On Hierarchies over the SLUR Class

Tomáš Balyo and **Štefan Gurský** and **Petr Kučera**[*] and **Václav Vlček**[†] [‡]
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Abstract

SLUR is a class of the CNF formulae on which the single lookahead unit resolution algorithm for satisfiability (SAT) testing never fails. It is known that the SLUR class contains classes of Horn, hidden Horn, (hidden) extended Horn and balanced formulae as a subclass. In recent paper it was proven that the SLUR class contains all canonical formulae. In this paper we will extend this result by showing it suffices that all prime implicates can be derived in one resolution step from input formula. We will also generalize previous hierarchy called SLUR(i) which is built on top of the SLUR class.

1 Introduction

The satisfiability problem (SAT) is to decide whether a given formula φ in CNF has a satisfying assignment, i.e. whether for some assignment t of values 0 (false) or 1 (true) to variables we have that $\varphi(t)$ evaluates to 1 (true). This problem was the first one shown to be NP-complete (Cook 1971; Garey and Johnson 1979). Thus, unless P=NP, no polynomial time algorithm can solve this problem. There are, however, many classes of formulae for which polynomial SAT algorithms are known. These classes of formulae include Horn formulae (Dowling and Gallier 1984; Itai and Makowsky 1987; Minoux 1988), hidden Horn formulae (Lewis 1978; Aspvall 1980), extended Horn formulae (Chandru and Hooker 1991), and CC-balanced formulae (Conforti, Cornuťjols, and Vuskovic 2006). These four classes share an interesting property: the satisfiability problem for formulae from these classes can be solved by unit resolution, namely by the single look-ahead unit resolution (SLUR) algorithm (Schlipf et al. 1995; Franco and Van Gelder 2003).

The SLUR algorithm works as follows. In each step it chooses an unassigned variable and a truth value nondeterministically and then repeatedly performs as many unit propagations as possible. If no empty clause (i.e. contradiction) is derived, then algorithm fixes this assignment, otherwise it tries the other value. If both possibilities leads to contradictions, the algorithm gives up. We say that a formula belongs to the SLUR class, if the SLUR algorithm does not give up for any possiblity of nondeterministic choices.

In (Čepek and Kučera 2010; Čepek, Kučera, and Vlček 2011) it was shown that problem of recognizing whether given formula belongs to class SLUR is coNP-complete. It was also shown there that every formula containing all prime implicates of the represented function is contained in the SLUR class. As a conseqeunce we have that every function has a SLUR representation.

In (Vlček 2009; Čepek, Kučera, and Vlček 2011) a hiearchy called SLUR(i) was built on top of the SLUR class. The idea of this hierarchy was that at i-th level we choose i variables simultaneously and we consider all 2^i possible assignments of truth values to these variables. We will generalize this hierarchy into the hierarchy called SLUR*(i). It can be seen as a variant of DPLL procedure (Davis, Logemann, and Loveland 1962) which is allowed to backtrack at most i levels back. We shall show that SLUR(i) is properly contained in SLUR*(i) as well as some more results concerning SLUR*(i) hierarchy structure.

In Section 2 we will present basic definitions and known results. In Section 3 we will show that formulae that contains almost all prime implicates are contained in the SLUR class. Then in Section 4 we will present the SLUR*(i) hierarchy and some of its properties. Finally, in Section 5 we will sum up the results and give some open questions.

2 Definitions and results

Boolean function on n variables is a mapping $f : \{0,1\}^n \mapsto \{0,1\}$. We say that function f is *satisfiable* if there is an $x \in \{0,1\}^n$ such that $f(x) = 1$.

A *literal* is either a variable or its negation. *Clause* is a disjunction of literals. We assume that no clause contains both positive and negative literals with the same variable. A clause which contains just one literal is called *unit clause*. Formula F is in *conjuntive normal form (CNF)* if it is a conjuction of clauses. We shall often treat a clause as a set of its literals and a CNF as a set of its clauses. It is a well known fact that every Boolean function can be represented by a formula in propositional logic (Boolean formula), par-

[*]The third author thankfully acknowledges a support by the Czech Science Foundation (grant P202/10/1188)

[†]The first, the second and the fourth author gratefully acknowledge a support by the Charles University Grant Agency (grant No. 266111).

[‡]This research was partially supported by SVV project number 263 314

ticularly formula in conjunctive normal form (CNF) see e.g. (Genesereth and Nilsson 1987).

