Enhancing Reasoning with the Extension Rule in CDCL SAT Solvers

Rodrigue Konan Tchinda, Clémentin Tayou Djamegni

ARIMA

épisciences

Enhancing Reasoning with the Extension Rule in CDCL SAT Solvers

Rodrigue Konan Tchinda[*1,2] **and Clémentin Tayou Djamegni**[1]

[1]University of Dschang, Cameroon
[2]University of Bamenda, Cameroon

*E-mail : rodriguekonanktr@gmail.com

DOI : 10.46298/arima.6434
Submitted on 28 April 2020 - Published on 23 October 2021
Volume : **33 - 2020** - Year : **2021**
Special Issue : **CRI 2019**
Editors : *Eric Badouel, Maurice Tchuenté, René Ndoundam, Paulin Melatagia*

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 and 2020 SAT competitions' benchmarks show the use of the extension rule in CDCL SAT solvers to be practically beneficial for both satisfiable and unsatisfiable instances.

Keywords

SAT; CDCL; Extension Rule

I 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) [6–8, 10] 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 [13, 17] which is known to have an exponential lower bound [5]. 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 [15, 16]. 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 II presents the necessary background for understanding the contribution. In Section III we review some related work. Our contribution is given in section IV and empirically evaluated in Section V. We finally conclude our work in Section VI while outlining some future research directions.

II 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. Given a CNF formula $\mathcal{F}$ and a literal l, we write $\mathcal{F}_{|l} = \{c | c \in \mathcal{F}, \{l, \neg l\} \cap c = \emptyset\} \cup \{c \backslash \{\neg l\} | c \in \mathcal{F}, \neg l \in c \text{ and } l \notin c\}$. $\mathcal{F}_{|l}$ denotes the simplified formula obtained from $\mathcal{F}$ by removing all clauses $c \in \mathcal{F}$ such that $l \in c$ and $\neg l$ from clauses containing it. This simplification can be extended to a set of literals $\{l_1, \cdots l_k\}$; thereby $\mathcal{F}_{|\{l_1,\cdots,l_k\}}$ is the formula obtained from $\mathcal{F}$ by successively applying the previous simplification rule on $l_1, l_2, \cdots$ and l_k i.e. $\mathcal{F}_{|\{l_1,\cdots,l_k\}} = (...(\mathcal{F}_{|l_1})_{|l_2}...)_{|l_k}$. *Unit propagation* is the application of the rule $\mathcal{F}_{|x}$ for each unit clause $\{x\} \in \mathcal{F}$ until a clause in $\mathcal{F}$ is falsified or $\mathcal{F}$ does not contain unit clauses anymore.

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 [2].

The most widespread algorithm today for solving SAT is known as CDCL (*Conflict-Driven Clause Learning*) [6–8, 10]. 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 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. The size of the proof from a given formula is defined as the number of inference steps in the proof. A well-known proof system is *resolution* (also referred to as *general resolution*) which makes use of the resolution rule [1] as inference rule. The strength of *resolution* can be further increased by adding the *extension rule*.

The *extension rule* first introduced by TSEITIN [2] 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 [5]. 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.

III RELATED WORK

There have been several work attempting to integrate the extension rule within CDCL SAT solvers. AUDEMARD *et al.* [15] argued that significant advances in SAT solving must come from implementation of stronger proof systems since exponential lower bounds are known for resolution [5, 20]. They used a restriction of ER called *Local Extended Resolution (LER)* by introducing the extension $z \leftrightarrow l_1 \vee l_2$ if there exist 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 $\neg l_2 \vee \alpha$. Some implementation-level optimizations of LER are proposed in [19].

HUANG [16] 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 [18], 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 [12, 14] 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.

IV EXTENDED CDCL

4.1 Motivation

The extension rule in combination with resolution has been theoretically shown to be useful for shortening the proof size of unsatisfiable formulas. The contributions mentioned in the literature try to reproduce this result in practice but the outcomes of most of them turn to be limited as they seldom match 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 is 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.

4.2 Description of the integration scheme

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 [9]). 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 can be seen

as an application of hyper-resolution [11] involving the original clause and some binary clauses encoding the extensions. It has as effect the shortening of the clause as well as the increase of its propagation power. In fact, as illustrated in [15], 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 1 ensures that after substitution, the resulting clause will remain asserting and the asserting level will be kept.

