Enhancing Reasoning with the Extension Rule in CDCL SAT Solvers

Rodrigue Konan Tchinda and Clémentin Tayou Djamegni

December 8, 2019

Enhancing Reasoning with the Extension Rule in CDCL SAT Solvers

Rodrigue Konan Tchinda and Clémentin Tayou Djamegni

Department of Mathematics and Computer Science
University of Dschang
Dschang
Cameroon
{rodriguekonanktr, dtayou}@gmail.com

ABSTRACT. The extension rule first introduced by G. TSEITIN is a simple but powerful rule that, when added to *resolution*, leads to an exponentially stronger proof system known as *extended resolution*(ER). Despite the outstanding theoretical results obtained with ER, its exploitation in practice to improve SAT solvers' efficiency still poses some challenging issues. There have been several attempts in the literature aiming at integrating the extension rule within CDCL SAT solvers but the results are in general not as promising as in theory. An important remark that can be made on these attempts is that most of them focus on reducing the sizes of the proofs using the extended variables introduced in the solver. We adopt in this work a different view. We see extended variables as a means to enhance reasoning in solvers and therefore to give them the ability of reasoning on various semantic aspects of variables. Experiments carried out on the 2018 SAT competition's benchmarks show the use of the extension rule in CDCL SAT solvers to be practically useful for both satisfiable and unsatisfiable instances.

RÉSUMÉ. La règle d'extension introduite pour la première fois par G. TSEITIN est une règle simple mais puissante qui, ajoutée à la *résolution*, conduit à un système de preuves plus puissant appelé *résolution étendue* (ER). Malgré les résultats théoriques remarquables obtenus avec ER, son exploitation pratique pour améliorer l'efficacité des solveurs SAT pose encore quelques problèmes. Plusieurs tentatives visant à intégrer la règle d'extension aux solveurs CDCL SAT existent dans la littérature, mais les résultats ne sont en général pas aussi prometteurs qu'en théorie. Une remarque importante à faire sur ces tentatives est qu'elles se concentrent pour la plupart sur la réduction de la taille des preuves à l'aide des variables étendues introduites dans le solveur. Nous adoptons dans ce travail un point de vue différent. Nous considérons les variables étendues comme un moyen d'améliorer le raisonnement dans les solveurs et donc de leur donner la capacité de raisonner sur différents aspects sémantiques des variables. Les expérimentations réalisées sur les instances tirées de la compétition SAT 2018 montrent que l'utilisation de la règle d'extension dans les solveurs CDCL est utile aussi bien pour les instances satisfiables que celles insatisfiables.

KEYWORDS : SAT, CDCL, extension rule

MOTS-CLÉS : SAT, CDCL, règle d'extension

1. Introduction

The Boolean satisfiability problem (SAT) consists in deciding whether a given propositional logic formula — generally expressed in conjunctive normal form or CNF — admits a model or not. There have been tremendous advances in its resolution during the last two decades and nowadays, SAT solvers are used in industry to solve several challenging problems. The key of this great success lies in a very subtle combination of several features within the so-called CDCL (Conflict-Driven Clause Learning) [10, 18, 19, 5] SAT solvers. The latter include conflicts analysis with clause learning, efficient unit propagation through watched literals, dynamic branching/polarity heuristics and sporadic restarts. SAT has also attracted theoreticians since it was the first problem proved to be NP-complete [3]. Hence, the existence or non-existence of an efficient algorithm for SAT will definitely give the answer to the question $\mathrm{P} =? \mathrm{NP}$ which is one of the seven millennium prize problems stated by the *Clay Mathematics Institute* for which an award of 1 Million USD is given to anyone solving one of them.