The set of variables of a formula F will be denoted by V_F. We will use capitals for formulae and the same lower case letters for the function that the given formula represents.

A mapping $v : U \to \{0,1\}$, where $U \subseteq V_F$ is a subset of variables is called a *partial assignment*. We will write $v(F)$ for the formula we get from F after we substitute to the variables of F and delete all the zeros from the clauses and all satisfied clauses from the formula. If we get the empty formula, then the original formula F is *satisfied with* v. On the other hand if a formula contains an empty clause, it is unsatisfiable. We will also write $F[x = 0]$ (or $F[x = 1]$) for a formula that we get after applying a partial assignment that assigns just 0 (1, respectively) to the variable x.

For two Boolean functions f and g of n variables we write $f \leq g$ if $\forall \vec{x} \in \{0,1\}^n f(\vec{x}) = 1 \Rightarrow g(\vec{x}) = 1$. Since each formula represents a function and each clause can be seen as a formula we can extend this notation to formulae and clauses as well. We say that a *clause* C_1 *subsumes a clause* C_2 if $C_1 \leq C_2$ holds, it is in fact equivalent to $C_1 \subseteq C_2$. A clause C is called an *implicate of* f if $f \leq C$ (e.g. each clause of a representation of f is its implicate). C is a *prime implicate* if there is no other implicate C' subsuming C. We say that a formula F is a *canonical representation of function* f if it consists of all prime implicates of f. A formula F is *irredundant* if removing any of its clauses changes the function it represents.

We say that two clauses have a *conflict in variable* x if there is a positive occurence of x in one clause and negative occurence in the other. Two clauses $C_1 = (\widetilde{C_1} \vee x)$ and $C_2 = (\widetilde{C_2} \vee x)$ are *resolvable on* x if $\widetilde{C_1}$ and $\widetilde{C_2}$ do not have a conflict in any variable. We write $R(C_1, C_2) = \widetilde{C_1} \vee \widetilde{C_2}$ and this disjuction is called a *resolvent*, clauses C_1 and C_2 are called *parent clauses*. It is a well-known fact that resolvent of two implicates is an implicate again (see e.g. (Buning and Letterman 1999)). Resolution in which one of the parent clauses is a unit clause is called *unit resolution*.

Let F be a CNF representing Boolean function f, we say, that C can be derived from F by a series of resolutions if there is a sequence of clauses $C_1, \ldots, C_k = C$ such that every $C_i, 1 \leq i \leq k$ either belongs to F, or $C_i = R(C_{j_1}, C_{j_2})$, where $j_1, j_2 < i$. It is a wellknown fact (see e.g. (Buning and Letterman 1999)) that every prime implicate of f can be derived from F. We define *depth* of resolution derivation of C from F as follows.

1. If $C \in F$, then depth of resolution derivation of C is 0.
2. If C can be derived from F by a series of resolutions $C_1, \ldots, C_k = C$, where $C = R(C_i, C_j)$ with $i, j < k$, then depth of the derivation of C is maximum of depths of resolution derivations of C_i and C_j increased by 1.
3. If C cannot be derived from F by a series of resolutions, then we define depth of resolution derivation of C as infinity.

Note, that definition of depth depends on given series of resolutions. We say, that C has *resolution depth* d with respect to CNF F, if C can be derived by a series of resolutions of depth d and there is no series of resolutions of depth smaller than d which would derive C. In particular C has resolution depth 0, if it belongs to F and C has depth 1, if there are clauses $C_1, C_2 \in F$ such that $C = R(C_1, C_2)$.

In the following lines we will focus on results about and extensions of Boolean formula's class called SLUR. In this paper we focus on the class of Boolean formulae called SLUR (single look-ahead unit resolution). This class was defined in (Franco and Van Gelder 2003; Schlipf et al. 1995). Its definition uses an algorithm similar to well known DPLL procedure (Davis, Logemann, and Loveland 1962). In fact DPLL procedure on a SLUR formula always runs without any backtracks larger than one level. The definition uses a nondeterministic polynomial time Algorithm 2. The basic operation used by this algorithm is unit propagation. Function $unitprop(F)$ for a given formula F in CNF returns a pair of values (F', t), where F' is the CNF formula that results from repeatedly performing unit resolution until no unit clauses remain in the formula, and t is the partial assignment which satisfies unit clauses found and eliminated during unit propagation. It is known, that unitprop can be implemented in time linear in the length of formula φ (Dalal and Etherington 1992).

