Efficient Reasoning for Inconsistent Horn Formulae

Joao Marques-Silva[1], Alexey Ignatiev[1,4], Carlos Mencía[2], and Rafael Peñaloza[3]

[1] University of Lisbon, Portugal ({jpms,aignatiev}@ciencias.ulisboa.pt)
[2] University of Oviedo, Spain (cmencia@gmail.com)
[3] Free University of Bozen-Bolzano, Italy (rafael.penaloza@unibz.it)
[4] ISDCT SB RAS, Irkutsk, Russia

Abstract. Horn formulae are widely used in different settings that include logic programming, answer set programming, description logics, deductive databases, and system verification, among many others. One concrete example is concept subsumption in lightweight description logics, which can be reduced to inference in propositional Horn formulae. Some problems require one to reason with inconsistent Horn formulae. This is the case when providing minimal explanations of inconsistency. This paper proposes efficient algorithms for a number of decision, function and enumeration problems related with inconsistent Horn formulae. Concretely, the paper develops efficient algorithms for finding and enumerating minimal unsatisfiable subsets (MUSes), minimal correction subsets (MCSes), but also for computing the lean kernel. The paper also shows the practical importance of some of the proposed algorithms.

1 Introduction

Horn formulae have been studied since at least the middle of the past century [18, 19]. More recently, Horn formulae have been used in a number of different settings, which include logic programming, answer set programming and deductive databases, but also description logics. In addition, there is a growing interest on Horn formula reasoning in formal methods [12, 13]. In the area of description logics, there is a tight relationship between description logic reasoning and Horn formulae. This is true for lightweight description logics [2,3,5,23,32,42,44,46], but there exists recent work suggesting the wider application of Horn formulae to (non-lightweight) description logic reasoning [9].

It is well-known that the decision problem for Horn formulae is in P [18], with linear-time algorithms known since the 80s [16, 20, 37]. Nevertheless, other decision, function and enumeration problems are of interest when reasoning about Horn formulae, which find immediate application in other settings, that include description logics. Moreover, related problems on Horn formulae have been studied earlier in other contexts [17, 31]. This paper extends earlier work on developing efficient algorithms for reasoning about Horn formulae [3, 5, 42]. Concretely, the paper investigates the complexity of finding and enumerating MUSes, MCSes, but also the complexity of computing the lean kernel [24–27]. The paper also studies MUS and MCS membership and related problems. In addition, the paper also investigates the practical significance of some of these new algorithms.

The paper is organized as follows. Section 2 introduces the notation used throughout the paper. Section 3 revisits the well-known linear time unit resolution (LTUR) algorithm, and proposes two algorithms used extensively in the remainder of the paper. Section 4 develops the main results in the paper. The practical significance of the work is briefly addressed in Section 5, before concluding in Section 6.

2 Preliminaries

This section introduces the notation and definitions used throughout the paper. We assume that the reader is familiar with the basic notions of propositional logic (see e.g. [11]). CNF formulae are defined over a finite set of propositional variables. A literal is a variable or its complement. A *clause* is a disjunction of literals, also interpreted as a set of literals. A *CNF formula* $\mathcal{F}$ is a finite conjunction of clauses, also interpreted as a finite set of clauses. In some settings, it is convenient to view a CNF formula as a multiset of clauses, where the same clause can appear more than once. The set of variables associated with a CNF formula $\mathcal{F}$ is denoted by $\text{var}(\mathcal{F})$. We often use $X \triangleq \text{var}(\mathcal{F})$, with $n \triangleq |X|$. $m \triangleq |\mathcal{F}|$ represents the number of clauses in the formula, and the number of literal occurrences in $\mathcal{F}$ is represented by $||\mathcal{F}||$. An *assignment* is a mapping from X from $\{0, 1\}$, and total assigments are assumed throughout. Moreover, the semantics of propositional logic is assumed. For a formula $\mathcal{F}$, we write $\mathcal{F} \nvDash \bot$ (resp. $\mathcal{F} \vDash \bot$) to express that $\mathcal{F}$ is satisfiable (resp. unsatisfiable).

In this paper we focus on Horn formulae. Intuitively, Horn formulae are sets of implications of the form $A_1 \wedge A_2 \wedge \ldots \wedge A_{k_j} \rightarrow I_j$, where all A_r are positive literals defined over the variables in X, and I_j is either a positive literal or $\bot$. Formally, a Horn formula $\mathcal{F}$ is a CNF formula where each clause contains at most one positive literal. Clauses without a positive literal are called *goal* clauses, and those with a positive literal are called *definite*. Given a (Horn) clause $c \in \mathcal{F}$, $P(c)$ denotes the set of variables appearing positively in c. For Horn clauses, $P(c)$ always contains at most one element. Likewise, $N(c)$ denotes the set of variables appearing negatively in c. We apply a similar notation for variables v. In this case, $N(v)$ (resp. $P(v)$) denotes the set of clauses where v occurs as a negative (resp. positive) literal.

We are interested in inconsistent formulae $\mathcal{F}$, i.e. $\mathcal{F} \vDash \bot$, such that some clauses in $\mathcal{F}$ can be *relaxed* (i.e. allowed not to be satisfied) to restore consistency, whereas others cannot. Thus, we assume that $\mathcal{F}$ is partitioned into two subformulae $\mathcal{F} = \mathcal{B} \cup \mathcal{R}$, where $\mathcal{R}$ contains the *relaxable* clauses, and $\mathcal{B}$ contains the *non-relaxable* clauses. $\mathcal{B}$ can be viewed as background knowledge, which must always be kept. As we will see in this paper, allowing $\mathcal{B} \neq \emptyset$ can affect the computational complexity and the runtime behavior of the tasks that we consider.

Given an inconsistent CNF formula $\mathcal{F}$, we are interested in detecting the clauses that are responsible for unsatisfiability among those that can be relaxed, as defined next.

Definition 1 (Minimal Unsatisfiable Subset (MUS)). *Let $\mathcal{F} = \mathcal{B} \cup \mathcal{R}$ denote an inconsistent set of clauses ($\mathcal{F} \vDash \bot$). $\mathcal{M} \subseteq \mathcal{R}$ is a* Minimal Unsatisfiable Subset *(MUS) iff $\mathcal{B} \cup \mathcal{M} \vDash \bot$ and $\forall_{\mathcal{M}' \subsetneq \mathcal{M}}$, $\mathcal{B} \cup \mathcal{M}' \nvDash \bot$. $\bigcup \text{MU}(\mathcal{F})$ denotes the union of all MUSes.*

Informally, an MUS provides the minimal information that needs to be added to the background knowledge $\mathcal{B}$ to obtain an inconsistency; thus, it explains the causes for this inconsistency. Alternatively, one might be interested in correcting the formula, removing some clauses to achieve consistency.