Proposition 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 1 uses the following lemma:

Lemma 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. 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 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 a heuristic) and when the maximum number of extensions (introduced as a parameter of the algorithm) is not yet reached. Hence, by *extension needed* we mean the moment the solver decides to make an extension typically through a heuristic. 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

Algorithm 1: Extended CDCL

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

```
1  begin
2      dl ← 0; Δ ← ∅;
3      while true do
4          conf ← unitPropagation(F ∪ Δ);
5          if conf ≠ null then
6              if dl = 0 then return UNSAT;
7              (c, btLevel) ← analyze(conf); /* return an asserting clause and the
                backtracking level */
8              c ← substituteExtendedLits(c);
9              Δ ← Δ ∪ {c};                                  /* learn clause c */
10             backtrack to btLevel;
11         else
12             if all variables are assigned then return SAT;
13             if time to restart then
14                 if extension needed and not reached max number of extensions then
15                     let δ be the current interpretation;
16                     choose a fresh variable x and a not yet extended pair {l1, l2} from δ such that
                        level(l1) > 0, level(l2) > 0, level(l1) ≠ level(l2) ;
17                     F ← F ∪ clauses(x ↔ l1 ∘ l2) ;      /* encode the extension as
                        clauses and add to the formula */
18                 restart();
19             pick an unassigned variable and assign it a value;
20             dl ← dl + 1;
```

Algorithm 2: substituteExtendedLits

Input: An asserting clause c
Result: A clause

```
1  begin
2      a ← the asserting literal of c ;
3      c' ← {a};
4      c ← c \ {a};
5      while ∃{l1, l2} ⊆ c such that the extension l ↔ l1 ∨ l2 is present in the solver do
6          c ← c \ {l1, l2};
7          c' ← c' ∪ {l};
8      c' ← c' ∪ c;
9      return c';
```

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 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.

V EXPERIMENTAL RESULTS

In order to evaluate ECDCL, we conducted a first stage of 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 *Glucose-3.0*[3]. *Glucose-3.0* 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 *Glucose-3.0_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 $K \in \{100, 500, 1000\}$. In our implementations, we used a very straightforward heuristic to decide when the extensions are made: it consists of making an extension at every restart performed when the current interpretation contains at least two decision levels, starting from the beginning of the search until the maximum number of allowed extensions is reached. This heuristic deserves a further study but we leave it for future investigations. When performing an 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 *Glucose-3.0* were left unchanged. The base solver, referred to as *Glucose-3.0_default* was used with its default configuration. Preprocessing was enabled for all versions of *Glucose-3.0* used in the experiments including the default one.

All our experiments were carried out on the StarExec[4] [21] 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.

[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/

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 *Glucose-3.0_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 satisfiable instances than the original solver) meaning that the extension rule greatly helped improve the resolution of satisfiable instances.

	Number of extended vars $\leq$ 100				**Number of extended vars $\leq$ 500**				**Number of extended vars $\leq$ 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: Performance of *Glucose-3.0* and *Glucose-3.0_exT_*$\circ$*_maxExtVars_K* on the 2018 SAT Competition's benchmarks when substitution is carried out on all learned clauses

This table also shows that the number of solved instances generally decreases as the limit on the number of extended variables increases when substitution is carried out on all learned clauses. 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 *Glucose-3.0_exT_*$\Leftrightarrow$*_maxExtVars_100*.

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 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 solved instances. Fig. 2 shows the scatter plot comparing the original solver *Glucose-3.0_default* and our best performing integration namely *Glucose-3.0_exT_*$\vee$*_maxExtVars_100*. It appears on this plot that the run time of many instances has been significantly improved: this is noticeable through the presence of many points in the upper triangle that are far from the diagonal. However there are still a number of instances where the original solver is faster.

We noticed during our experiments that the substitution operations were very time consuming despite our optimizations. To address this issue, we decided to restrict the substitutions to the learned clauses that were most likely to remain in solvers for a longer time. Thus, we restricted the substitutions to clauses whose LBD values at the time of their learning were less than or equal to an empirically chosen threshold set to 20. With this new optimization, we conducted a second phase of experiments, this time with an extended set of benchmarks including those of