Definition 1 *We say that a function F is in the SLUR class if the SLUR algorithm (Algorithm 2) does not return "give up" for any of nondeterministic choices in steps 5 and 16.*

Algorithm 2 SLUR(F)

Input: *A CNF formula F with no empty clause*
Output: *A partial truth assignment satisfying F, "unsatisfiable", or "give up".*

1: $(F, t) := unitprop(F)$
2: **if** F contains an empty clause **then return** "unsatisfiable" **endif**
3: **while** F is not empty
4: **do**
5: Select a variable x present in F
6: $(F_1, t_1) := unitprop(F \wedge \overline{v})$
7: $(F_2, t_2) := unitprop(F \wedge v)$
8: **if** both F_1 and F_2 contain an empty clause **then return** "give up" **endif**
9: **if** F_1 contains an empty clause
10: **then**
11: $(F, t) := (F_2, t \cup t_2)$
12: **else if** F_2 contains an empty clause
13: **then**
14: $(F, t) := (F_1, t \cup t_1)$
15: **else**
16: Choose one of the following two steps:
17: $(F, t) := (F_1, t \cup t_1)$
18: $(F, t) := (F_2, t \cup t_2)$
19: **endif**
20: **enddo**
21: **return** t

SLUR(i)

First idea how to extend the SLUR class is not choosing just one variable in step 5 of SLUR algorithm but choose more variables at once and search for partial assignments that does not lead to an empty clause after applying and performing unitprop. The other thing one notices is that if all partial assignments in the first run of while–cycle (line 3 of the SLUR algorithm) creates an empty clause then the algorithm can return "unsatisfiable".

Using these two ideas a hierarchy called SLUR(i), where i is the number of variables chosen in one run of while cycle, was defined in (Čepek, Kučera, and Vlček 2011; Vlček 2009). It can be easily proven that this hierarchy does not collapse and each formula is contained in some level for hierarchy (it is sufficient to take i greater than number of variables in the formula). See (Čepek, Kučera, and Vlček 2011; Vlček 2009) for details.

CANON(i)

The last definition we will need to formulate our main result is the following.

Definition 3 *Let F be a CNF and let f be a Boolean function represented by F. We say that F belongs to class CANON(i), for $i \geq 0$, if every prime implicate of f has resolution depth with respect to F not greater than i.*

Note, that each $F \in \text{CANON}(0)$ contains all prime implicates of f (where f is a Boolean function represented by F). If F is moreover prime, it is in fact the canonical representation of f. According to (Čepek and Kučera 2010; Čepek, Kučera, and Vlček 2011), every $F \in \text{CANON}(0)$ belongs to the SLUR class. In next section we will show, that this remains true even for CANON(1), but it is not true for CANON(2).

3 Main result: CANON(1) $\subseteq$ SLUR

In (Čepek and Kučera 2010; Čepek, Kučera, and Vlček 2011) is shown that every formula containing all prime implicates is actually in the SLUR class. This result is equivalent to $\text{CANON}(0) \subseteq \text{SLUR}$. The main result of this paper is to extend this result to $\text{CANON}(1) \subseteq \text{SLUR}$. We will start with showing that the class CANON(1) is closed under partial assignments, which is the key part of the proof.

Lemma 4 *Let $F \in$ CANON(1) and let x be any variable of F. Then both $F[x := 0]$ and $F[x := 1]$ are also in CANON(1).*

Proof We will show only the case $x := 0$, the case $x := 1$ is similar. Let us denote $F' = F[x := 0]$, our goal is to prove that F' is in CANON(1), i.e. each of its prime implicates is either in F' or can be derived from it in one resolution step. Let f denote the function represented by F and let f' denote the function represented by F'.

Let us fix an arbitrary prime implicate C' of f' and let us show, that $C' \in F'$ or there are two clauses $C_1, C_2 \in F'$ such that $C' = R(C_1, C_2)$. That is exactly the property required for F' to belong to CANON(1) and thus by this our proof will be completed.