Despite their current great efficiency, there are still some instances that are out of the reach of current CDCL SAT solvers. The increasing quest of efficiency is achieved by equipping solvers with new techniques and heuristics but the latter are limited from a theoretical perspective. Indeed, CDCL SAT solvers can be formalized as proof systems and it has been shown that the resulting proof system is p-equivalent to general resolution [2, 14] which is known to have an exponential lower bound [7]. This means that CDCL SAT solvers cannot do better than what can be done with general resolution and in particular, that exponential lower bounds known for resolution hold for CDCL SAT solvers as well. To overcome this limitation, a promising research direction is to equip solvers with proof systems that are stronger than resolution. One such proof system is *extended resolution* (ER) which makes use of the *extension rule*.

Several work aiming at integrating the extension rule within CDCL SAT solvers exist in the literature [8, 1]. Most of them focus on the use of the extension rule as a means to reduce the size of the proofs produced by solvers using extended variables introduced in the latter. Seen like that, it might appear that the extension rule is only beneficial for unsatisfiable formulas. However, extended variables introduced in the solver can be seen as a means to increase the level of abstraction and hence enhance reasoning in the solver so that it helps improve the resolution of both satisfiable and unsatisfiable formulas.

We are interested in this work in designing a new integration scheme of the extension rule within CDCL SAT solvers in order to enhance their reasoning capabilities by the use of the extension rule.

The main contributions of this paper are the following: (1) we design a new integration scheme of the extension rule within CDCL SAT solvers called Extended CDCL (ECDCL in short) aiming at enhancing solvers' reasoning. (2) We prove that the substitution of extended literals performed on an asserting clause does not alter its asserting characteristic nor its asserting level. (3) We implemented ECDCL on top of a state-of-the-art SAT solver and conducted an empirical evaluation.

The rest of this paper is organized as follows: Section 2 presents the necessary background for understanding the contribution. In Section 3 we review some related work. Our contribution is given in section 4 and empirically evaluated in Section 5. We finally conclude our work in Section 6 while outlining some future research directions.

2. Background

A Boolean variable is one that domain is $\{true, false\}$. A literal is either a Boolean variable x or its negation $\neg x$. A clause c is a finite disjunction of literals ($c = l_1 \vee ... \vee l_k$) and a CNF formula $\mathcal{F}$ is a finite conjunction of clauses ($\mathcal{F} = c_1 \wedge \cdots \wedge c_n$). A CNF formula can be also seen as a set of clauses where each clause is thought of as a set of literals. In this way, we can use set operators on CNF formulas and clauses. An interpretation $\mathcal{I}$ of a CNF formula $\mathcal{F}$ is a function that maps each variable in $\mathcal{F}$ to a truth value in $\{true, false\}$. A clause is said to be satisfied under an interpretation $\mathcal{I}$ if at least one of its literals is satisfied under $\mathcal{I}$. A CNF formula is said to be satisfied under an interpretation $\mathcal{I}$ if all its clauses are satisfied under $\mathcal{I}$. A CNF formula $\mathcal{F}$ is satisfiable if there can be found an interpretation under which it is satisfied; otherwise it is unsatisfiable. The Boolean satisfiability problem (SAT) consists in deciding whether a given CNF formula is satisfiable or not. The latter definition, which considers only formulas in the CNF representation, is not a restriction since every propositional logic formula can be efficiently translated into an equisatisfiable CNF formula [17].

The most widespread algorithm today for solving SAT is known as CDCL (*Conflict-Driven Clause Learning*) [10, 18, 19, 5]. The principle of CDCL can be summarized as follows: the algorithm performs a sequence of unit propagations until a fixed point is reached (i.e. no further unit propagation can be made) or a conflict is found (i.e. a clause is falsified). If no conflict was found, the algorithm proceeds by making a decision and subsequently increases the decision level. Each assigned variable is associated to a *decision level* and a literal l is said to be of *level* k if it is the kth decision literal or is deduced by unit propagation after setting the kth decision literal. If a conflict is found, then procedure *analyze* is invoked to examine it in order to produce an *asserting clause* (i.e. a clause that is falsified under the interpretation being constructed and that contains only one literal of the conflicting decision level) as well as the decision level at which the solver must backtrack in order to continue the search. Afterward, the solver learns the asserting clause and backtracks accordingly. From time to time, the algorithm performs restarts which consist in backtracking at *decision level zero* and begin a new search while keeping some information of the previous round (such as learned clauses, variable activities, etc.) which might help speed up the new search.

