfiability after 1700 backtracks when only 22,000 variables were assigned fixed values. It took *Minisat* 420,000 backtracks and more than one hour of runtime to assign the *rest* of the variables of this formula

In Table 4, we summarize the performance of *Minisat* and *FI* on the formulas of Table 2 in terms of implied and decision assignments (given in millions). This table shows that *FI* made considerably fewer assignments of either kind for both satisfiable and unsatisfiable formulas. Note that this result can be explained by better identification by *FI* of unsatisfiable cores. After *FI* has fixed some assignments a small unsatisfiable core may appear in the current formula. The same occurs after Minisat has made some assignments to free variables. However, *FI* is more likely to pick a core variable than *Minisat* (see Subsection 4.1). Making assignments to variables outside the core may lead to redundant BCP calls.

Table 2. Comparison of *FI* with other SAT-solvers on large BMC formulas (timeout 10 hours)

Category	#inst	*Zchaff-2001*		*BerkMin561*		*Minisat*		*FI*	
		#cnfls $\times 10^3$	time (hrs) [aborted]	#cnfls $\times 10^3$	time (hrs) [aborted]	#cnfls $\times 10^3$	time (hrs)	#cnfls $\times 10^3$	time (hrs)
Sat.	28	> 488	> 65 [4]	$> 2,204$	> 40 [2]	3,406	16.4	**756**	**4.5**
Unsat.	51	$> 1,603$	> 87 [1]	$> 1,772$	> 27 [1]	**1,269**	**4.0**	1,448	9.5
Total	79	$> 2,091$	> 152 [5]	$> 3,976$	> 67 [3]	4,675	20.4	**2,204**	**14.0**

6.3 Can *FI* be simulated by a random windowing of variables?

In the previous section, we saw that in the number of backtracks, *FI* is competitive with state-of-the-art SAT-solvers. One might think that this result can be attributed to the great robustness of decision making based on computing conflict activity of variables (or literals). It might be the case that even if a SAT-solver branches on a variable with a reasonably high activity score (instead of the most active variable), it still may get the same or similar results. Then even though *FI* has a dramatically smaller pool of variables to select from, the set $Vars(M(\boldsymbol{p}))$ may still contain variables with good activity scores.

Table 3. Comparison of *FI* and *Minisat* on a sample of large BMC formulas

Name	#vars $\times 10^3$	#clauses $\times 10^3$	*Minisat*				*FI*			
			#decis. $\times 10^3$	#imp. $\times 10^6$	#cnfl. $\times 10^3$	time (s)	#dec. $\times 10^3$	#impl. $\times 10^6$	#cnfl. $\times 10^3$	time (s)
always.100	249	758	379	168	21	211	83	109	16	**157**
mcbdm.40	273	773	231	50	6	84	59	19	9	**28**
cfet.22	613	1,808	1,763	98	14	385	368	23	31	**58**
ccc.16	317	1,081	163	187	22	**150**	44	375	21	1,559
*sched.30	974	2,653	3,805	834	23	937	245	643	26	**737**
*prop9.100	1,001	2,981	521	599	40	**411**	305	549	48	1,190
*iuf.200	2,083	6,443	134,305	3,891	593	9,873	57	24	6	**59**
*fft.2000	260	736	54,883	2,188	148	2,433	26	947	1	**8**

Table 4. Comparison of *Minisat* and *FI* in terms of decision and implied assignments

Category	*Minisat*		*FI*	
	#decis. $\times 10^6$	#impl. $\times 10^6$	#decis. $\times 10^6$	#impl. $\times 10^6$
Sat.	91	10,616	10	5,979
Unsat.	587	40,003	15	12,979
Total	679	50,619	25	18,958

To resolve this issue we ran some experiments on formulas of Table 2 (we observed similar results on other classes of benchmarks). The idea was to compare *FI* with a SAT-solver whose only difference from *FI* was in the way the set of branching variables was restricted. In *FI*, we pick the next variable to branch on out of $Vars(M(\boldsymbol{p}))$ (i.e. the variables of the clauses falsified by the current complete assignment $\boldsymbol{p}$). We compared *FI* with a SAT-solver that *randomly* chooses a window of $|Vars(M(\boldsymbol{p}))|$ variables and then picks the most active literal among the window variables. If the good performance of *FI* is due to great robustness of decision making based on variable/literal activity, then the performance of these two SAT-solvers should be close.

The results of experiments are shown in Tables 5 and 6. Both tables have identical structure and give data for unsatisfiable and satisfiable formulas. The second and third columns describe the number of variables and clauses (in thousands). The fourth and fifth columns give the number of conflicts and the average window size (in percent) for *FI* computed over all decision assignments. For every decision, the window size was computed for the current set of variables whose values had not been fixed yet. For example, if, in the current formula, there are 100,000 of such variables and $|Vars(M(\boldsymbol{p}))|$ =1000, then the window size is 1%. For the formula *pa.25* (first formula of Table 5) the average window size (computed over all decision assignments) was 0.43%.