Definition 2 (MCS,MSS). *Let $\mathcal{F} = \mathcal{B} \cup \mathcal{R}$ denote an inconsistent set of clauses ($\mathcal{F} \vDash \bot$). $\mathcal{C} \subseteq \mathcal{R}$ is a* Minimal Correction Subset *(MCS) iff $\mathcal{B} \cup \mathcal{R} \setminus \mathcal{C} \nvDash \bot$ and $\forall_{\mathcal{C}' \subsetneq \mathcal{C}}$, $\mathcal{B} \cup \mathcal{R} \setminus \mathcal{C}' \vDash \bot$. We use $\bigcup \mathrm{MC}(\mathcal{F})$ to denote the union of all MCSes. $\mathcal{S} \subseteq \mathcal{R}$ is a* Maximal Satisfiable Subset *(MSS) iff $\mathcal{B} \cup \mathcal{S} \nvDash \bot$ and $\forall_{\mathcal{S}' \supsetneq \mathcal{S}}$, $\mathcal{B} \cup \mathcal{S}' \vDash \bot$.*

It is well known that there is a close connection between MUSes, MCSes, and MSSes. Indeed, it is easy to see that a set $\mathcal{C}$ is an MCS iff $\mathcal{R} \setminus \mathcal{C}$ is an MSS. Moreover, there exists a minimal hitting set duality between MUSes and MCSes [43]. In particular this means that $\bigcup \mathrm{MU}(\mathcal{F}) = \bigcup \mathrm{MC}(\mathcal{F})$.

The lean kernel [24–27] represents an (easier to compute) over-approximation of $\bigcup \mathrm{MU}(\mathcal{F})$, containing all clauses that can be included in a resolution refutation of $\mathcal{F}$, with $\mathcal{F} \vDash \bot$. The lean kernel of a CNF formula is tightly related with the maximum autarky of the formula, one being the complement of the other [24–27]. Computation of the lean kernel for Horn formulae is analyzed in Section 4.3.

For arbitrary CNF formulae, there exists recent work on extracting MUSes [7,10], and on extracting MCSes [6,33,35,36]. The complexity of extracting MUSes for Horn formulae has also been studied in the context of so-called axiom pinpointing for light-weight description logics [5,41,42]. In particular, it has been shown that axiom pinpointing for the $\mathcal{EL}$ family of description logics [4,5] can be reduced to the problem of computing MUSes of a Horn formula with $\mathcal{B} \neq \emptyset$ (see [2,3,44] for details).

We also assume that the reader is familiar with the basic notions of computational complexity; for details, see [22,39,40]. Throughout the paper, the following abbreviations are used. For decision problems [39], NPC stands for NP-complete, and P^{NP} (or Δ_2^{p}) denotes the class of problems that can be decided with a polynomial number of calls (on the problem representation) to an NP oracle. Similarly, $\mathrm{P}^{\mathrm{NP}}[\mathsf{log}]$ denotes the class of problems that can be decided with a logarithmic number of calls to an NP oracle (where n denotes the size of the problem instance). For enumeration problems [22], OP stands for *output polynomial* and PD stands for *polynomial delay*, denoting respectively algorithms that run in time polynomial on the size of the input and already computed solutions (i.e. the output), and algorithms that compute each solution in time polynomial solely on the size of the input. Finally, for function (or search problems), the notation used for characterizing the complexity of decision problems is prefixed with F [39]. For example, $\mathrm{FP}^{\mathrm{NP}}$ denotes the class of function problems solved with a polynomial number of calls to an NP oracle. Similarly, $\mathrm{FP}^{\mathrm{NP}}[\mathsf{log}]$ (respectively $\mathrm{FP}^{\mathrm{NP}}[\mathsf{wit,log}]$) denotes the class of function problems solved with a logarithmic number of calls to an NP oracle (respectively to a witness producing NP oracle [14]).

3 Basic LTUR and Saturation

It is well-known that consistency of Horn formulae can be decided in linear time on the size of the formula [16,20,37]. A simple algorithm that achieves this linear-time behavior on the number of literals appearing in the formula, is known as *linear time unit resolution* (LTUR) [37]. Motivated by the different uses in the remainder of the paper, a possible implementation is analyzed next. In addition, we also introduce an extension

```
Function LTUR ($\mathcal{F}$)
    Input  : $\mathcal{F}$: input Horn formula
    Output: falsified clause, if any; $\alpha$: antecedents
    $(Q, \eta, \gamma, \alpha) \leftarrow$ Initialize($\mathcal{F}$)
    while $Q \neq \emptyset$ do                              // $Q$: queue of variables assigned 1
        $v_j \leftarrow$ ExtractFirstVariable($Q$)
        foreach $c_i \in N(v_j)$ do
            $\eta(c_i) \leftarrow \eta(c_i) - 1$
            if $\eta(c_i) = 0$ then
                if $\gamma(c_i)$ then return $(\{c_i\}, \alpha)$
                $v_r \leftarrow$ PickVariable($P(c_i)$)
                if $\alpha(v_r) = \emptyset$ then
                    AppendToQueue($Q, v_r$)
                    $\alpha(v_r) \leftarrow \{c_i\}$
    return $(\emptyset, \alpha)$
```

Algorithm 1: The LTUR algorithm

of LTUR that saturates the application of unit propagation rules, without affecting the linear-time behavior. We then show how the result of this saturation can be used to trace the causes of all consequences derived by LTUR.