We formally characterize the state of a CDCL SAT solver by the tuple $(\mathcal{F}, \Delta, \delta)$ where $\mathcal{F}$ is the formula being solved, Δ the learned clause database and δ the partial interpretation being constructed by the solver. We denote the level of a literal/variable x relatively to a solver state S by $level(x)$. Given an asserting clause c w.r.t. a state S, its *asserting literal* is the literal with the highest decision level and its *asserting level* is the second highest decision level of literals in c. In this paper, the state should be clear from the context when not explicitly specified.

A propositional proof system is a polynomial time algorithm V, such that for every propositional formulas $\mathcal{F}$, $\mathcal{F}$ is unsatisfiable iff there exists a string P (a proof of unsatisfiability or refutation of $\mathcal{F}$) such that V accepts the input $(\mathcal{F}, P)$. In the rest of this paper, we omit the word *propositional* and refer to *propositional proof system* simply as *proof system*. Given two proof systems V_1 and V_2, V_1 p-simulates V_2 iff there exists a polynomial-time computable function f such that V_2 accepts $(\mathcal{F}, P)$ iff V_1 accepts $(\mathcal{F}, f(P))$. Two proof systems are p-equivalent if they p-simulate each other. A well-known proof system is *resolution* (also referred to as *general resolution*) which makes use of the resolution rule [15] as inference rule. The strength of *resolution* can be further increased by adding the *extension rule*.

The *extension rule* first introduced by TSEITIN [17] allows the use of literals as abbreviation for longer formulas. Concretely, let $\mathcal{F}$ be a CNF formula, $\{x, l_1, l_2\}$ be a set of literals such that neither x nor $\neg x$ appears in $\mathcal{F}$. The extension rule allows to introduce definitions of the form $x \leftrightarrow l_1 \vee l_2$ by adding the clauses $\neg x \vee l_1 \vee l_2$; $x \vee \neg l_1$ and $x \vee \neg l_2$ to $\mathcal{F}$. This rule when added to *resolution*, turns it to an exponentially stronger proof system known as *extended resolution* (ER). A typical example of formulas that are hard for resolution are pigeonhole formulas which do not admit any short (i.e. polynomial size) resolution proof [7]. However, short proofs of pigeonhole formulas exist when using the extension rule [4]. The challenge when using the extension rule is to determine which variables to choose for extension so as to produce short proofs. Even with the right variable choices, the resolution steps that should be performed to achieve this goal still constitute an important issue.

3. Related Work

There have been several work attempting to integrate the extension rule within CDCL SAT solvers. AUDEMARD *et al.* [1] argued that significant advances in SAT solving must come from implementation of stronger proof systems since exponential lower bounds are known for resolution [7, 11]. They used a restriction of ER called *Local Extended Resolution (LER)* by introducing the extension $z \leftrightarrow l_1 \vee l_2$ if there exists previously derived clauses in the form $\neg l_1 \vee \alpha$ and $\neg l_2 \vee \beta$ where α and β are disjunction of literals such that $l \in \alpha \Rightarrow \neg l \notin \beta$. A clear limitation of this approach is that it uses clauses of particular form that might seldom appear in the set of derived clauses. In addition, looking for such clauses can be difficult and costly. For the latter reasons, the authors in their implementation restricted this lookup to a small window of recent clauses, only looking for those of the form $\neg l_1 \vee \alpha$ and $\neg l_2 \vee \alpha$.