The column before the last shows the results of the SAT-solver that used random windowing. For example, for the formula *pa.25*, for all decisions we used windows of the size 0.43% (the average size of *FI* windows for this formula). For instance, if the current set of available branching variables was 100,000, then this SAT-solver would generate a random window W of 430 variable and then picks the most active literal among the variables of W. For the formula *pa.25*, the SAT-solver with random windowing proved unsatisfiability after 374 backtracks (conflicts). The last column gives the result when the most active literal was selected among *all* available branching variables (i.e. the window size was 100%).

The results of Table 5 and 6 show the advantage of windowing of *FI* over random windowing. So one can not claim that good performance of *FI* is due to robustness of decision making based on computing conflict activity of literals. Interestingly, the advantage of *FI* windowing is greater for satisfiable formulas. Only in one case (*fetch.30*), the SAT-solver with random windowing had fewer backtracks before finding a satisfying assignment. For the rest of the formulas, *FI* found a satisfying assignment with a considerably smaller number of backtracks. The results of this subsection show that windows specified by complete assignments are "special". So these results can be considered as a strong argument in favor of proof encryption semantics.

Note that, in general, the SAT-solver choosing the most active literal out of *all* available variables, had fewer backtracks than *FI*. But as we saw in the previous subsection the number of decision and implied assignments (not shown here) made by such a SAT-solver may be much larger than for *FI*.

Table 5. Different choice of windows for unsatisfiable formulas

Name	#Vars $\times\ 10^3$	#Clauses $\times\ 10^3$	*FI*		*no complete assgn.*	
			#cnfl.	window size (%)	#cnfl. (window)	#cnfl (all vars)
pa.25	337	995	127	0.43	374	88
pa.50	727	2,146	1,436	2.02	46,590	480
mcbdm.20	127	360	157	2.03	549	163
mcbdm.40	273	773	2,019	1.22	12,280	8,700
cmt.100	63	178	194	0.03	9,753	469
cmt.200	127	358	403	0.01	27,529	1,715
always.20	61	186	158	1.28	238	244
always.40	136	415	2,955	2.89	4,763	1,994
pc_top.40	379	1,126	1,066	0.41	5,447	752
pc_top.50	480	1,424	3,609	0.52	11,786	3,747
lsu.10	89	239	3,253	1.01	7.015	1,024
stdata.10	110	302	2,764	1.65	7,901	725

Table 6. Different choice of windows for satisfiable formulas

Name	#Vars $\times 10^3$	#Clauses $\times 10^3$	*FI*		*no complete assgn.*	
			#cnfl.	window size (%)	#cnfl. (window)	#cnfl (all vars)
sched.10	188	508	159	0.36	4,992	60
sched.20	384	1,044	1,367	0.51	21,038	389
byteen.10	69	184	1,007	1.25	7,093	1,566
byteen.15	110	292	5,196	1.55	21,277	5,171
bar.10	371	1,102	1,157	0.08	56,594	36
gmtx.50	645	1,887	708	2.42	12,298	199
muls.20	307	944	647	0.20	594,669	57
fetch.30	168	501	325	2.29	19	168
write.40	233	695	1,949	1.24	46,773	1,172
prop7.50	496	1,475	1,527	1.35	86,349	148
resp_grant.20	302	892	877	2.97	9,775	82

Table 7. Formulas with unsatisfiable core (1 hour timeout)

Names	*BerkMin561* #conflicts	*Minisat* #conflicts	*FI* #conflicts
f10k10_1	$> 2,354,579$	$> 1,586,998$	756
f10k10_2	$> 2,530,886$	$> 1,509,013$	935
f10k10_3	$> 2,014,759$	$> 1,519,403$	908
f10k10	4	6	1,209
f3k50_1	$> 6,542,130$	$> 3,188,959$	48
f3k50_2	$> 6,538,252$	$> 4,300,869$	18,127
f3k50_3	$> 6,479,246$	79	18,890
f3k50	74	71	15,447

6.4 Finding unsatisfiable subformulas

In this subsection, we consider the performance of *BerkMin*, *Minisat* and *FI* on 8 artificial formulas with small unsatisfiable subformulas. The results are shown in Table 7. Formulas $f10k10\_i$ and $f3k50\_i$, i=1,2,3 were obtained from formulas $f10k10$ and $f3k50$ by random permutation of variables. The formula $f10k10$ was obtained by adding to a hard random formula F_1 of 10,000 variables the clauses of a random unsatisfiable formula F_2 of 10 *new* variables. The formula $f3k50$ was obtained in the same way as $f10k10$. The only difference was that formula F_1 had 3,000 variables and F_2 had 50 variables. (Note that we got results similar to those of Table 7 even when variables of F_1 and F_2 had a “weak” overlap.) Timeout limit was set to 1 hour.