Because $F' = F[x := 0]$, we can observe, that $C' \vee x$ is an implicate of f. Indeed, let v be an arbitrary assignment satisfying f and let us show, that $C' \vee x$ is satisfied by v, too. If $v(x) = 0$, then v satisfies f' and thus $C'(v) = 1$. If $v(x) = 1$, then clearly $(C' \vee x)(v) = 1$. This implies, that there has to be a prime implicate C of f such that

$$C \leq C' \vee x \tag{1}$$

It follows, that $C[x := 0] \leq (C' \vee x)[x := 0] = C'$ and because C' is a prime implicate of f', while $C[x := 0]$ is an implicate of f', we get, that in fact

$$C[x := 0] = C'. \tag{2}$$

If $C \in F$, then clearly $C' = C[x := 0] \in F$.

Let us now assume that $C \notin F$. From our assumption that $F \in \text{CANON}(1)$ it follows, that there are $C_1, C_2 \in F$ such that $C = R(C_1, C_2)$. We will divide the proof into several cases.

1. Clause C does not contain variable x.
   (a) If the resolution step does not use variable x as a conflict one, then also $x \notin C_1, C_2$, which immediately means that both $C_1 = C_1[x := 0]$ and $C_2 = C_2[x := 0]$ are present in F' including their conflict variable. We can therefore do the same resolution step as before and get $C = C' = R(C_1, C_2)$.
   (b) If on the other hand the resolution step uses x as a conflict variable, then we can write $C_1 = A \vee x$, $C_2 = B \vee \bar{x}$, and $C = R(C_1, C_2) = A \vee B$ for some clauses A, B, which do not have a conflict. After substitution to x we get $C_1[x := 0] = A$ and $C_2[x := 0] = 1$. It follows that $C_1[x := 0] = A \leq A \vee B = C \leq C' \vee x$, where the last inequality follows from (1). Now since $C_1 \in F$, we get that $C_1[x := 0] = A \in F'$ and because C' is a prime implicate of f', it must be the case that in fact $C' = A \in F'$.
2. It remains to consider the case when C contains x, but $C \notin F$. This case is in fact similar to the case when C does not contain x and it was derived by a resolution which did not use x as a conflict variable. Let us again assume, that $C = R(C_1, C_2)$, where $C_1, C_2 \in F$. Therefore $C_1[x := 0], C_2[x := 0]$ are two resolvable clauses which belong to F' and we have that $C' = R(C_1[x := 0], C_2[x := 0])$.

$\diamondsuit$

We will also need a simple observation about unsatisfiable clauses from CANON(1).

Lemma 5 *If F is unsatisfiable and $F \in$ CANON(1), then either F contains an empty clause, or an empty clause is generated during $unitprop(F)$.*

Proof Let f denote the function represented by F. If F is unsatisfiable, then f has the only one prime implicate and that is an empty clause. Let us denote it $\emptyset$. Due to the assumption that $F \in \text{CANON}(1)$, we get that either $\emptyset \in F$, or $\emptyset = R(C_1, C_2)$, where $C_1, C_2 \in F$. In the latter case the only possibility how an empty clause can be generated in one resolution step is, if $C_1 = x$ and $C_2 = \overline{x}$ for some

variable x (or symmetrically $C_1 = \overline{x}$ and $C_2 = x$). This means, that an empty clause would be generated during unit propagation. $\diamondsuit$

Now are ready to prove the main result of this section.

Theorem 6 *If F is in CANON(1) then it is also in the SLUR class.*

Proof If F is unsatisfiable, then by Lemma 5 Algorithm 2 (SLUR) would correctly recognize it after unit propagation in step 2.

Let us assume, that F is satisfiable. Inductive use of Lemma 4 ensures, that at every step of the algorithm every formula considered belongs to CANON(1). This is because every formula originates from F by partial assignment, this is also true for unit propagation. Note also, that using $F \wedge \overline{v}$ corresponds to $F[v := 0]$ and using $F \wedge v$ corresponds to $F[v := 1]$. If at the beginning of the while cycle formula F is satisfiable, then one of F_1 and F_2 is satisfiable and if one of them is unsatisfiable, it contains an empty clause by Lemma 5. Thus at the beginning of the next cycle F is again satisfiable and at the end the SLUR algorithm finds satisfying assignment. $\diamondsuit$

This result can not be extended even more to class CANON(2), as follows from the following example. The following formula F belongs to CANON(2), but it is not SLUR.

$$F = (x \vee y \vee a) \wedge (x \vee \overline{y} \vee b) \wedge (\overline{x} \vee y \vee c) \wedge (\overline{x} \vee \overline{y} \vee d). \quad (3)$$

It can be checked, that all other implicates which can be derived by resolution from F are the following:

$$(x \vee a \vee b), (y \vee a \vee c), (\overline{y} \vee b \vee d), (\overline{x} \vee c \vee d), (a \vee b \vee c \vee d) \quad (4)$$

The last clause has resolution depth 2, and the remaining clauses have resolution depth 1, thus $F \in \text{CANON}(2)$. However, it is not SLUR. If the SLUR algorithm first chooses a, b, c, and d and sets them all to 0, then it gets a complete quadratic CNF, which is unsatisfiable and thus the SLUR algorithm would give up. This implies, that it is not true, that CNFs from CANON(2) would all be SLUR.