HUANG [8] proposed Extended Clause Learning (ECL), a general scheme which is a modification of the CDCL algorithm where decisions to use the extension rule might be made (guided by a heuristic) when the number of assigned literals is greater than 2. Besides ECL, they proposed a concrete heuristic where the extension rule was used after learning clauses γ of size greater than 2. Concretely, if the decision to make an extension is taken, then γ is split into $\alpha \vee \beta$ such that $|\alpha| \geq 2$ and $|\beta| > 0$ and the solver learns the clauses $x \vee \beta$, $x \leftrightarrow \alpha$ where x is a fresh variable. A restart is performed after each extension introduced in the solver. The drawback of this is that it alters the restart strategy of the solver. Hence, if the heuristic used to decide the time to make extensions is not well designed, it might compromise the completeness of the CDCL SAT solver. For instance, if we decide to make an extension after each conflict, the solver will never reach more than one conflict and the search will hardly progress in this situation.

In [9], JABBOUR *et al.* proposed a method that mimics the principle behind extended resolution by detecting hidden Boolean functions introduced in the CNF during the encoding phase [13, 6] and by using them to shorten learned clauses through substitution. This approach however does not use fresh variables at all and substitution is restricted to only literals which are the input arguments of a detected Boolean function.

4. Extended CDCL

The extension rule in combination with resolution has been theoretically shown to be useful for shortening the proof size of unsatisfiable formulas. The work mentioned in the literature try to reproduce this result in practice but the outcome of most of them turns to be limited as it seldom matches the expectations. When looking at the extension rule as a means to reduce the size of the proof, it might seem that it will be useful only for unsatisfiable formulas. We adopt here a different view. Extensions are introduced within a solver in order to increase its reasoning capabilities with the ultimate goal of enhancing solving times. When solving a CNF formula, current CDCL SAT solvers proceed by assuming a selected variable to be *true* or *false* and by evaluating the consequences of this assumption on the formula being solved. Proceeding this way limits reasoning to a single semantic aspect of variables, notably their truth values. We want the solver to be able to carry out reasoning on other semantic aspects of formulas' variables. That is, we want the solver in addition to make other types of assumptions such as assuming that two or more variables are equivalent, simultaneously *true* or *false*, one variable implies the other etc. To achieve this without modifying the way solvers proceed, we are going to use extensions to encapsulate these semantic aspects. Hence, the algorithm of the solver will not change since it will still continue to carry out reasoning as usual; that is, assigning *true/false* values to variables. The difference however will be that when this reasoning is performed on an extended variable, it will denote other semantic aspects. For instance, suppose the extension $x \leftrightarrow l_1 \Leftrightarrow l_2$ has been made in the solver. When the solver picks the extended variable x and assigns it value $true$, this means that it is assuming l_1 and l_2 to be equivalent. Hence, after setting x to *true*, anytime in the search where one of the variables in $\{l_1, l_2\}$ will be given a value, then the other will also be given the same value via unit propagation. In this way, we could expect an improvement of reasoning in the solver.

In order to integrate the extension rule in the CDCL framework, we should answer the following questions: which variables should be chosen for extension and when should we perform these extensions?

We propose to use extensions at restarts and to choose for each extension, two variables from two different decision levels (in our implementation, we chose the most active decision variables i.e. the ones with the highest VSIDS scores [12]). At this level, extensions can be performed using any binary connective; for instance $x \leftrightarrow l_1 \vee l_2$, $x \leftrightarrow l_1 \wedge l_2$, $x \leftrightarrow l_1 \Rightarrow l_2$, $x \leftrightarrow l_1 \Leftrightarrow l_2$ etc. The solver can then make assumptions on the extended variables resulting in an implicit meaning on variables on which extension is performed (for instance, they are/are not equivalent, one implies/does not imply the other, they are both *false/true*, etc.) and evaluates the consequences on the other variables of the formula being solved. By choosing variables of different decision levels instead of any two variables, we aim to give the solver the ability to carry out reasoning on variables that apparently seem not to be dependent and avoid some useless extensions such as extensions where the value of one literal is already known or extensions where the extended variable is immediately forced to a given value. We further provide a substitution mechanism that can be cheaply performed in order to favor the use of extended variables within the solver. This substitution consists in replacing any pair $\{l_1, l_2\}$ of literals in a clause c by the literal l provided that the extension $l \leftrightarrow l_1 \vee l_2$ has been made in the solver. This substitution has as effect the shortening of the clause as well as the increase of its propagation power. In fact, as illustrated in [1], if we consider the clause $c = l_1 \vee l_2 \vee \alpha$ and the extension

$l \leftrightarrow l_1 \vee l_2$, then the clause $c' = l \vee \alpha$ obtained from c by replacing $l_1 \vee l_2$ with l unlike the clause c itself will become unit once all literals in α are set to *false*. The substitution mechanism is performed on each asserting clause derived after conflict analysis and Proposition 4.1 ensures that after substitution, the resulting clause will remain asserting and the asserting level will be kept.

Proposition 4.1. *Let $S = (\mathcal{F}, \Delta, \delta)$ be the state of a solver and $c = a \vee \alpha$ an asserting clause w.r.t. S where a is the asserting literal. Let $x \leftrightarrow l_1 \vee l_2$ be an extension such that $\{l_1, l_2\} \subseteq \alpha$. Then, the clause $c' = a \vee x \vee \beta$ where $\beta = \alpha \setminus \{l_1, l_2\}$ is asserting w.r.t. S. Furthermore, the asserting levels of c and c′ are identical.*

The proof of Proposition 4.1 uses the following lemma:

Lemma 4.1. *Let $x \leftrightarrow l_1 \vee l_2$ be an extension introduced in a solver. If literals x, l_1 and l_2 are all assigned, with x set to* false*, then the decision level of x is the maximum decision level of l_1 and l_2.*

Proof. Since $x = false$ and $x \leftrightarrow l_1 \vee l_2$, then l_1 and l_2 are set to *false* as well. If x is first set to *false* by the solver or set to *false* after setting either of l_1 or l_2 to *false* (no matter the decision level), then the values of l_1 and/or l_2 will immediately be deduced by unit propagation through the clauses $x \vee \neg l_1$ and $x \vee \neg l_2$. In this case, the decision levels of x and l_1 or x and l_2 will be identical. If l_1 and l_2 are first set to *false*, then $x = false$ will be inferred by unit propagation via the clause $\neg x \vee l_1 \vee l_2$. This occurs as soon as the last literal of $\{l_1, l_2\}$ is assigned. In either of the previous cases, the level of x is the same as that of the most recently assigned literal of $\{l_1, l_2\}$, that is $level(x)$ is the maximum of $level(l_1)$ and $level(l_2)$. □

Proof of Prop. 4.1. $c = a \vee \alpha$ is an asserting clause, hence all its literals are *false*. Since $x \leftrightarrow l_1 \vee l_2$ is an extension introduced in the solver and $\{l_1, l_2\} \subseteq c$, then x is *false* as well. By Lemma 4.1, $level(x) = max(level(l_1), level(l_2))$. In addition, $level(l_1) < level(a)$ and $level(l_2) < level(a)$, hence, $level(x) < level(a)$ which means that a still has the highest decision level in $c' = a \vee x \vee \beta$ where $\beta = \alpha \setminus \{l_1, l_2\}$. c' therefore remains asserting. Furthermore, the second highest decision level of literals in c' remains the same as in c since $max(\{level(y), y \in \alpha\}) = max(\{level(y), y \in (\beta \cup \{x\})\})$. □

The resulting scheme that we call *Extended CDCL* (ECDL in short) is described in Algorithm 1. All in this algorithm are as in CDCL except that we introduce extensions at restarts (lines 14–17) and substitution of literals with extended literals for asserting clauses derived from conflicts (line 8). At line 17, the extension operator $\circ$ can be any binary connective and in this paper we take $\circ \in \{\vee, \wedge, \Rightarrow; \Leftrightarrow\}$. Note that an extension is not systematically added at each restart but only when the solver finds it necessary (through an heuristic) and when the maximum number of extensions (introduced as a parameter of the algorithm) is not yet reached. Function *substituteExtendedLits* which carries out substitution is described in Algorithm 2. In order to perform substitution, all extensions are converted so that they use the connective $\vee$. Hence, the extensions $l \leftrightarrow l_1 \wedge l_2$ and $l \leftrightarrow l_1 \Rightarrow l_2$ are respectively converted to $\neg l \leftrightarrow \neg l_1 \vee \neg l_2$ and $l \leftrightarrow \neg l_1 \vee l_2$. As far as the extension $l \leftrightarrow l_1 \Leftrightarrow l_2$ is concerned, we use two auxiliary variables $\{l', l''\}$ for its conversion which lead to the extensions $\neg l \leftrightarrow \neg l' \vee \neg l''$, $l' \leftrightarrow \neg l_1 \vee l_2$ and $l'' \leftrightarrow l_1 \vee \neg l_2$. It is worth mentioning that substitution here is not performed on unary or binary clauses since it requires that the clause contains at least two literals in addition to the asserting literal. Notice that function *substituteExtendedLits* will return the clause

Algorithm 1: Extended CDCL

Input: A CNF formula $\mathcal{F}$
Result: SAT or UNSAT

begin
  $dl \leftarrow 0$; $\Delta \leftarrow \emptyset$;
  while *true* **do**
    $conf \leftarrow$ unitPropagation($\mathcal{F} \cup \Delta$);
    if $conf \neq null$ **then**
      if $dl = 0$ **then return** UNSAT;
      $(c, \text{btLevel}) \leftarrow$ analyze($conf$); `/* return an asserting clause and the backtracking level */`
      $c \leftarrow$ substituteExtendedLits(c);
      $\Delta \leftarrow \Delta \cup \{c\}$; `/* learn clause c */`
      backtrack to $btLevel$;
    else
      if *all variables are assigned* **then return** SAT;
      if *time to restart* **then**
        if *extension needed and not reached max number of extensions* **then**
          let δ be the current interpretation;
          choose a fresh variable x and a not yet extended pair $\{l_1, l_2\}$ from δ such that $level(l_1) > 0, level(l_2) > 0, level(l_1) \neq level(l_2)$;
          $\mathcal{F} \leftarrow \mathcal{F} \cup clauses(x \leftrightarrow l_1 \circ l_2)$; `/* encode the extension as clauses and add to the formula */`
        restart();
      pick an unassigned variable and assign it a value;
      $dl \leftarrow dl + 1$;

Algorithm 2: substituteExtendedLits

Input: An asserting clause c
Result: a clause

begin
  $a \leftarrow$ the asserting literal of c ;
  $c' \leftarrow \{a\}$;
  $c \leftarrow c \setminus \{a\}$;
  while $\exists \{l_1, l_2\} \subseteq c$ *such that the extension* $l \leftrightarrow l_1 \vee l_2$ *is present in the solver* **do**
    $c \leftarrow c \setminus \{l_1, l_2\}$;
    $c' \leftarrow c' \cup \{l\}$;
  $c' \leftarrow c' \cup c$;
  return c';

as is if called with an unary or a binary clause or even when no extension has yet been introduced in the solver. Another thing to point out in Algorithm 1 is that extensions are made with only two literals. This does not mean that reasoning in solvers is limited to at most two literals at a time. In fact, in Algorithm 1, extensions can be made using other extended literals as well. Hence, extensions with more literals such as $l \leftrightarrow l_1 \circ l_2 \circ \cdots \circ l_n$ can be represented by a sequence of two-literal extensions. The drawback here is that several fresh variables need to be introduced in the solver for that.

Unlike Extended Clause Learning (ECL) which performs a restart after each extension made after conflicts analysis, ECDCL does not alter the restart policy of the CDCL solver and hence eliminates the completeness issue related to an uncontrolled restart strategy. Furthermore, since we limit the number of extensions that can be used, ECDCL remains complete.