As we mentioned in Subsection 4.1, one of the advantages of operating on complete assignments is easy identification of unsatisfiable subformulas. For a formula $F_1 \wedge F_2$ of Table 7, no matter how *FI* picks a complete assignment $\boldsymbol{p}$, the set $M(\boldsymbol{p})$ will contain a clause of the unsatisfiable subformula F_2. In this experiment, we modified the first decision-making heuristic of *FI* (described in Subsection 2.4) as follows. For every clause C of the formula, its activity was maintained. This activity was computed as the number of conflicts in which C was involved. Every 100 decisions *FI* picked the clause C' of $M(\boldsymbol{p})$ with the lowest activity and fixed an assignment to a variable of C'. This way we made *FI* to look for unsatisfiable subformulas (because the modification above forced *FI* to fix an assignment to a variable of F_2 once in a while). Note that this modification does not change *FI*'s performance on other formulas much because in 99% cases *FI* makes a regular decision. As one can see from Table 7, *BerkMin* and *Minisat* easily solved 2 and 3 formulas respectively, but for the rest of the formulas they got stuck in the hard subformula F_1. On the other hand, *FI* easily solved all 8 formulas.

7. Conclusions and future directions

In this paper, we introduce a proof encryption semantics whose main idea is to treat a set of complete assignments as an "encryption" of a resolution proof. We believe that this semantics better explains what resolution-based SAT-solvers with clause learning do. We describe a SAT-solver *FI* inspired by the proof encryption semantics. *FI* can be viewed as a regular DPLL-like SAT-solver that maintains a complete assignment to reduce the choice of variables to branch on. Experiments show that for large industrial formulas even our primitive implementation of *FI* is competitive with *Minisat* and *BerkMin.* Experiments also give a strong evidence in favor of proof encryption semantics. Namely they show that *FI* can not be efficiently simulated by random windowing of branching variables.

References

[1] L. Bachmair and H. Ganzinger. Resolution theorem proving in A.Robinson, A.Voronkov editors, *The Handbook of Automated Reasoning*, chap. 2, vol. **1**, pp. 19-99. Elsevier Science Pub., 2001.

[2] P. Beame, H. A. Kautz, and A. Sabharwal. Towards Understanding and Harnessing the Potential of Clause Learning. *JAIR* 2004, number 22, pp.319-351.

[3] M. Davis, G. Logemann, and D.Loveland. A Machine program for theorem proving. *Communications of the ACM*, 1962, vol. **5**, pp. 394-397.

[4] N. Eén. and N. Sörensson. An extensible SAT-solver. *Proceedings of SAT-2003 in LNCS* **2919**, pp.503-518.

[5] H. Fang and W. Ruml. Complete Local Search for Propositional Satisfiability. *Proc. of 19th National Conference on Artificial Intelligence*, 2004, pp.161-166.

[6] E. Goldberg. Testing satisfiability of CNF formulas by computing a stable set of points. *CADE*-2002, pp. 161-180.

[7] E. Goldberg. On bridging simulation and formal verification. *VMCAI*-2008, San Francisco, USA, *LNCS* **4905**, pp.127-141.

[8] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust SAT-Solver. *DATE*-2002, pp. 142-149.

[9] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. *DATE*-2003, pp. 886-891.

[10] C. P. Gomes, B. Selman, H. A. Kautz. Boosting combinational search through randomization. *AAAI*-1998, pp. 431-437.

[11] D. Habet, C. M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for SAT. *International Conference on Principles and Practice of Constraint Programming*, 2002, pp. 172-184.

[12] E. Hirsch, A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Math. and Artif. Intelligence* 2005, vol. **43**(1-4), pp.91-111.

[13] B. Mazure, L. Sais, and R. Gregoire. Boosting complete techniques thanks to local search methods. *Annals of Math. and Artif. Intelligence* 1998, vol. **22**, pp. 319-331.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *Design Automation Conference*, 2001, pp. 530-535.

[15] S. Prestwich and I. Lynce. Local Search for Unsatisfiability. SAT-2006, *LNCS* **4121**, pp. 283-296.

[16] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *AAAI*-92, pp. 440-446.

[17] B. Selman, H. A. Kautz, and B.Cohen. Noise strategies for improving local search. *AAAI*-94, Seattle, pp. 337-343.

[18] B. Selman, H. A. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. *IJCAI*-97, Nagoya, Aichi, Japan.

[19] J. P. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions of Computers*, 1999, vol **48**, pp. 506-521.

[20] H. Zhang. SATO: An efficient propositional prover. *Proceedings of the International Conference on Automated Deduction*, 1997, pp. 272-275.

[21] L. Zhang, C. Madigan, M. Moskewicz, S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, *ICCAD*-2001, San Jose, pp. 279-285.

[22] http://www.cadence.com/webforms/cbl_software/index.aspx