Above CNF F has another interesting property. It is the only prime and irredundant CNF representing the same function f. And it is also the only one, which is not SLUR, it suffices to add any other implicate from list (4) to F to make it SLUR.

If e.g. we add $(x \vee a \vee b)$ to F, then the SLUR algorithm would recognize an unsatisfiable formula during unit propagation after setting a, b, c, and d to 0 and thus it would not give up. If x or y would be assigned a value before a, b, c, or d, or if one of a, b, c, or d would be assigned value 1, then SLUR algorithm would get a satisfiable quadratic formula, which is SLUR. The cases of the next three implicates in (4) are symmetric. Adding the implicate $(a \vee b \vee c \vee d)$ makes the formula belong to CANON(1) and therefore SLUR, because it is the only implicate with resolution depth 2. This is also an example of a function, that does not have a prime and irredundant SLUR representation.

4 SLUR*(i)

Let us return to the SLUR(i) hierarchy, which was defined in (Vlček 2009; Čepek, Kučera, and Vlček 2011). Let us consider the following formula

$$\begin{aligned} F = &(x \vee \overline{y}) \wedge (\overline{x} \vee y) \\ &\wedge (x \vee y \vee a \vee b) \wedge (x \vee y \vee \overline{a} \vee b) \\ &\wedge (x \vee y \vee a \vee \overline{b}) \wedge (x \vee y \vee \overline{a} \vee \overline{b}) \end{aligned} \quad (5)$$

It can be observed, that this formula is not in the SLUR(2) class, if we choose x and y and then we choose assignment $x = y = 0$, then the SLUR(2) algorithm gives up as it is left with a complete and thus unsatisfiable quadratic formula. Here SLUR(2) actually does not take any advantage from the fact that it can choose two variables at once, because it chooses two equivalent variables. If however after choosing a value for x, the SLUR(2) algorithm would be allowed to perform unit propagation, then it would not choose y as the second variable and it would recognize, that in case $x = 0$ the rest is an unsatisfiable formula. This example leads us to hierarchy consisting of classes SLUR*(i), in which the algorithm also chooses i variables at each step, but it performs unit propagation between each of these choices rather then after all of them. Formally, SLUR*(i) class is defined using Algorithm 9.

Definition 7 *Formula F is a member of SLUR*(i) class if Algorithm 9 does not return "give up" for any of nondeterministic choices made during its run.*

Firstly we will show the recursive function (Algorithm 8) that takes care of searching the i-decision assignments and then the SLUR*(i) algorithm.

Algorithm 8 test(F, k)

Input: *A CNF formula F, number of decision k which remain to be made by the algorithm.*

Output: *A partial assignment which have not led to an empty clause after k decisions, or UNSAT if no such assignment exists.*

1: $(F, t) := unitprop(F)$
2: **if** F contains an empty clause **then return** UNSAT **endif**
3: **if** k=0 **then return** empty assignment **endif**
4: e :=an undetermined literal (positive or negative)
5: $t'_1 := test(F_1, k-1)$
6: **if** previous test did not return UNSAT **then return** $t \cup t_1 \cup t'_1$ **endif**
7: $t'_2 := test(F_2, k-1)$
8: **if** previous test did not return UNSAT **then return** $t \cup t_2 \cup t'_2$ **endif**
9: **return** UNSAT

Algorithm 9 SLUR*(i, F)

Input: *A CNF formula F without any empty clause*
Output: *A partial truth assignment satisfying F, "unsatisfiable", or "give up".*

1: $(F, t) := unitprop(F)$
2: **if** F contains an empty clause **then return** "unsatisfiable" **endif**
3: **while** F is not empty
4: **do**
5: t' :=test(F, i)
6: **if** previous test returned UNSAT
7: **then**
8: **if** it is the first run of the *while* cycle
9: **then**
10: **return** "unsatisfiable"
11: **else**
12: **return** "give up"
13: **endif**
14: **endif**
15: $t := t \cup t'$
16: **enddo**
17: **return** t

Note, that all nondeterminism is now stored in step 4 of procedure *test* (Algorithm 8). In this step by choosing literal instead of a variable we also give no preference to whether the first value tested will be 1 or 0. The test procedure is in fact a DPLL procedure (see (Davis, Logemann, and Loveland 1962) for details), in which we bound our search by given depth. If i is a fixed constant, algorithm SLUR*(i) runs in polynomial time, though it is naturally exponential with increasing i.