5. Experimental Results

In order to evaluate ECDCL, we conducted experiments on the 400 application benchmarks [1] drawn from the 2018 SAT competition [2]. For these experiments, we implemented ECDCL on top of the state-of-the-art CDCL SAT solver *Glucose3.0* [3]. *Glucose3.0* was chosen because it is one of the most used today as a base for many CDCL SAT solvers. We distinguished several different versions obtained by varying the type of extension and the maximum number of extended variables allowed in the solver: We designated by *Glucose3.0_exT_○_maxExtVars_K* the version of our solver where the extension operator is $\circ$ with $\circ \in \{\vee, \wedge, \Rightarrow; \Leftrightarrow\}$ and the maximum number of extended variables allowed set to $K \in \{100; 500; 1000\}$. When performing and extension $l \leftrightarrow l_1 \circ l_2$, we set the VSIDS score of l to the sum of the scores of l_1 and l_2 so that when making a decision in the solver, the extended variable l is prioritized overs l_1 and l_2. It is worth mentioning that in our modified solvers, all the other parameters of *Glucose3.0* were left unchanged. The base solver, referred to as *Glucose_3.0_default* was used with its default configuration.

All our experiments were carried out on the StarExec [4] [16] cluster infrastructure running Red Hat Enterprise Linux Server version 7.2 (Maipo). Each node of this infrastructure has 128 GB of memory and two Intel processors with 4 cores (2.4 GHz) each. For each solver, we set a time limit of 1800 seconds and a memory limit of 24GB for the resolution of each benchmark.

The results obtained at the end of these experiments are summarized in Table 1. The table indicates for each extension type and maximum number of extended variables allowed, the number of satisfiable instances solved (#S), the number of unsatisfiable instances (#U), the total number of instances solved (#T) as well as the PAR-2 score. The PAR-2 score is defined as the *sum of all runtimes for solved instances* $+ 2 \times$ *timeout for unsolved instances*. The line *default* in the table represents the performance of *Glucose3.0_default* which are repeated for every limit on the number of extended variables to ease the comparisons. We see in this table that all our solvers outperformed the original solver on the total number of instances solved (up to 14 additional instances solved for our best performing solver). The second remark is that our solvers are very efficient on satisfiable instances (our best performing solver on satisfiable instances solved 12 more sat-

1. http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip
2. http://www.satcompetition.org/
3. https://www.labri.fr/perso/lsimon/downloads/softwares/glucose-3.0.tgz
4. https://www.starexec.org/

isfiable instances than the original solver) meaning that the extension rule greatly helped improve the resolution of satisfiable instances.

	Number of extended vars ≤ 100				Number of extended vars ≤ 500				Number of extended vars ≤ 1000			
	#S	#U	#T	PAR-2	#S	#U	#T	PAR-2	#S	#U	#T	PAR-2
exT_$\vee$	**83**	**71**	**154**	**2436,36**	**82**	69	**151**	**2454,71**	76	**69**	145	2499,94
exT_$\wedge$	73	**71**	144	2515,47	78	**70**	148	2472,11	75	68	143	2518,84
exT_$\Rightarrow$	80	70	150	2455,51	79	**70**	149	2484,04	75	67	142	2521,33
exT_$\Leftrightarrow$	74	67	141	2531,56	75	68	143	2505,41	**84**	68	**152**	**2456,92**
default	72	68	140	2526,22	72	68	140	2526,22	72	68	140	2526,22

Table 1. *Results, #S, #U and #T are respectively the number of satisfiable, the number of unsatisfiable and the total number of instances solved*

This table also shows that the number of solved instances generally decreases as the limit on the number of extended variables increases. There is however an exception for the extension operator $\Leftrightarrow$ where the performance increased with the limit on the number of extended variables.

When considering another performance metric, notably the average PAR-2 score [5] currently used for ranking solvers at SAT competitions, we also notice that all our solvers outperformed the original solver except for *Glucose3.0_exT_$\Leftrightarrow$_maxExtVars_100*.

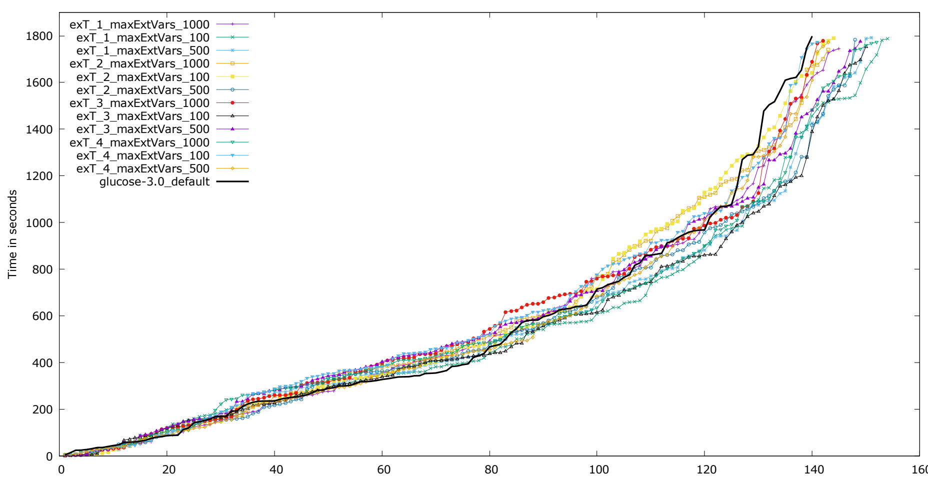

Figure 1. *Cactus plots, for the extension type, 1 = $\vee$, 2 = $\wedge$, 3 = $\Rightarrow$, 4 = $\Leftrightarrow$*

Fig. 1 shows the cactus plots of all the solvers on the 400 application benchmarks of the 2018 SAT competition. It clearly appears on these plots that the use of the extension rule in the solver was beneficial for our solvers since they still outperformed the original solver *Glucose_3.0_default* for various solving time limits in term of number of instances solved.

5. Solvers with the lowest scores are the best performing

6. Conclusion and Future Work

We presented in this paper a new integration scheme of the extension rule within CDCL called *extended CDCL* (ECDCL) which, unlike the state-of-the-art integrations, uses extensions to enhance reasoning in solvers. ECDCL also allows to substitute literals in asserting clauses with extended literals while preserving their asserting nature as well as the asserting levels. We showed experimentally that the extension rules helps improve the resolution of both satisfiable and unsatisfiable instances.

This work opens doors to many other investigations. For instance, an interesting research direction is to determine if our algorithm seen as a proof system is theoretically strictly more powerful than general resolution. Additionally, it might be interesting to see whether the proof system implemented in our algorithm p-simulates ER. Some extensions introduced in the solver might already exist in the original formula in the form of Boolean function [9]. So it would be interesting to detect and use them instead of making new extensions with fresh variables which unnecessarily increase the formula size.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments and valuable suggestions on a previous version of this paper.

References

[1] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[2] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[3] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[4] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

[5] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[6] Éric Grégoire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Saïs. Automatic extraction of functional dependencies. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 122–132. Springer, 2004.

[7] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297 – 308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[8] Jinbo Huang. Extended clause learning. *Artif. Intell.*, 174(15):1277–1284, October 2010.

[9] Saïd Jabbour, Jerry Lonlac, and Lakhdar Saïs. Extending resolution by dynamic substitution of boolean functions. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, pages 1029–1034. IEEE, 2012.

[10] JP Marques-Silva and KA Sakallah. Grasp-a new search algorithm for satisfiability. iccad, 1996.

[11] Mladen Mikša and Jakob Nordström. Long proofs of (seemingly) simple formulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 121–137. Springer, 2014.

[12] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[13] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting structural knowledge from cnf formulas. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 185–199, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[14] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[15] John Alan Robinson et al. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[16] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: a cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, pages 367–373. Springer, 2014.

[17] G Tseitin. On the complexity ofderivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[18] Hantao Zhang. Sato: An efficient prepositional prover. In *Automated Deduction-CADE-14*, pages 272–275. Springer, 1997.

[19] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.