3.1 Linear Time Unit Resolution

LTUR can be viewed as one-sided unit propagation, in the sense that only variables assigned value 1 are propagated. The algorithm starts with all variables assigned value 0, and repeatedly flips variables to 1, one at a time. Let $\eta : \mathcal{F} \to \mathbb{N}_0$ associate a counter with each clause, representing the number of negative literals *not* assigned value 0. Given an assignment, a goal clause $c \in \mathcal{F}$ is falsified if $\eta(c) = 0$. Similarly, a definite clause $c \in \mathcal{F}$ is unit and requires the sole positive literal to be assigned value 1 when $\eta(c) = 0$. LTUR maintains the η counters, propagates assignments due to unit definite clauses and terminates either when a falsified goal clause is identified or when no more definite clauses require that their sole positive literal be assigned value 1. The procedure starts with unit positive clauses c_r for which $\eta(c_r) = 0$. Clearly, a Horn formula without unit positive clauses is trivially satisfied. In the following, $\gamma : \mathcal{F} \to \{0, 1\}$ denotes whether a clause $c \in \mathcal{F}$ is a goal clause, in which case $\gamma(c) = 1$. Finally, $\alpha : \text{var}(\mathcal{F}) \to 2^{\mathcal{F}}$ is a function that assigns to each variable v a set of clauses $\alpha(v)$ that are deemed responsible for assigning value 1 to v. If the value of v is not determined to be 1, then $\alpha(v) = \emptyset$. As is standard in CDCL SAT solving [11], $\alpha(v)$ will be referred to as the *antecedent* (set) of v. The organization of LTUR is summarized in Algorithm 1. The initialization step sets the initial values of the η counters, the α values, and the value of the γ flag. Q is initialized with the variables in the unit positive clauses. For every $v \in Q$, $\alpha(v)$ contains the unit clause v. For all other variables, the value of α is $\emptyset$. Clearly, this initialization runs in time linear on the number of literals. The main loop analyzes the variables assigned value 1 in order. Notice that a variable v is assigned value 1 iff $\alpha(v) \neq \emptyset$. For each variable $v \in Q$, the counter η of the clauses where v occurs as a negative literal is decreased. If the $\eta(c) = 0$ for some clause c, then either the formula is inconsistent, if c is a goal clause, or the positive literal of c is assigned

```
Function LTURs ($\mathcal{F}$)
    Input  : $\mathcal{F}$: input Horn formula
    Output: $\mathcal{U}$: falsified clauses; $\alpha$: antecedent sets
    $(Q, \eta, \gamma, \alpha, \mathcal{U}) \leftarrow$ Initialize($\mathcal{F}$)
    while $Q \neq \emptyset$ do                                  // $Q$: queue of variables assigned 1
        $v_j \leftarrow$ ExtractFirstVariable($Q$)
        foreach $c_i \in N(v_j)$ do
            $\eta(c_i) \leftarrow \eta(c_i) - 1$
            if $\eta(c_i) = 0$ then
                if $\gamma(c_i)$ then
                    $\mathcal{U} \leftarrow \mathcal{U} \cup \{c_i\}$
                else
                    $v_r \leftarrow$ PickVariable($P(c_i)$)
                    if $\alpha(v_r) = \emptyset$ then
                        AppendToQueue($Q, v_r$)
                    $\alpha(v_r) \leftarrow \alpha(v_r) \cup \{c_i\}$
    return $(\mathcal{U}, \alpha)$
```

Algorithm 2: The LTURs algorithm

value 1 and added to Q. The operation of LTUR is such that $|\alpha(v)| \leq 1$ for $v \in \text{var}(\mathcal{F})$. It is easy to see that LTUR runs in linear time on the number of literals of $\mathcal{F}$ [37]: each variable v is analyzed only once, since v is added to the queue Q only if $\alpha(v) = \emptyset$, and after being added to Q, $\alpha(v) \neq \emptyset$ and $\alpha(v)$ will not be set to $\emptyset$ again. Thus, v will not be added to Q more than once. For each variable v, its clauses are analyzed at most once, in case v was added to Q. Thus, the number of times literals are analyzed during the execution of LTUR is $\mathcal{O}(||\mathcal{F}||)$.

It is often convenient to run LTUR incrementally. Given $\mathcal{F} = \mathcal{R} \cup \mathcal{B}$, one can add a unit positive clause at a time, and run LTUR while consistency is preserved. If no inconsistency is identified for any of the unit positive clauses, the total run time is $\mathcal{O}(||\mathcal{F}||)$. In contrast, if inconsistency is identified for some unit positive clause, consistency may be recovered by undoing only the last sequence of variables assigned value 1. Incremental LTUR plays an important role in the algorithms described in Section 4, concretely for MUS and MCS extraction.

3.2 LTUR Saturation

We will often resort to a modified version of LTUR, which we call LTUR *saturation* (LTURs). LTURs also runs in linear time on the number of literals, but exhibits properties that are relevant when analyzing inconsistent Horn formulae. The basic idea is *not* to terminate the execution of LTUR when a falsified goal clause is identified. Instead, the falsified clause is recorded, and the one-sided unit propagation of LTUR continues to be executed. The procedure only terminates when Q is empty. Besides $\mathcal{U}$, in this case the value of $\alpha(v)$ is updated with any clause that can serve to determining the assignment of v to value 1.

Algorithm 2 summarizes the main steps of LTURs. When compared with LTUR, the main difference is the set $\mathcal{U}$, initially set to $\emptyset$, and to which the falsified clauses are added to. Using the same arguments presented in Section 3.1, it can be shown that LTURs also runs in time $\mathcal{O}(||\mathcal{F}||)$.

```
Function TraceClauses ($\mathcal{U}, \alpha$)
    Input  : $\mathcal{U}$: falsified clause(s); $\alpha$: antecedents
    Output: $\mathcal{M}$: traced clauses from $\mathcal{F}$, given $\mathcal{U}$ and $\alpha$
    $(S, \mathcal{M}, \phi) \leftarrow$ Initialize($\mathcal{U}$)
    while not Empty($S$) do
        $c_i \leftarrow$ PopClause($S$)
        foreach $v_r \in N(c_i)$ do
            foreach $c_a \in \alpha(v_r)$ do
                if $\phi(c_a) = 0$ then
                    $\phi(c_a) \leftarrow 1$
                    PushClause($S, c_a$)
                    $\mathcal{M} \leftarrow \mathcal{M} \cup \{c_a\}$
    return $\mathcal{M}$
```

Algorithm 3: Tracing antecedents

Table 1: Summary of results

$\mathcal{B}$?	1 MUS	MUS Enum	1 MCS	MCS Enum	Lean Kernel	$\in$ MUS	$\in$ MCS	$\bigcup$MU, $\bigcup$MC
$\mathcal{B} = \emptyset$	linear	*PD*	linear	PD	linear	*NPC*	*NPC*	FP^{NP}[wit,log]
$\mathcal{B} \neq \emptyset$	*poly*	*not OP*	poly	*not OP*	linear	*NPC*	*NPC*	FP^{NP}[wit,log]

3.3 Tracing Antecedents

Another important step when analyzing inconsistent Horn formulae is to trace antecedents. Algorithm 3 describes an approach for tracing that is based on knowing at least one antecedent for each variable assigned 1. Thus, this method can be used after running LTUR or LTURs. In the algorithm, ϕ is used as a flag to ensure each clause is traced at most once. The stack S is initialized with the clauses in $\mathcal{U}$. Algorithm 3 implements a depth-first traversal of the graph induced by the antecedent sets, starting from the clauses in $\mathcal{U}$. Hence, the algorithm runs in $\mathcal{O}(||\mathcal{F}||)$ time.

4 Efficient Reasoning for Inconsistent Horn Formulae

In this section we analyze a number of computational problems related with the analysis of inconsistent Horn formulae. The main results obtained are summarized in Table 1. Some of the results, depicted with slanted text, are adapted from the literature. We briefly recall these results before presenting our contributions.

An algorithm for enumerating all MUSes of an inconsistent Horn formula without background knowledge can be obtained through a straightforward modification of the method presented in [42]. In the presence of background knowledge (that is, when $\mathcal{B} \neq \emptyset$), it was shown in [5, Theorem 4] that no output polynomial algorithm exists for enumerating all MUSes (unless P $=$ NP). A similar approach was used to prove that MCSes for formulae with background knowledge cannot be enumerated in output polynomial time in [41, Theorem 6.15], unless P $=$ NP. In [42, Theorems 17, 18] is was also shown that deciding MUS membership for a clause is NP-complete. A simple algorithm for finding one MUS requires one inconsistency check for every relaxable clause in the formula [8, 15]. The linear runtime of LTUR [16, 37] guarantees that one MUS can be computed in quadratic time. This upper bound was further refined to $\mathcal{O}(|\mathcal{M}| \cdot ||\mathcal{F}||)$, where $\mathcal{M}$ is the size of the largest MUS, in [3]. In the following we provide more details on these results, and prove the remaining claims from Table 1.

4.1 MUS Extraction and Enumeration

We first focus on the problems related to extracting and enumerating MUSes. As mentioned already, to compute one MUS one can simply perform a linear number of inconsistency tests—one for each clause in $\mathcal{R}$. This yields an overall quadratic behaviour. As we show next, in the absence of background knowledge, one MUS can be extracted in linear time.

Proposition 1 (MUS Extraction, $\mathcal{B} = \emptyset$). *Let $\mathcal{F} \vDash \bot$, with $\mathcal{B} = \emptyset$. One MUS of $\mathcal{F}$ can be computed in time $\mathcal{O}(||\mathcal{F}||)$.*

Proof. Horn formulae are decided by LTUR, that implements (one-sided) unit propagation and runs in $\mathcal{O}(||\mathcal{F}||)$. It is well-known that unsatisfiable subsets computed with unit propagation are MUSes [29, Proposition 1]. Thus, tracing antecedents, starting from the falsified goal clause $c \in \mathcal{F}$ returned by LTUR, yields an MUS of $\mathcal{F}$. Algorithm 3 illustrates an implementation of clause tracing that runs in $\mathcal{O}(||\mathcal{F}||)$. Overall, both LTUR and clause tracing are run once. Hence, an MUS is extracted in time $\mathcal{O}(||\mathcal{F}||)$. $\square$

One important observation is that the polynomial delay enumeration algorithm presented in [42] uses an arbitrary polynomial-time MUS extraction algorithm as a blackbox. Thus, the linear time extraction method presented above can be exploited to enumerate all MUSes more efficiently.

When the formula $\mathcal{F}$ contains background knowledge, MUS extraction becomes more expensive. As shown recently in [3] through an insertion-based algorithm, in this case an MUS can be computed in $\mathcal{O}(|\mathcal{M}| \cdot ||\mathcal{F}||)$ time, where $\mathcal{M}$ is the size of the largest MUS. This is achieved by running LTUR incrementally, allowing the run time of successive consistent runs of LTUR to amortize to $||\mathcal{F}||$. Unfortunately, background knowledge has a more important effect on the enumeration problem. Indeed, as shown in [5], if $\mathcal{B} \neq \emptyset$, it is impossible to enumerate all MUSes in output polynomial time, unless P = NP.

4.2 MCS Extraction and Enumeration

A simple algorithm for computing one MCS consists in running LTUR first over all the clauses in $\mathcal{B}$, and then incrementally adding each clause c in $\mathcal{R} = \mathcal{F} \setminus \mathcal{B}$ to the execution of LTUR. As soon as LTUR detects an inconsistency, the latest clause c inserted is known to belong to an MCS and is added to a set $\mathcal{C}$; this clause is retracted, and the process continues. When all clauses have been tested, $\mathcal{C}$ contains an MCS, and its complement is an MSS. Overall, by running LTUR incrementally, the consistent calls to LTUR amortize to $||\mathcal{F}||$. For the inconsistent calls to LTUR, i.e. those producing clauses added to $\mathcal{C}$, one needs to undo the incremental run of LTUR, which in the worst-case runs in $||\mathcal{F}||$. This process needs to be executed once for each clause in $\mathcal{C}$. Taking into account that the (amortized) running time of LTUR is $\mathcal{O}(||\mathcal{F}||)$, an MCS can be computed in time $\mathcal{O}(|\mathcal{C}| \cdot ||\mathcal{F}||)$, where $\mathcal{C}$ is the size of the largest MCS in $\mathcal{F}$.

As it was the case for MUSes, the MCS extraction procedure can be improved to run in linear time in the case where $\mathcal{B} = \emptyset$. Consider again the execution of LTURs (see Al-

```
Function MCSEnum ($\mathcal{F}, \tau$)
  Global : $\mathbb{M}$: MCS register
  Input  : $\mathcal{F}$: input Horn formula; $\tau$: clause tags
1 $(\mathcal{U}, \alpha) \leftarrow$ LTURs($\mathcal{F}$)          // Use LTUR saturation to find MCS candidate
2 if $\mathcal{U} = \emptyset$ then return
3 $(\mathcal{C}, \mathcal{V}) \leftarrow$ PickVariables($\mathcal{U}$)     // $\mathcal{V}$: variables of negative literals in $\mathcal{U}$
4 if MCSRegistered($\mathcal{C}, \mathbb{M}$) then return
5 RegisterMCS($\mathcal{C}, \mathbb{M}$)          // Record computed MCS in MCS register
6 foreach $v \in \mathcal{V}$ do
7     $\mathcal{W} \leftarrow$ DropClauses($\mathcal{F}, \neg v$)        // Drop clauses with literal $\neg v$
      // Next, drop literal $v$ in clauses of $\mathcal{W}$ and tag clauses
8     $(\mathcal{W}, \tau) \leftarrow$ DropLitsTagCls($\mathcal{W}, \tau, v$)
9     MCSEnum($\mathcal{W}, \tau$)          // Recursive call of MCS enumeration
```

Algorithm 4: MCS enumeration with polynomial delay

gorithm 2). Since $\mathcal{B} = \emptyset$, any clause can be included in an MCS. This observation yields the following result.

Proposition 2. *Given $\mathcal{F}$, with $\mathcal{F} \vDash \bot$, the set $\mathcal{U}$ computed by LTURs is an MCS of $\mathcal{F}$.*

Proof. First, we show that $\mathcal{U}$ is a correction set. Observe that $\mathcal{U}$ is composed of goal clauses. Goal clauses do not serve to propagate other variables to value 1 and do not serve to prevent variables from being assigned value 1 when running LTURs. Thus, removing these clauses will not elicit further propagation of variables to value 1. Since these are the only falsified clauses, if the clauses in $\mathcal{U}$ are removed, what remains is satisfiable; hence $\mathcal{U}$ is a correction set. To show that $\mathcal{U}$ is minimal, observe that if any clause in $\mathcal{U}$ is not removed, then it will remain unsatisfied, again because removing clauses in $\mathcal{U}$ does not alter the variables assigned value 1. Thus, $\mathcal{U}$ is a correction set for $\mathcal{F}$ and it is minimal, and so it is an MCS for $\mathcal{F}$. □

Corollary 1. *LTURs computes an MCS in linear time on the number of literals.*

In different settings, enumeration of MUSes and MCSes is paramount [5, 42, 44]. We can use the linear time algorithm for MCS extraction to develop a polynomial delay algorithm for enumerating MCSes of a Horn formula when $\mathcal{B} = \emptyset$. We know how to compute one MCS in linear time, by applying LTURs and falsifying goal clauses. The question is then how to iterate the computation of MCSes. The approach we follow is to transform the formula, so that different sets of falsified goal clauses are obtained. Given a goal clause c with a negative literal ℓ on variable v, the transformation is to remove the clauses with literal $\neg v$, and remove the literal v from each clause c' containing that literal. The resulting clause $c_r = c' \setminus \{v\}$ becomes a goal clause. The newly created goal clause is *tagged* with the variable v. As the formula is transformed, $\tau(c)$ indicates whether c is tagged, taking value $\bot$ if not, or being assigned some variable otherwise. As a result, when LTURs is used to find a set of falsified goal clauses, the computed MCSes must also account for the set of tags associated with these falsified goal clauses.

Algorithm 4 summarizes the main steps of the MCS enumeration algorithm with polynomial delay. At each step, LTURs is used to find a set of falsified *goal* clauses

Table 2: MCS enumeration example

Formula $\mathcal{F}$	Depth	False Clauses	MCS	Literals	Picked Literal
$\mathcal{F}_1 \triangleq \{(x_1), (\neg x_1), (x_2), (\neg x_2)\}$	0	$\{(\neg x_1), (\neg x_2)\}$	$\{(\neg x_1), (\neg x_2)\}$	$\{x_1, x_2\}$	x_1
$\mathcal{F}_2 \triangleq \{(), (x_2), (\neg x_2)\}$	1	$\{(), (\neg x_2)\}$	$\{(x_1), (\neg x_2)\}$	$\{x_2\}$	x_2
$\mathcal{F}_3 \triangleq \{(), ()\}$	2	$\{(), ()\}$	$\{(x_1), (x_2)\}$	$\emptyset$	–
$\mathcal{F}_1$	0	–	–	$\{x_1, x_2\}$	x_2
$\mathcal{F}_4 \triangleq \{(x_1), (\neg x_1), ()\}$	1	$\{(\neg x_1), ()\}$	$\{(\neg x_1), (x_2)\}$	$\{x_1\}$	x_1
$\mathcal{F}_5 \triangleq \{(), ()\}$	2	$\{(), ()\}$	~~$\{(x_1), (x_2)\}$~~	$\emptyset$	–

which, together with the tag associated with each falsified goal clause (if any), represents an MCS for $\mathcal{F}$. $\mathcal{V}$ represents the negative literals in falsified goal clauses, and will be used to create additional subproblems. In contrast, $\mathcal{C}$ represents the literals in $\mathcal{V}$ and also includes the variables in the tags of the falsified goal clauses. If the computed MCS has already been seen before, then the algorithm returns without further action. Otherwise, the MCS has not been computed before, and it is recorded in a global register of MCSes $\mathbb{M}$. This MCS is then used to generate a number of modified formulas, from which additional MCSes of the original $\mathcal{F}$ can be computed. Observe that line 7 and line 8 in Algorithm 4 can be viewed as *forcing* some variable to be assigned value 0, hence blocking the one-sided unit propagation of LTUR.

Example 1. Consider the formula: $\mathcal{F} \triangleq \{(x_1), (\neg x_1), (x_2), (\neg x_2)\}$. Table 2 summarizes the execution of Algorithm 4. For each recursion depth (0, 1 or 2), the literal picked is the literal selected in line 6 of Algorithm 4. Given the falsified clauses identified by LTURs, the actual MCS is obtained from these clauses, each augmented with its own tagging literal, if any.

Proposition 3. *Algorithm 4 is sound and complete; i.e., it computes all MCSes.*

Proof (Sketch). Algorithm 4 iterates the enumeration of MCSes by selecting some variable v assigned value 1 by LTURs (see line 6), and then changing the formula such that the variable is removed. This step corresponds to replacing the variable v with value 0, or alternatively by forcing v to be assigned value 0. This process changes the set of falsified clauses. These observations enable us to prove soundness and completeness.
Soundness. The proof is by induction on i, the number of variables forced to be assigned value 0 in, which uniquely identify each MCS. For the base case ($i = 1$), consider the MCS associated with no variables yet forced to be assigned value 0. Force some 1-valued variable in the MCS to be assigned value 0. Re-run LTURs. The set of falsified clauses is an MCS, provided we augment the falsified goal clauses with their tag variable. For the inductive step, suppose we have i variables forced to be assigned value 0 and some current MCS. Force some other 1-valued variable in the MCS to be assigned value 0. Run LTURs. Again, the set of falsified clauses is an MCS.
Completeness. Consider the operation of moving from one MCS to another, obtained from assigning some variables to value 0. This consists in reverting some unit propagation step, where a variable was assigned value 1 and is now forced to be assigned

value 0. The algorithm reverts unit propagation steps in order, creating a search tree. Each node in this search tree represents one MCS and is expanded into k children. Each child node is associated with one of the 1-valued literals in the MCS to be forced to be assigned value 0. Thus, Algorithm 4 will enumerate all subsets of variables assigned value 1, which lead to a conflict being identified, and so all MCSes are enumerated. □

Proposition 4. *Algorithm 4 enumerates the MCSes of an inconsistent Horn formula $\mathcal{F}$ with polynomial delay.*

Proof (Sketch). At each iteration, Algorithm 4 runs LTURs in linear time, and transforms the current working formula, also a linear time operation, once for each literal in the target set of literals. The algorithm must check whether the MCS has already been computed, which can be done in time logarithmic on the number of MCSes stored in the MCS registry, e.g. by representing the registry with a balanced search tree. The additional work done by Algorithm 4 in between computed MCSes is polynomial on the formula size. The iterations that do not produce an MCS are bounded by a polynomial as follows. An MCS can be repeated from a recursive call, but only after one new MCS is computed. Each new MCS can recursively call Algorithm 4 $\mathcal{O}(|\mathcal{F}|)$ times, in the worst-case each call leading to an MCS not being computed. Thus, the overall cost of recursive calls to Algorithm 4 leading to an MCS not being computed is polynomial. Therefore, Algorithm 4 computes MCSes of an inconsistent Horn formula with polynomial delay. □

4.3 Finding the Lean Kernel

The lean kernel is the set of all clauses that can be used in some resolution refutation of a propositional formula, and has been shown to be tightly related with the concept of maximum autarky [24–26]. Autarkies were proposed in the mid 80s [38] with the purpose of devising an algorithm for satisfiability requiring less than 2^n steps. Later work revealed the importance of autarkies when analyzing inconsistent formulas [24, 25, 27, 34]. Indeed, the lean kernel represents an over-approximation of $\bigcup$MU [24–27], that is in general easier to compute for arbitrary CNF formulae. As shown in this section, the same holds true for Horn formulae.

Example 2. Consider the Horn formula: $\mathcal{F} \triangleq \{(a), (\neg a \lor b), (\neg b \lor x), (\neg x \lor b), (\neg b \lor c), (\neg c), (\neg b \lor d)\}$. It is easy to see that the lean kernel of $\mathcal{F}$ is: $\mathcal{K} \triangleq \{(a), (\neg a \lor b), (\neg b \lor x), (\neg x \lor b), (\neg b \lor c), (\neg c)\}$. Indeed, there exists a resolution proof that resolves on x once and on b twice in addition to resolving on a and c. On the other hand, $\mathcal{F}$ has only one MUS, and hence $\bigcup$MU is given by: $\mathcal{U} \triangleq \{(a), (\neg a \lor b), (\neg b \lor c), (\neg c)\}$.

The most efficient practical algorithms for computing the maximum autarky, and by extension the lean kernel, exploit intrinsic properties of maximum autarkies and reduce the problem to computing one MCS [34]. Other recently proposed algorithms require asymptotically fewer calls in the worst case [28]. The reduction of maximum autarky to the problem of computing one MCS involves calling a SAT solver on an arbitrary CNF formula a logarithmic number of times, in the worst-case. In contrast, it is possible to obtain a polynomial (non-linear) time algorithm by exploiting LTUR [37] and

```
Function LeanKernel ($\mathcal{F}$)
    Input  : $\mathcal{F}$: input Horn formula
    Output: $\mathcal{K}$: lean kernel of $\mathcal{F}$
1   $(\mathcal{U}, \alpha) \leftarrow$ LTURs($\mathcal{F}$)                          // See Algorithm 2
2   $\mathcal{K} \leftarrow$ TraceAntecedents($\mathcal{U}, \alpha$)              // See Algorithm 3
3   return $\mathcal{K}$
```

Algorithm 5: Computing the Lean Kernel

the maximum autarky extraction algorithm based on the iterative removal of resolution refutations [26]. This simple polynomial time algorithm can be further refined to achieve a linear time runtime behavior for computing the lean kernel for Horn formulae, even in the presence of background knowledge, as shown next.

Algorithm 5 exploits LTUR saturation for computing a set of clauses $\mathcal{K}$ that corresponds to the lean kernel. The algorithm simply traverses all possible antecedents, starting from the falsified clauses until the unit positive clauses are reached. The set of all traced clauses corresponds to the lean kernel, as they are all clauses that can appear in a resolution refutation, by construction (see Proposition 5). Notice that the correctness of this algorithm does not depend on the presence or absence of background knowledge. Thus, the lean kernel can be computed in linear time also for formulas with $\mathcal{B} \neq \emptyset$.

Example 3. Consider again the formula $\mathcal{F}$ from Example 2. After applying LTURs to $\mathcal{F}$, we obtain the antecedent sets of all activated variables. Observe that, among others, $(\neg b \vee x)$ and $(\neg x \vee b)$ are antecedents of x and b, respectively. After tracing the antecedents of the falsified clauses, we obtain the set of clauses $\mathcal{U}$.

Proposition 5. *Algorithm 5 computes the lean kernel of the input Horn formula $\mathcal{F}$.*

Proof. Recall that LTURs only assigns the value 1 to a variable v when this is necessary to satisfy some clause in $\mathcal{F}$. In order to trace the causes for inconsistency, all such clauses are stored by LTURs as antecedents for the variable activation. Thus, for every clause c in $\alpha(v)$ there exists a proof derivation for the assignment of 1 to v that uses this clause c. Thus all the traced clauses from $\mathcal{F}$ given the falsified clauses and α appear in some resolution refutation of $\mathcal{F}$; that is, they belong to the lean kernel.

Conversely, notice that resolving two Horn clauses yields a new Horn clause. Moreover, the number of variables in a clause can only be reduced by resolving with a clause containing only a single (positive) variable. Thus, every resolution refutation of $\mathcal{F}$ can be transformed into a sequence of steps of LTUR leading to a conflict, with the resolving clauses appearing as antecedents for each activation. In particular, all clauses in the lean kernel are found while tracing antecedents of falsified clauses. □

4.4 MUS and MCS Membership

As stated already, the lean kernel is an over-approximation of $\bigcup$MU [25] that can be computed in linear time. In contrast, finding the precise set $\bigcup$MU is significantly harder. In fact, deciding whether a given clause c belongs to $\bigcup$MU is NP-complete, even if $\mathcal{B} = \emptyset$.

Definition 3 (MUS/MCS Membership). *Let $\mathcal{F}$ be a formula and $c \in \mathcal{F} \setminus \mathcal{B}$. The* MUS membership problem *is to decide whether there exists an MUS $\mathcal{M}$ of $\mathcal{F}$ such that*

$c \in \mathcal{M}$. *The* MCS membership problem *is to decide whether there exists an MCS $\mathcal{C}$ of $\mathcal{F}$ such that $c \in \mathcal{C}$.*

It was previously shown that MUS membership is a computationally hard problem. Indeed, for Horn formulae this problem is already NP-complete [42, Theorems 17, 18], and for arbitrary CNF formulae, its complexity increases to Σ_2^p-complete [30].

Interestingly, the hitting set duality between MUSes and MCSes [43] implies that a clause is in some MCS if and only if it is in some MUS. In other words, the identity $\bigcup \mathrm{MU}(\mathcal{F}) = \bigcup \mathrm{MC}(\mathcal{F})$ holds. From this fact, it automatically follows that MCS membership is also NP-complete [41, Theorem 6.5].

Proposition 6. *The MCS membership problem is* NP*-complete.*

Through the non-deterministic algorithm that decides MUS membership, it is then possible to prove, using the techniques from [21] that $\bigcup \mathrm{MU}$ can be computed through logarithmically many calls to a witness oracle.

Proposition 7. $\bigcup \mathrm{MU}(\mathcal{F})$ *is in* $\mathrm{FP}^{\mathrm{NP}}[\mathsf{wit,log}]$.

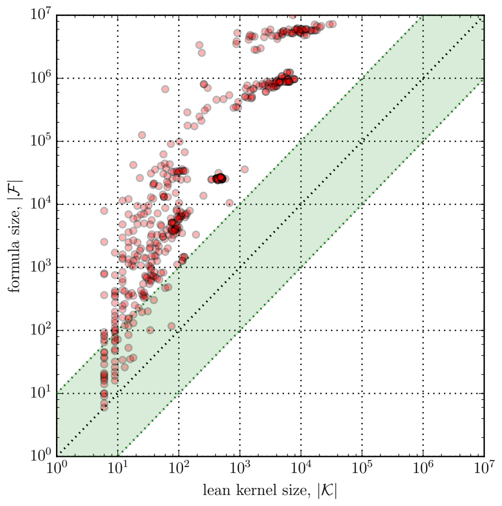
Proof (Sketch). Notice first that $\bigcup \mathrm{MU}(\mathcal{F})$ can be computed through a linear number of parallel queries to an NP oracle. More precisely, for every clause c in $\mathcal{F}$ we decide the MUS membership problem for c. As shown in [21, Remark 4], these parallel queries can be replaced by a logarithmic number of calls to a witness oracle. It then follows that $\bigcup \mathrm{MU}(\mathcal{F})$ is in $\mathrm{FP}^{\mathrm{NP}}[\mathsf{wit,log}]$. □

Remark 1. Since $\bigcup \mathrm{MU}(\mathcal{F}) = \bigcup \mathrm{MC}(\mathcal{F})$, then $\bigcup \mathrm{MC}(\mathcal{F})$ is also in $\mathrm{FP}^{\mathrm{NP}}[\mathsf{wit,log}]$.

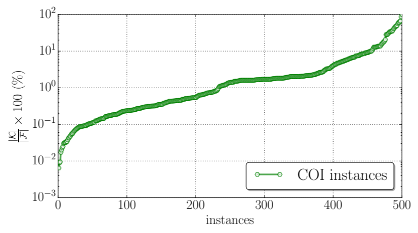
5 Experimental Results

To illustrate the importance of the algorithms developed in Section 4, we investigate the size of the lean kernel (see Algorithm 5) for 1000 unsatisfiable Horn formulae that encode axiom pinpointing problems in the description logic $\mathcal{EL}^+$ using the encoding from [44–46]. These instances were used in [3] as benchmarks for enumerating MUSes of Horn formulae (with $\mathcal{B} \neq \emptyset$). There are two kinds of instances that correspond to two different reduction techniques proposed in [45, 46], namely COI and the more effective x2 optimization. The experiments include 500 instances of each kind. Earlier work [3, 45, 46] showed that the size of the formulae has a great impact in the efficiency of MUS enumeration, being the COI instances (much) harder to solve than the x2 instances.

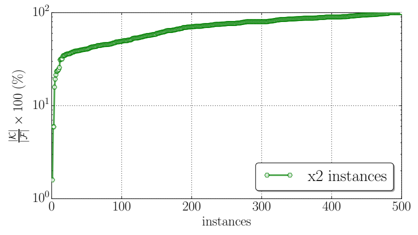
We computed the lean kernel of each of the 1000 test formulae and compared the size of the result with the size of the original formula. The results of these comparisons are depicted in Figure 1. The scatter plot in Figure 1a summarizes the formulae reductions for the COI instances. The cactus plots summarize the formulae reductions for the COI and x2 instances, respectively. As can be observed, for the COI instances with more than 100 clauses, the size of the lean kernel (i.e. an over-approximation of the clauses that are relevant for computing MUSes and MCSes) is often around 1% (or less) of the original formula. Figure 1b confirms that for around 50% of the instances, the lean kernel size is less than 1% of the original problem instance size. In other words, the lean kernel is at least two orders of magnitude smaller than the input formula in



(a) Scatter plot for COI instances



(b) Cactus plot for COI instances



(c) Cactus plot for x2 instances

Fig. 1: Formula reductions for COI and x2 instances

most of these cases. In practical terms, this means that for many of the Horn formulae used for axiom pinpointing in the recent past based on COI reduction, around more than 99% of the clauses are irrelevant for the computation of MUSes and MCSes. The results obtained for the x2 instances (Figure 1c) are not as dramatic. This was expected as the x2 reduction is more effective in removing irrelevant clauses from the formula. However, even in this case the lean kernel is strictly smaller than the input formula in all but 19 instances. From these 19 instances, one has 25 clauses, and all others contain 6 clauses; thus, it is not surprising that no reduction was achieved through the lean kernel. Interestingly, about half of the instances observed a reduction of over 20%, and in some extreme cases the size of the formula was reduced in more than 90%. To the best of our knowledge, these are the first practical problem instances for which the size of the maximum autarky (i.e., the complement of the lean kernel) is non-negligible.

6 Conclusions

We have developed several new results related to reasoning about inconsistent Horn formulae. These results complement earlier work [3, 5, 42], and find application in a number of settings, including axiom pinpointing of lightweight description logics. In particular, we presented a polynomial delay algorithm for enumerating all MCSes, and a linear-time method for computing the lean kernel of a formula.

We illustrate the relevance of our work by analyzing Horn formulae that encode axiom pinpointing problems in the description logic $\mathcal{EL}^+$. The experimental results show that commonly used Horn formulae [2, 3, 44] contain a very large proportion of irrelevant clauses, i.e. clauses that do not interfere with consistency. With the exception of a few outliers related with very small formulae, most formulae have around 99% of irrelevant clauses, which can be identified with a linear time algorithm. From a practical perspective, a natural step is to exploit the linear time lean kernel identification in state of the art axiom pinpointing tools [1–3, 32] and other problems where MUS enumeration and membership are important.

References

1. M. F. Arif, C. Mencía, A. Ignatiev, N. Manthey, R. Peñaloza, and J. Marques-Silva. BEACON: an efficient sat-based tool for debugging $\mathcal{EL}^+$ ontologies. In *SAT*, pages 521–530, 2016.
2. M. F. Arif, C. Mencía, and J. Marques-Silva. Efficient axiom pinpointing with EL2MCS. In *KI*, pages 225–233, 2015.
3. M. F. Arif, C. Mencía, and J. Marques-Silva. Efficient MUS enumeration of Horn formulae with applications to axiom pinpointing. In *SAT*, pages 324–342, 2015.
4. F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *IJCAI*, pages 364–369, 2005.
5. F. Baader, R. Peñaloza, and B. Suntisrivaraporn. Pinpointing in the description logic $\mathcal{EL}^+$. In *KI*, pages 52–67, 2007.
6. F. Bacchus, J. Davies, M. Tsimpoukelli, and G. Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841, 2014.
7. F. Bacchus and G. Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV*, pages 70–86, 2015.
8. R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, pages 276–281, 1993.
9. A. Bate, B. Motik, B. C. Grau, F. Simancik, and I. Horrocks. Extending consequence-based reasoning to SRIQ. In *KR*, pages 187–196, 2016.
10. A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.
11. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
12. N. Bjørner, F. Fioravanti, A. Rybalchenko, and V. Senni, editors. *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014*, volume 169 of *EPTCS*, 2014.
13. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *FoLC II*, pages 24–51, 2015.
14. S. R. Buss, J. Krajíček, and G. Takeuti. Provably total functions in the bounded arithmetic theories R^i_3, U^i_2, and V^i_2. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory, and Computational Complexity*, pages 116–161. OUP, 1995.
15. J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.
16. W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.
17. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995.
18. L. J. Henschen and L. Wos. Unit refutations and Horn sets. *J. ACM*, 21(4):590–605, 1974.
19. A. Horn. On sentences which are true of direct unions of algebras. *J. Symb. Log.*, 16(1):14–21, 1951.
20. A. Itai and J. A. Makowsky. Unification as a complexity measure for logic programming. *J. Log. Program.*, 4(2):105–117, 1987.
21. M. Janota and J. Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233:73–83, 2016.
22. D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

23. Y. Kazakov, M. Krötzsch, and F. Simancik. The incredible ELK - from polynomial procedures to efficient reasoning with ontologies. *J. Autom. Reasoning*, 53(1):1–61, 2014.
24. H. Kleine Büning and O. Kullmann. Minimal unsatisfiability and autarkies. In Biere et al. [11], pages 339–401.
25. O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.
26. O. Kullmann. On the use of autarkies for satisfiability decision. *Electronic Notes in Discrete Mathematics*, 9:231–253, 2001.
27. O. Kullmann, I. Lynce, and J. Marques-Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In *SAT*, pages 22–35, 2006.
28. O. Kullmann and J. Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, pages 138–155, 2015.
29. C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4):456–484, 2010.
30. P. Liberatore. Redundancy in logic I: CNF propositional formulae. *Artif. Intell.*, 163(2):203–232, 2005.
31. P. Liberatore. Redundancy in logic II: 2CNF and Horn propositional formulae. *Artif. Intell.*, 172(2-3):265–299, 2008.
32. N. Manthey, R. Peñaloza, and S. Rudolph. Efficient axiom pinpointing in $\mathcal{EL}$ using SAT technology. In *DL*, 2016.
33. J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622, 2013.
34. J. Marques-Silva, A. Ignatiev, A. Morgado, V. M. Manquinho, and I. Lynce. Efficient autarkies. In *ECAI*, pages 603–608, 2014.
35. C. Mencía, A. Ignatiev, A. Previti, and J. Marques-Silva. MCS extraction with sublinear oracle queries. In *SAT*, pages 342–360, 2016.
36. C. Mencía, A. Previti, and J. Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979, 2015.
37. M. Minoux. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Inf. Process. Lett.*, 29(1):1–12, 1988.
38. B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.
39. C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.
40. C. H. Papadimitriou. NP-completeness: A retrospective. In *ICALP*, pages 2–6, 1997.
41. R. Peñaloza. *Axiom-Pinpointing in Description Logics and Beyond*. PhD thesis, Dresden University of Technology, Germany, 2009.
42. R. Peñaloza and B. Sertkaya. On the complexity of axiom pinpointing in the $\mathcal{EL}$ family of description logics. In *KR*, 2010.
43. R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
44. R. Sebastiani and M. Vescovi. Axiom pinpointing in lightweight description logics via Horn-SAT encoding and conflict analysis. In *CADE*, pages 84–99, 2009.
45. R. Sebastiani and M. Vescovi. Axiom pinpointing in large $\mathcal{EL}^+$ ontologies via SAT and SMT techniques. Technical Report DISI-15-010, DISI, University of Trento, Italy, April 2015. Available as `http://disi.unitn.it/~rseba/elsat/elsat_techrep.pdf`.
46. M. Vescovi. *Exploiting SAT and SMT Techniques for Automated Reasoning and Ontology Manipulation in Description Logics*. PhD thesis, University of Trento, 2011.