It is easy to show that the SLUR*(i) hierarchy does not collapse, i.e. for every $i \geq 1$ the inclusion

$$\text{SLUR*}(i) \subsetneq \text{SLUR*}(i+1)$$

is strict, in fact exactly the same argumentation as for the previous SLUR(i) hierarchy can be used (Čepek, Kučera, and Vlček 2011; Vlček 2009). Modification of the proof to SLUR*(i) hierarchy is contained in the following lemma.

Lemma 10 *For each i there is a formula F_{i+1} such that $F_{i+1} \in$ SLUR*$(i+1) \setminus$ SLUR*(i).*

Proof Let us take F_{i+1} as the zero function written as a CNF formula on $i+2$ variables $V = \{x_1, \ldots, x_{i+2}\}$. There are all possible combination of positive and negative literals:

$$F_{i+1} = \bigwedge_{P \subseteq V} \left(\bigvee_{v \in P} v \vee \bigvee_{v \in V \setminus P} \overline{v} \right).$$

Now we can see that assigning values to any i-tuple of variables some of the clauses disappear and the rest of the formula is quadratic and unsatisfiable. E.g. if we assign arbitrary zero-one values to $x_1, \ldots, x_i$, we get $(x_{i+1} \vee x_{i+2}) \wedge (x_{i+1} \vee \overline{x_{i+2}}) \wedge (\overline{x_{i+1}} \vee x_{i+2}) \wedge (\overline{x_{i+1}} \vee \overline{x_{i+2}})$. Unit propagation after each variable assignment does not give us anything new, because all clauses are too long. The last $unitprop$ will not derive the empty clause on such CNF and the algorithm SLUR*(i,F_{i+1}) will have to give up in the next step.

On the other hand, if we assign values to any $(i+1)$-tuple we will get a CNF of the $x \wedge \overline{x}$ form. $Unitprop$ will derive the empty clause on such a formula and so the algorithm SLUR*($i+1$,F_{i+1}) returns "unsatisfiable". $\diamondsuit$

It can be immediately seen, that every CNF F on n variables belongs to SLUR*(n). It can be also observed that by doing unit propagation before each choice, we do not loose anything and thus

$$\text{SLUR}(i) \subseteq \text{SLUR*}(i)$$

for every $i \geq 1$, in particular SLUR $\subseteq$ SLUR*(1). The example formula defined in (5) at the beginning of this section shows, that in fact the inclusion SLUR$(2) \subseteq$ SLUR*(2) is strict, this example can in fact be generalized to show the following lemma. (Note, that in case $i = 1$ we do not gain anything and thus SLUR$(1) =$ SLUR*(1)).

Lemma 11 *For every $i > 1$ we have [SLUR$(i) \setminus$ SLUR$(i-1)] \cap$ SLUR*$(2) \neq \emptyset$.*

Proof Let $i > 1$ be a fixed constant and let us consider the following formula:

$$\begin{aligned} F \quad = \quad & (y_1 \vee \overline{y_2}) \wedge \ldots \wedge (y_{i-1} \vee \overline{y_i}) \wedge (y_i \vee \overline{y_1}) \\ & \wedge (y_1 \vee \ldots \vee y_i \vee a \vee b) \wedge (y_1 \vee \ldots \vee y_i \vee \overline{a} \vee b) \\ & \wedge (y_1 \vee \ldots \vee y_i \vee a \vee \overline{b}) \wedge (y_1 \vee \ldots \vee y_i \vee \overline{a} \vee \overline{b}) \end{aligned}$$

This formula is logically equivalent to

$$(y_1 \leftrightarrow y_2 \leftrightarrow \ldots \leftrightarrow y_i)$$
$$\wedge \Big(y_1 \vee \ldots \vee y_i \vee \big((a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \big) \Big)$$

If the SLUR(i) algorithm chooses at first the i-tuple $y_1, \ldots, y_i$, and then it sets all these variables to 0, then it will get an unsatisfiable complete quadratic formula on variables a and b, which means, the SLUR(i) algorithm will give up.