[5]Solvers with the lowest scores are the best performing

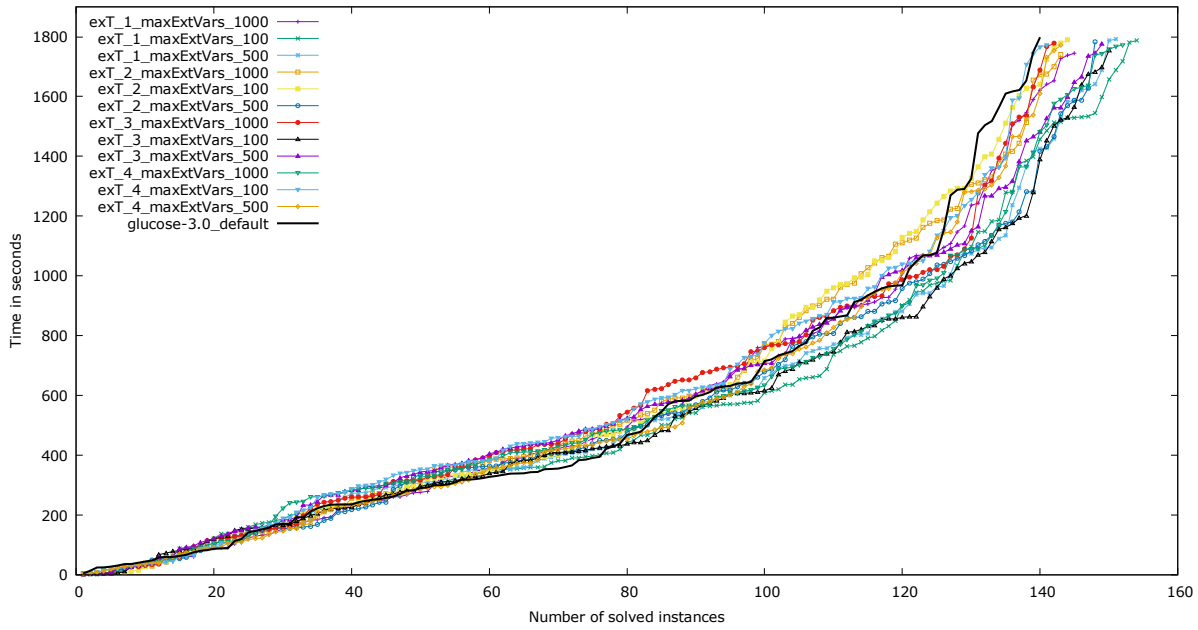

Figure 1: Cactus plots, for the extension type, 1 = $\vee$, 2 = $\wedge$, 3 = $\Rightarrow$, 4 = $\Leftrightarrow$

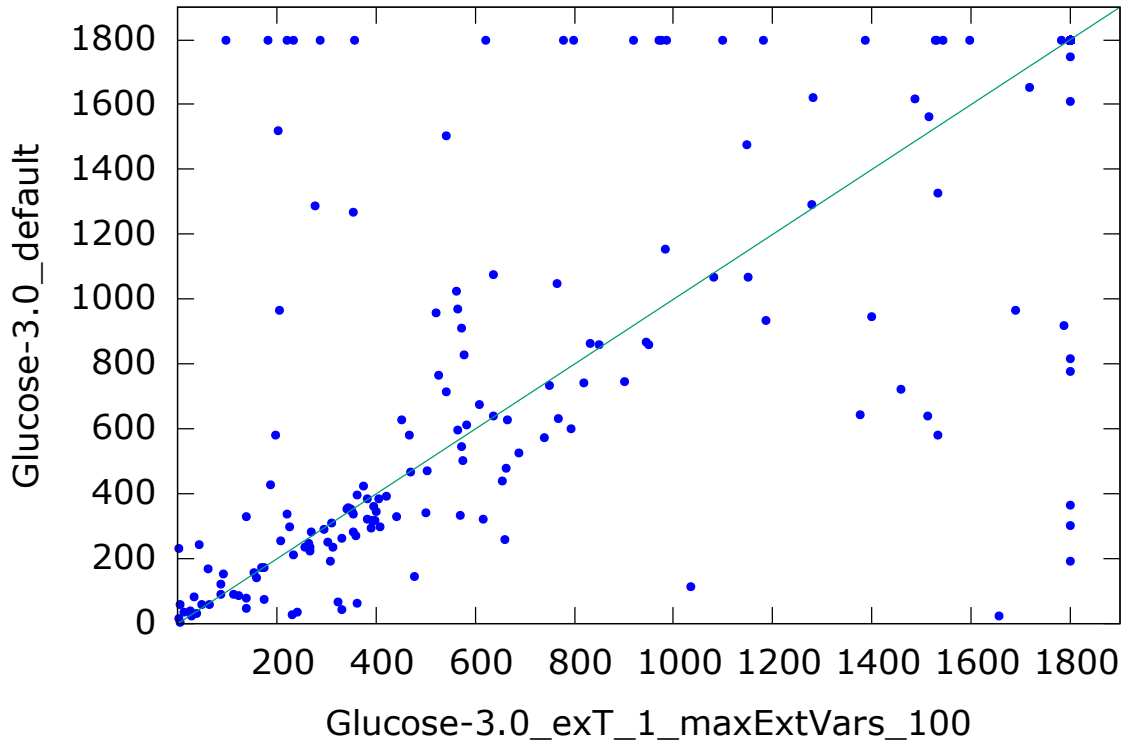

Figure 2: Scatter plot Glucose-3.0_exT_$\vee$_maxExtVars_100 vs Glucose-3.0_default

the SAT competitions 2018[6] and 2020[7]. Moreover, we integrated ECDCL with this optimization in one of the current top performing SAT solvers, *MapleLCMDistChronoBT*[8], in order to

[6] http://sat2018.forsyte.tuwien.ac.at/benchmarks/Main.zip
[7] https://satcompetition.github.io/2020/downloads/sc2020-main.uri
[8] http://sat2018.forsyte.tuwien.ac.at/solvers/

evaluate the impact on its performance. The versions of the solvers used in these experiments were named in the same way as before, i.e. *MapleLCMDistChronoBT_exT_∘_maxExtVars_K* is the version of *MapleLCMDistChronoBT* integrating ECDCL where the extension operator is $\circ$ with $\circ \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ and the maximum number of extended variables allowed set to $K \in \{100, 500, 1000\}$. The default version of *MapleLCMDistChronoBT* as far as it is concerned was named *MapleLCMDistChronoBT_default*. For all these versions of *MapleLCMDistChronoBT* including the default one used in our experiments, we set the `--chrono` option to -1 and enabled preprocessing. The experiments were conducted under the same conditions as described in the first phase and the results obtained are reported in Table 2 and Table 3. Each of these tables has two parts respectively for the benchmarks of the SAT Competition 2018 (SC2018) and for the SAT Competition 2020 (SC2020).

		Number of extended vars $\leq$ 100				**Number of extended vars $\leq$ 500**				**Number of extended vars $\leq$ 1000**			
		#S	#U	#T	PAR-2	#S	#U	#T	PAR-2	#S	#U	#T	PAR-2
SC2018	exT_$\vee$	**78**	**72**	**150**	**2465.88**	80	**74**	**154**	**2426.68**	80	**74**	**154**	**2427.34**
	exT_$\wedge$	72	71	143	2494.51	**83**	71	154	2433.98	**81**	70	151	2448.30
	exT_$\Rightarrow$	75	69	144	2493.24	79	73	152	2455.69	80	69	149	2466.28
	exT_$\Leftrightarrow$	71	71	142	2498.26	81	71	152	2443.35	73	72	145	2490.11
	default	72	68	140	2526.22	72	68	140	2526.22	72	68	140	2526.22
SC2020	exT_$\vee$	**50**	61	111	2778.94	**54**	64	**118**	**2732.46**	50	62	112	2765.89
	exT_$\wedge$	49	67	**116**	**2749.18**	50	60	110	2788.62	**51**	62	113	2757.20
	exT_$\Rightarrow$	45	**68**	113	2761.84	52	**66**	**118**	2747.27	43	63	106	2821.51
	exT_$\Leftrightarrow$	49	67	**116**	2750.60	47	64	111	2773.05	58	63	**121**	**2706.09**
	default	47	65	112	2773.01	47	65	112	2773.01	47	**65**	112	2773.01

Table 2: Performance of *Glucose-3.0* and *Glucose-3.0_exT_∘_maxExtVars_K* on the 2018 and 2020 SAT Competitions' benchmarks when substitutions are carried out only on learned clauses having LBD lower than or equal to 20

		Number of extended vars $\leq$ 100				**Number of extended vars $\leq$ 500**				**Number of extended vars $\leq$ 1000**			
		#S	#U	#T	PAR-2	#S	#U	#T	PAR-2	#S	#U	#T	PAR-2
SC2018	exT_$\vee$	114	**86**	200	2025.28	111	84	195	2045.07	113	84	197	2042.22
	exT_$\wedge$	108	**86**	194	2066.72	**119**	**88**	**207**	**1953.83**	**116**	87	**203**	1986.98
	exT_$\Rightarrow$	**118**	**86**	**204**	**1983.04**	111	**88**	199	2038.26	110	**88**	198	2034.79
	exT_$\Leftrightarrow$	**118**	84	202	1993.29	117	85	202	1987.66	112	86	198	2031.29
	default	115	**86**	201	1986.70	115	86	201	1986.70	115	86	201	**1986.70**
SC2020	exT_$\vee$	**63**	74	**137**	**2585.92**	57	69	126	2656.81	60	70	130	2636.93
	exT_$\wedge$	60	73	133	2616.19	54	70	124	2670.61	58	70	128	2648.66
	exT_$\Rightarrow$	62	71	133	2608.55	61	72	133	2601.37	59	72	131	2617.42
	exT_$\Leftrightarrow$	59	71	130	2621.29	**62**	72	**134**	2610.89	**62**	72	**134**	2599.70
	default	59	**75**	134	2592.51	59	**75**	**134**	**2592.51**	59	**75**	**134**	**2592.51**

Table 3: Performance of *MapleLCMDistChronoBT* and *MapleLCMDistChronoBT_exT_∘_maxExtVars_K* on the 2018 and 2020 SAT Competitions' benchmarks when substitutions are carried out only on learned clauses having LBD lower than or equal to 20

We can see in Table 2 that for the SAT Competition 2018 benchmarks, the PAR-2 scores as well as the number of solved unsatisfiable instances have been significantly improved compared to those of the first phase experiments in Table 1. This clearly indicates that it is more favorable to perform substitutions on clauses with low LBD scores. In Tables 2 and 3 it appears that the most notable improvements still reside at the level of satisfiable instances both for those of the 2018 and the 2020 SAT Competitions. We see for instance in Table 2 that the solver *Glucose-3.0_exT_∧_maxExtVars_500* (resp. *Glucose-3.0_exT_∨_maxExtVars_500*) solved 11 (resp. 7) satisfiable instances of the SC2018 (resp. SC2020) more than the base solver *Glucose-3.0_default* and in Table 3 that *MapleLCMDistChronoBT_exT_∧_maxExtVars_500*

(resp. *MapleLCMDistChronoBT_exT_∨_maxExtVars_100*) solved 4 (resp. 4) more satisfiable instances of the SC2018 (resp. SC2020) than the base solver *MapleLCMDistChronoBT_default*. Even if we can see clear improvements on the unsatisfiable instances of the SAT Competition 2018 in Table 2, we still notice a decrease in performance at the level of those of the SAT Competition 2020. This same observation is made on Table 3 for the solvers *MapleLCMDistChronoBT_exT_∘_maxExtVars_K*. From Table 3, we can see that the gain in performance for the solvers *MapleLCMDistChronoBT_exT_∘_maxExtVars_K* is not as important as that for *Glucose-3.0_exT_∘_maxExtVars_K* solvers and even sometimes we observe a decrease in performance. This is justifiable because the base solver *MapleLCMDistChronoBT* is a highly optimized and therefore more difficult to improve.

The scatter plots in Fig 3 compare the performance of some of our best integrations with the base solvers. We see on these plots in general that many instances were solved with run times almost similar to those of the base solvers. This is noticeable through the presence of several points near the diagonals. However, with the presence of many points above the diagonals, we note that the solving times of many instances have been improved. It is also important to mention that for all these plots, there are many points at the top and right borders indicating the instances solved by one of the solvers and not by the other.

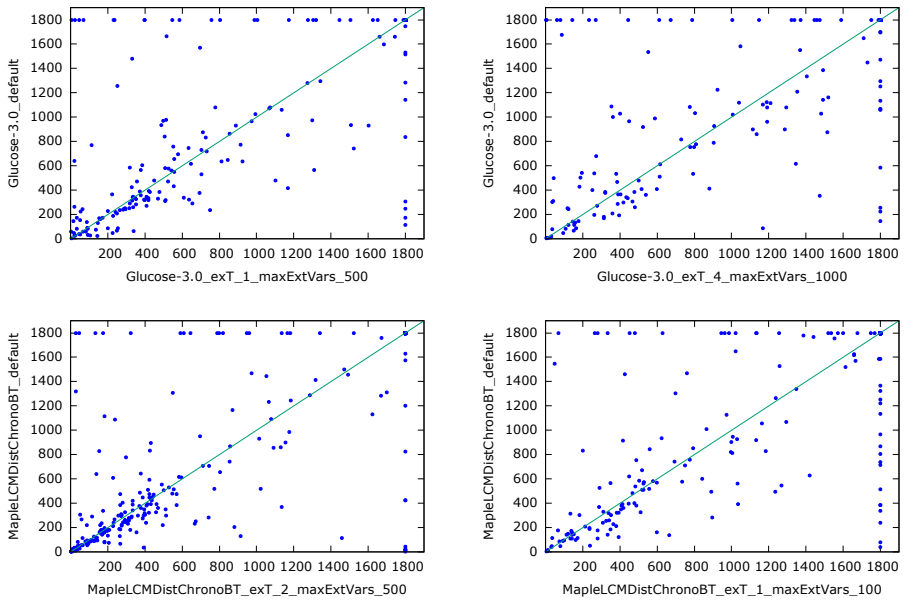

Figure 3: Scatter plots comparing some of our best integrations with the base solvers. At the left (resp. right) we have the comparison using the SC2018 (resp. SC2020) benchmarks

VI 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 rule helps improve the resolution of satisfiable instances and in some cases unsatisfiable instances as well.

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 functions [18]. 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

Publications

[1] J. A. Robinson et al. "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM* 12.1 (1965), pages 23–41.

[2] G. Tseitin. "On the complexity ofderivation in propositional calculus". In: *Studies in Constrained Mathematics and Mathematical Logic* (1968).

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

[4] S. A. Cook. "A Short Proof of the Pigeon Hole Principle Using Extended Resolution". In: *SIGACT News* 8.4 (Oct. 1976), pages 28–32. ISSN: 0163-5700.

[5] A. Haken. "The intractability of resolution". In: *Theoretical Computer Science* 39 (1985). Third Conference on Foundations of Software Technology and Theoretical Computer Science, pages 297–308. ISSN: 0304-3975.

[6] J. Marques-Silva and K. Sakallah. "Grasp-a new search algorithm for satisfiability. ICCAD". In: IEEE Computer Society Press, 1996, pages 220–227.

[7] H. Zhang. "SATO: An efficient prepositional prover". In: *Automated Deduction-CADE-14*. Springer, 1997, pages 272–275.

[8] C. P. Gomes, B. Selman, H. Kautz, et al. "Boosting combinatorial search through randomization". In: *AAAI/IAAI* 98 (1998), pages 431–437.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pages 530–535.

[10] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. "Efficient conflict driven learning in a boolean satisfiability solver". In: *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press. 2001, pages 279–285.

[11] F. Bacchus. "Enhancing Davis Putnam with extended binary clause reasoning". In: *AAAI/IAAI* 2002 (2002), pages 613–619.

[12] R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. "Recovering and Exploiting Structural Knowledge from CNF Formulas". In: *Principles and Practice of Constraint Programming*. Edited by P. Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 185–199. ISBN: 978-3-540-46135-7.

[13] P. Beame, H. Kautz, and A. Sabharwal. "Towards understanding and harnessing the potential of clause learning". In: *Journal of Artificial Intelligence Research* 22 (2004), pages 319–351.

[14] É. Grégoire, R. Ostrowski, B. Mazure, and L. Saıs. “Automatic extraction of functional dependencies”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2004, pages 122–132.

[15] G. Audemard, G. Katsirelos, and L. Simon. “A Restriction of Extended Resolution for Clause Learning SAT Solvers”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Volume 24. 1. 2010, pages 15–20.

[16] J. Huang. “Extended Clause Learning”. In: *Artificial Intelligence* 174.15 (Oct. 2010), pages 1277–1284. ISSN: 0004-3702.

[17] K. Pipatsrisawat and A. Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: *Artificial Intelligence* 175.2 (2011), pages 512–525.

[18] S. Jabbour, J. Lonlac, and L. Saıs. “Extending resolution by dynamic substitution of boolean functions”. In: *24th International Conference on Tools with Artificial Intelligence*. Volume 1. IEEE. 2012, pages 1029–1034.

[19] N. Manthey. “Extended resolution in modern SAT solving”. In: *Joint Automated Reasoning Workshop and Deduktionstreffen*. 2014, pages 26–27.

[20] M. Mikša and J. Nordström. “Long proofs of (seemingly) simple formulas”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2014, pages 121–137.

[21] A. Stump, G. Sutcliffe, and C. Tinelli. “StarExec: a cross-community infrastructure for logic solving”. In: *International Joint Conference on Automated Reasoning*. Springer. 2014, pages 367–373.