On the other hand Algorithm 9 which performs unit propagation after each pick of a variable will assign equivalent values to all $y_1, \ldots, y_i$ variables after it will come across the first one of them. So it can use the remaining step to deal with the rest of the formula. Again, no problem can arise, if the first chosen variable is a or b. This means that formula F belongs to SLUR*(2). $\diamondsuit$

The following is now an easy corollary.

Corollary 12 *For every $i > 1$ we have that SLUR$(i) \subsetneq$ SLUR*(i).*

Let us now return to the classes CANON(i). Let F be CNF defined in (3), we have seen that this CNF belongs to CANON(2), but it is not SLUR, now we can even observe, that F does not belong to SLUR(2), this is because if the SLUR(2) algorithm chooses first a and b and sets them to 0, then c and d and sets them to 0, what remains is a complete unsatisfiable quadratic formula. Moreover, in this case unit propagation after choosing value for a or c does not help

and thus F does not even belong to SLUR*(2). By concatenating copies of F by disjoint union, we could in fact get an example of a formula showing the following lemma (we omit formal proof).

Lemma 13 *For every $i \geq 1$ we have that $[SLUR^*(i) \setminus SLUR^*(i-1)] \cap CANON(2) \neq \emptyset$.*

This means, that CANON(2) is not a subclass of any level of SLUR*(i) hierarchy.

5 Conclusion

We focused on SLUR formulae and their generalizations into two hierarchies. We have studied relations among newly defined hierarchies of CANON(i) and SLUR*(i) formulae and the hierarchy SLUR(i) defined in (Vlček 2009; Čepek, Kučera, and Vlček 2011). There is still open question, whether there are natural classes which would contain hierarchy CANON(i) and whether there is a satisfiability algorithm for each of these classes, which would be polynomial for a fixed i. Following definition of CANON(i), there is such an algorithm based on resolution. So we would like to see, if there is a satisfiability algorithm based on unit propagation and the SLUR algorithm.

References

Aspvall, B. 1980. Recognizing disguised nr(1) instances of the satisfiability problem. *Journal of Algorithms* 1(1):97 – 103.

Buning, H. K., and Letterman, T. 1999. *Propositional Logic: Deduction and Algorithms*. New York, NY, USA: Cambridge University Press.

Chandru, V., and Hooker, J. N. 1991. Extended horn sets in propositional logic. *J. ACM* 38(1):205–221.

Conforti, M.; Cornuťjols, G.; and Vuskovic, K. 2006. Balanced matrices. *Discrete Mathematics* 306(19-20):2411–2437.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, 151–158. New York, NY, USA: ACM.

Dalal, M., and Etherington, D. W. 1992. A hierarchy of tractable satisfiability problems. *Information Processing Letters* 44(4):173–180.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Commun. ACM* 5:394–397.

Dowling, W., and Gallier, J. 1984. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming* 3:267 – 284.

Franco, J., and Van Gelder, A. 2003. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Appl. Math.* 125:177–214.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman and Company.

Genesereth, M. R., and Nilsson, N. J. 1987. *Logical foundations of artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Itai, A., and Makowsky, J. 1987. Unification as a complexity measure for logic programming. *Journal of Logic Programming* 4:105 – 117.

Lewis, H. R. 1978. Renaming a set of clauses as a horn set. *J. ACM* 25:134–135.

Minoux, M. 1988. Ltur: A simplified linear time unit resolution algorithm for horn formulae and computer implementation. *Information Processing Letters* 29:1 – 12.

Schlipf, J. S.; Annexstein, F. S.; Franco, J. V.; and Swaminathan, R. P. 1995. On finding solutions for extended horn formulas. *Inf. Process. Lett.* 54:133–137.

Čepek, O., and Kučera, P. 2010. Various notes on slur formulae. In *Proceedings of the 13th Czech-Japan Seminar on Data Analysis and Decision Making in Service Science*, 85–95.

Čepek, O.; Kučera, P.; and Vlček, V. 2011. Properties of slur formulae. sent to SOFSEM 2011.

Vlček, V. 2009. Třídy booleovských formulí s efektivně řešitelným satem. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic.