Interactive Extraction of Minimal Unsatisfiable Cores Enhanced By Meta Learning

Lehrstuhl Algorithmik
Wilhelm-Schickard-Institut für Informatik
Eberhard Karls Universität Tübingen

Johannes Dellert

Thesis submitted for the degree of
Informatik Diplom

First Examiner:
Prof. Dr. Michael Kaufmann

Supervisors:
MSc Bioinform. Christian Zielke
Dipl.-Inform. Stephan Kottler

Tübingen, January 2013

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbständig und nur mit den angegebenen Quellen und Hilfsmitteln (einschließlich des WWW und anderer elektronischer Quellen) angefertigt habe. Alle Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht.

(Johannes Dellert)

Acknowledgements

First and foremost, I would like to express my gratitude to my main supervisor Christian for the encouragement during the many difficult phases and setbacks in the course of the evolution of the ideas for this thesis, and his careful reading of various chapter drafts. Many little errors would have slipped my attention without his precise thought and keen eye. The inevitable errors that remain are of course entirely mine. I am also indebted to Christian for providing me with new extensions to MiniSat whenever I needed one, along with swift support, in particular the last-minute fix of a bug that would almost have prevented me from running my benchmarks.

As the main supervisor in the initial phase of my work on this thesis, Stephan Kottler has done a lot to channel my ambitions onto fertile grounds. Some of the initial ideas fleshed out in this work arose as results of the very interesting discussions we had.

Further thanks go to Kilian for the years we spent working together on the Kahina system, which has once again proven to be a surprisingly versatile, yet stable environment for rapid prototyping and testing out interesting ideas.

I am also indebted to Martin Lahl for providing me with his implementation of deletion-based MUS extraction as a starting point for my own code, and thoroughly answering all the questions I had to get me started with the practical aspects of MUS extraction.

Thanks are also due to my friends who from time to time forced me out of my cocoon for a refreshing and invigorating walk in the woods, and to my family for providing the stable and secure environment in which I have been able to cultivate my academic interests.

Finally, I would like to thank my wife Anna for her love and allround support throughout the past two exhausting years of preparing for final exams and writing two theses. I shall be happy to return the favour now that she has the more stressful job.

Contents

Chapter 1

Introduction

In many areas of technology and science, relevant questions can be conceived as the search for solutions to sets of formal constraints. The applications for which formal methods are used are manifold, ranging from formal verification and automated configuration of hardware or software to planning problems and other areas of artificial intelligence.

Among the large variety of possible formalisms for expressing such constraints, propositional logic is the most basic and at the same time one of the most successful ones. Informally, propositional logic talks about atomic facts which can be either true or false. A formula of propositional logic models logical relations between such facts, and can therefore be seen as a constraint over possible combinations of truth values for these facts. The question whether there is a combination of atomic facts for which a propositional formula holds is called the satisfiability (SAT) problem. The SAT problem is of central importance to many branches of computer science as the prototypical NP-complete problem to which many other hard combinatorical problems can be reduced. While NP-completeness implies that there is probably no polynomial-time algorithm which is able to solve every instance of the SAT problem, modern SAT solvers can solve many large theories as they occur in applications, even some that express constraints over millions of facts.

While stating application problems directly in propositional logic can be cumbersome, many more expressive types of constraints can be encoded in and solved as SAT instances. As a result, many tools for automated reasoning about more complex types of constraints build on today's highly developed SAT solving technology as their computational backbone, further increasing the central importance of SAT solving for the field of formal methods.

A currently very active branch of research in formal methods focuses on the case where sets of constraints are unsatisfiable, i.e. when it can be proven by formal means that no solution exists. In many applications, such unsatisfiable constraint sets arise from design errors which we wish to detect. A fruitful approach to gaining information about such errors is to try to extract from a set of conflicting constraints some sort of minimal explanation that explains which constraints or which interactions between constraints are problematic.

For the case of propositional satisfiability, a straightforward possibility is to define the desired minimal explanations as Minimal Unsatisfiable Subsets (MUSes) of a constraint set. One of the standard approaches to extracting MUSes starts with some unsatisfiable subset (US) and tries to remove constraints until no constraint can be removed any longer without making the subset satisfiable.

This thesis explores how this process of MUS extraction, which has so far only been implemented in command-line tools without any interface for user interaction, can be made interactive. A range of additional techniques for processing and displaying the relevant information about a MUS extraction process is developed in this context. A meta learning technique is developed for reusing information derived from previous reduction attempts in order to provide only reduction options that will actually lead to new information.

The thesis is divided into six chapters. It begins by presenting the central concepts and the state of the art in MUS extraction in Chapter 2. The exposition starts with the basic definitions for propositional logic, then discusses different definitions of minimal unsatisfiable cores and presents the most recent algorithms for MUS extraction. The last section of the chapter introduces some relevant advanced concepts that some of the work in this thesis relies on.

Chapter 3 then motivates the paradigm of interactive MUS extraction, explaining the central ideas and describing the core elements of a prototype implementation. The prototype is centered around an explicit visualization of the explored parts of the search space and allows to select any point in it as a starting point for US reduction operations. The system architecture also supports the parallel execution of automated search agents on a common search space, and includes a plug-in interface for custom reduction heuristics.

In Chapter 4, we introduce a meta instance which stores the determined connections between the presence or absence of individual clauses in USes. The search space for the MUS extraction problem is analysed under the aspect of reducibility knowledge that can be shared between derived USes. A general scheme for extracting and distributing this information is developed, which is partially implemented in the prototype. The chapter concludes with an outlook on possible further applications of the meta instance concept.

Chapter 5 sets out to find efficient compression schemes for large meta instances, and the two block inference mechanisms it develops for this purpose turn out to infer and represent relevant additional knowledge about MUS structures. The block definition schemes are also examined as generalizations of the meta instance concepts which yield natural extensions of the prototype for interactively extracting other types of minimal unsatisfiable cores.

In Chapter 6, an attempt is made to evaluate the concept of interactive MUS extraction on industrial data. The unavailability of suitable test sets leads to the development of a SAT encoding for the debugging of context-free grammars in natural language processing. The instances thus generated turn out to have very interesting properties that set them apart from other benchmark sets and potentially make them a valuable contribution to the SAT community. At the same time, these properties limit the validity of the observations made using the interactive MUS extraction prototype for other applications.

A final chapter summarizes the theoretical and practical results of this work, describes the current state of the prototype implementation, and presents possible directions for future research about interactive MUS extraction.

Chapter 2

Preliminaries

This chapter lays the theoretical and technical foundations for the work presented in this thesis. It also contains the bigger part of the discussion of previous work. Starting with a concise recapitulation of basic facts about propositional logic and SAT solving, further sections present relevant parts of the theory behind minimal unsatisfiable cores as well as basic algorithms for their extraction. The last section presents a few more advanced concepts and current results from the field of MUS extraction which are also used in this work.

2.1 Propositional Logic

This section contains definitions of the core concepts as well as some basic properties of propositional logic, to the extent that will be needed in the following chapters. Since some familiarity with the basic concepts of logic must be presupposed, the main purpose of this part is to introduce the notational conventions adopted in this thesis.

2.1.1 Syntax and Semantics

For any formal method, we first need to formally specify the syntax and the semantics of the logical formalism we build upon. The **syntax** defines how expressions of our logical language are structured, and the **semantics** is a formal description of the way in which such expressions are evaluated with respect to our domain of interest.

To define the syntax of propositional logic, we first need a countably infinite set of **variables** V, whose elements we will normally denote by $v, v_1, v_2, \ldots$. To this we add two **constant** symbols $\top$ and $\bot$ and a set of **operators** symbols $\{\neg, \wedge \text{ and } \vee\}$. $\neg$ is called **negation**, $\wedge$ **conjunction**, and $\vee$ represents **disjunction**. To indicate the order of execution of these operators, we will furthermore need **bracket** symbols (and). With these symbols in place, we define the set of **formulae** of propositional logic as follows:

Definition 2.1.1. *(**Syntax of Propositional Logic**)*

- *The constants $\top$ and $\bot$ are formulae.*
- *Each variable symbol $v \in V$ is a formula.*
- *For each formula ϕ, $\neg\phi$ is a formula.*
- *For formulae ϕ and ψ, $(\phi \wedge \psi)$ and $(\phi \vee \psi)$ are formulae.*
- *Nothing else is a formula.*

Variables and constants are also called **atoms**. A **literal** is an atom or an atom preceded by the negation symbol $\neg$. Literals will usually have the names $l_1, l_2, \ldots$, whereas the Greek letters ϕ and ψ, often with subscripts, will be used for formulae. We will generally handle brackets liberally, based on the conventional operator precedence ordering $\neg \prec \wedge \prec \vee$. For instance, we will interpret a formula $\phi_1 \vee \neg\phi_2 \wedge \phi_3$ as $\phi_1 \vee ((\neg\phi_2) \wedge \phi_3)$.

The semantics of propositional logic is defined relative to an **assignment** $\vartheta : V \to \{0, 1\}$ of variables to the numbers 0 and 1, where 0 represents the **truth value** "false", and 1 "true". We will encounter both partial and complete assignments in this thesis. The truth value of a formula ϕ under a complete assignment ϑ is defined recursively via an evaluation function $eval(\phi, \vartheta)$:

Definition 2.1.2. *(**Semantics of Propositional Logic**)*
*For a complete assignment ϑ and formulae ϕ, ψ, we define the **evaluation** function eval by*

- $eval(\bot, \vartheta) := 0$ *and* $eval(\top, \vartheta) := 1$
- $\forall v \in V : eval(v, \vartheta) := \vartheta(v)$
- $eval(\neg\phi, \vartheta) := 1 - eval(\phi, \vartheta)$
- $eval(\phi \wedge \psi, \vartheta) := \min\{eval(\phi, \vartheta), eval(\psi, \vartheta)\}$
- $eval(\phi \vee \psi, \vartheta) := \max\{eval(\phi, \vartheta), eval(\psi, \vartheta)\}$

A complete assignment is also called a **model**, as it can be taken to represent a state of affairs. Via the evaluation function, a formula encodes a set of truth conditions for a model, thereby giving a description of all possible states of affairs where that formula is true. Some additional terminology is commonly used for talking about the relationships between formulae and assignments:

Definition 2.1.3. *(**Satisfaction, Entailment and Equivalence**)*

- *A complete assignment ϑ **satisfies** or is a **model of** a propositional formula ϕ iff ϕ evaluates to "true" under the assignment, i.e. $eval(\phi, \vartheta) = 1$ holds.*
- *A formula ϕ **entails** another formula ψ if for every complete assignment ϑ with $eval(\phi, \vartheta) = 1$, $eval(\psi, \vartheta) = 1$ holds as well. In this case, we write $\phi \vDash \psi$.*
- *Two formulae ϕ and ψ are **(semantically) equivalent** iff both $\phi \vDash \psi$ and $\psi \vDash \phi$ hold, i.e. ϕ and ψ entail each other. Semantic equivalence is written as $\psi \equiv \phi$.*

Note that $\psi \equiv \phi$ means that any model of ϕ must also be a model of ψ, and vice versa.

2.1.2 CNF and Definitional CNF

A range of syntactic transformations can be applied to propositional formulae without destroying semantic equivalence. For instance, the distributivity laws allow nested conjunctions and disjunctions to be flattened, and De Morgan's laws can be used to move negation symbols inwards. This syntactic freedom can be exploited to restrict oneself to formulae which possess certain syntactic properties that facilitate processing.

Some such formula classes with particularly simple structure and useful properties have received the status of **normal forms**. Here we will only introduce the normal form which is most relevant to our purposes, because it is the form that many standard algorithms of automated reasoning expect their input to be provided in.

Definition 2.1.4. *(**Clauses and CNF**)*

- *A **clause** is a formula of the form $(l_1 \vee l_2 \vee \cdots \vee l_k)$, i.e. a disjunction of literals. k is called the **size** of the clause. We will designate clauses by $C_1, C_2, \ldots$, and will often write them as sets of literals $C = \{l_1, \ldots, l_k\}$.*

- *A formula is said to be in **Conjunctive Normal Form (CNF)** if it has the form $(C_1 \wedge C_2 \wedge \cdots \wedge C_m)$, i.e. a conjunction of clauses. We will often treat a CNF formula as a **clause set** $\{C_1, C_2, ..., C_m\}$, and use Latin capital letters F and G, possibly with subscripts, to refer to clause sets.*

While it is possible to transform any propositional formula into an equivalent formula in CNF, during the process the transformed formula can grow exponentially, quickly becoming intractably large. It can be shown (see [1]) that this problem necessarily occurs in the worst case of any algorithm which tries to maintain semantic equivalence while transforming arbitrary formulae into CNF.

In our context, however, we will mostly be interested in the question whether a formula is satisfiable at all, and not whether it has exactly the same models as some other formula. In this situation, a weaker form of equivalence is sufficient:

Definition 2.1.5. *(**Equisatisfiability**) Two formulae ϕ_1 and ϕ_2 are called **equisatisfiable** if ϕ_1 is satisfiable whenever ϕ_2 is.*

If maintaining equisatisfiability is enough, we can circumvent the exponential blowup during CNF transformation using a method which is originally due to Tseitin [2], but is today most often used in the variant by Plaisted & Greenbaum [3]. The method introduces additional variables as shortcuts for representing subformulae, but only enforces one direction of the equivalence conditions used to define these variables:

Definition 2.1.6. *(**Tseitin Transformation to Definitional CNF**)*
*The **Tseitin encoding** of a propositional formula ϕ is a clause set defined as the output of a function $tseitinPos(\phi)$ which is computed recursively as follows:*

- $var(\psi) := l$ *for a literal $\psi = l$, and a unique new variable for non-literals*

- $tseitinPos(l) := tseitinNeg(l) := \{\}$ *for each literal l*

- $tseitinPos(\neg\psi) := tseitinNeg(\psi)$ *for any formula ψ*

- $tseitinPos(\phi_1 \wedge \phi_2) := \{\{\neg var(\phi_1, \phi_2), var(\phi_1)\}, \{\neg var(\phi_1, \phi_2), var(\phi_2)\}\}$
  $\cup\, tseitinPos(\phi_1) \cup tseitinPos(\phi_2)$ *for any two formulas ϕ_1, ϕ_2*

- $tseitinPos(\phi_1 \vee \phi_2) := \{\{\neg var(\phi_1, \phi_2), var(\phi_1), var(\phi_2)\}\}$
  $\cup\, tseitinPos(\phi_1) \cup tseitinPos(\phi_2)$ *for any two formulas ϕ_1, ϕ_2*

- $tseitinNeg(\phi_1 \wedge \phi_2) := \{\{var(\phi_1, \phi_2), \neg var(\phi_1), \neg var(\phi_2)\}\}$
  $\cup\, tseitinNeg(\phi_1) \cup tseitinNeg(\phi_2)$ *for any two formulas ϕ_1, ϕ_2*

- $tseitinNeg(\phi_1 \vee \phi_2) := \{\{var(\phi_1, \phi_2), \neg var(\phi_1)\}, \{var(\phi_1, \phi_2), \neg var(\phi_2)\}\}$
  $\cup\, tseitinNeg(\phi_1) \cup tseitinNeg(\phi_2)$ *for any two formulas ϕ_1, ϕ_2*

Plaisted & Greenbaum give the proof that this algorithm transforms any formula to CNF while maintaining equisatisfiability for the more general case of first-order logic. A variant of the Tseitin Transformation will be used later when an interactive MUS extraction method for formulae in CNF is generalized to arbitrary propositional formulae.

2.1.3 SAT Solving

With the syntax and semantics of propositional logic in place, we can now turn to the computational aspects that the work in this thesis builds on. We start by formally defining the SAT problem, followed by a short overview the field of SAT solving. Then, the custom variant of the SAT solver MiniSat by Christian Zielke, which the implementations builds on, is described in some detail.

2.1.3.1 The SAT Problem

Definition 2.1.7. *(**Satisfiability and Validity**) A propositional formula ϕ is **satisfiable** if it is satisfied by at least one model. If no model satisfies ϕ, we say that ϕ is **unsatisfiable**, and we write $\not\models \phi$. If ϕ is satisfied by every model, we say it is **valid**, and we write $\models \phi$.*

Definition 2.1.8. *(**The Propositional Satisfiability Problem (SAT)**)*
*The **SAT problem** is the problem of deciding whether a propositional formula ϕ is satisfiable or unsatisfiable. Any formula ϕ for which we want to solve the SAT problem, especially in the context of benchmark sets, is commonly called a **SAT instance**.*

Unlike in the case of more expressive logics, the number of possible models over the set of variables occurring in some formula is finite. Furthermore, by induction over the formula size it is clear that *eval* terminates after a finite number of steps on each model. These two observations already imply that the SAT problem is **decidable**, so there exists a procedure which can decide in a finite number of steps whether an arbitary propositional formula is satisfiable or unsatisfiable. Such a procedure is called a **SAT solving** algorithm.

The SAT problem is of central importance to computer science because it is the prototypical case of an **NP-complete** problem. Many NP-complete problems can very directly be translated to SAT, and SAT has been reduced to many problems in order to prove their NP-completeness. Since it is still unknown whether efficient procedures for NP-complete problems exist, we must assume that it is impossible to develop a SAT solving algorithm which is polynomial in the worst case.

2.1.3.2 SAT Solving

Although SAT solving is NP-complete, SAT instances derived from real-world problems are often a lot more easily solvable than one would expect from this theoretical result. In fact, modern SAT solvers such as Glucose [4], PrecoSAT [5], and SATzilla [6] can solve many SAT instances from industrial applications, often with millions of clauses, in reasonable time.

A variety of SAT solving algorithms and strategies have been proposed and implemented during the past decades. The single most successful paradigm builds on the classic DPLL algorithm (see [7] for the original exposition) enhanced by conflict-directed clause learning in a variant first introduced by Marques-Silva and Sakallah [8]. The basic mechanisms of these algorithms, such as unit propagation and refutation proofs by resolution, must be presupposed here for reasons of brevity. For a good overview of these topics, the reader is referred to the discussion in [9].

The standard input format for SAT instances is the DIMACS format, a simple plain text format for representing propositional formulae in CNF which arose in connection with the international SAT competition [10], a biannual event where SAT solvers are compared and evaluated on a variety of benchmarks. A SAT solver minimally outputs whether the input instance is satisfiable or not. In the satisfiable case, most SAT solvers optionally specify a satisfying assignment for the input formula. Some SAT solvers are also able to print out some sort of explanation in the unsatisfiable case, e.g. a refutation proof trace or a list of the new clauses derived during conflict-directed clause learning.

2.1.3.3 A Custom Variant of MiniSat

The prototype implementation of interactive MUS extraction described in this thesis builds on the solver MiniSat by Niklas Eén and Niklas Sörensson [11], which is one of the most popular freely available SAT solvers because it is small, extensible, and well-documented [12] while still displaying competitive performance.

Since some of the algorithms we will employ require us to analyse proofs in the unsatisfiable case, we build on a custom variant of MiniSat version 1.14 as the most recent version for which a variant with support for proof logging is available. Christian Zielke has extended this version by three additional features in order to fulfil the requirements posed to a SAT solver by the prototype implementation.

Most importantly, Zielke's version allows to specify an additional input file with a vector of so-called **freeze literals**, which can be seen as a predefined partial assignment the solver may not modify while searching for a model. This mechanism makes it possible to deactivate clauses in a SAT problem enhanced by **selector variables**, i.e. an additional variable for each clause whose negation is appended to the clause it selects. If the selector variable is set to false in the freeze file, the corresponding clause instantly evaluates to true, with the same consequences as if we had created a new input file with that clause missing. Temporarily cutting away parts of a SAT instance using the selector variable mechanism is a lot faster than it would be to copy the entire SAT instance into a new file each time we want to select a different subproblem.

The second extension concerns the proof format. By default, the proof logging option of MiniSat version 1.14 produces a list of learned clauses along with references to the clauses that took part in deriving each new clause. Zielke extended this proof output by a list of the resolution variables used in these derivations, which was needed for a new MiniSat-based implementation of the autarky finding algorithm discussed in Section 2.4.3.2.

The third extension adds support for a separate output of learned literals. Whereas in the unsatisfiable case, these could also have been extracted from the existing proof logging output, the adapted version also prints out the units which are derived and unit-propagated in the satisfiable case. In later chapters, this output of learned units is used as one alternative for implementing the meta-learning enhancement to interactive MUS extraction.

Many of the algorithms presented in this thesis build on calls to a SAT solver. In pseudocode, a call to the custom variant of MiniSat will be written $\langle res, proof, model, units\rangle := sat(F, frzLits)$ for a CNF formula F and a set of freeze literals $frzLits$. res will receive one of the values sat or $unsat$ to represent satisfiability and unsatisfiability. In the $unsat$ case, $proof$ will represent a proof object, on which we can execute a function $used\_clauses$ to retrieve the subset $F' \subseteq F$ which was used by MiniSat to prove unsatisfiability, and a function $used\_vars$ to retrieve the resolution variables. In the sat case, $model$ will be a non-empty set of literals representing a total assignment which satisfies F. In both cases, $units$ contains the unit clauses derived during the solver run.

2.2 Minimal Unsatisfiable Cores

As mentioned in the introduction, the main contribution of this thesis is the development of an interactive approach to the extraction of minimal explanations for the unsatisfiability of propositional formulae. The most popular and direct approaches to generating such explanations are based on extracting what is called a **minimal unsatisfiable core (MUC)**, which in most general terms can be defined as a subproblem of the original problem that

is still unsatisfiable, but only has satisfiable subproblems. There are quite a few different possibilities to formally define such cores, depending on the scenario and on the form in which the SAT problems are given. Three types of MUCs which are of particular importance to our purposes are motivated and formally defined in this section.

2.2.1 Minimal Unsatisfiable Subsets of CNF Formulae

The intuitive notion of a MUC is easiest to define if the unsatisfiable formula is already represented as a set of constraints from which we can select arbitrary subsets. This is of course the case for propositional formulae in CNF. In this context, we can define MUCs in the obvious way as minimal unsatisfiable subsets of a clause set:

Definition 2.2.1. *(**Minimal Unsatisfiable Subset**) A subset $F' \subseteq F$ of an unsatisfiable clause set F is a **minimal unsatisfiable subset (MUS)** of F if it is unsatisfiable, and every subset $F'' \subset F'$ is satisfiable.*

In propositional logic, conjunction is monotonic with respect to unsatisfiability, meaning that an unsatisfiable clause set cannot be made satisfiable by adding additional clauses, and that a satisfiable clause set stays satisfiable when we remove clauses from it. This leads to an important local property of MUSes, which can also act as an alternative definition that is a lot easier to check algorithmically:

Property 2.2.2. *(**Alternative Characterization of MUSes**) An unsatisfiable clause set F is a MUS iff it is unsatisfiable, and for every clause $C \in F$, $F \backslash \{C\}$ is satisfiable.*

SAT instances which occur and need to be solved in applications often consist of a general part describing some well-tested software or hardware system which stays the same across instances, and a particular part encoding one specific combination of settings or inputs which is different for every instance. In such a situation, one often desires minimal explanations in terms of clauses from the particular part only, since the clauses from the general part tend to only extend the MUS size without explaining much.

The standard way to axiomatize the desired type of MUS in such a situation is to partition the instance into a set of **relevant clauses** (often corresponding to the part that changes across instances) and a set of **don't-care clauses** containing all the clauses that wouldn't explain much if they were part of a MUS. If we then want to find unsatisfiable subsets which are minimal in terms of a set of relevant clauses, we use the following definition, which simply includes all don't-care clauses into every US, but does not count them:

Definition 2.2.3. *(**MUS with respect to relevant clauses**) Let F be an unsatisfiable clause set, and $R \subset F$ be a set of relevant clauses. A subset $R' \subseteq R$ of the relevant clauses is a **MUS of** F **with respect to** R if $R' \cup F \backslash R$ is unsatisfiable, and for every subset $R'' \subset R'$, $R'' \cup F \backslash R$ is satisfiable.*

This definition can also be expressed (and implemented) in terms of clause removals from the set R, in full analogy with the characterization of standard MUSes.

2.2.2 GMUSes for Group CNF

In many applications, some parts of a system are best treated as blackboxes, as we are often not interested in the internal workings of well-tested simple components, but only in the behaviour and the interactions of such components. For instance, in hardware verification, the behaviour of an adder or a multiplexer will be described by a set of clauses encoding the boolean function it computes. If we want to locate some error in the wiring between such components, having many clauses from the internal representations of these components as part of a MUS will make the relevant higher-level problem much harder to spot.

The existence of groups of clauses within a CNF formula which belong more closely together was formalized by Alexander Nadel [13] in the notion of a group SAT instance, and a type of MUS which only enforces minimality with respect to the contained groups is then straightforwardly defined:

Definition 2.2.4. *(**Group SAT instance**) A **group SAT instance** is a SAT instance F which is partitioned into (i.e. the disjoint union of) a set of don't care clauses D, and a number of clause sets $G_1, \ldots, G_k$ which are called groups.*

Definition 2.2.5. *(**Group MUS**) A **group MUS (GMUS)** of a group SAT instance $F = D \cup \bigcup G$ with $G = \{G_1, \ldots, G_k\}$ is a subset $G' \subseteq G$ such that $D \cup \bigcup G'$ is unsatisfiable, but for every $G'' \subset G'$, $D \cup \bigcup G''$ is satisfiable.*

Because of the high relevance of GMUS extraction for practical applications, the SAT competition, which introduced a MUS track for the first time in 2011, now features separate subtracks for plain MUS extraction and GMUS extraction.

2.2.3 Minimal Unsatisfiable Cores in Non-Clausal Formulas

Whereas most current work on MUC finding concentrates on the development of efficient methods and tools for MUS and GMUS extraction, there are some applications for which other types of MUCs are potentially more interesting. Though any propositional formula can be transformed into an equivalent (or equisatisfiable) clause set, much of the structure of the original formula tends to get lost in the process. A formula may originally have a modular and often human-readable structure, but the clauses resulting from CNF conversion are barely interpretable in isolation, and often impossible to group into meaningful subsets, because the application of distributivity laws tends to spread variables all over the clause set which were formerly only used in one small part of the formula.

One idea to approach this problem is to define unsatisfiable cores not in terms of clause subsets, but in terms of subformulae. The obvious approach is to view non-clausal formulae as syntactic trees, and to define a minimal unsatisfiable formula as an unsatisfiable formula that becomes satisfiable as soon as we remove an arbitrary subtree. Viktor Schuppan [14] presents a formalization of this approach for formulae of the temporal logic LTL. He proposes to distinguish the cases of subtree removal under conjunctive and disjunctive nodes. The definition of conjunctive and disjunctive nodes refers to the **polarity** of a node, defined as positive if the number of negations above that node is even, and negative if it is odd. By De Morgan's laws it is clear that a conjunction node of negative polarity needs to be treated as disjunctive, and vice versa. Schuppan's definition of tree-based MUCs now builds on replacing disjunctive subtrees by $\bot$, and conjunctive subtrees by $\top$. Note that this is semantically equivalent to simply removing the respective disjunct or conjunct, although syntactically, it retains some of the structure of the input formula.

The problem is that this definition lacks an important desirable property of clause-based MUSes. Removing a disjunct from a disjunctive node makes the formula more instead of less constrained, meaning that a satisfiable formula can become unsatisfiable again when a subtree is removed, i.e. we lose the desired monotonicity. Unsatisfiable formulae which are minimal according to this definition therefore tend to reduce all disjunctive nodes to just one child, often leaving only a meaningless skeleton around a pair of complementary literals that occurred at arbitrary positions in the original formula, which does not at all explain the original formula's unsatisfiability.

As part of their comprehensive work on the theory of MUS extraction [15], Hans Kleine Büning and Oliver Kullmann give an alternative definition of minimal unsatisfiable formulae with nicer properties. They circumvent the need to consider polarities by operating only on

formulae in **negation normal form (NNF)**, where negation may only occur immediately in front of atoms. They then define an **or-subtree** of the formula tree to be a subtree whose root is either a disjunction node or a literal immediately below a conjunction node. The or-subtrees are exactly the subtrees which can be removed without any risk of making a satisfiable formula unsatisfiable again. This leads to the following definition:

Definition 2.2.6. *(**Minimal Unsatisfiable Formula**) A propositional formula ϕ in NNF is called **minimal unsatisfiable** if ϕ is unsatisfiable, and eliminating an arbitrary or-subtree from its formula tree makes ϕ satisfiable.*

This definition is of course trivial to extend to non-NNF formulae by re-introducing the concept of polarity. An or-subtree is then simply defined as having a disjunction root of positive polarity or a conjunction root of negative polarity, or being a literal immediately below a conjunction root of positive or a disjunctive root of negative polarity. Nevertheless, we will use the NNF variant in Chapter 5 for our discussion of interactive MUC extraction in non-CNF formulae.

2.3 Basic MUS Extraction Algorithms

The notion of a MUS leads to several different algorithmic tasks, which considerably differ in complexity and achievable performance, but also in the methods used for approaching them.

The easiest task is to find a single MUS, and is commonly just called **MUS extraction**. Note that a minimal unsatisfiable subset is not required to be the smallest in the sense that no smaller unsatisfiable subsets exist. This reduces the task to an essentially linear traversal of the search space until we arrive at a subset which fulfills the definition. This task can be solved quite efficiently in practice, and the most popular methods and techniques to achieve this are the main subject of this section.

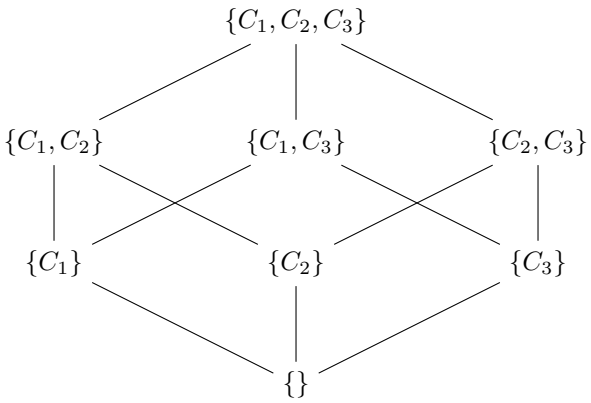
A more ambitious goal is to find a MUS of minimum cardinality, which is also called a **smallest MUS (SMUS)** or a **minimum unsatisfiable subset**. Algorithms in this field first relied on non-chronological backtracking [16], but a branch-and-bound approach [17] has turned out to be a lot more efficient. The practical relevance of this problem seems to be quite limited, as a MUS of minimal size is a lot harder to find than an arbitrary MUS, but often does not contain a significant amount of additional information.

Finally, the most ambitious task is to exhaustively enumerate **all MUSes** of an unsatisfiable clause set. Existing approaches to this problem rely on clever enumeration to economize the checking of subset candidates [18], or on exploiting the duality of MUSes and maximal satisfiable subsets in an interleaved [19] or a two-level [20] approach to a hypergraph transversal problem. Given the fact that a clause set can contain exponentially many different MUSes, no efficient general method for finding all MUSes can exist. Even in large industrial instances, however, the number of different MUSes is often surprisingly low, giving some practical relevance to the existing tools for finding all MUSes, because the set of all MUSes contains comprehensive information about an error's nature.

While the work in this thesis builds on the existing algorithms for extracting a single MUS, some degree of non-determinism will be added by the interactive search interface. In principle, the interface will make the entire search space accessible to the user, although it will of course be impractical to exhaust it.

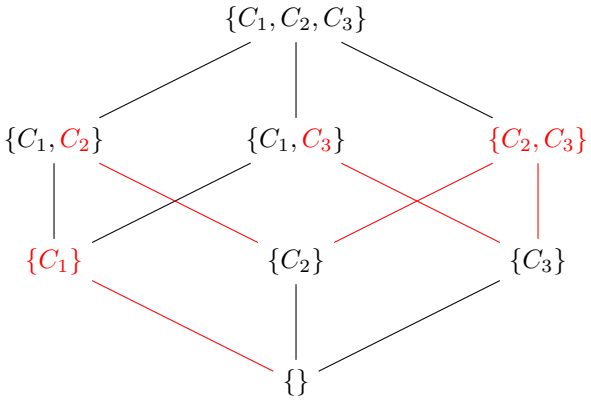
2.3.1 MUS Extraction and the Powerset Lattice

In the experience of the author, it helps to explicitly conceive the task of MUS extraction as a search task in the **powerset lattice**, i.e. the set of all subsets connected by the subset relation. As our running example for our discussion of MUS extraction, we will take a set of three clauses $\{C_1, C_2, C_3\}$, causing our search space to look like this:



For navigating such powerset lattices, by convention we will speak of downward movement when we remove elements to arrive at smaller subsets, and of upward movement to talk about the addition of elements, shifting to larger subsets. Note that before the search process starts, we know that the top node of the lattice represents an unsatisfiable set, and that the bottom node is trivially satisfiable. Between these two extremes, we find a vast unexplored landscape of satisfiable and unsatisfiable subsets.

Monotonicity implies that no subset of a satisfiable clause set can be unsatisfiable, so there will always be a **transition boundary** between unsatisfiable subsets in the upper part, and satisfiable subsets in the lower part of the powerset lattice. In some unsatisfiable subset F', a clause $C \in F'$ such that $F' \backslash \{C\}$ becomes satisfiable is called a **critical clause** or a **transition clause**. According to Property 2.2.2, a MUS is therefore an unsatisfiable subset where all clauses are critical, meaning that the minimal unsatisfiable subsets are positioned along the transition boundary. The task of finding one or more MUSes can thus be conceived as efficiently traversing the powerset lattice in search of this boundary. Assuming that our clause set has the two MUSes $\{C_1\}$ and $\{C_2, C_3\}$, the powerset lattice with the critical clauses and the transition boundary marked in red would look like this:



Another important fact about the search space is that above the transition boundary, criticality is monotonic under clause removal, and non-criticality is monotonic under clause

addition. This means that during the search, whenever we find out that some clause is critical in an US, we know that it will stay critical in all its unsatisfiable subsets, and whenever some clause isn't critical in some US, it means that it will not have been critical in any US on the path from the current US up to and including the original clause set.

2.3.2 Deletion-Based Approaches

Many of the most performant algorithms for MUS extraction from CNF instances can be classified as **deletion-based**. In such algorithms, starting from some unsatisfiable subset, we gradually try to remove clauses while making sure that the candidate set stays unsatisfiable. When the candidate set cannot be further reduced, we have arrived at a MUS by the alternative definition. In the powerset lattice, this corresponds to starting at the top and moving down towards the transition boundary, testing for and detecting the transition clauses from above.

The advantage of deletion-based approaches is that we can make larger reduction steps by analysing the refutation proofs produced by the SAT solver. If we generate such a proof and only select those clauses which were used in it as axioms, we are guaranteed to receive an unsatisfiable subset smaller or equal to the reduced clause set.

This technique is often referred to as **clause set refinement** in the literature. It was made popular by Alexander Nadel [13], who uses it in the following simple, but very efficient algorithm for deletion-based MUC extraction:

Algorithm 1 Deletion-Based MUS Extraction Using Selector Variables

Input: an unsatisfiable SAT instance $F = \{C_1, \ldots, C_m\}$ in CNF
Output: some minimal unsatisfiable subset $S \subseteq F$

```
F' := {}                                   ▷ stores the problem extended by meta variables
for each clause C_i ∈ F do
    F' := F' ∪ {C_i ∨ ¬s_i} for a new selector variable s_i
end for
⟨res, proof, φ, units⟩ := sat(F', {s_1, ..., s_m})
for each C_j ∉ used_clauses(proof) do                    ▷ clause set refinement
    F' := F' ∪ {{¬s_j}}
end for
US := {i | C_i ∈ used_clauses(proof)}                    ▷ the clauses of unknown status
MUS := {}                                                ▷ collects critical clauses
while (US is not empty) do
    k := select one index ∈ US\MUS
    US := US\{k}
    ⟨res, proof, φ, units⟩ := sat(F', {s_i | i ∈ US ∪ MUS})        ▷ reduction attempt
    if res = sat then                                    ▷ unsuccessful reduction, so k is critical
        MUS := MUS ∪ {k}
    else
        US := {i | C_i ∈ used_clauses(proof)}            ▷ clause set refinement after successful reduction
        for each C_j ∉ used_clauses(proof) do
            F' := F' ∪ {{¬s_j}}
        end for
    end if
end while
return S := {C_i | i ∈ MUS}
```

In the pseudocode, MUS collects the indices of clauses known to be critical, and US contains the indices representing the current candidate US. The complexity of MUS extraction algorithms is commonly measured in the number of necessary SAT calls. Obviously, the number of SAT calls for this algorithm is in $O(m)$, i.e. linear in the size of the input problem.

For the prototype implementation of interactive MUS extraction, parts of an implementation of this algorithm by Martin Lahl [21] were reused. For the purposes of interactive reduction, the implementation had to be extended by explicit representation and storage of the intermediary USes as well as the collected criticality information. Furthermore, the algorithm's data structures and the interface to MiniSat were adapted to allow for running several deletion-based MUS extraction processes concurrently. The resulting parallel system architecture is detailed in Section 3.4.1.

Note that Nadel's algorithm can trivially be adapted (and was in fact originally designed) for the case of GMUS. We simply introduce only one selector variable for each group, and construct F' by extending all clauses of the same group by the same selection variable. This ensures that the clauses of one group are always removed from the candidate set together. As the work in this thesis will show, the underlying idea of steering and modifying deletion-based MUS extraction by manipulating the selector variables can also be exploited in other interesting ways.

2.3.3 Insertion-Based Approaches

A second class of MUC extraction algorithms works in a manner dual to the deletion-based approach. In **insertion-based** algorithms, we start with a satisfiable subset of the constraint set, gradually expanding it by additional clauses until our candidate set becomes unsatisfiable. In the powerset lattice, we therefore start at the bottom and move up towards the transition boundary, testing for and detecting transition clauses from below.

While they tend to require a higher number of solver calls, an advantage of insertion-based approaches is that they profit immensely from the use of **incremental SAT solving**. An incremental SAT solver can store the data it derived while processing a clause set, and use it to process further input problems more efficiently if each problem only differs from the previous problem by the addition of clauses. In an insertion-based approach which gradually adds more clauses to a candidate set, the individual calls to a SAT solver can therefore be performed incrementally at a much lower cost.

The first algorithm for insertion-based MUS extraction was presented by Hans van Maaren and Siert Wieringa [22]. Their algorithm operates in rounds, repeatedly inflating a satisfiable under-approximation until a new transition clause is found. During the inflation, redundant clauses are detected and pruned away for the next iteration. For detecting redundant clauses, the algorithm makes use of the observation [23] that a clause C is redundant in a CNF formula F if $F \backslash \{C\} \cup \neg C$ is unsatisfiable, where $\neg C$ for a clause $C = \{l_1, \ldots, l_k\}$ is a shorthand for the set of unit clauses $\{\{\neg l_1\}, \ldots, \{\neg l_k\}\}$ obtained by negating C. This result can be exploited to create a more constrained SAT instance without losing any satisfying assignments, often considerably reducing the time needed for each SAT solver run.

The last non-redundant element that could be added during inflation before F'' becomes unsatisfiable, is identified as a transition clause. Note that the outer while loop terminates when *lastAppended* stays empty, i.e. as soon as during inflation, every element remaining in $F' \backslash MUS$ was found to be redundant. If k is the size of the MUS we find, the number of SAT solver calls is in $O(k \cdot m)$, since each iteration of the outer while loop adds an element to MUS, and the inner for loop needs to iterate through the entire input clause set in the worst case. In practice, $F' \backslash MUS$ will clearly be much smaller than that.

Algorithm 2 Insertion-Based MUS Extraction

Input: an unsatisfiable SAT instance $F = \{C_1, \ldots, C_m\}$ in CNF
Output: some minimal unsatisfiable subset $S \subseteq F$

```
MUS := {}                                   ▷ MUS under construction, collects transition clauses
F' := F                                     ▷ candidate transition clauses
while (|MUS| < |F'|) do
    F'' := MUS
    lastAppended := {}
    for C_i ∈ F'\MUS do
        ⟨res, proof, φ, units⟩ := sat(F'' ∪ ¬C_i, {})      ▷ incremental redundancy checking
        if res = sat then
            F'' := F'' ∪ {C_i}
            lastAppended := {C_i}                          ▷ remember the last non-redundant element
        end if
    end for
    MUS := MUS ∪ lastAppended
    F' := F''
end while
return MUS
```

Marques-Silva and Lynce [24] present a variant which improves the number of SAT solver calls to $O(m)$ for this paradigm as well, showing that insertion-based approaches are not necessarily slower than deletion-based ones. The savings are due to the use of **relaxation variables**, which work just like selection variables except that a ≤ 1 constraint over them ensures that only one clause is added to MUS in each iteration. If more than one relaxation variable is needed to achieve satisfiability, this implies that we are exploring the search space towards more than one MUS, so one is selected and the other one explicitly blocked.

2.3.4 A Recent Hybrid Approach

In [24], Marques-Silva and Lynce emphasize that unlike SAT solvers, algorithms for MUC extraction have not yet reached industrial strength. As a first step towards a remedy of this situation, they present a novel hybrid approach to MUC extraction, which integrates a great variety of ideas from earlier approaches. Apart from being the state of the art, this hybrid algorithm is of further interest to our discussion here because it blurs the distinction between the previous two paradigms. While the general layout of the algorithm is still insertion-based, techniques from the deletion-based approach such as clause set refinement are used to efficiently manage a set of candidate transition clauses. The method remains heavily inspired by van Maaren and Wieringa in that it adapts their efficient redundancy check. A disadvantage of the hybrid approach is that it cannot make use of incremental SAT solving any more.

Note that unlike in the purely deletion-based approach, we cannot use clause set refinement every time we managed to throw out a candidate transition clause. The reason for this is that the redundancy check extends the SAT instance by additional unit clauses which cause a spurious unsatisfiability if C_i was redundant, but it can happen that the clause was only redundant under the condition that some other elements of F' were part of the checked instance, which are however not needed for the unsatisfiability proof as long as the unit clauses are there. Only if none of these unit clauses was involved in proving the unsatisfiability can we be sure that the reduced set was still unsatisfiable in the sense needed for clause set refinement, i.e. without needing additional elements of F'.

Algorithm 3 Hybrid MUS Extraction

Input: a trimmed unsatisfiable SAT instance $F = \{C_1, \ldots, C_m\}$ in CNF
Output: some minimal unsatisfiable subset $S \subseteq F$

```
MUS := {}                                   ▷ MUS under construction, collects transition clauses
F' := F                                     ▷ candidate transition clauses
while (F' is not empty) do
    C_i := selectClause(F')
    F' := F'\{C_i}
    ⟨res, proof, φ, units⟩ := sat(MUS ∪ F' ∪ ¬C_i, {})      ▷ redundancy checking
    if res = sat then
        MUS := MUS ∪ {C_i}
    else if used_clauses(proof) ∩ ¬C_i = ∅ then
        F' := used_clauses(proof)\MUS       ▷ clause set refinement
    end if
end while
return MUS
```

Concerning the complexity, it is clear that each clause from the input formula is analysed exactly once, which means that the number of SAT solver calls is in $\Theta(m)$. In the practical experiments conducted by Marques-Silva and Lynce [24], their hybrid algorithm performs significantly better than all previous approaches on a range of benchmarks.

2.4 Advanced Concepts in MUS Extraction

In the last section of this introductory chapter, we turn our attention to three further relevant branches of current research on MUS extraction. The first is an important algorithmic technique which represents a breakthrough in better exploiting the information returned by the SAT solver during MUS extraction, the second consists of a very useful classification of clauses in unsatisfiable clause sets, and the third concerns some deeper theoretical results about the nature of the MUS extraction problem.

2.4.1 Model Rotation

Marques-Silva and Lynce [24] present an additional very useful technique for speeding up deletion-based or hybrid MUS extraction, which uses the model returned by the SAT solver to get a lot more information out of unsuccessful reduction attempts. We present a variant called **recursive model rotation** because it continues to flip assignments in the model and then checks whether each model variant satisfies all but a single isolated clause, which is thereby found to be critical:
Model rotation can and should be executed after each unsuccessful reduction attempt in deletion-based or hybrid MUS extraction, and it significantly decreases the number of SAT solver calls necessary to arrive at a MUS. Intuitively, the procedure works because a critical clause C_j in an unsatisfiable clause set F is characterized by having an **associated assignment**, i.e. an assignment φ which satisfies $F\backslash\{C_j\}$, but not C_j. Model rotation exploits this property by cheaply deriving from an associated assignment other assignments which can quickly be tested for associatedness with other clauses. A formal proof of the fact that a clause is critical iff it has an associated assignment can be found as a prerequisite of the correctness proof for recursive model rotation by Belov & Marques-Silva [25].

Siert Wieringa [26] gives an alternative description of recursive model rotation based on traversals of the flip graph, achieves further improvements to the algorithm based on these insights, and provides some analysis of benchmark instances which (partially) explain its

Algorithm 4 Recursive Model Rotation

Input: unsatisfiable $F = \{C_1, \ldots, C_m\}$, critical clause $C_k \in F$, model φ of $F \backslash \{C_k\}$
Output: a set of further critical clauses $Crit \subseteq F$

```
Crit := {}
for each variable v in C_k do
    φ(v) := 1 − φ(v)                          ▷ flip the variable in the model
    nSat := {C_i | φ(C_i) = 0}                ▷ collect clauses not satisfied by model
    if nSat = {C_j} for some 1 ≤ j ≤ m then   ▷ φ is an associated assignment for C_j
        Crit := Crit ∪ {C_j}
        Crit := Crit ∪ modelRotation(F, C_j, φ)   ▷ recursive case with new critical clause
    else
        φ(v) := 1 − φ(v)                      ▷ flip the variable back
    end if
end for
return Crit
```

high usefulness in practice. Marques-Silva and Lynce [24] as well as Lahl [21] determine in benchmarks that recursive model rotation is the single most effective technique for speeding up MUS extraction, motivating its implementation and use for this thesis.

2.4.2 Classification of Clauses

Careful inspection of the structure of typical MUS extraction problems shows that all the different MUSes of an unsatisfiable SAT instance tend to overlap, containing a common core set of clauses, which is satisfiable on its own. To arrive at some MUS, this common core must be made unsatisfiable by adding some combination of further clauses. Kullmann, Lynce and Marques-Silva [27] build on this observation to formally distinguish three degrees of necessity for clauses with respect to MUSes, each with a corresponding dual notion of clause redundancy. These definitions give us some very useful vocabulary for talking about the role of different clauses in interactive MUS extraction. In the entire section, we use F for some unsatisfiable SAT instance, and $MU(F)$ for the set of all the MUSes of F.

2.4.2.1 Necessary and Unnecessary Clauses

Definition 2.4.1. *(**Necessary Clause**) A clause $C \in F$ is called **necessary** if every resolution refutation of F must use C as an axiom. This is the case iff $F \backslash \{C\}$ is satisfiable.*

The set of necessary clauses forms the mentioned core that every MUS must contain, and can therefore be written as $\bigcap MU(F)$. Given m clauses in F, $\bigcap MU(F)$ is trivial to compute by m calls to a SAT solver, since we can simply check if $F \backslash \{C\}$ is satisfiable for each clause C. Using model rotation in the satisfiable and proof analysis in the unsatisfiable case, this process can of course be enormously accelerated.

The dual to this strongest notion of necessity is a very weak notion of redundancy of a clause, only demanding that unsatisfiability is maintained when it is the single clause we remove from the original F, but not ensuring that any two unnecessary clauses can be removed at the same time:

Definition 2.4.2. *(**Unnecessary Clause**) A clause $C \in F$ is called **unnecessary** if there is some resolution refutation of F which does not use C as an axiom. By completeness of resolution, this is the case iff $F \backslash \{C\}$ is still unsatisfiable.*

2.4.2.2 Potentially Necessary and Never Necessary Clauses

Definition 2.4.3. *(**Potentially Necessary Clause**) A clause $C \in F$ is called **potentially necessary** if there exists an unsatisfiable $F' \subseteq F$ such that C is critical in F'.*

Potentially necessary clauses are thus all clauses which are either necessary, or become critical when some other clauses are removed. Since MUSes are USes in which all clauses are critical, this means that all clauses in a MUS must be at least potentially necessary in the original formula, so we can write the set of all potentially necessary clauses in F as $\bigcup MU(F)$. This set could serve to define the problem of finding an explanation of infeasibility in a canonical way, since it contains everything that is needed to explain the infeasibility, but has exactly one solution. Unfortunately, $\bigcup MU(F)$ appears to be very hard to compute, although the complexity is still unknown. According to Kullmann et al. [27], the best approaches do not go much beyond employing an efficient algorithm for computing all MUSes, and then directly computing their union.

The corresponding notion of redundancy is already a lot stronger than that of an unnecessary clause, as while still allowing that the clause be used in some refutation proof of F, it demands that it must not be necessary for proving the unsatisfiability of any subset of F:

Definition 2.4.4. *(**Never Necessary Clause**) A clause $C \in F$ is called **never necessary** if in all unsatisfiable subsets $F' \subseteq F$, C is unnecessary.*

The definition implies that we can safely remove any combination of clauses known to be never necessary, although this might make the unsatisfiability a lot harder to prove. Never necessary clauses can thus safely be thrown away when looking for MUSes, although a little caution is in order, because they can contribute to much shorter proofs of unsatisfiability, which might be a lot easier to understand than the proof of a MUS. We will not explore this issue further here, as it is a general problem with the approach of taking minimal subsets as formal approximations to small explanations of infeasibility.

2.4.2.3 Usable and Unusable Clauses

Definition 2.4.5. *(**Usable Clause**) A clause $C \in F$ is called **usable** in F if there exists some tree resolution refutation of F which uses C as an axiom.*

This weakest notion of necessity states that the clause may be useful for shortening the unsatisfiability proofs for some USes, but its addition does not make any satisfiable subset unsatisfiable. The set of usable clauses of a clause set F is called its **lean kernel** $N_a(F)$, and a clause set F with $F = N_a(F)$ is called **lean**. If a clause set is not lean, the clauses outside its lean kernel are called *unusable*, which we define in the obvious way:

Definition 2.4.6. *(**Unusable Clauses**) A clause $C \in F$ is called **unusable** in F if there exists no tree resolution refutation of F which uses C as an axiom.*

The unusable clauses are those which cannot be part of any MUS, and they do not even potentially help to explain the unsatisfiability of any US. A useful preprocessing technique, especially if we are interested in finding more than one MUS, is to prune away all unusable clauses by finding and only operating on the lean kernel. A reasonably efficient method of extracting the lean kernel is presented in the following section.

2.4.3 Autarkies and Autarky Reduction

The most advanced current theory of minimal unsatisfiable subsets, chiefly developed by Oliver Kullmann in a series of papers culminating and summarized in [15], builds on the notion of *autarkies*, which can informally be viewed as partial assignments that satisfy all the clauses they touch. Autarkies are a central concept for formalizing redundancies, and can

be used to detect clauses which are in some sense independent of an infeasibility. Some very basic concepts and results about autarkies, along with the particularly useful application to lean kernel extraction, are introduced in this section.

2.4.3.1 Autarkies

Definition 2.4.7. *(**Autarkies and autark subsets**) A partial assignment φ is an **autarky** of an unsatisfiable clause set F if any clause $C \in F$ which contains variables assigned by φ is satisfied by φ. A subset $F' \subset F$ is called **autark** if there is an autarky φ such that $F' = F \backslash (\varphi * F)$, where $\varphi * F$ denotes the result of applying φ to F.*

Autarkies are closed under composition, giving rise to a submonoid *Auk* of autarkies in the monoid of partial assignments. On the elements of this **autarky monoid**, a partial order can additionally be defined through the subset relation. The maximal elements of this partial order are called **maximal autarkies**. In general, there can be more than one maximal autarky for a clause set F.

Dually, we can see the autark subsets as a partial order via the subset relation. The empty set is an autark subset of F, and for autark subsets F_1 and F_2, the union $F_1 \cup F_2$ is again autark. This implies that there is a unique largest autark subset of F, which can be written as $F \backslash (\varphi * F)$ for any maximal autarky.

2.4.3.2 Finding Maximal Autarkies

Building on his previous work on autarky theory, Oliver Kullmann [28] introduced a simple algorithmic approach to determining a maximal autarky of a clause set. For a set of variables V and a clause set F, the operation $V * F$ in the pseudocode is executed by removing from all clauses in F all positive or negative occurrences of any variable from V, and $\bot$ represents the empty clause. Note that $V * F$ is not unit propagation, since literals of both polarities are crossed out, and clauses are only removed if they become empty in the process.

Algorithm 5 Maximal Autarky Extraction

Input: an unsatisfiable SAT instance $F = \{C_1, \ldots, C_m\}$ in CNF
Output: the maximal autarky φ for F

```
⟨res, proof, φ, units⟩ := sat(F, {})
while (res = unsat) do
    V := used_vars(proof)
    F := V * F \ {⊥}
    ⟨res, proof, φ, units⟩ := sat(F, {})
end while
return φ
```

The formal proof of the algorithm's completeness and correctness relies on the comprehensive theory elaborated by Oliver Kullmann [29]. Only some core observations can be discussed here for reasons of brevity. The first is that if by repeatedly crossing out the resolution variables used in a proof, we arrive at a satisfiable clause set, then any partial assignment which only touches the remaining variables will be an autarky, so that the algorithm can only return an autarky. Secondly, Theorem 3.16 in [29] says that F has no autarky left iff every clause left in F can be used for some resolution refutation of F (i.e. every clause is usable), showing that the result is maximal.

A very interesting alternative approach to finding a maximal autarky was presented by Mark Liffiton and Karem Sakallah [30]. Their algorithm frames the task as an explicit optimization problem in the space of partial assignments, and relies on a clever instrumentation

scheme to give a SAT solver the ability to enable and disable both clauses and variables during standard search. A maximal autarky is then extracted using AtMost constraints for the assignment size as a sliding objective for optimization. This approach is reported to outperform Kullmann's algorithm, but it arguably is a lot more challenging to implement, and autarky extraction is not time-critical in the context of this thesis.

2.4.3.3 Reduction to the Lean Kernel

A very important characterisation of the lean kernel $N_a(F)$ is that it is the complement of the unique largest autark subset of F. Recall that the lean kernel contains all the usable clauses. Again, Theorem 3.16 [29] shows that the set of all unusable clauses $F \backslash N_a(F)$ is identical to the largest subset that can be pruned away via autarky reduction, i.e. the largest autark subset. This result means that we can reduce any unsatisfiable clause set to its lean kernel by finding a maximal autarky φ and simply applying it. Note that by definition, the application of an autarky prunes away all the clauses which contain variables it assigns, so that we do indeed get a subset of unmodified clauses.

The only available implementation of the maximal autarky extraction algorithm builds on Kullmann's OKsolver [31], a highly experimental system which is distributed as part of a giant software package with so many dependencies that it is very difficult to get to cooperate with current versions of various system libraries. For the purposes of this thesis, autarky reduction was therefore implemented from scratch on the basis of the customized MiniSat variant provided by Christian Zielke.

Concerning the practical relevance of autarky reduction, Liffiton & Sakallah [30] show the value of reducing to the lean kernel as a preprocessing step before starting any MUS extracting algorithm. On the other hand, Marques-Silva & Lynce [24] state that in their experiments, autarky reduction on candidate sets at later stages of the MUS extraction was not effective enough to warrant the additional investment of computation time, especially since the much cheaper clause set refinement already reduced the largest part of these autarkies.

Despite the limited practical relevance of autarky theory, the handle to automated clause classification that it provides gives us some valuable measures for describing structural properties of unsatisfiable clausal SAT instances. In general, it is possible to say that the MUS extraction task becomes the more interesting and demanding, the more the lean kernel, the set of potentially necessary clauses, and the set of necessary clauses differ in size. These three sizes are therefore interesting properties of benchmark instances. Kullmann et al. [27], evaluate a set of MUS benchmarks derived from automotive product configuration data [32] with respect to these and other figures. In the instances, the lean kernel is often substantially larger than the set of potentially necessary clauses, but the set of necessary clauses is not much smaller than that set. This implies that the MUSes in the instances are all of similar size, and they overlap to a very high degree. Still, the sometimes significant difference in size between the smallest MUS and the largest MUS makes these instances an interesting and popular test case for approaches which are interested in considering more than one MUS, and we will use them as examples for the discussion of interactive MUS extraction.

Chapter 3

Interactive MUS Extraction

3.1 Motivation

The MUS extraction algorithms introduced in the previous chapter have so far only been implemented as console tools which can be configured to some degree via parameters, but do not offer any possibilities for user interaction in the middle of the search process. The existing approaches to influencing the MUS extraction process have confined themselves to specialized heuristics which extract some relevant set of MUSes. Good examples of such approaches are the method presented by Grégoire, Mazure & Piette [33] for determining inconsistent covers, and the work by O'Sullivan et al. [34] on generating representative sets of explanations.

The central idea of this thesis is that the search process itself should be made more dynamic, enabling specialists to use much of their domain knowledge already while extracting MUSes as small explanations of infeasibility. In hardware verification, for instance, in a situation where the descriptions of several different hardware components are part of an US, an error diagnosis specialist will have intuitions which component is more likely to have contributed to the error at hand, and will therefore find it useful to be able to influence which of the descriptions is thrown out next in a deletion-based MUS extraction run. Note that when a single MUS is to be extracted, this does not only influence the speed of the reduction, but depending on the specialist's decisions, we might end up with vastly different unsatisfiable cores, one of which might be much more straightforward to interpret than the other.

The core idea of **interactive MUS extraction** therefore is to make a deletion-based MUS extraction process transparent and controllable by the user. First and foremost, this means that the user will be able to select which of the clauses (or clause groups) in the current US the algorithm should try to remove next. To be able to make informed decisions, the user needs to be provided with as much information as possible about the structure and the content of the current US at any point.

Moreover, especially for very complex problem instances, the user should be able to revert decisions that led to unwanted consequences without having to restart the reduction process. For instance, it often happens that by some selected reduction, some other relevant part of the US falls away as well, although the user intended to arrive at a MUS containing that part. Thus, the user should at any time be able to explore alternative paths through the powerset lattice. In this chapter, these considerations lead to the development of a system where multiple USes can not only be stored and explored, but also interactively reduced in parallel. Subsequent chapters then build on this novel infrastructure for interactive reduction.

3.2 Basic Architecture

This section gives an overview of the software architecture on the basis of which the prototype of the interactive MUS extraction system was implemented. The first part of the section motivates the choice of an existing software framework chiefly written by the author as the basis of the implementation. The second part describes how the interface to MiniSat was implemented, whereas the final and most essential part of the section describes the ways in which US reductions and the information about critical clauses are stored and processed within the system. Taken together, these parts of the software architecture form the machinery behind the graphical interface presented in Section 3.3.

3.2.1 The Kahina Framework

The **Kahina framework** [35] started out as an interactive graphical debugging environment for logic programming in Prolog, with special focus on the needs of symbolic grammar engineering, a branch of computational linguistics which deals with the development of formal models of (fragments of) natural language syntax. The main purpose of Kahina is to provide a grammar engineer with graphical visualizations of parsing processes in the form of explicit control flow visualizations. Kahina's main asset is its comprehensive support for post-mortem inspection of computation step details. Kahina is written entirely in Java, and relies on vendor-specific interfaces for communication with different Prolog implementations. The graphical components build on the Swing library. Kahina's software architecture is designed with a special focus on modularity and extensibility, and has already proven to be efficiently integrable with several different grammar implementation systems [36].

Kahina suggested itself as the basis for the prototype implementation because it already provided many relevant view components, such as visualization components for trees and directed acyclic graphs, as well as a simple abstraction layer for defining complex graphical user interfaces. The conceptional similarity between post-mortem inspection of a control flow graph and interactive inspection of a lattice of unsatisfiable clause subsets led to the advantage that large chunks of existing Kahina-based code could just be reused, and that only a few minor extensions to Kahina's core package became necessary for better support of concurrency and view update management. Another very relevant reason for choosing Kahina over some other more well-established toolkit was the author's intimate knowledge of Kahina's internals, acquired in three years of work as the main developer and maintainer, which greatly reduced the time that would otherwise have been needed for exploring and becoming familiar with a new codebase, and left more time for experiments.

In the first stage of the project, the Kahina system was extended by additional data types for clause sets, variable assignments, propositional non-CNF formulae, and refutation proofs. Additional new components include input and output functionality for DIMACS files, a text format for propositional formulae, and the proof format used by CoPAn [37]. A few new visualization components were developed as well, such as a fold-out tree view for refutation proofs, and a general graph viewer for inspection of clause graphs and variable graphs, although the latter did not make it into the final system because these visualizations turned out to be not very helpful for instances of any interesting size.

Another strong point of the Kahina framework is its support for automatization of user interactions. In a Kahina-based debugging system, tracing commands can be automatized via patterns over step properties. In the Prolog case, such patterns are used to automatically skip over the execution details of well-tested predicates, or to emulate the breakpoint mechanisms of debugging environments for other languages.

3.2.2 Interface to MiniSat

The basic interface for single steps of a deletion-based MUS extraction algorithm was implemented after the model of Martin Lahl [21]. The input and output for each MiniSat call is exchanged via temporary files. While reading the problem from a file for each reduction seems wasteful at first, the freeze variable mechanism allows the system to only write the instance extended by selector variables to disk once, only manipulating much smaller freeze files afterwards. MiniSat's high speed at reading from files does the rest to make this simple approach reasonably fast.

While Lahl's original version only had to support one mode for calls to MiniSat (reading in the extended input file as well as the freeze file and printing out the result and the proof into new temporary files), in the new architecture MiniSat is called in more than just one scenario. For the implementation of autarky reduction, only the output of resolution variables was needed, and for an application we shall see in Chapter 4, the output of a file containing all the derived unit clauses was required. Unlike in Lahl's code (where the SAT solver calls were directly executed within the program's main loop for simplicity), all the methods interacting with the custom MiniSat variant were encapsulated into an auxiliary class which provides wrapper methods for the different types of solver calls.

A slightly more complicated issue has been the support for parallelism. A major goal for the new architecture was to support the distribution of SAT calls over several processors, allowing multiple branches to be explored in parallel. For this purpose, SAT solver calls in the context of US reduction where wrapped into a `USReductionTask` class which implements Java's `Runnable` interface. All instances of `USReductionTask` can therefore be started and executed as worker threads. For the MiniSat interface, the only necessary change was to keep the temporary files for each SAT solver call separate, which was solved in the trivial way by adopting a naming scheme that includes a numeric ID which uniquely represents the calling `USReductionTask` instance. The much more interesting problem of coordinating the access of different reduction task threads to a common data structure for storing reduction states will be discussed later in this chapter.

3.2.3 Representing Reduction States

When used as the basis for a debugging system, Kahina adapts the standard view of a computation as a set of computation steps connected by a control structure. In the case of imperative programming, this control structure is a call tree. For declarative programming with backtracking, we get an additional search tree structure over the computation steps. In any paradigm such as dynamic programming where partial results are stored and reused by later steps, the call tree effectively becomes a directed acyclic graph (dag).

As already suggested in the introductory remarks for this chapter, the USes encountered on reduction paths are the obvious choice for defining the meaningful steps of a MUS extraction process, and were therefore chosen as the primary data points. Since every US is a unique subset of the original clause set, it can be represented and uniquely identified by a set of clause indices representing the clauses it contains. In the implementation, all this reduction step information is stored in instances of a `MUSStep` class inheriting from the basic `KahinaStep`, which gives the reduction steps and thereby the encountered USes the status of primary information units in Kahina. This status has the advantage that view components for displaying the contents of the currently selected US were very easy to write and straightforward to integrate on the basis of existing application code.

In the case of US reduction, a deterministic deletion-based MUS extraction algorithm will only create a linear structure of US states connected by the subset relation. But as soon as

we allow the exploration of multiple deletion alternatives, the structure of the encountered USes will branch out into a tree. Since there can be different sequences of deletions leading to the same US, the structure is a dag which can semi-formally be defined like this:

Definition 3.2.1. *(**Reduction graph**) Let F be an unsatisfiable clause set. A **reduction graph** $R = (V, E)$ for F has as its vertices a set of unsatisfiable subsets $V \subset 2^F$, and contains an edge $(V_1, V_2) \in E$ iff V_2 was the result of removing some non-critical clause from V_1, possibly followed by clause set refinement or autarky reduction.*

The fact that by this definition, for any $(V_1, V_2) \in E$ we have $V_1 \supset V_2$ and thereby $|V_1| > |V_2|$, implies that the reduction graph is indeed acyclic, and the deterministic nature of the SAT solver we are using (if it is run with the same seed on the same machine) implies that no two different children can be reached by the deletion of the same non-critical clause, meaning that for any US node V, we have at most one outgoing edge for each non-critical clause in V.

These properties motivate our choice of the basic data structure for storing the edges of the reduction graph. A very compact way to store all the relevant information about a US and the deletion attempts which were made on it was found to be a **reduction table**, a mapping of clause IDs into integers which represent the reduction graph edges or **reduction links** to other USes. The most important operation on this data structure is the addition of a new reduction link. In the case of a successful deletion of a clause C in a US V_1 which is thereby reduced to a smaller US $V_2 \subseteq V_1 \backslash \{C\}$, the step corresponding to V_2 is retrieved from the step database, or added to the database if V_2 has not been encountered so far. The unique step ID for V_2 is then added to the reduction table of V_1 as the value under the ID of the deleted clause C.

As the search space is being explored, the integer-based link structure between the various US steps grows to represent an ever larger reduction graph of US connections established by successful deletion attempts. For the efficient implementation of the reduction table, the choice of a data structure which supports thread-safe modification and scales well for a large number of concurrent operations was a central concern. Moreover, for efficient traversal, the key set of the map implementation needed to provide an iterator which enumerates all the clause IDs for each entry in ascending order. A good solution for these requirements was found in the `ConcurrentSkipListMap` class from the `java.util.concurrent` package, the recommended concurrent analogue to `TreeMap`, which is not thread-safe.

3.2.4 Storing and Propagating Reducibility Information

While the way in which the reduction graph is stored and managed has already been described, we have not yet considered the question how to store information on critical clauses, and how to distribute reducibility information throughout this dag, given the observation from Section 2.3.1 that criticality is downward monotonic, and irreducibility is upward monotonic in the powerset lattice.

The very efficient way adopted in the prototype to encode the information that a clause was determined to be critical in the reduction table of some US is to simply set its reduction link to -1. This special value can be seen as the ID of an imaginary node in the reduction dag which represents all satisfiable subsets. Whenever a reduction link is set to -1, by the monotonicity of criticality we can propagate this value down to any subset of the current US, which includes all its descendants in the reduction graph. Whenever a new link is created in the reduction graph, it then suffices to simply propagate all the criticality information from the parent to the child. Note that not all nodes representing subsets of some US will necessarily be reachable as descendants of that US in the reduction graph. This is an instance of a more general problem which will be resolved in Chapter 4.

The same general idea about propagation applies to any information about clauses found to be non-critical in some US. This information can be handed up to all supersets of the current US, the only question being what the propagated reduction links in higher nodes are supposed to point to. Unlike in the case of criticality, there is a marked difference between the information that some clause was found to be non-critical, and knowing and storing in which lower node MUS reduction will end up when trying to reduce that clause at that particular US. Not modelling the difference between what we will call **explicitly reduced** and **fall-away** clauses would be contrary to the design goal of making the entire search space accessible, as some parts of the powerset lattice might become unreachable if we simply link each clause that fell away during reduction to the reduced US everywhere. The right strategy is of course to only propagate fall-away information for a non-critical clause, no matter whether it fell away or whether it was explicitly reduced.

To differentiate between the two cases, -2 was defined as an additional value to be stored in the reduction tables, indicating that that clause is merely known to be a fall-away clause. Reducing such a fall-away clause will possibly lead to a new unexplored reduction state, giving a strong reason for the user interface to allow just that.

3.3 User Interface

With the decisions on the data structures for interactive MUS extraction made, we can now turn to the question how this information is best presented to a user, and how the desired interactivity can be implemented in an intuitive way. The answers given to these questions by the current prototype are the subject of this section. After a concise statement and motivation of the central ideas, the relevant view components and interaction possibilities of the prototype are discussed in turn, and a few examples of their usage are given.

3.3.1 Overall Design

The role of the reduction graph as the central place of storage for the current knowledge about the search space motivates the idea of also using a visualization of this data structure as the main orientation point in the user interface. One of the prerequisites of interactive MUS extraction is that the user should be able to jump to any part of the search space and continue to explore it from there at any time. This free access to the search space can be provided by allowing the selection of nodes in a visualization of the reduction graph. The unsatisfiable subset associated with the selected node would then be displayed in a second view component which also visualizes the contents of that node's reduction table.

To execute reduction attempts, the user should be allowed to interact with individual representations in the display of the current US. The selection of some US in the reduction graph and of some clause in the US display is already enough to define the next reduction attempt which should be performed under the hood by a SAT solver call. The newly derived reducibility information as well as the potential new US are then used to update the corresponding views, giving the user visual feedback about the reduction attempt's outcome and preparing the system for the next user interaction.

The general setup of the graphical user interface for interactive MUS extraction as presented so far is simple enough to stay intuitive, but additional visual complexity might easily detract from this. More advanced functionality such as the possibility to exploit model rotation or autarky reduction is therefore best hidden in a second interaction layer only accessible via context menus. These context menus can also make available lesser-used options e.g. for influencing the display properties of the two view components.

A second observation which has turned into one of the guiding ideas for the workflow of the prototype is that there tend to be exploratory phases, in which many user-defined reduction attempts are executed in a rather uninformed fashion. Confronted with a corner of the search space about which nothing is known yet, a user will usually just walk into it in a random direction for a while, which is often the best way to gain intuitions about the local structure of the search space. However, this type of uninformed exploration is not optimally supported by an interface which requires the user to manually execute every single exploration step, especially in a situation where reductions can be executed much faster than the user can enqueue them. Allowing some type of interplay between single reduction attempts which are manually executed by the user and much faster automated reduction attempts started by a simple algorithmic specification promises to increase overall productivity.

For representing clause sets such as in the US view, a `JList`-based format was chosen because this Swing component can display very large sets of list items without requiring expensive computations. The default format for list entries consists of the clause ID (numbered according to the order in the input DIMACS file) and a set of positive and negative integers representing the literals of the clause, just like in the original DIMACS file. If the DIMACS file contains comment lines of the format `c [variable] [symbol]` between the header and the clause list, these will be imported as a **symbol table**, and the respective symbols will be displayed instead of the variable IDs. This support for displaying symbolic information instead of bare variable IDs is obviously necessary for interactive MUS extraction based on domain knowledge.

3.3.2 The Reduction Graph View

As motivated above, a **reduction graph view** was implemented to form the central component for navigating between different unsatisfiable subsets, and was designed as an explicit representation of the explored parts of the powerset lattice. By default, each node just displays information about the size of the US it represents (as a number displayed at the left of each node label), and the number of clauses about which we have no reducibility information yet (in parentheses). The colour of each node further encodes whether the corresponding subset was determined to be a MUS (red), a non-minimal US where all reduction alternatives have already been tried (green), or neither of the two (white).

Figure 3.1 shows an example of a reduction graph in the middle of a reduction process. The example instance is based on an unsatisfiable instance of an answer set programming (ASP) encoding of natural language parsing by Yulia Lierler and Peter Schüller [38], which was converted to a SAT instance using the ASP grounder Gringo [39] as well as a chain of auxiliary ASP tools developed at the Helsinki University of Technology [40].

The reduction graph in our example has a top node of size 6074, which of course corresponds to the number of clauses in the unsatisfiable SAT instance we are trying to reduce. Only for 285 of these clauses we do not yet know anything about their criticality for the entire clause set. Note that the number of clauses without reducibility information decreases on each path through the graph, reflecting the knowledge we have gained with each reduction attempt. At the bottom of the reduction graph, we see that six different MUSes have already been found, containing between 340 and 369 clauses. Just above some of these MUSes, there are nodes coloured in dark green, reflecting that these subsets have been explored in their entirety. Only for the MUSes and these nodes, we can be certain that no further branches can be created by additional reduction attempts.

Observe that between the root and the coloured nodes, there is a complex landscape of branches representing all the successful reduction choices, giving some impression of the vast non-determinism potentially involved in MUS extraction. Many nodes are still coloured in

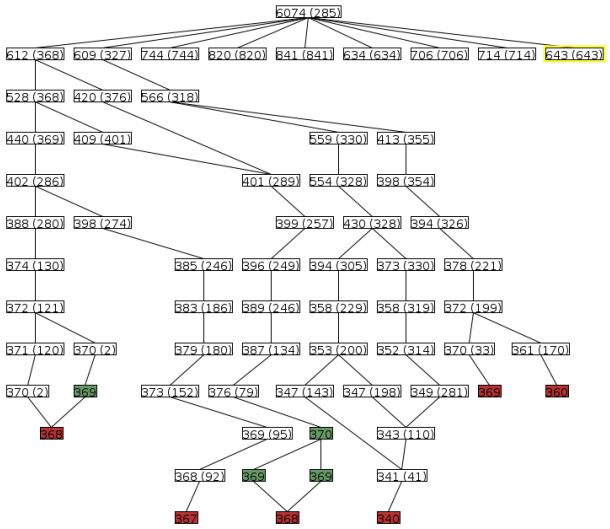


Figure 3.1: A first example of the reduction graph view.

white, indicating that there are still unexplored reduction choices left for each of them. In principle, the entire search space is made accessible via this reduction graph, and the user is able to interactively navigate and explore it by inspecting the subsets associated with nodes, and by starting reduction attempts which may uncover new corners of the search space, generating additional nodes in the reduction graph.

3.3.3 US Inspection and Reduction

The general design considerations led to the idea that whenever some US is selected in the reduction graph view, an additional **US view** should display its contents. In the current prototype, the US view is not much more than a list of clauses whose font colour encodes the respective value in the reduction table for that clause. Critical clauses are displayed in red, explicitly reduced clauses in a dark green, fall-away clauses in a lighter green, and all other clauses in black. For advanced interactions, one or more clauses can be selected, and are then highlighted by a yellow background colour.

In Figure 3.2, we see a part of the visualization of a large US in the ASP example. In the visible section, the three clauses with the IDs 640, 641, and 646 have already been determined to be critical, either by unsuccessful reduction attempts from this node, or from some superset of the current US. Successful reduction attempts have already been made for the clauses with the IDs 648 and 650. Except for clause 653, all other clauses have already fallen away during some successful reduction, so we know that these are not critical, although deleting them from the current US might lead to new previously unseen USes. Clause 653 is the only clause in this section about which we do not have any criticality information yet, and would therefore be a typical target for the next reduction attempt.

Apart from displaying the unsatisfiable subset currently selected in the reduction graph, the US view is also responsible for providing the interactive reduction functionality. The most direct method of starting a reduction attempt is to double-click on one of the clauses coloured in black or light green. On a US consisting of not more than a few thousand clauses, the underlying SAT solver call is so fast that the result will appear almost instantly. If the SAT solver finds a model, meaning that the reduction attempt failed, the clause will be coloured in red, and the criticality information will be propagated downwards in the reduction graph.

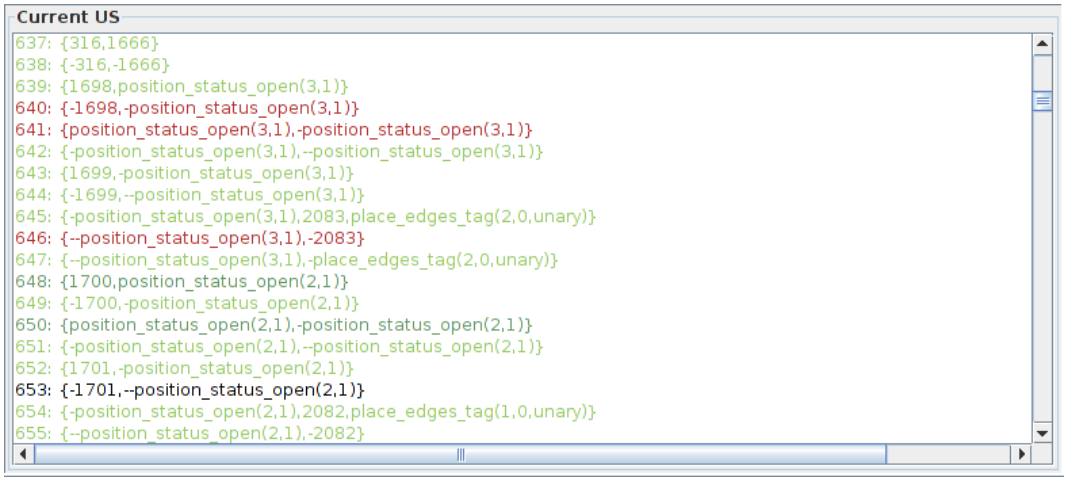


Figure 3.2: A first example of the US view.

If the reduction attempt was successful, the clause itself will receive a dark green colour, most often along with a few other ones falling away and switching to light green as a result of clause set refinement. In any case, the newly derived uncriticality information will be propagated upwards, and a new link will appear in the reduction graph, either to a previously existing node (so there are now multiple paths to some US) or to a new node, indicating that a novel US was found and made accessible via the reduction graph.

To illustrate the development of the reduction graph during interactive MUS extraction, for reasons of compactness we do not use the ASP instance previously used, but take a simple example instance from the Daimler test set [32] instead. Like some instances in that test set, it has more than a single MUS, and is thus complex enough to illustrate all the central mechanisms of interactive MUS extraction. In Figure 3.3, we see the state of the reduction graph and the selected US before starting our reduction attempts. Note that a MUS of size 93 has already been found at this stage, and that we are currently exploring an alternative branch of the reduction graph. About this branch, not much is known yet, as we have no information about the status of 94 of the 97 remaining clauses. For this reason, all the clauses in the visible section of the current US view are still coloured in black.

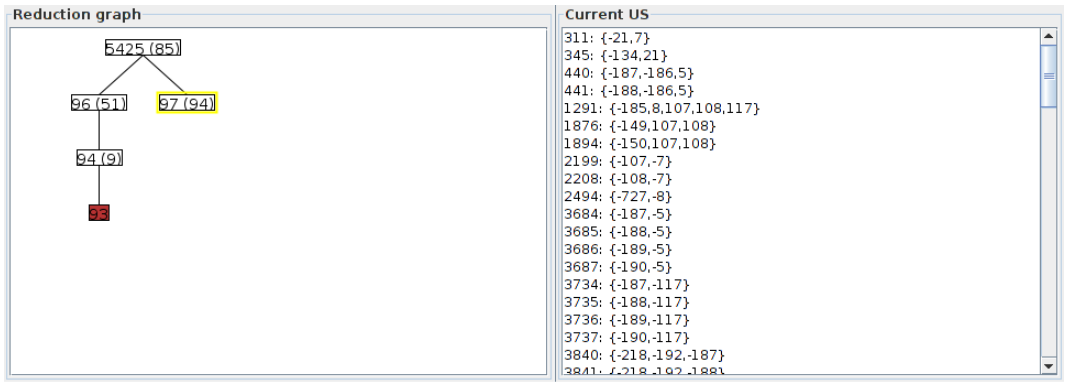


Figure 3.3: Display before the example reductions.

Let us first look at the result of an unsuccessful reduction attempt in Figure 3.4. After the user attempted to reduce the clause with ID 1291 by double-clicking on it in the current US view, the clause has changed its colour to red, indicating that the SAT instance generated and solved under the hood was determined to be satisfiable, and that the clause we tried to delete is therefore now known to be critical. Note that the counter for clauses of unkown status in the selected node of the reduction graph reflects this by its decrease from 94 to 93. Since the criticality information can only be propagated downwards, nothing else has changed in the reduction graph.

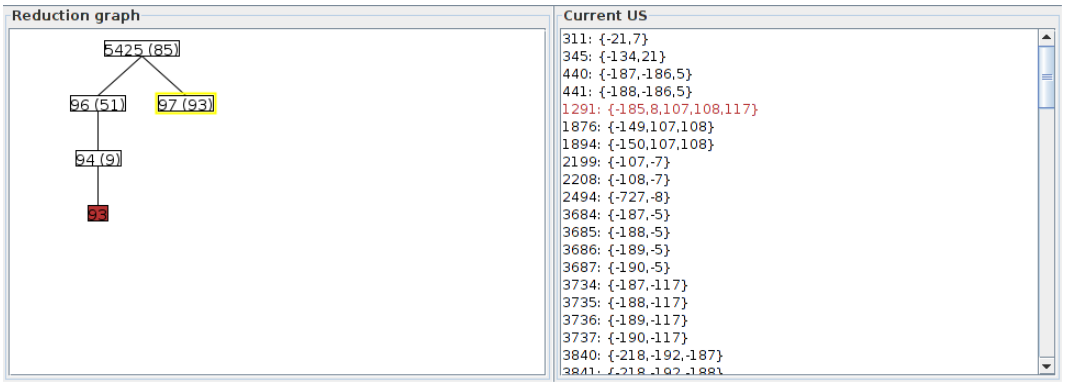


Figure 3.4: Display after the unsuccessful example reduction.

This is different in the case of a successful reduction attempt like the one we see executed in Figure 3.5. Here, we attempted to reduce the clause with ID 311, and as we can see, the attempt was successful, resulting in a new node in the current branch of the reduction graph. The new node is of size 94, showing that along with clause 311, two other clauses from the old US have fallen away. When a new node is added to the reduction graph during interactive reduction, that node is automatically selected to allow for quick manual exploration of a new branch. In the new node, we see that clause 1291 is already known to be critical, so this information has been propagated downwards from the old US. The old US has gained the information that three of its clauses are not critical, causing the decrease of the number of clauses of unknown status from 93 to 90. Note that the number of nodes of unknown status in the top node has not decreased, which means that the upwards propagation of fall-away information did not yield any new knowledge in the top node. The reason for this must be that the same clauses have already been successfully deleted on the other branch, indicating an overlap between the already determined MUS and the one we are currently moving towards. In Chapter 5, we will develop an additional view component to make this kind of overlap information much more explicitly visible.

The advantage of the explicit US view is that it can be used to provide more functionality than just the execution of single reduction attempts. Some more powerful operations which act not on a single clause, but on subsets of the current US, were made accessible via the US view's context menu. The first of these options offers what could be called methods of **semi-automatization**. In essence, it allows the user to initiate a batch processing of reduction attempts, where all the clauses which are currently selected in the US view are reduced in turn. The main application of this is to quickly open up several new branches in the reduction graph at once, or to speed up criticality checks. This option is called semi-automatization because we will be introducing a much more powerful form of automatization in the last section of this chapter.

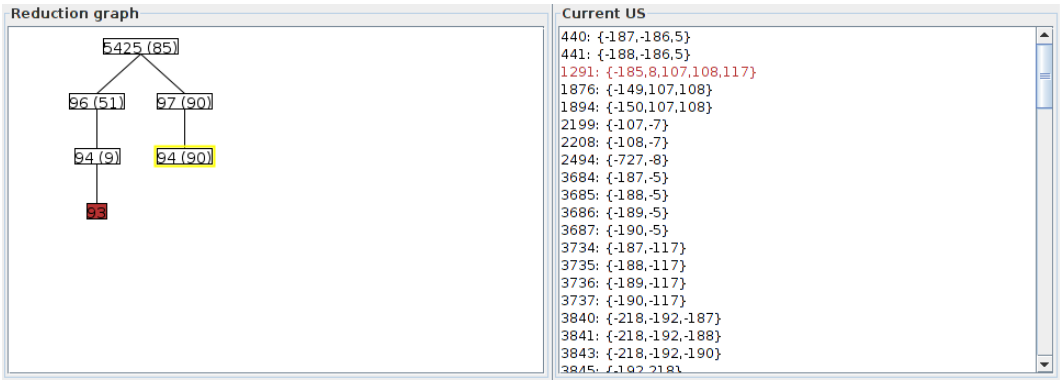


Figure 3.5: Display after the successful example reduction.

The second option is called **simultaneous reduction**, an obvious generalization of the default deletion-based method where multiple clauses can be deactivated at once by setting their selection variables. The corresponding context menu option causes the system to attempt the deletion of all the currently selected entries in the clause list at once. If the attempt was successful, a new node (or link) will appear in the reduction graph, just like in the case of deleting a single clause. The major difference is that after a simultaneous reduction, none of the clauses in the old US will receive the status of an explicitly reduced clause, again to avoid the unaccessibility of possible intermediate USes in the reduction graph. If a simultaneous reduction attempt fails, no criticality information can be derived, because it is possible that some of the clauses could have been deleted as long as other would have stayed in. Simultaneous reduction will become a very helpful technique in Chapter 5, when additional information about clause groups which belong closely together will be made visible in an additional view.

The support for a semi-automatic or simultaneous reduction of clause subsets immediately leads to the question how interesting subsets can be selected as efficiently as possible. In any case, the interface needs to provide more than just the option of manually selecting chunks of clauses in the list. The basic paradigm in which this is achieved in the current prototype may be called **selection refinement** or **subselection**. Starting from some selected subset of the clauses in the current US, the advanced selection interface of the current version gives the user the possibility to subselect only the clauses of a given status, the clauses with a given number of literals, the clauses containing a given literal, just the first or the last few clauses of the selection, or a random subset of a given size. All these options are available through a hierarchy of submenus in the US view's context menu.

To demonstrate the use of this advanced selection interface, and to illustrate the interaction possibilities arising from it in combination with semi-automatization, we will conclude this section with a small workflow example. Having already found a first MUS for the ASP example instance, assume that we want to explore a few alternative paths which start from a node in the middle of the reduction graph. To do this, we want to semi-automatically reduce some of the clauses of yet unknown status in the selected US. The first step is to select all the clauses via the corresponding option in the US view's context menu.

Figure 3.6 shows how we can use the subselection mechanism to refine this selection to all clauses of unkown status. In Figure 3.7, this first subselection step has been executed, so that the fall-away clauses in the visible part of the US view have lost the yellow background colour which previously marked them as selected. In the reduction graph, we can see that

our current selection now contains 345 clauses. Since we only want to explore a few additional paths, semi-automatically reducing by all these clauses would be a bit too much. Instead, we first apply another subselection command. We could choose to subselect either the first few or the last few clauses, but this is likely to be a low-quality sample of the overall variety in alternative reduction paths. Instead, in Figure 3.7 we opt for a random selection of twenty clauses of unknown status.

If we now choose the option of reducing the selected clauses individually in the context menu, the system will perform a batch processing of reduction attempts for each of the twenty selection clauses. In Figure 3.8, we see the results of this semi-automated reduction step. The twenty different selection attempts have led to five additional branches in the reduction graph, each representing a US of a different size. Twelve additional clauses have been determined to be critical, as we can see in the decrease of the number of clauses of unknown status in the old branch. Only $(345 - 304 - 12) = 29$ additional clauses have fallen away in the five new branches together, suggesting a high overlap between the new USes. Altogether, after this systematic and goal-directed semi-automatized exploration of but a few new branches we have gained quite a bit more knowledge about the search space and thereby about the nature of our ASP instance.

3.3.4 Model Rotation and Autarky Reduction

Model rotation and autarky reduction (as defined in Section 2.4) were both implemented based on the adapted version of MiniSat. In compliance with the general design considerations, these two advanced methods were also made accessible as operations via the context menu of the US view.

Note that autarky reduction as well as attempts to delete multiple clauses at once may create links in the reduction graph which are not associated with any attempt to reduce a specific clause, and are therefore not represented in any reduction table. This is the main reason why the reduction graph view is not updated directly from the reduction tables, but the reduction graph is maintained in a separate data structure.

In Figure 3.9, as an example of the benefits of model rotation, we repeat the unsuccessful reduction attempt of Figure 3.4, only this time choosing via the context menu to enhance this reduction step by model rotation. In the visible part of the US view, two additional clauses apart from the clause we attempted to reduce are now displayed in red, and the numbers on the nodes of the reduction graph tell us that eight clauses in total have been determined as critical. Situations where by a single user interaction and a single call to a SAT solver, quite a few clauses are determined to be critical by model rotation, are not

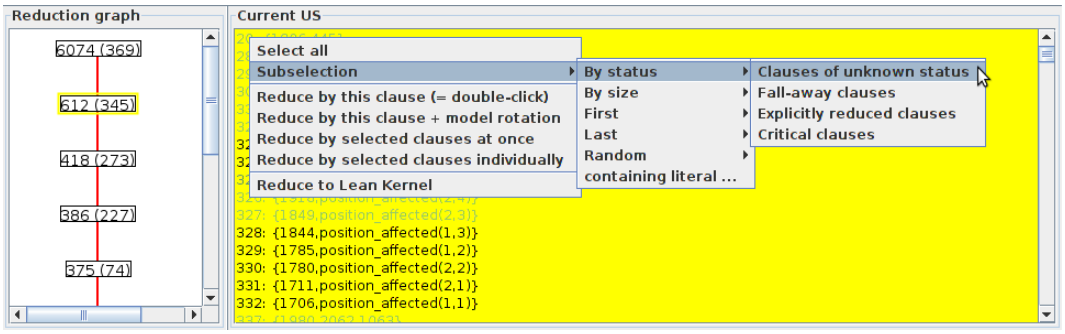


Figure 3.6: Subselecting all clauses of unknown status in a US.

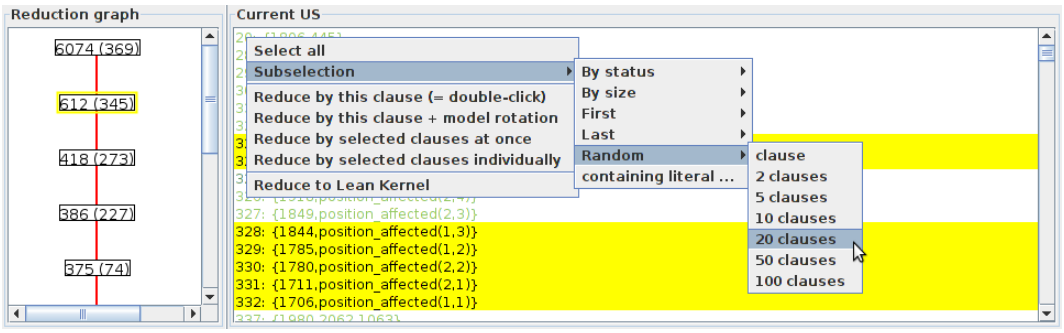


Figure 3.7: Subselecting twenty random clauses of unknown status.

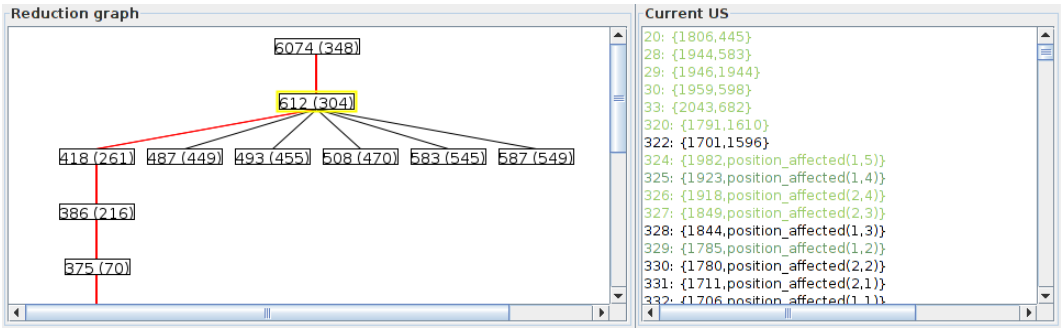


Figure 3.8: The result of semi-automatically reducing by the selection.

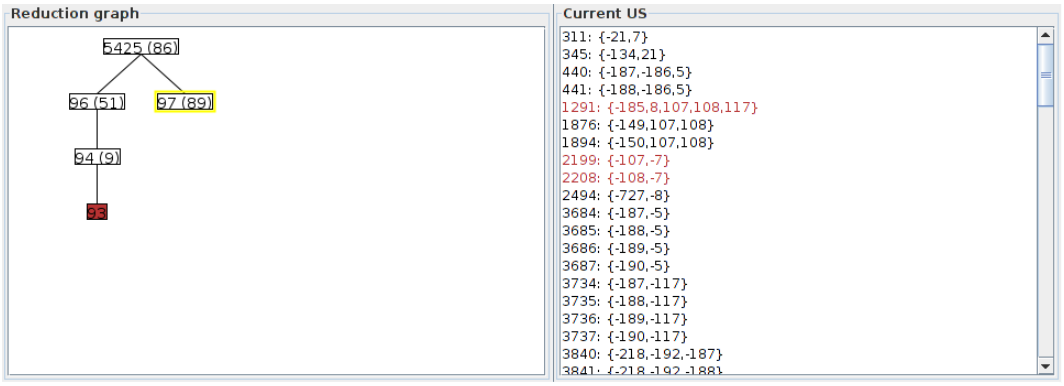


Figure 3.9: Display after unsuccessful example reduction followed by model rotation.

at all uncommon. The technique is thus not only a very worthwhile enhancement of any deterministic MUS extraction algorithm, but is also very relevant for interactive reduction, because the otherwise tedious process of testing a large number of clauses for criticality, especially in the last phase on the interactive reduction process where we have already closed in on some specific MUS, can be tremendously accelerated.

In Figure 3.10, we see on the left the result of applying lean kernel extraction (see Section 2.4.3.3) to our Daimler test instance, and the result of a random successful deletion attempt followed by clause set refinement on the right. If our main goal is to find a small MUS as fast as possible, clause set refinement is clearly superior, since it immediately reduces the instance to an US of size 96, whereas the lean kernel still contains 1627 clauses. On the other hand, remember that clause set refinement boils down the instance to only those clauses which were needed for a single arbitrary proof, whereas lean kernel extraction gives us all the clauses which can be used in some refutation proof, thereby producing a pruned instance which still subsumes all the MUSes of the original instance. Depending on the scenario, both approaches to clause set pruning may be useful. If we are interested in several or even all MUSes, lean kernel extraction can be a very valuable preprocessing step, whereas if we want to quickly find a way towards the transition boundary, clause set refinement is a lot more helpful.

3.4 Automated Reduction

As mentioned in the general design considerations, a user will often want to quickly explore some part of the search space without having to manually execute hundreds of reduction attempts, especially in contexts where domain knowledge has not yet become relevant, so that the manual reduction attempts would be executed in a very mechanistic way. Making the semi-automated reduction of selected subsets possible has already somewhat alleviated this problem, but even these more abstract reduction steps tend to become rather repetitive and predictable for large problems.

Guided by the general observation that repetitive tasks are always good candidates for automatization, this section introduces an automated reduction mechanism in the form of reduction agents which freely interact with user-controlled manual interactive reduction, and presents and discusses some of the implementation details. As we shall see, the reduction agent approach has the additional benefit of allowing the user to define and observe the behaviour of standard MUS extraction algorithms in the form of pluggable deletion heuristics.

3.4.1 Reduction Agents & Parallelized Architecture

The very general approach to fully automated reduction implemented in the prototype relies on the concept of a **reduction agent** which in essence acts like an autonomous additional

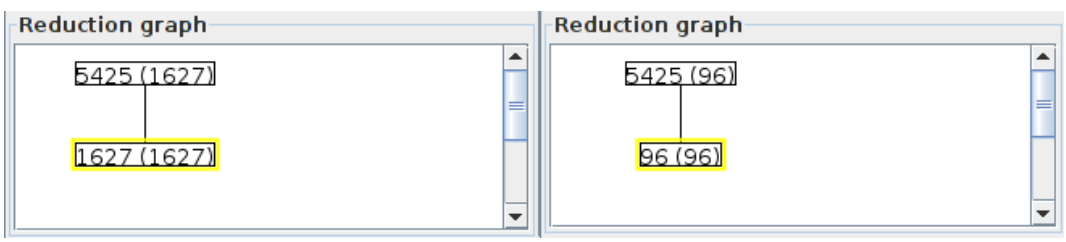


Figure 3.10: Clause set pruning by lean kernel extraction and clause set refinement.

user who was given a set of simple instructions. A reduction agent issues a sequence of reduction commands, receives back information about the success of these attempts along with data about the reached USes, and hands on its reduction results to a Kahina state object which administers and regulates the access to a shared reduction graph. We shall later see that by plugging in different instruction sets, agents can be programmed to behave just like standard deletion-based MUS extraction algorithms.

The reduction agents are the reason why we have been referring to concurrency issues throughout this chapter. The architectural challenge was to build a stable system which can handle a number of threads operating on a common data structure, while at all times staying responsive to user input in the form of manual and semi-automatic reduction commands. The resulting parallel system architecture is displayed as a schematic architecture graph in Figure 3.11. The boxes which represent separate threads or processes have a grey background, whereas the few shared data structures are kept in white. The reduction work is done by a pool of reduction threads which are managed by a common controller class. This controller receives reduction task statements from the user as well as from a number of reduction agents which were started by the user and work their ways through the powerset lattice in parallel. The new reduction tasks are queued and distributed to a number of reduction threads. Each of these threads creates its own freeze file and issues a system call to MiniSat, waits for the completion of the MiniSat process, reads in the result file and/or the proof file, and extracts the relevant information. The information whether the executed reduction was successful or not is handed back to the issuing reduction agent, and the newly derived reducibility information is added to the reduction graph.

This last stage is the most problematic for concurrent processing, since the modification operations on the reduction graph are not atomic. This does not only easily lead to consistency errors when multiple reduction threads try to add their data at the same time, but it is also relevant for the GUI update thread which reads out the reduction graph from time to time in order to update the reduction graph visualization. If the graph structure is changed while it is being traversed for the view update, this can cause the GUI update thread to encounter an exception and die, which in turn causes the user interface to freeze. The only way to prevent this issue was to be very generous with locks on the reduction graph data structures. This prevents the mentioned inconsistencies from occurring, but it leads to starvation and livelock issues if too many reduction threads are started at the same time. Depending on the machine, the architecture of the prototype therefore runs into problems when more than a handful of reduction agents are started. While this is unproblematic for simple use cases, the desired scalability to many more reduction agents will only be achievable with a much more involved data structure for the reduction graph, most likely involving an update queue and a technique similar to double buffering for the visualization.

To provide the user with some information about the running agents, an overview of all the active reduction agents is displayed in an additional view that was added to the bottom of

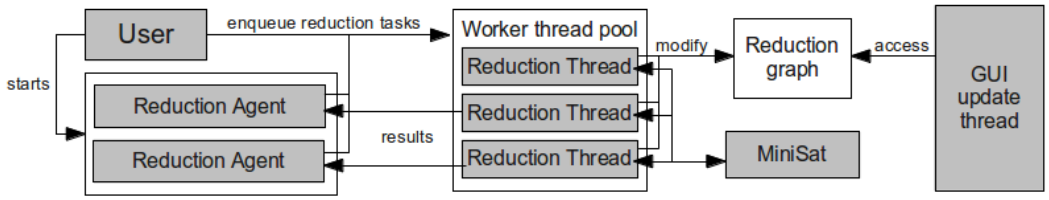


Figure 3.11: Overview of the parallel reduction agent architecture.

the main window. This view is a scrollable list of frames which compactly represent relevant information about the currently running reduction agents, each displaying information on the US the respective agent is currently operating on, as well as statistics about the number of successful and unsuccessful SAT calls. These statistics are not based on the SAT calls which were actually executed, but on the number of SAT calls the reduction agent would have had to make if there had been no information about the reduction graph beforehand. Internally, a largely unified treatment of both types of SAT calls was achieved by providing support for **simulated reduction attempts** which from the perspective of the reduction agent differ in no perceptible way from genuine reduction steps except that being mere step data retrievals, they are executed much faster.

In Figure 3.12, we see an example of the reduction agent overview in the middle of a reduction. Of the four reduction agents in the list, one has already terminated after reaching a MUS, one was terminated by the user using the Stop button in the middle of the process before reaching a MUS, and the two remaining ones are still running. The meaning of the heuristics specification, the Hide button and the coloured Change button will become clear in the following sections.

3.4.2 Agent Traces in the Reduction Graph

To visualize the actions of reduction agents in the reduction graph, one can visualize what we will call **agent traces**, i.e. the downward paths through the powerset lattice by which the different agents explore the search space. For this purpose, a **signal colour** can be assigned to every reduction agent, and the trace of the respective agent in the reduction graph will be highlighted in that colour. If several different agents happen to share a common path segment, this segment will be drawn using parallel lines with multiple colours.

In Figure 3.13, we see another example of a reduction graph for the ASP instance, this time including two different agent traces. There is a red trace on the left side of the graph which ends in a MUS, and another trace which has not yet arrived at a MUS. The thin black lines to the other nodes are the result of manual reductions which were performed without the help of the reduction agent system. Using the Hide button in the information panel of an agent, the coloured trace of each agent can be removed from the reduction graph, leaving behind only a thin black line which looks as if the reductions had been executed manually. This hide functionality is a valuable tool for decluttering the reduction graph of optically dominant details that have become irrelevant. In the remainder of this section, we will see how agent traces can be used to gain insights on the performance of different deletion-based MUS extraction algorithms.

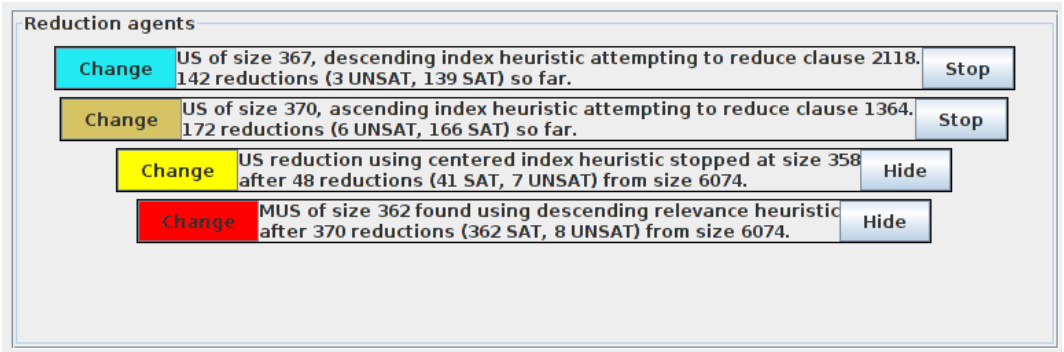


Figure 3.12: Example of the reduction agent overview.

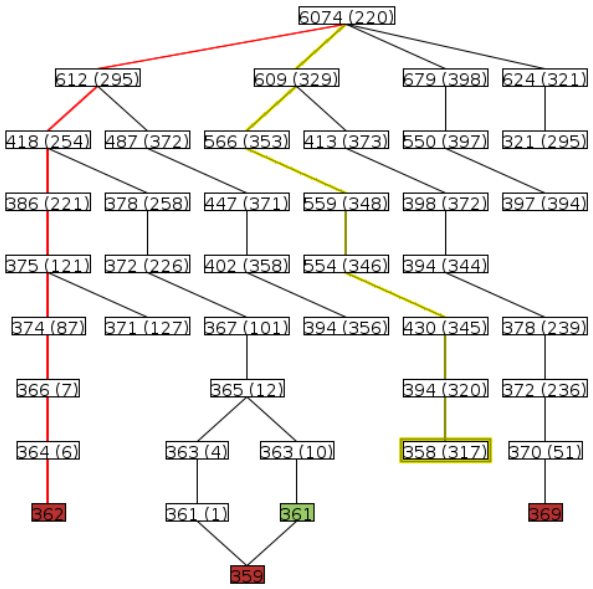


Figure 3.13: Example of a reduction graph with reduction agent traces.

3.4.3 Interface for Plugging Heuristics

When deletion-based MUS extraction was introduced in Section 2.3.2, the way in which the **deletion candidate** k is selected was not specified at all in the generic algorithm. There are of course many possible ways to define selection functions or **deletion heuristics**, which differ widely in their degree of informedness, but also in complexity. As in the case of selecting the next unit for propagation in the DPLL algorithm, there is a tradeoff between retrieving and processing as much information as possible to arrive at an informed decision, and the computation time invested into making that decision. In the case of DPLL, unit propagation is so cheap that investing time into choosing between units rarely pays off, which is reflected by the tendency of rather simple heuristics to display superior performance [41]. However, given the potentially high costs of SAT solver calls, this result cannot be generalized to deletion heuristics in MUS extraction without further investigation. It is very conceivable that intelligent heuristics will help a lot in speeding up deletion-based MUS extraction. However, a series of experiments carried out by Martin Lahl [21] in order to investigate this issue was inconclusive, providing some indication that not much can be gained by the use of intelligent heuristics in deletion-based MUS extraction either.

When developing new ideas for intelligent deletion heuristics, being able to quickly inspect their behaviour on real-world instances is an asset. To facilitate such experiments, the reduction agent mechanism of the prototype was extended by a very general Java interface named `ReductionHeuristics`. User-defined heuristics can quickly be created as Java classes implementing this interface. By default, heuristics are only given access to the contents of the current US, i.e. we simulate isolated runs by forcing the reducer to store any additional information for itself. Based on only the information about the current US and possibly information it has stored about previous USes on its path, all that a `ReductionHeuristics` implementation needs to define is some sequence of deletion candidates for reduction attempts. In any sensible heuristic, this sequence will of course have some additional properties. For instance, there will be no repetitions of clause indices, since this would imply wasted SAT calls. By convention, a deletion heuristic returns $k = -1$ to signal that it assumes to have found a MUS, which can then be tested via interactive inspection and reduction.

A handful of different simple deletion heuristics come pre-defined with the prototype. These include four of the six different deletion heuristics investigated by Martin Lahl [21], with only the ones based on proof reduction missing. The heuristics are based either directly on the clause IDs or on a relevance ordering of the selection variables based on their frequency in the last resolution proof. Both of these lists can be traversed either from top to bottom, from the bottom up, or in a spiraling movement starting in the middle of the list. Taken together, we therefore arrive at six different predefined deletion heuristics, all of which are listed in Figure 3.14. Each of these heuristics can be motivated by structural considerations, but none of them can be expected to perform better than all the others across instances.

Whenever a new reduction agent is defined, it can be set to receive its instructions from some `ReductionHeuristics` object. Figure 3.15 shows the dialog which was added for creating and starting new reduction agents. The most important option is the possibility to select one of the predefined heuristics from a drop-down menu. The signal colour for each new agent is initialized with a random point in RGB space, but it can be redefined via the Change button which allows the user to define a different colour using Swing's standard colour chooser dialog. Finally, the user is given the option to independently activate or deactivate model rotation and autarky pruning for the reduction agent.

3.4.4 Comparing the Behaviour of Deletion Heuristics

To give an example of the insights one can gain from a graphical inspection of the behaviour of different heuristics, we will now take a look at yet another example reduction graph for the ASP instance. It is interesting to observe how the predefined deletion heuristics go very different paths through the reduction graph, leading to considerable differences in the number of successful and unsuccessful SAT solver calls as well as in the size of the MUS that is found.

In Figure 3.16, we see a reduction graph which only consists of five different agent traces. All the agents were started at the top node, but each of them used a different predefined deletion heuristic. None of the agents was configured to apply model rotation or autarky reduction, and each one happened to end up in a different MUS.

In this test case, the descending relevance heuristic turned out to need the lowest number of steps through the reduction graph, as it was often able to throw away large chunks of a US. This is not surprising given that this heuristic always tries to reduce clauses which were used in many places in the refutation proof. A successful deletion of such a clause must lead to a very different proof, potentially causing many other clauses to become unnecessary and fall away. However, note that the descending relevance heuristic has also led to a rather large

ascending index heuristic	goes through the US clauses by ascending ID
descending index heuristic	goes through the US clauses by descending ID
centered index heuristic	spirals through the US clauses starting in the middle
ascending relevance heuristic	goes through the US clauses in order of relevance, starting with the least relevant clause
descending relevance heuristic	goes through the US clauses in order of relevance, starting with the most relevant clause
centered relevance heuristic	starts with the US clauses of medium relevance, then spiraling out to ever more and less relevant ones

Figure 3.14: Table of predefined deletion heuristics.

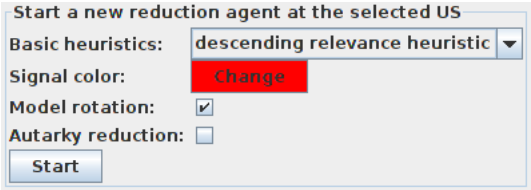


Figure 3.15: The dialog for starting a new reduction agent.

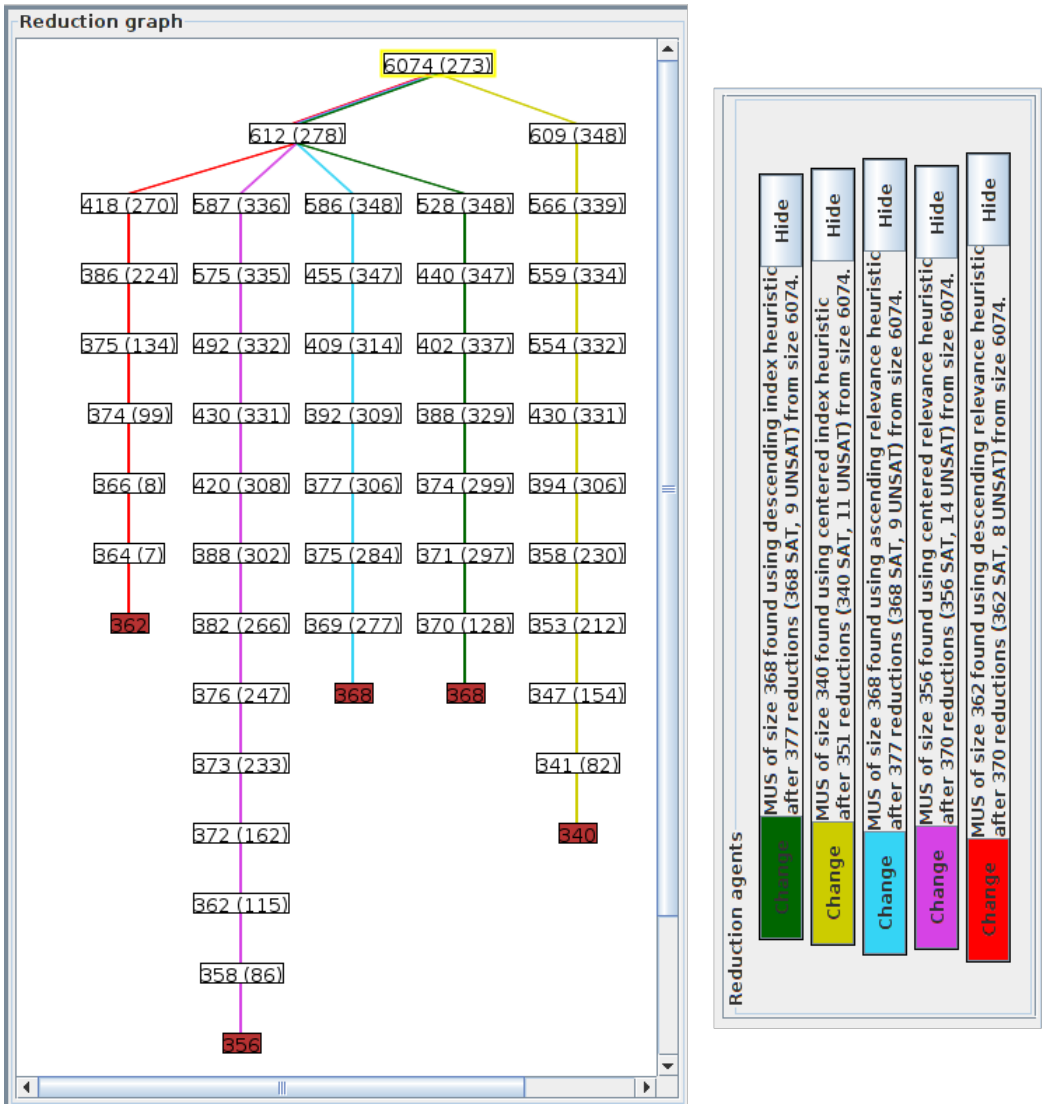


Figure 3.16: Comparing deletion heuristics using the reduction graph.

MUS, which may have been caused by the fact that it tends to throw away clauses which are helpful for short refutation proofs, making a greater number of other clauses necessary to compensate for their absence.

The centered relevance heuristic, which tries to keep a balance between the relevance of the deletion candidates for the proof and the potential impact of deletion on MUS size, was more successful in reducing MUS size, but it needed more steps for its path through the reduction graph. In this test case, the heuristic which can with some justification be called the most successful one is the centered index heuristic. This heuristic led to the smallest MUS, while at the same time requiring the lowest total number of SAT calls. Since the heuristic has this advantage in other instances of this particular ASP encoding as well, this can be taken as a hint that the encoding tends to lead to an agglomeration of fall-away material in the middle of the clause list, making it worthwhile to start deletion attempts in the middle of the instances.

Although none of these observations can be generalized to arbitrary instances without quantitative evaluation, the behaviour of the different heuristics usually stays rather similar across SAT instances derived from a single application. Trying out different deletion heuristics on a few sample instances of a benchmark set can therefore help to quickly determine the characteristic structure of a given SAT encoding with respect to the MUS extraction search space, and to assess the expected performance of different heuristics for practical applications.

3.5 Conclusion

In this chapter, we introduced and saw an implementation of a first version of the interactive approach to MUS extraction developed in this thesis. While we have not yet seen any practical application of the paradigm, its prospective advantages can already be motivated on principal grounds. The ability to influence the MUS extraction process while it is running obviously complements existing tools which are geared towards performant extraction, since it adds flexibility and the possibility to incorporate domain knowledge to deletion-based MUS extraction. The explicit visualization of the search space makes it easy to detect structural properties of novel encodings.

A major strength of the initial prototype presented in this chapter is the very general heuristics interface that allows to experiment with many different heuristics, which can be quickly defined by implementing a lean Java interface, and then immediately be observed in action. The parallel architecture allows to run many such agents at the same time, causing the system to scale well and making it possible to fully exploit the processing power of multicore hardware architectures. The integrated accessibility of state-of-the-art techniques such as model rotation and autarky pruning further adds to the value of the prototype as an exploratory tool.

The main disadvantage of the first prototype is that it does not yet provide any visualization of overlaps between USes, which makes it hard to see how relevant the differences between the various branches of a reduction graph really are. The lack of detail in the reduction graph also makes it difficult to understand the properties of different deletion candidates, since the relevant information can only be gained via time-consuming inspection of individual USes. While it allowed for a straightforward implementation of a first version of interactive reduction, the rudimentary display of clause sets as a colour-coded list does not add much to the user's understanding of the problem structure either. While the selection functionality can to some degree be used to increase inspection efficiency, the sometimes tight connections between individual clauses are not visible in any way. Another disadvantage that we have already alluded to is that we are so far only propagating reducibility information along es-

tablished reduction links, but not to all subsets or supersets as the monotonicity properties would allow us to do. This leads to a loss of information which can lead to a waste of SAT solver calls for deriving information which would actually already have been there.

In the following chapters, many of these shortcomings of the initial prototype will be addressed. Chapter 4 introduces meta learning as a general and theoretically founded approach to a better utilisation of the available reducibility information by sharing it between branches in a systematic manner. Chapter 5 will add visualization components which reveal a lot more information about the connections between different USes, and Chapter 6 will present a first practical application of the extended prototype as a proof of concept for the benefits of interactive MUS extraction.

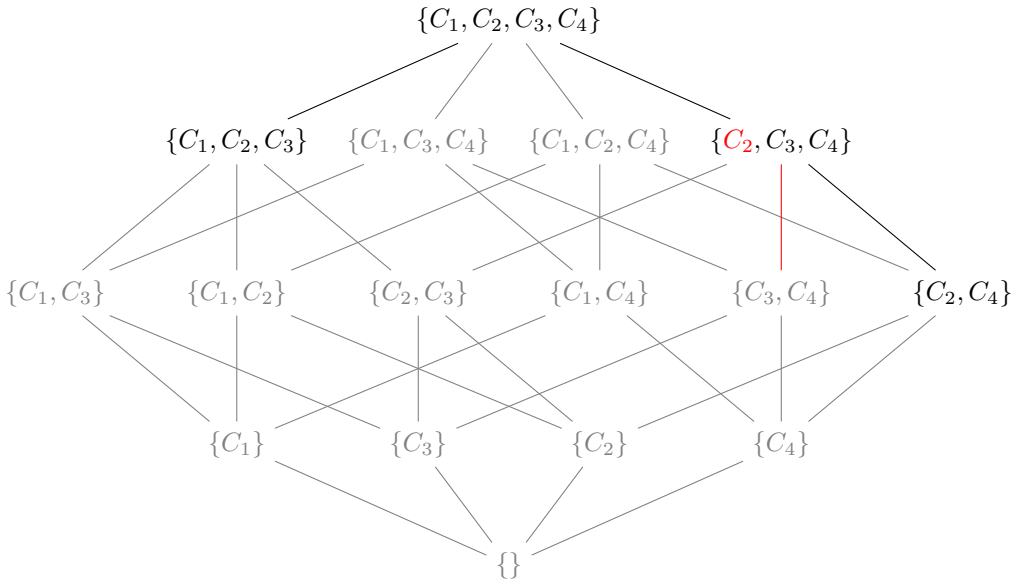
Chapter 4

Introducing Meta Learning

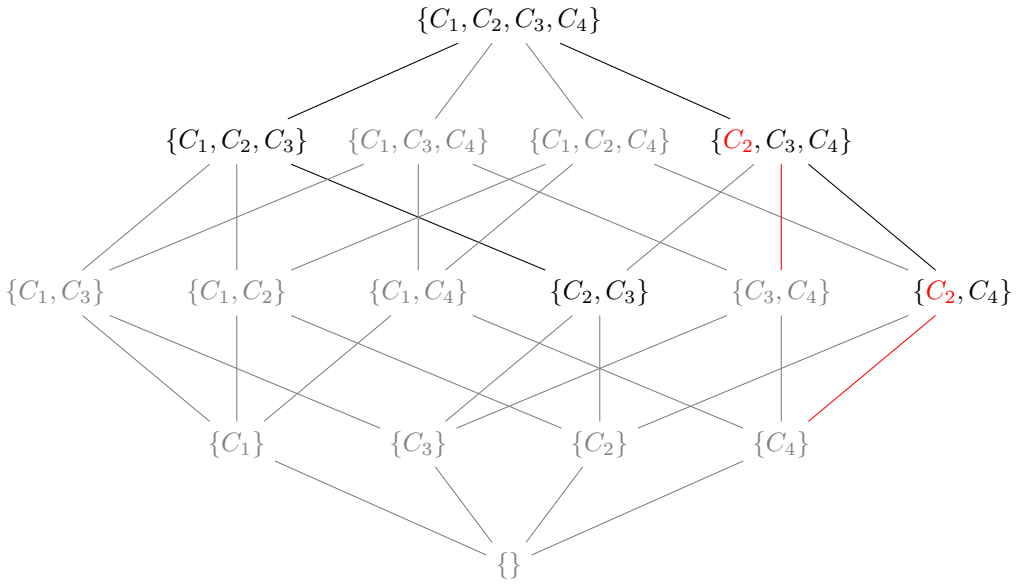
4.1 Motivation

When we discussed the issue of propagating reducibility information through the reduction graph during interactive reduction in the last chapter, we mentioned that this can still lead to unnecessary SAT calls because the edges in the reduction graph do not necessarily cover all the subset relations between the different USes we encountered. In this chapter, we will develop a solution to this problem which also leads to a range of additional benefits for the interactive MUS extraction paradigm.

We start our closer analysis of this problem by considering the powerset lattice of another small example instance $\{C_1, C_2, C_3, C_4\}$. To simplify the exposition, we will assume that clause set refinement is of no relevance here, so that each single reduction step will at most lead one step down in the powerset lattice. Assume a state of interactive reduction where the reduction graph spans the edges coloured in black, and the grey edges and subsets have not yet been explored. In the subset $\{C_2, C_3, C_4\}$, the clause C_2 has just been determined to be critical, which we again symbolize by colouring critical clauses and the corresponding transition edges in red.



The criticality of C_2 in $\{C_2, C_3, C_4\}$ is propagated downwards in the reduction graph to the subset $\{C_2, C_4\}$, but not to the other subsets. Now assume that in the next reduction step, we successfully delete C_1 from the set $\{C_1, C_2, C_3\}$. The propagation and the reduction attempt together lead us to the following new state of the reduction graph:



In the implementation of interactive MUS extraction in Chapter 3, in this situation, we do not derive the information that C_2 is critical in the new US $\{C_2, C_3\}$, causing us to waste a call to a SAT solver when we later try to delete C_2. This type of problem is quite virulent in larger instances, because it is likely to occur whenever one branch of the reduction graph arrives at a US which would also be reachable by reduction starting in some other node where some unsuccessful reductions have already been performed.

In our minimal example (clause sets of size 4 are the smallest case where the problem can occur in non-trivial subsets), the issue does not appear too problematic because it seems feasible to simply test for all the subsets of $\{C_2, C_3, C_4\}$ whether they are already in the reduction graph, and marking C_2 as critical in all of them. The problem with this is of course that the number of possible subsets is exponential in the size of a set, so for instances of any interesting size we will need a much more efficient approach to propagating this information only to those subsets which are already represented by nodes in the graph.

The propagation of criticality information through the powerset lattice is just one instance of the more general phenomenon that the deletion of some clause or some combination of clauses from a clause set can make other clauses necessary. For interactive reduction, it seems worthwhile to collect such observations and to exploit them in order to systematically infer new information about the status of clauses in one node of the reduction graph from the information collected on other branches.

The approach developed in this chapter aims to achieve this by explicitly representing and processing these connections in the form of boolean **constraints over selector variables**. Before, selector variables had the only purpose of extending the clauses of the original instance in such a way that clauses can effectively be switched off by setting the corresponding selector variable in the freeze file to false. In this chapter, we will go beyond that by reusing the selector variables as direct handles for talking about the presence or absence of clauses

in subsets. For instance, we will see that, as in our motivating example, if during our reduction attempts we find a set of clauses $\{C_1, C_2\}$ which cannot be reduced at the same time, we can remember this information in the form of $\{s_1, s_2\}$, an additional clause which contains positive literals for the two selection variables corresponding to C_1 and C_2, enforcing the presence of either of the two in any subset we henceforth consider. We will be using a collection of such clauses to store all the derived reducibility information, and use it to efficiently determine all the clauses already known to be critical in new unsatisfiable subsets.

Although the terms are slightly overused, we will take **meta instance** to refer to the second SAT instance over the selector variables which we shall maintain, and **meta learning** to denote the addition of clauses to the meta instance whenever new reducibility information has been derived. The clauses of the meta instance, whether added during meta learning or in other contexts, will be called **meta constraints**.

The remainder of this chapter is divided into four sections. In Section 4.2, we will systematically develop answers to the question which connections between selector variables we can learn from the results of individual reduction attempts. Section 4.3 then deals with the question how meta learning can be implemented and best made use of in our interactive MUS extraction system. This includes the problem of compiling all the derived knowledge into clausal meta constraints, and the question how the criticality information relevant in different use contexts can efficiently be extracted from the meta instance. Section 4.4 then provides an outlook on some of the many other ways in which meta constraints can be put to use. Finally, Section 4.5 summarizes the results of the chapter, taking stock of the problems resolved and the issues remaining.

4.2 What Can We Learn?

We have seen that the reducibility information we find out about one node of the reduction graph tends to be of use in many other nodes as well. In this section, we will systematically develop an answer to the question how this information can be expressed in terms of formulae over selector variables. The two cases of a successful and an unsuccessful reduction are of course very different in the kinds of knowledge they allow us to infer, which is why in this section, we will be considering both cases separately.

Throughout this chapter, we will use the convention that an unsatisfiable SAT instance F was extended to F' by extending each clause $C_i := \{l_1, \ldots, l_n\} \in F$ by a negated selector variable s_i to form $\{l_1, \ldots, l_n, \neg s_i\} \in F'$. By convention, the correspondence between clauses and selector variables is therefore expressed by shared indices. Therefore, setting the selector variable s_i to false is equivalent to deactivating the clause C_i, and setting it to true enforces the presence of C_i. Whenever we operate on a second set of clauses and corresponding selector variables, we will use the same convention for clauses D_j and selector variables t_j, respectively.

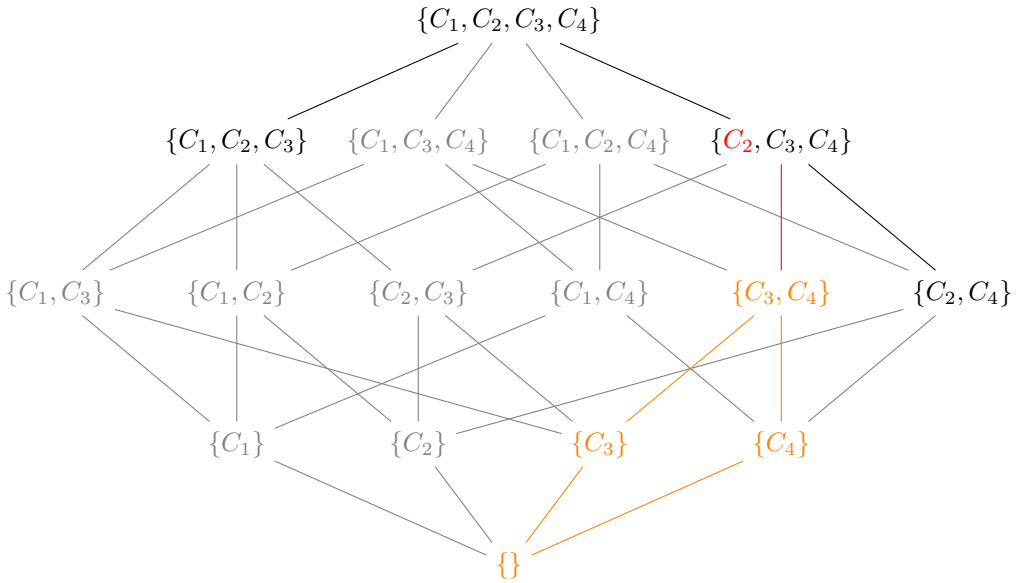
While there is a variety of conditions a meta instance could be used to express, for reasons that will become clear as we proceed, we will use the meta instance to express **conditions for clause unsatisfiability**. Formally, we will know that a clause set $\{C_1, \ldots, C_n\}$ is satisfiable if for $\{D_1, \ldots, D_m\} := F \backslash \{C_1, \ldots, C_n\}$, the meta instance G extended by assumptions over selector variables modelling the set $\{C_1, \ldots, C_n\}$, i.e. $G \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_m\} \cup \{s_1\} \cup \cdots \cup \{s_n\}$, is unsatisfiable. This does not imply that the meta instance will be unsatisfiable under such assumptions for all satisfiable subsets, or that its satisfiability under these assumptions tells us anything about the satisfiability of the subset. In this case, the meta instance would already encode the full search space, defeating its purpose. Instead, we will be using the meta instance as a repository of our incomplete knowledge about the search space.

4.2.1 Unsuccessful Reductions

Assume that we have attempted to delete a clause C_i from an unsatisfiable clause set $\{C_1, \ldots, C_m\}$, and that this has resulted in a satisfiable instance, meaning that C_i is now known to be critical in $\{C_1, \ldots, C_m\}$. This criticality information depends on all the clauses $\{D_1, \ldots, D_n\} := F \backslash \{C_1, \ldots, C_m\}$ which had already been reduced or fallen away in the state where we attempted the deletion. The new information can therefore be expressed as $(\neg t_1 \wedge \cdots \wedge \neg t_n) \rightarrow s_i$. Note that this new meta constraint can directly be written as a single clause $\{t_1, \ldots, t_n, s_i\}$, which we can simply add to the meta instance.

The new meta constraint $\{t_1, \ldots, t_n, s_i\}$ can be read as preventing all the elements of the set $\{D_1, \ldots, D_n, C_i\}$ from being removed at the same time, by requiring that one element of the set must be present in any unsatisfiable subset. In our motivating example, we would learn the meta clause $\{s_1, s_2\}$, expressing that either C_1 or C_2 must be present in any unsatisfiable subset of $\{C_1, C_2, C_3, C_4\}$.

A helpful way to think about the nature of criticality information in the powerset lattice of clause subsets is based on the observation that after each reduction attempt, the monotonicity properties allow us to reduce the search space by an entire **wedge** through the powerset lattice. Let us consider how the criticality information learnt in the case of an unsuccessful reduction can be conceived as such a wedge. We have just seen that an unsuccessful deletion of C_i from a clause set $\{C_1, \ldots, C_m\}$ allows us to derive a new meta constraint $\{t_1, \ldots, t_n, s_i\}$, where $t_1, \ldots, t_n$ describes the clause set $\{C_1, \ldots, C_m\}$ by stating the presence of any element from its complement. Now consider what happens in an arbitrary subset of $\{C_1, \ldots, C_m\}$. The monotonicity of criticality with respect to the subset relation predicts that C_i will still be critical in every such subset. But this is exactly what our constraint expresses, because any subset can be described by a set $T \supset \{t_1, \ldots, tn\}$ of selector variables, leading to clauses which are subsumed by $\{t_1, \ldots, t_n, s\}$. In fact, the set $\{C_1, \ldots, C_m\} \backslash \{C_i\}$ corresponds to the tip of what we will call an **upward wedge of satisfiability (sat wedge)** through the elements of the powerset lattice, and this wedge is explicitly removed from the search space by the meta constraint $\{t_1, \ldots, t_n, s_i\}$. In our motivating example, by colouring all sets known to be satisfiable in orange, the sat wedge which gets cut from the search space when we learn the meta constraint $\{s_1, s_2\}$ can be visualized in the powerset lattice like this:



Note that the nodes had to be slightly reordered to demonstrate that the sets we have determined to be satisfiable can be seen to form an upward wedge with the tested subset $\{C_3, C_4\}$ at the top. For a single wedge, we can always reorder the nodes in our two-dimensional display of the powerset lattice in such a way that the wedge structure becomes visible. For displaying more than one wedge at the same time, we might already need an additional dimension to visualize the lattice in a way that explains our geometric notion of a wedge.

A few further relevant observations about sat wedges deserve to be mentioned. The first is that smaller meta clauses yield larger wedges. Obviously, with each literal less in the meta clause the number of subsets covered by the wedge doubles. This not only explains why for interactive reduction, we always want to detect critical clauses as high up in the powerset lattice as possible, but it also leads to the idea that it might be worthwhile to try to enlarge the wedges by attempting to delete literals from meta constraints. We will explore this idea in some detail in Section 4.3.1.

Turning back to the example at hand, let us see how the meta constraint $\{s_1, s_2\}$ helps us to derive the criticality information we were missing when using mere propagation through the reduction graph. Assume again that in the next reduction step, we successfully reduce $\{C_1, C_2, C_3\}$ to $\{C_2, C_3\}$. If we now want to determine whether C_2 is critical in $\{C_2, C_3\}$, we can perform a first check for the satisfiability of $\{C_3\}$ by testing the satisfiability of $\{\{s_3\}, \{\neg s_1\}, \{\neg s_2\}, \{\neg s_4\}\}$ given our knowledge that $\{s_1, s_2\}$. Sure enough, the clause set $\{\{s_3\}, \{\neg s_1\}, \{\neg s_2\}, \{\neg s_4\}, \{s_1, s_2\}\}$ is unsatisfiable, allowing us to find out by solving a trivial SAT instance that C_2 must be critical in $\{C_2, C_3\}$. As intended, we do not need to waste a much more costly SAT solver call on the original instance any longer.

More generally, we find that checking a combination of selector literals representing a clause subset against a collection of meta clauses learnt from unsuccessful reduction attempts is equivalent to checking whether the subset lies in one of the sat wedges that we know of:

Theorem 4.2.1. *Let $F_1, \ldots, F_k \in 2^F$ be satisfiable subsets of an unsatisfiable clause set F, where for each $1 \leq j \leq k$ we write $F_j := \{C_{j1}, \ldots, C_{jm_j}\}$ and $F \backslash F_j := \{D_{j1}, \ldots, D_{jn_j}\}$. Let $G := \{\{t_{11}, \ldots, t_{1n_1}\}, \ldots, \{t_{k1}, \ldots, t_{kn_k}\}\}$ be the CNF instance derived by meta learning in the reduction steps which established the satisfiability of the sets $F_1, \ldots, F_k$. Then, for any set $S \subset F$ with $F \backslash S := \{D_1, \ldots, D_n\}$, there is a $1 \leq i \leq k$ with $S \subseteq F_i$ iff the extended meta instance $G \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_n\}$ is unsatisfiable.*

Proof. $\Rightarrow$: Let $S := F \backslash \{D_1, ..., D_n\} \subseteq F \backslash \{D_{i1}, \ldots, D_{in_i}\} =: F_i$. Taking the complement on both sides, we know that $\{D_1, ..., D_n\} \supseteq \{D_{i1}, \ldots, D_{in_i}\}$ and therefore $\{t_1, ..., t_n\} \supseteq \{t_{i1}, \ldots, t_{ik_i}\}$. This means that G is extended by at least the unit clauses $\{\neg t_{i1}\}, \ldots, \{\neg t_{ik_i}\}$. After propagating these k_i units, the clause $\{t_{i1}, \ldots, t_{ik_i}\}$ in G has become the empty clause, proving the unsatisfiability of $G \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_n\}$.
$\Leftarrow$: Let $G \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_n\}$ be unsatisfiable. By the completeness of resolution, we must be able to deduce the empty clause by a finite sequence of resolution steps. As the non-unit clauses in the extended meta instance do not contain any negative literals, all these steps must have occurred between the non-unit clauses and the negative literals. To arrive at an empty clause, we must have deleted all literals from $\{t_{i1}, \ldots, t_{in_i}\}$ for some $1 \leq i \leq k$, which means that for the set $\{t_1, ..., t_n\}$ of selector variables in the negative units we must have $\{t_1, ..., t_n\} \supseteq \{t_{i1}, \ldots, t_{in_i}\}$. But given the definition of the selector variables, this is equivalent to $S = F \backslash \{D_1, ..., D_n\} \subseteq F \backslash \{D_{i1}, \ldots, D_{in_i}\} = F_i$. □

According to this theorem, the meta instance we are building so far can be used to check whether a newly encountered clause set is a subset of any previously determined satisfiable subset, showing that we have found the desired general solution to the problem of complete propagation of criticality information. Later, we will see that the meta instance is even more useful if we go beyond using it as an effective encoding of sat wedge membership.

4.2.1.1 Model Rotation

Let us now consider the case of model rotation. If using this method, we have found an entire set $\{C_{i_1}, \ldots, C_{i_k}\}$ of new critical clauses, this gives us the constraint $(t_1 \wedge \cdots \wedge t_n) \rightarrow (s_{i_1} \wedge \cdots \wedge s_{i_k})$ or $\neg t_1 \vee \cdots \vee \neg t_m \vee (s_{i_1} \wedge \cdots \wedge s_{i_k})$ which must hold for any unsatisfiable subset in the reduction graph. Converting this into clausal form, we receive the k meta constraints $\{t_1, \ldots, t_m, s_{i_1}\}, \ldots, \{t_1, \ldots, t_m, s_{i_k}\}$, i.e. from the perspective of meta learning, the results of model rotation are exactly equivalent to what we would have learnt from the k corresponding unsuccessful reductions of a single clause. In the following discussion, we therefore no longer need to consider the case of model rotation separately.

4.2.1.2 Unsuccessful Simultaneous Reduction

Another case which we need to consider is when the simultaneous deletion of a set of k clauses $\{C_{i_1}, \ldots, C_{i_k}\}$ from a clause set $\{C_1, \ldots, C_m\}$ has led to a satisfiable problem, meaning that we have gained the knowledge that the k clauses may not be deleted together. Note that in terms of criticality, this does not imply for any of the clauses in $\{C_{i_1}, \ldots, C_{i_k}\}$ that it was critical in $\{C_1, \ldots, C_m\}$, nor that it was not. We have therefore learnt a more complex connection between clause deletions which cannot simply be represented by manipulating reduction tables in other parts of the reduction graph.

Still, our selector variable approach is powerful enough to express the new information. Instead of a conjunction of positive selection literals enforcing the presence of clauses as in the case of model rotation, we now have a disjunction in the resulting meta formula $(t_1 \wedge \cdots \wedge t_n) \rightarrow (s_{i_1} \vee \cdots \vee s_{i_k})$, expressing that under the condition that all the clauses in the complement of $\{C_1, \ldots, C_m\}$ have been deleted, at least one of the clauses $C_{i_1}, \ldots, C_{i_k}$ cannot be deleted any more. Again, we have a trivial translation to a single clausal meta constraint $\{t_1, \ldots, t_n, s_{i_1}, \ldots, s_{i_k}\}$. Note that this clause is subsumed by any of the clauses $\{t_1, \ldots, t_n, s_{i_j}\}$ we would get if we later determine one clause $C_{i_j} \in \{C_{i_1}, \ldots, C_{i_k}\}$ to be critical. This reflects the fact that the information we get from an unsuccessful simultaneous reduction is rather weak, and immediately becomes obsolete if we determine a single one of the reduced clauses to be critical.

4.2.2 Successful Reductions

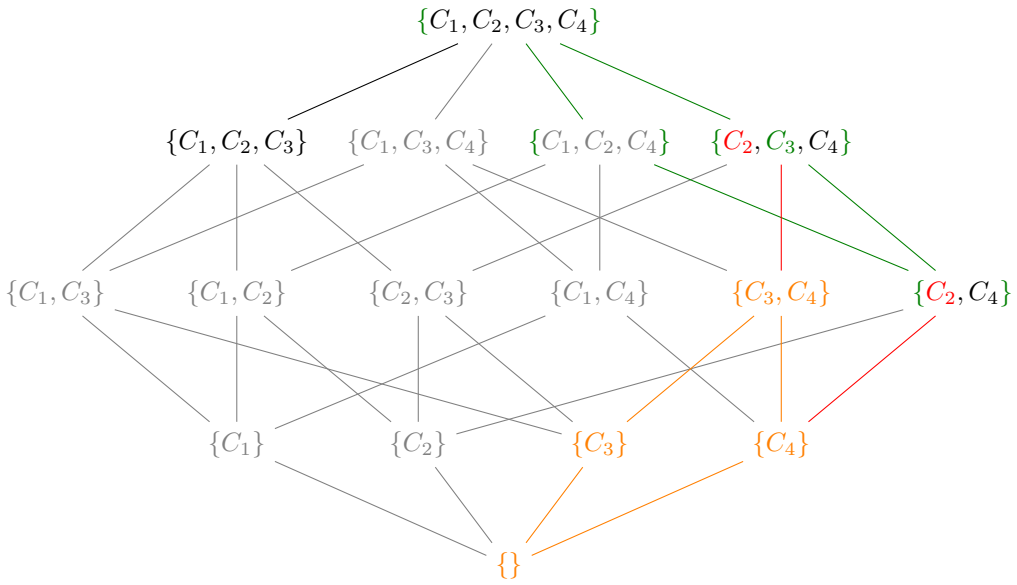
Let us now assume that our attempt to delete a clause C_i from an unsatisfiable clause set $\{C_1, \ldots, C_m\}$ was successful, so that we have arrived at a new unsatisfiable subset $C' \subseteq \{C_1, \ldots, C_m\} \backslash \{C_i\}$, where equality holds iff clause set refinement has not caused any further clauses to fall away. With this result, we have found out that we can safely delete all the clauses in $\{D_1, \ldots, D_n\} := \{C_1, ..., C_m\} \backslash C'$ from any unsatisfiable clause set which contains all the clauses in C'. If we express this connection in terms of selector variables, we can simply say that it suffices for unsatisfiability if all clauses in $\{C_1, ..., C_m\} \backslash C'$ are present, i.e. if $t_1 \wedge \cdots \wedge t_n$ holds.

Note that this formula is of a very different nature from what we have derived for the case of unsuccessful reductions. Our goal remains to encode in the meta problem conditions for unsatisfiability, but we are now dealing with a constraint that is a **sufficient condition**, not a necessary condition as before. In fact, $t_1 \wedge \cdots \wedge t_n$ is a **minterm** which would need to be added as a disjunct to the meta problem to represent one way to fulfill the condition that a subset be unsatisfiable. But this disjunctivity means that we cannot simply integrate this information by adding a set of clausal constraints to the meta instance, touching a core element of the problems of representability which will be a focus of discussion in the context of implementing meta learning in Section 4.3.1.

If we interpret in terms of the powerset lattice the information that $t_1 \wedge \cdots \wedge t_n$ is a sufficient condition for unsatisfiability, in full analogy with the unsuccessful reduction case we get a **downward wedge of unsatisfiability (unsat wedge)**, because the minterm also holds in all supersets of $\{C_1, ..., C_m\} \backslash C'$. With every successful reduction attempt, we therefore get unsatisfiability guarantees for a number of other sets. The smaller the set we find to be still unsatisfiable, the larger the number of supersets in the unsat wedge becomes. Again, for each conjunct less in the minterm, the number of unsatisfiable subsets covered doubles. This once more illustrates how much the detection of one small unsatisfiable subset can help us in detecting others, as large chunks of the search space are then known to be above the transition boundary.

While the enlargement of sat wedges will require additional implementation effort, in our deletion-based MUS extraction paradigm, generalizing the minsets to enlarge unsat wedges is exactly what we are doing by our standard operation of executing reduction attempts. This makes the explicit storage of unsat wedges less obviously necessary than that of sat wedges, later conveniently allowing us to forego the latter in the implementation.

Returning to our running example, let us see what we learn from the successful reduction of $\{C_2, C_3, C_4\}$ to $\{C_2, C_4\}$. The minset is simply $s_2 \wedge s_4$, and the unsat wedge it represents (coloured in green) comprises the subset $\{C_1, C_2, C_4\}$ about which we knew nothing before.



The bad news is that this new information about the subset $\{C_1, C_2, C_4\}$ cannot be derived via a satisfiability check on the meta instance as we defined it up to now. While the new minset makes the meta instance satisfiable under the assumptions $\{\{s_1\}, \{s_2\}, \{\neg s_3\}, \{s_4\}\}$, The satisfiability of the meta instance under assumptions can never guarantee that the corresponding reduction is possible. So how could we perform this type of inference? If our meta instance were composed only of minterms over selector variables, we could use it to detect unsat wedge membership, allowing a full propagation of fall-away information in a way dual to what we achieved for criticality information from unsuccessful reductions. Assume that in the next step of our reduction, we want to find out whether C_3 can be deleted from the top node $\{C_1, C_2, C_3, C_4\}$, so we need test for the unsatisfiability of $\{C_1, C_2, C_4\}$. The unsatisfiable subsets we already know are encoded as $(s_2 \wedge s_4) \vee (s_1 \wedge s_2 \wedge s_3)$. Adding the assumption minset $(s_1 \wedge s_2 \wedge \neg s_3 \wedge s_4)$ representing $\{C_1, C_2, C_4\}$, we find that the meta instance remains satisfiable, showing us that the tested set lies in one of the unsat wedges.

Again, this piece of fall-away information would not have arrived in the new node by upward propagation alone, since the edge between $\{C_1, C_2, C_4\}$ and $\{C_2, C_4\}$ has not been added to the reduction graph yet. We were only able to make this inference because by collecting only minterms, we have in fact arrived at an alternative meta instance whose satisfiability given an assumption minset guarantees the unsatisfiability of the corresponding subset, but which cannot guarantee satisfiability, dual to the meta instances we used before.

We have seen that we have a nice duality between downward unsat wedges and upward sat wedges which can capture all the information we want to distribute about the status of individual clauses, but that we need a conjunction of clauses to directly encode the sat wedges, and a disjunction of minterms to collect and efficiently represent unsat wedges. Furthermore, we have seen that the conjunction of clauses only encodes sufficient conditions for satisfiability, and that the disjunction of minterms only encodes sufficient conditions for unsatisfiability. This duality makes it impossible to meaningfully represent both types of information in one meta problem so that we would either have to maintain two meta instances, or choose to only completely represent either criticality or fall-away information. In the next section, we will see good reasons for opting for the second option in the context of interactive MUS extraction.

4.2.2.1 Successful Simultaneous Reduction

As for unsuccessful reductions, let us last consider the case of successful simultaneous reduction. Assume that a set of k clauses $\{C_{i_1}, \ldots, C_{i_k}\}$ was simultaneously deleted from a clause set $\{C_1, \ldots, C_m\}$. For the result set $\{D_1, \ldots, D_n\} := \{C_1, \ldots, C_m\} \backslash \{C_{i_1}, \ldots, C_{i_k}\}$ we have found to be unsatisfiable, we can again store a sat wedge by learning the minterm $t_1 \wedge \cdots \wedge t_n$. Though the result set might be smaller, there is thus no principal difference to the case of a successful deletion of a single clause. Note that this observation also allowed us to not consider the consequences of clause set refinement in this section, since it does not change anything except the size of the derived minterm.

4.3 Implementation

To add the new concept of meta learning to the prototype implementation of interactive MUS extraction described in the previous chapter, we will proceed in four steps. In Section 4.3.1, we will spend some thoughts on which types of constraints derived by meta-learning can directly represented in the meta instance, and we will motivate the decision to not represent unsat wedges, therefore not storing any meta constraints which would serve to guarantee unsatisfiability. Section 4.3.2 then explores different ways of extracting selector units from the meta instance, while Section 4.3.3 deals with the application of full SAT solving on the meta instance. The implementation of meta learning in the interactive MUS extraction system is described and discussed in Section 4.3.4. Finally, Section 4.3.5 shows an example of meta learning in the implementation.

4.3.1 Representing and Maintaining the Meta Problem

In the last section, we have seen that the constraints over selector variables which result from unsuccessful reductions can easily be added to a clausal meta instance that encodes sufficient conditions for unsatisfiability, but we cannot integrate the non-clausal minsets derived from successful reductions into the same meta instance, not only because the mixed problem could grow exponentially when converted to clausal form, but also because if we try to encode all known sufficient conditions for satisfiability and unsatisfiability in the same meta problem, we end up encoding neither of the two.

As already mentioned, we could now choose to work with two meta problems, one being in conjunctive normal form and one in the form of a disjunction of minterms (for which the term disjunctive normal form (DNF) is commonly used), one encoding the satisfiability of some previously unseen subsets by sat wedges and the other of some unsatisfiable ones by unsat wedges. But if we only want to maintain one meta instance, we will need to make a choice which type of conditions to express in it.

For the implementation, we will opt for the CNF encoding of sat wedges, but not only because we want to avoid the overhead of maintaining two different meta problems. Even if we could directly derive the unsatisfiability of a subset of previously unknown status from the meta problem, for the technique of clause set refinement, a crucial element for efficient interactive reduction, we need to inspect a proof of unsatisfiability, which we can only get from a SAT solver call on the full original instance. This severely limits the possible pay-off of unsat wedge membership testing, since most successful tests would still have to be followed by a solver call on the original instance to generate the proof. At any rate, the few situations where bare unsatisfiability information derived from the meta instance could be useful (such as when the tested subset is only one clause larger than an unsatisfiable subset which is already in the reduction graph), do not justify the overhead involved in maintaining a second meta instance. We will therefore only be using one meta problem in CNF to represent sat wedges in the implementation.

Some technical difficulties are involved in storing and maintaining the meta instance in an efficient and compact manner. The main challenge in this regard is to avoid storing redundant information. The problem of redundancy is very severe in a meta instance which encodes a collection of wedges, as many of these wedges will be contained in others, and it would be sufficient to only store those edges which are not completely contained in some larger wedge.

In terms of the defining clauses, wedge redundancy exactly corresponds to what is commonly called **subsumption** in the literature. A clause C_1 is said to **subsume** another clause C_2 iff the conditions imposed by C_1 on satisfying assignments are stronger than those imposed by C_2, in the sense that every assignment fulfilling C_1 will fulfill C_2 as well. For clauses which do not contain complementary literals, this means that a clause C_1 subsumes another clause C_2 iff $C_1 \subset C_2$. In our meta instance, the subset relation between clauses corresponds to the containment relation between the wedges they represent, as illustrated in Figure 4.1. Assume that our meta instance encodes a small sat wedge, which we represent by the grey upward triangle. Now assume that we learn another meta clause containing a subset of the literals in the existing wegde. This new meta clause encodes a larger wedge (displayed in white) which includes the old one. To avoid redundancies, we must detect the wedges contained in the new wedge and remove the clauses corresponding to them from the meta instance while adding the new clause.

The direct correspondence between the containment relation between sat wedges and the subsumption relation between the respective clauses allow us to use standard approaches for filtering out subsumed clauses by what is called **subsumption checking** to detect these redundant clauses in the meta instance. Subsumption checking is used sparingly by SAT solvers because it is considered too costly to execute as often as would be necessary to maintain subsumption-freeness at all times. For our meta problem, this is not an issue, since unlike during SAT solving, we are not constantly modifying the clauses of the problem, but the only possible source of redundancy is if it is caused by the addition of a new clause, which makes subsumption checking tractable and worthwhile.

To develop an algorithm for efficiently maintaining the subsumption freeness of a clause set to which more clauses are added one by one, Lintao Zhang [42] distinguishes between two

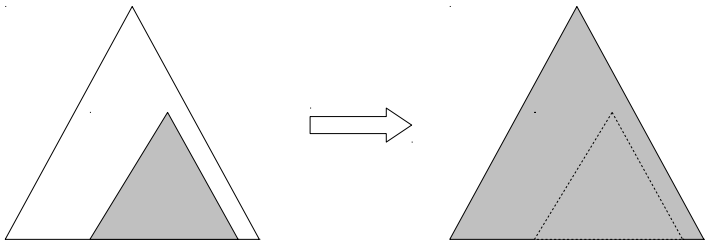


Figure 4.1: Illustration of subsumption checking while adding a sat edge.

different kinds of subsumption checking. The first kind is called **backward subsumption checking**, where for a new clause to be added, we check whether it subsumes any of the clauses already present, which are then removed from the instance because they have been made redundant. By contrast, in **forward subsumption checking** we determine whether the new clause is already subsumed by some clause in the set, making it redundant and allowing us to simply not add it. For both types of subsumption checking, different approaches were proposed in the literature, and Zhang's algorithm applies very different techniques to both cases.

In our use case, we have the fortunate situation that only backward subsumption checking will be necessary. If we arrived at a situation where forward subsumption checking were necessary, i.e. if there were a known sat wedge containing the new wedge, this would imply that we have checked for the satisfiability of a subset which we should already have known to be satisfiable, since the subsuming sat wedge must already have been part of our meta knowledge. This concurs with our intuition that deletion-based MUS extraction steps can only cause existing sat wedges to grow, but never to shrink.

For the implementation of backward subsumption checking, a variant of the algorithm presented by Lintao Zhang [42] was chosen. In essence, the algorithm loops through the literals of the new clause, efficiently retrieves the lists of existing clauses containing each literal, and gradually refines a set of subsumption candidates by efficiently intersecting it with these lists. After all the literals are processed, the candidate set contains all backward-subsumed clauses. The original version relies on a sparse-matrix representation of the clause set to retrieve the lists of clauses for each literal, whereas my implementation uses an indexing data structure which was already being maintained for other purposes, such as quick variable deletion for the autarky reduction algorithm.

While subsumption checking can be seen as a general method for CNF simplification, our specific situation would also allow us to perform some additional operations on the meta problem. These operations are based on forming and testing hypotheses about additional connections between selector variables, thereby constituting ways to derive information about the search space beyond the purely deletion-based paradigm.

A first approach in this direction is based on the trivial fact that more satisfiable subsets are covered by larger wedges, which makes larger wedges represented by shorter meta clauses more desirable. This suggests an operation of **strengthening** clauses in the meta instance by attempting to remove literals from them, and then testing whether the corresponding assignment of truth values to the selector variables still leads to a satisfiable instance. If it does, we have managed to enlarge the wedge represented by the strengthened clause. Strengthening can also be seen as a simplification operation because strengthened clauses

are more likely to subsume other existing clauses, possibly causing other clauses to fall away during subsumption checking.

Viewing the meta instance as a collection of upward sat wedges, another approach would be to attempt to form larger wedges from smaller ones by a **merge** operation. If we have found satisfiable subsets F_1 and F_2 and have therefore added sat wedges represented by meta clauses C_1 and C_2 to the meta instance, it could well be that already the union $F_1 \cup F_2$ was satisfiable. In terms of meta clauses, we can test this by forming the intersection $C_1 \cap C_2$ and testing the corresponding assignment to the selector variables. If the result is satisfiable, we have created a potentially much larger sat wedge which subsumes both C_1 and C_2, often leading to significant simplification by letting the merged wedges and possibly more fall away.

A problem with the merge operation is that one application may cause other applications to be blocked. For instance, it may be the case that we already know of three satisfiable subsets F_1, F_2, F_3, and could find out by merging that both $F_1 \cup F_2$ and $F_2 \cup F_3$ are satisfiable, but not $F_1 \cup F_2 \cup F_3$. If we use the standard merge operation on $F_1 \cup F_2$ or $F_2 \cup F_3$ first, the other merge operation becomes unavailable. If instead, after the successful check we add the sat wedge headed by $F_1 \cup F_2$ to the meta instance, but keep the one headed by F_2 around, it remains possible to also check $F_2 \cup F_3$ by a second merge operation. This method of only applying subsumption checking for all the new clauses after performing a series of merge operations could be called **non-greedy merge**.

Schematic representations of all three operations are given in Figure 4.2. In each case, the grey triangles represent some sat wedges which were already known before the operation, and the white triangles stand for the new wedges derived by the different operations. In a next step, subsumption checking would then remove all grey wedges covered by white wedges, making clear why the operations lead to simplifications of the meta instance. In the rightmost sketch representing an instance of the non-greedy merge operation, the middle triangle plays the role of the wedge headed by F_2.

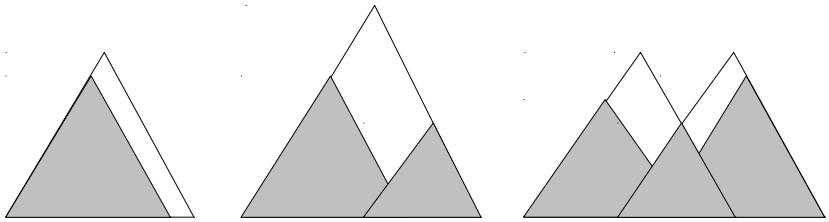


Figure 4.2: Illustration of wedge strengthening, merge and non-greedy merge.

There are some interesting connections between these operations on the meta instance and the different paradigms for MUS extraction. When deciding to only store sat wedges in our meta instance, we already saw that a successful deletion step is equivalent to enlarging an unsat wedge. Conversely, the strengthening operation we just introduced on a sat wedge is equivalent to the successful addition of a clause to a satisfiable subset in an insertion-based MUS extraction algorithm.

These connections show that an interactive reduction system which supports the additional operations could maintain a more compact meta instance, but it would not be purely deletion-based any more, causing some of the design concepts established for the prototype to collapse. The obvious way to integrate those operations given the existing infrastructure would be to define and implement **expansion agents** as the dual to the existing reduction

agents, and have agents of both types approach the transition boundary simultaneously from both sides. Expansion agents would profit immensely from the second instance we discarded, because it would allow them to share criticality information just as efficiently and completely as our current meta instance format does to reduction agents. These ideas seem interesting enough to be further pursued, but they could not be carried out within the time constraints for this thesis.

4.3.2 Retrieving Transition Clauses

Having decided what information to store in the meta problem and how to represent it, we now turn to the question how the desired information can be extracted from the meta instance during interactive reduction. Recall that our goal is to determine in some US $F' \subseteq F$ with $F \backslash F' := \{D_1, \ldots, D_n\}$ (with $n = 0$ if $F = F'$) all the clauses C_j which are implied to be critical by the meta instance G because their removal would lead to a set covered by one of the sat wedges. In Theorem 4.2.1, we saw that this check can be performed by calling a SAT solver on the extended meta instance $G \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_n\} \cup \{\neg s_j\}$. The most obvious way to determine the known transition clauses by means of the meta problem therefore is to perform for each C_j a SAT solver call on the meta instance with the assumptions $\{\neg t_1, ..., \neg t_n, \neg s_j\}$.

This approach could be used to speed up the processing of reduction steps by often sparing us a much more expensive SAT call on the full problem in the satisfiable case. The disadvantage is that in the case of the extended meta instance being satisfiable, the full SAT problem would still need to be solved, causing us to have wasted computation time on the first SAT solver call. Moreover, for the purposes of interactive MUS extraction, we would like to determine many or even all of these forbidden reductions whenever a node is displayed, allowing us to store the criticality information and colour the deletion clauses in red before the user even gets to select a deletion candidate. This makes it very desirable to find a much more efficient way to extract all the criticality information contained in the meta instance with respect to some US in the form of a set of transition clause IDs.

Once such an approach is developed and implemented, the prototype can be extended to perform this retrieval of transition clause IDs whenever a new US is added to the reduction graph, but also whenever the user selects some existing node. We are thus not constantly trying to redistribute all the derived information to all the nodes in the same fashion as the propagation of criticality information before, but we only derive all the information we have whenever the user (or some other agent) enters some US. We now turn to the challenge of implementing this retrieval of transition clauses in an efficient but complete fashion.

4.3.2.1 Using the Modified SAT Solver

In order to develop an approach which suits our needs, we first note that given the way we have set up the meta instance, a clause C_i is guaranteed to be a transition clause in some set $F \backslash \{D_1, \ldots, D_n\}$ if no model of the extended instance F' can be found where $\neg s_i$ as well as $\neg t_1 \wedge \cdots \wedge \neg t_n$ hold at the same time. Therefore, every literal $\neg s_j$ which follows from $F' \cup \{\neg t_1\} \cup \cdots \cup \{\neg t_n\}$ tells us that C_j is a transition clause in $F \backslash \{D_1, \ldots, D_n\}$. If we want to avoid having to call for each C_j a SAT solver first on the meta problem and often a second time on the original instance, we therefore need a method which allows us to quickly determine all the $\neg s_j$ that follow from $F' \cup \{t_1\} \cup \cdots \cup \{t_n\}$.

So how can we get access to these implied negative literals? One obvious idea is to use an existing tool part of whose task it is to derive the unit clauses following from a clause set. A SAT solver based on the DPLL algorithm does just that during the propagation phase. Using our custom variant of MiniSat, it is easy to dump the units learnt during a solver run

into a temporary file, and by reading this file it is easy to get a list of literals which follow from the meta instance. This was implemented within the prototype, but it did not yield a complete solution to our problem.

Unfortunately, the literals we can extract in this way are only a subset of the ones implied by the meta problem. The problem is that while finding a model of the meta instance, the SAT solver has three different ways of arriving at variable assignments for the model under construction. The first is unit propagation, and the second (called **splitting**) consists in making an arbitrary decision for the assignment of a variable, which can later be reverted if this choice turns out to lead to an unsatisfiable subproblem. The third way is called **pure literal elimination**, which is to set a variable to the respective truth value if it only occurs in the formula with one polarity. The effects of unit propagation are covered by extracting the learnt units, but splitting and literal elimination are more problematic. Eliminated pure literals are not considered to be learnt clauses, and will therefore not end up in the temporary file. One problem with splitting is that all clauses learnt after some splitting will include the decisions made, causing no further units to be learnt. But more crucially, if a choice made during splitting would later have followed in any case, it is one of the literals we want to extract, but it will not show up in the list of learnt unit clauses. Both these problems for using a SAT solver to extract implied literals can be seen as caused by the fact that a SAT solver is content as soon as it finds some model, and does not need to differentiate whether the assignment decisions made on the way to it were necessary or mere assumptions which helped to build a model.

4.3.2.2 Using a Java Implementation of Unit Propagation

Fortunately, it turns out that, given the structure of the meta instance, systematically propagating the selector literals representing a subset through the meta instance is enough to derive all the implicitly stored reducibility information for the subset, even without the branching part we would need for a complete DPLL decision procedure. To see why this is so, let us reconsider the proof of Theorem 4.2.1. The first part of the proof shows that mere propagation of the unit clauses containing the selector literals which characterize some subset S will suffice to detect the unsatisfiability in case S is covered by one of the sat wedges. But this gives us exactly all the reducibility information we desire, as a check against all the sat wedges is already all we need to determine whether some reduced version of the subset is already known to be satisfiable.

The Java implementation of unit propagation which was added to the prototype for this purpose relies on a simple, but reasonably efficient unit propagation algorithm first presented by James M. Crawford and Larry D. Auton [43]. Relying on the existing indexing data structure that allows for quick lookup of all the clauses which contain a given literal, and using counters for keeping track of the number of literals still remaining in each clause, their algorithm was straightforward to implement.

The implementation was found to be clearly fast enough for the intended application, although of course much more efficient algorithms for this essential component of any modern SAT solver exist. A first step towards further optimizing unit propagation would be to use the more recent algorithm by Hantao Zhang and Mark E. Stickel [44], which is however reported to only improve performance by a constant factor of about 2 on typical instances, making the effort not seem very worthwhile. More recent and involved algorithms for unit propagations could lead to significant performance gains, but these would not add much to the responsiveness of the prototype, since its main performance bottlenecks already lie elsewhere, particularly in the costly redraws of the various visualization components.

As expected, running the new implementation of unit propagation on the test sets yields a number of negative selection literals which exactly correspond to the clauses in the current subset that are implied to be critical by the meta knowledge. By exploiting unit propagation, we have therefore arrived at the desired efficient approach to extracting all the reducibility information encoded in our clausal meta instance for some specific US, and this new functionality could directly be integrated to quickly detect and colour all known transition clauses whenever the user or another agent enters some US.

4.3.3 Full SAT Solving Against Meta Constraints

With the infrastructure for complete propagation of criticality information firmly in place, we can now turn to a different more general application scenario for meta learning. This scenario is based on generating models of the meta instance. Each such model is a combination of selector variables that possibly encodes an interesting unsatisfiable subset. Depending on the assumptions which we put into the model, we get new and possibly smaller subset candidates to test for satisfiability against the full instance. If the check fails, we have determined another unsatisfiable subset, which we might integrate into the reduction graph of our interactive MUS extractor. Otherwise, we have detected a satisfiable subset which heads a sat wedge that we can use to refine the meta instance by applying meta learning just as before, including the backward subsumption check.

One problem of using models of the meta instance is that we need to make sure that the models we generate are indeed new. After finding a new US, we therefore need to extend the meta problem by a clause which explicitly forbids the combination of selector variables representing that US. Since we only want to exclude that specific US, but not an entire wedge under it like before, the clause we use for excluding a subset $S := \{C_1, \ldots, C_m\} \subset F$ with $F \backslash S := \{D_1, \ldots, D_n\}$ is $\{\neg s_1, \ldots, \neg s_m, t_1, \ldots, t_n\}$. Adding this clause to the meta instance will prevent the SAT solver running on the meta instance from producing the selector variable characterization of the same S as a model again. A problem of this approach is that each of these clauses has as many literals as the original instance has clauses, and storing many of them will therefore significantly increase the runtime of the SAT solver on the meta problem. To some degree, this problem can be remedied by the block-based storage techniques which are going to be developed in Chapter 5.

Since we want to use models of the meta instance in order to find US candidates which are of interest for MUS extraction, minimization needs to be enforced by always looking for US candidates below some subset which is already known to be unsatisfiable. This can easily be done by adding negative selector literals for all the clauses outside that subset to the assumption list given to the SAT solver. Moreover, for fast minimization, the SAT solver we employ on the meta instance should always try to set to false as many selector variables as possible. This could be achieved by ensuring that during splitting, the negative assignment should be chosen and explored first. The downside is that as long as we do not yet have strong sat wedges in the meta instance, we will often get candidate subsets which are much too small and therefore unlikely to be unsatisfiable. In practice, it will be important to maintain a careful balance between fast minimization and remaining above the transition boundary. The observation that in instances with many MUSes, all of them tend to be of similar size, suggests that if we are interested in more than one MUS this could be done by looking for models of approximately the same size as the already determined MUSes.

Note that this procedure constitutes a fundamentally different approach to finding MUS candidates from what we have seen so far. While for MUS extraction in the standard sense, the advantages of this approach are not obvious, the meta instance models start to get interesting when we are allowed to add arbitrary additional constraints over selector variables. This subject will be explored further in Section 4.4.

4.3.4 Integrating the Meta Instance into the Interface

In order to integrate meta learning into the user interface of the interactive MUS extraction prototype, a second instance of the existing clause set display component containing the current state of the MUS instance at each point was added to the graphical user interface. Whereas internally, the selector variables have IDs in the range from $n+1$ to $n+m$ (where n is the number of variables and m the number of clauses in the original instance), there is of course no need to keep the IDs separate for an instance which consists only of selector variables. It is therefore possible to make the contents and the behaviour of the meta instance easier to understand by shifting the displayed variable IDs down by n positions, such that the IDs of the selector variables displayed in the meta instance coincide with the IDs of the corresponding clauses in the original instance.

For the current purpose of complete propagation of criticality information alone, there would be no need to expose the meta instance in this way to the user. The entire meta instance and the interactions with it could be handled completely under the hood without unnecessarily cluttering the user interface. In an experimental system, however, it is appropriate to give the user some room for experimentation. Additional options are therefore provided by a context menu, as displayed in Figure 4.3. The first item gives the user access to the model-based US candidate generation we just introduced. Whenever this item is selected, the prototype will use a SAT solver to generate a model of the meta instance under the assumptions defined by the currently selected US. This model is interpreted as a new US candidate which is then checked for satisfiability. If the candidate is indeed a US, a new node is added to the reduction graph below the currently selected US. Otherwise, meta learning is applied as after a normal unsuccessful reduction step.

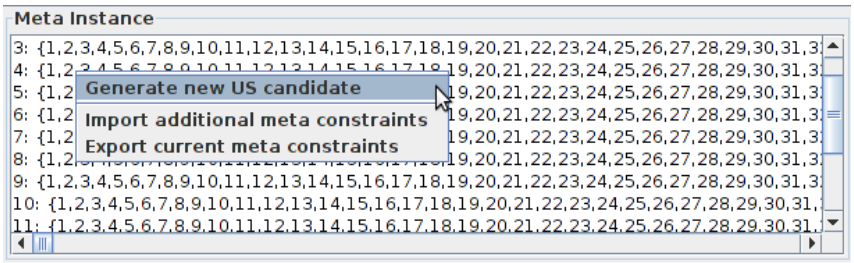


Figure 4.3: An example of the meta instance view with visible context menu.

Model-based candidate generation only really makes sense if we can add arbitrary boolean conditions over selector variables. In the current version, the context menu therefore allows the user to import additional meta constraints from another CNF file in DIMACS format. Another option allows to store the derived meta knowledge in a DIMACS file, giving the user the possibility to have the collected meta constraints persist across reduction sessions. In a future version which also supports the operations of wedge strengthening and merging, these would also be made accessible via the context menu, allowing the user to select the clauses to be strengthened or merged, and then executing the desired operation.

4.3.5 An Example of Meta Learning in the Prototype

For purposes of demonstration in this and in the next chapter, a very small example instance was hand-crafted to have as many interesting properties as possible while still being small enough to be displayed in its entirety on screenshots. In this section, this example is used to show how meta learning and backward subsumption checking are implemented in practice.

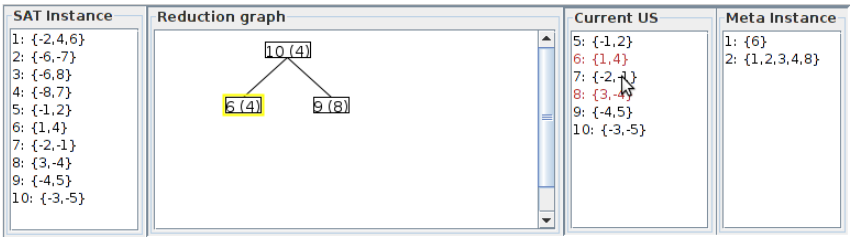


Figure 4.4: Initial situation for an example of meta learning on a simple instance.

We start our example with the situation in Figure 4.4, where we have already made two different successful reduction attempts in the top node, and we have unsuccessfully tried to delete C_6 from the top node, leading to the sat wedge $\{s_6\}$. In the currently selected US, we have furthermore attempted to delete C_8 without success, which has caused us to learn the sat wedge $\{s_1, s_2, s_3, s_4, s_8\}$. Next, we attempt to delete C_7 in the same US.

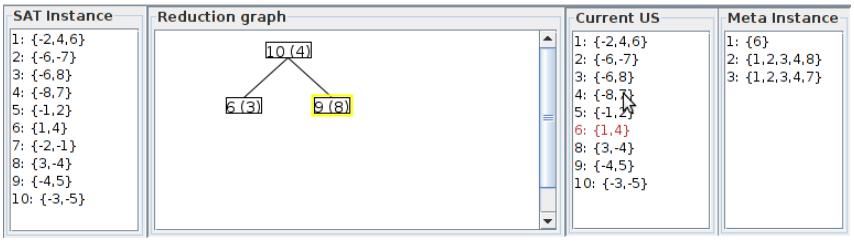


Figure 4.5: Situation after the unsuccessful deletion of clause C_7 in the example.

The result of this reduction attempt is visible in Figure 4.5, where we also have selected the other leaf US in the reduction graph. The new sat wedge $\{s_1, s_2, s_3, s_4, s_7\}$ in the meta instance reflects that our attempt to delete C_7 in the US $\{C_5, C_6, C_7, C_8, C_9, C_{11}\}$ was not successful. Let us see what happens if we attempt to delete C_4 in the newly selected US.

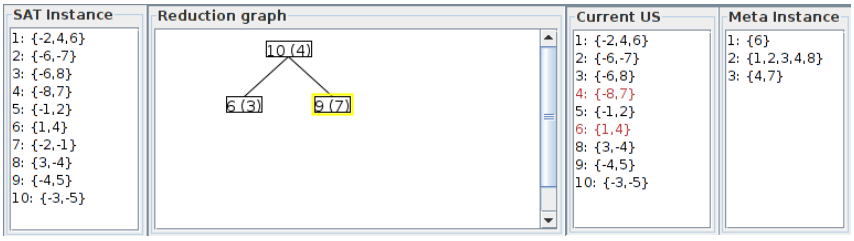


Figure 4.6: Consequences of backward subsumption by a new sat wedge $\{s_4, s_7\}$.

Again, the reduction attempt is unsuccessful. From this we can learn the sat wedge $\{s_4, s_7\}$. Figure 4.6 shows what happens when this sat wedge is added to the meta instance. The new sat wedge subsumes the existing sat wedge $\{s_1, s_2, s_3, s_4, s_7\}$, which is therefore removed during backward subsumption checking. We now have the knowledge that C_4 and C_7 can never be removed together.

4.4 Additional Meta Constraints

In this final section of the chapter on meta learning, we will go beyond the main topic and turn to the question what else apart from more complete exploitation of criticality information we can gain from maintaining a meta instance. The main aspect under which this question is explored here is to employ the meta problem for directly specifying constraints on the presence of clauses in the MUSes we extract. This general idea leads to a number of possible applications, some of which are presented in this section. In all these applications, we use the meta instance to axiomatize some additional properties which we want our MUSes to exhibit.

The applications proposed in this section only serve to demonstrate potential benefits of adding constraints over selector variables, whereas none of them has actually been implemented as part of the prototype. The main reason for this is the unclear connection to interactive MUS extraction caused by the lack of monotonicity in some interesting properties, while all the monotonic properties can somewhat indirectly be emulated by the existing mechanism for importing additional meta constraints from CNF files. For interactive reduction, we only used the meta instance to axiomatize unsatisfiability, whereas now, we are additionally using it to axiomatize further criteria for the sets we want to find. If these criteria are not monotonic in the sense that they also hold for each superset of a set where they hold, we cannot prevent that the satisfiability check against the meta instance will prematurely reject unsatisfiable subsets that would by future reductions have reached the criterion. Therefore, some of the ideas presented in this section only make sense in the mode of full SAT solving against meta constraints, where they are used to encode constraints on the combinations of selector variables which arise as models of the meta problem, and which we use as suggestions for US candidates.

4.4.1 Inclusion or Exclusion of Specific Clauses

We first come back to one of the problems we set out to solve via interactive MUS extraction. The problem of unwittingly causing clauses to fall away which were supposed to be part of the desired explanation of infeasibility was solved by interactive MUS extraction by allowing to revert reduction decisions and to explore alternative paths. However, because this approach does not change anything about the fact that we are making reduction attempts which can in principle cause arbitrary other clauses to fall away, a lot of trial and error might still be involved until the user arrives at a MUS containing a specific set of desired clauses.

Using meta constraints, it becomes possible to explicitly specify such requirements. If we have a subset $\{C_1, \ldots, C_m\}$ which we desire to be part of the extracted MUS, we can simply add the unit constraints $\{s_1\}, \ldots, \{s_m\}$ to the meta instance in order to ensure that the desired clause set must be part of each unsatisfiable subset we encounter. Note that we do not necessarily arrive at a MUS in the original sense by this method, but we are guaranteed that the USes we arrive at are minimal under the condition that all the clauses in $\{C_1, \ldots, C_m\}$ remain present. Note that the property thus expressed is monotonic, since all supersets of a set containing $\{C_1, \ldots, C_m\}$ trivially contain these clauses, too. The addition of positive selector literals as unit constraints does therefore not interfere with the validity of deletion-based interactive MUS extraction as implemented in the prototype.

An analogous scenario arises when we are sure that some clauses, though part of the original instance, cannot be part of any interesting MUS. Just as we enforced the presence of a subset $\{C_1, \ldots, C_m\}$, we can also enforce its absence by adding the unit constraints $\{\neg s_1\}, \ldots, \{\neg s_m\}$ to the meta instance. This is our first example of a property which is not monotonic, as with these constraints added, the unsatisfiability of the meta instance

will not tell us any more that all subsets of the tested subset violate the desired conditions, but it might well be that we end up in a legal US after removing some more clauses. This way of enhancing the meta instance is therefore not useful for deletion-based interactive MUS extraction, but needs to be used in the context of full SAT solving against the meta constraints for deriving US candidates.

Despite this difference, both of the scenarios we just discussed could be subsumed under the name **MUS extraction modulo unit assumptions**. The conditions we can express in this way are more general than in the previously discussed approach of merely distinguishing between relevant and don't-care clauses, because this can be emulated by simply enforcing the presence of all the don't-care clauses and ignoring them in the resulting MUSes.

4.4.2 Expressing and Generalizing GMUS Extraction

Let us now carry our approach of complementing the existing connections between selector variable assignments by additional constraints a little further. To give an example of the power of this approach, let us reconsider the task of GMUS extraction. The standard approach to ensuring that certain groups of clauses are always removed together was to assign the same selector variable to the clauses of a group. The possibility to define additional constraints over the selector variables allows us to emulate this behaviour even though we have instrumented every clause by a different selector variable.

Assume that we want the clauses $C_1, \ldots, C_4$ to belong to the same group. We can enforce these clauses to be either all absent or present by adding clauses $\{\neg s_1, s_2\}$,$\{\neg s_2, s_3\}$,$\{\neg s_3, s_4\}$, and $\{\neg s_4, s_1\}$ to the meta problem. Note that the circular dependency between these clauses enforces that as soon as one of the s_i is assigned the value false, repeated unit propagation forces all the three other s_i to also become false, and analogously for the value true. This encoding is very efficient because it is linear in the number of clauses. Still, it seems a lot more straightforward to use one common selector variable for each group, and we have not yet gained any advantage over this standard approach.

The advantages only become visible when we start generalizing the notion of a GMUS. Note that a standard GMUS problem effectively defines a partition of the instance, as every clause is assigned to exactly one group. But let us assume that we have an application where these groups overlap, e.g. two groups $\{C_1, C_2, C_3\}$ and $\{C_1, C_4, C_5\}$ where the presence of C_1 only forces the clauses from either one of the groups to be present, but the absence of C_1 forces all clauses from both groups to be absent. Such connections between groups might be applied in isolating faulty components which share parts, for example while modelling hardware or software interfaces.

In our example, the necessary conditions can easily be expressed e.g. by the meta constraints $\{\neg s_1, s_2, s_4\}$, $\{\neg s_2, s_3\}$, $\{\neg s_3, s_1\}$, $\{\neg s_3, s_2\}$, $\{\neg s_4, s_5\}$, $\{\neg s_5, s_1\}$, and $\{\neg s_5, s_4\}$. This disjunctive dependency of the clause C_1 could not be expressed within the standard GMUS paradigm. Depending on the application, even more complex connections between overlapping groups might need to be expressed. For such purposes, being able to freely define dependencies between clauses in terms of meta constraints will be very helpful.

4.4.3 Enforcing Desired MUS Size

Much more complex constraints are of course possible as well. One of the potentially most attractive options is to express an upper bound k on the size of the desired MUS, allowing us to explicitily check whether any MUS of a given size exists. The basic procedure for adding such a parametrization is simple enough: we use an efficient encoding for AtMost constraints (such as the ones described by Carsten Sinz [45] or Ben-Haim et al. [46]) to

express the requirement $\sum_{i=1}^{m} s_i \leq k$, and add the resulting clauses to the meta problem. Running the SAT solver on the meta problem would then only generate US candidates of a size smaller than k, and once the meta instance is made unsatisfiable by the sat wedges learnt from unsuccessful reduction attempts, this means that no further USes of size $\leq k$ exist.

Note that by repeating the process for ever smaller k, this parametrized US search could in principle be used to find minimum unsatisfiable subsets. It might not even be the most inefficient way, as between iterations, the sat wedges generated by failed US candidates could be reused to speed up the process for lower k. Time constraints made it impossible to implement and evaluate this algorithm in the context of this thesis. But the idea serves well to illustrate the potential value of extending MUS extraction systems by the option of defining and adding arbitrary constraints over selector variables.

4.5 Conclusion

In this chapter, we have solved the problem of efficiently propagating all the available criticality information to all subsets in our reduction graph. While an analogous method for solving the dual problem of propagating fall-away information to all supersets suggested itself, the choice was made not to implement it, because we would then have to maintain an additional complex data structure, and we saw that if we employ clause set refinement, the potential benefits do not warrant the additional overhead in memory and computation time.

To achieve complete propagation, we have introduced a mechanism for explicitly tracking and exploiting connections between assignments of values to the selector variables, which is based on maintaining a second clausal meta instance over the selector variables. Efficient ways of storing connections between selector variables and retrieving the information relevant for a given MUS were developed and implemented in the prototype. Furthermore, we have seen how the other standard paradigms of MUS extraction can be seen as procedures for simplifying the clausal meta instance. Finally, we have taken a glimpse at the additional possibilities which the paradigm of expressing and solving constraints over selector variables gives us for axiomatizing interesting additional properties for the MUSes we attempt to find.

For the purposes of standard interactive MUS extraction, however, the benefits of the meta learning approach beyond complete propagation of partial knowledge have not become too clear yet. The next chapter will reveal that the meta instance can help us to understand more about the structure of unsatisfiable subsets, and allows us to provide interactive variants of both GMUS extraction and finding minimal unsatisfiable subformulae merely by adding additional structure to the meta instance, and using this structure to restrict the possible interactions in ways which reflect the special problem structure.

A general issue with the first version of meta learning implemented in this chapter is that for instances of any interesting size, the meta clauses become very large, as each of them enumerates all the selector variables corresponding to clauses in the complement of the satisfiable subset at the top of the sat wedge. As a result, maintenance of the meta instance, and especially subsumption checking, quickly becomes rather costly and therefore slow. This issue will also be addressed in the next chapter, where the additional structure given to the meta problem will also serve as a compression scheme for large meta clauses.

Chapter 5

Block-Based Meta Learning

This chapter motivates and explores at some depth the idea of imposing additional structure on the meta instance as it was introduced in the previous chapter. The concrete approach of this idea builds on the notion of a **block** of selector literals, i.e. a disjunction of selector literals which always occur together in the clauses of the meta instance. The selector literals occuring in the meta instance will be grouped together in blocks according to some scheme which translates information derived during interactive reduction into block assignments in a meaningful way. With the help of these blocks, we will be able to resolve some of the most pressing problems of interactive MUS extraction as implemented so far in the prototype system. On the more theoretical side, we will establish some interesting connections between block-based meta learning and the problems of GMUS extraction as well as the extraction of minimal unsatisfiable subformulae from non-CNF instances.

5.1 Motivation

In this introductory section, we will look at the idea of forming blocks of selector variables from three different angles, motivating and setting the stage for the deeper exploration of the idea in the remaining sections of the chapter. The first section only views the blocks as abbreviations in what can be seen as a compression scheme, allowing us to exploit the large overlaps between meta clauses to represent them in a more space-efficient manner, with barely any overhead in extracting the relevant information. The second section then proceeds to interpret the blocks inferred by this compression scheme semantically as representing groups of clauses which in some sense belong more closely together. The closeness has a natural interpretation in terms of refutation proof subtrees. The third perspective lifts the concept of constraints over selector variables to dependencies between blocks, leading to a less granular view of the enforcement relation between the presence of different clauses which makes the relevant structure of unsatisfiable SAT instances easier to visualize and comprehend.

5.1.1 Efficiency Considerations

In the previous chapter, we mentioned as one of the main problems that the clauses learnt during meta learning tend to become very large, since each of them enumerates all the individual selector variables which correspond to the elements of the complement of some satisfiable subset. The sheer size of these meta clauses has had a detrimental impact on the runtimes of subsumption checking as well as the unit propagation approach we used for extracting the desired criticality information.

The key to a remedy of this problem is the trivial observation that there tend to be large overlaps between the literals in the meta clauses. Even the most extreme case of two large meta clauses only differing in a single literal is quite common, since this is what results from unsuccessful attempts to delete different clauses from the same unsatisfiable subset.

Using some automated scheme which groups the selector literals that often occur together in meta clauses into blocks, we will be introducing block variables as shorthands for referring to large disjunctions of selector variables in our meta instance. This approach can be viewed as using the block representation as a **compression scheme**. The space savings achieved in this way will turn out to be so significant that this measure alone makes both subsumption checking and unit propagation for extracting selector literals feasible on benchmark instances of any interesting size.

5.1.2 Conspiracies and the Role of Refutations

Beyond seeing blocks merely as a compression device, there is a way to interpret the clause sets represented by the selector literals which are grouped together in one block in a more semantically motivated way. As the block structure progressively gets more granular, the algorithms for inferring the block structure that we will use tend to group together clauses which often occur together as subtrees of refutation proofs. The reasons for this connection between inferred blocks and refutation proofs lies in the fact that the operation of clause set refinement prunes unsatisfiable subsets to only those clauses which occur in some unsatisfiability proof, and that block structure inference is performed according to a criterion of efficient compression, causing clause subsets to be grouped together which fell away together during an application of clause set refinement.

Clauses which belong together in this way will be said to **conspire**, suggesting that each of the clauses represented by a selector variable in some block contributes in a hidden way to the derivation of a clause which is of use for building refutations. Visualizing which clauses form such conspiracies will become an important element of our attempts to make the internal structure of unsatisfiable SAT instances more transparent during interactive MUS extraction. Also, the inferred clause conspiracies will become the central handle for concise descriptions of sat wedges and thereby of our knowledge about the search space.

5.1.3 Dependencies Between Blocks

The resulting new version of the interactive MUS extraction system can be viewed as keeping track of and revealing **block dependencies**, i.e. dependencies between the reducibility or criticality of blocks. The extension of the prototype by visualization components for two different block structures will support the user in thinking about the MUS extraction problem in this way. This includes changes to the workflow, as the user will normally not select individual clauses for reduction attempts (although this option continues to exist), but attempt to remove entire blocks, which are automatically refined in case the reduction was unsuccessful, differentiating between the part of the block which could be removed in the respective situation, and the part that was determined to be critical.

Many choices are involved in defining an algorithm that infers a block structure from the information that is derived during interactive reduction. Two obvious choices are whether blocks may be allowed to subsume other blocks, and whether they are allowed to overlap. Allowing both quickly leads to an uncontrollable proliferation of blocks which makes the generation of compressed meta clauses too costly. Therefore, we will first discuss the variant of forbidding both nestings and overlaps, leading to the block partition approach presented and discussed in Section 5.2. If we allow nestings, but continue to forbid overlaps, we get a recursive tree structure over blocks. This approach will be explored in Section 5.3.

5.2 Block Partitions

Our first approach to maintaining a block structure over the selector literals of the meta instance builds on inferring a partition of the clauses in the CNF instance. We will define a **block partition** as a set $\mathcal{B} := \{B_1, \ldots, B_k\}$ of clause subsets or **blocks** $B_j \subseteq F$, where for all $1 \leq i \neq j \leq k$, we have $B_i \cap B_j = \emptyset$, i.e. the blocks are defined to be disjoint. Furthermore, we impose the condition that $\bigcup B_j = F$, meaning that every clause needs to be assigned to some block. The **trivial block partition** $\mathcal{B} := \{B_1\} := \{F\}$ plays an important role as the starting point for the inference algorithm we are about to define, and therefore receives a special symbol $\mathcal{B}_0$.

The reader will have noticed that we are defining blocks in terms of clause subsets and not in terms of disjunctions of selector literals, as we did in the introductory remarks. This is warranted by the fact that in the form of the meta instance we will be building on, only positive selector variables occur in the meta clauses, so that we have an exact correspondence between blocks in the sense of clause subsets and blocks in the sense of disjunctions of the corresponding selector variables. While the algorithm in Section 5.2.1 would also be applicable for blocks that contain negative selector literals as well, the visualization and integration with interactive MUS extraction discussed in Section 5.2.2 needs to rely on the narrower sense of a block as we formally defined it here. Sometimes, we will use the correspondence between clauses C_i and selector variables s_i to implicitly identify blocks with sets of positive selector variables, often leading to a much simpler notation.

Turning to the question how a block partition is used to compress the meta instance, we first note that we only need to maintain equivalence in two important respects. Firstly, the full meta instance and the compressed instance should be equi-satisfiable, and secondly, the sets of selector units derivable by unit propagation should be identical.

Definition 5.2.1. ***(Block variables and block access function)***
Let $G := \{D_1, \ldots, D_n\}$ be a meta instance for the CNF instance $F := \{C_1, \ldots, C_m\}$ with meta clauses over positive selector literals $s_1, \ldots, s_m$. Let $\mathcal{B} := \{B_1, \ldots, B_k\}$ be a block partition of F. For each B_j, we define a corresponding ***block variable*** *b_j, accessible by a function $blockVar(B_j) := b_j \; \forall 1 \leq j \leq k$. Moreover, we define a* ***block access*** *function by $blocks(s_i) := \{B_j \mid C_i \in B_j\}$.*

Note that because of the disjointness of the block partition, $blocks(s_i)$ will be a singleton set for each s_i here, which we sometimes identify with the single element of that set.

Definition 5.2.2. ***(Meta instance representability by a block partition)***
For an integer threshold $n_{max} \geq 0$, a meta instance $G := \{D_1, \ldots, D_n\}$ for a CNF instance F is called ***n_{max}-representable*** *in a block partition $\mathcal{B}$ over F iff for each $1 \leq i \leq n$ and each $s_{ij} \in D_i$ with $|blocks(s_{ij}) \cap D_i| > n_{max}$, we have $blocks(s_{ij}) \subseteq D_i$.*
For each set of positive selector variables D and each $s \in D$, we furthermore define the function $\textbf{blockRep}(s, D) := \begin{cases} \{b_j\} & if\ \{B_j\} = blocks(s) \subseteq D \\ \{s\} & else \end{cases}$.

Note that the expression $|blocks(s_{ij}) \cap D_i| > n_{max}$ uses all the notational conventions we have introduced so far to express the condition that the overlap between D_i and the selector variables for the clauses contained in the block that s_{ij} is assigned must exceed the threshold value n_{max} for the inclusion relation to be enforced. This definition can be seen as ensuring a degree of compatibility (parametrized by n_{max}) between a meta instance and a block partition, because shortening a clause by a shortcut only works if the definition of the shortcut (almost) completely fits into the clause. The higher we set n_{max}, the larger the allowed overlaps with blocks that are not completely contained in D_i become. In practice, we will mostly work with the threshold values $n_{max} := 1$, allowing the presence of a single selection variable from each block not covered, or $n_{max} := 0$, not allowing any overlaps at

all. Building on parametrized representability, we can now formally define a compression scheme for meta instances:

Definition 5.2.3. *(**Meta instance compression by a block partition**)*
*Let $G := \{D_1, \ldots, D_n\}$ be a meta instance for the CNF instance $F := \{C_1, \ldots, C_m\}$, and let $\mathcal{B} := \{B_1, \ldots, B_k\}$ be a block partition over F in which G is n_{max}-representable. We define the **compression** G' of G by $\mathcal{B}$ as consisting of*

- *the clause $\{\neg b_j, s_{j1}, \ldots, s_{jm_j}\}$ for each $B_j := \{C_{j1}, \ldots, C_{jm_j}\}$, and*
- *the clause $\bigcup_{j=1}^{n_i} blockRep(s_{ij}, D_i)$ for each meta clause $D_i := \{s_{i1}, \ldots, s_{in_i}\}$.*

The first type of clause, encoding the implication $b_j \to (s_{j1} \vee \cdots \vee s_{jm_j})$, is also called a **block definition clause**, and the second type of clause is called a **compressed sat wedge**. Let us show that the compression scheme does indeed have the desired properties:

Theorem 5.2.4. *(**Equisatisfiability of meta instance compression**)*
For a CNF instance F and a block partition $\mathcal{B}$ of F, an arbitrary meta instance G that is n_{max}-representable in $\mathcal{B}$ is satisfiable iff the compression G' of G by $\mathcal{B}$ is satisfiable.

Proof. $\Rightarrow$: Let ϑ be a model of G. We show that ϑ can be extended to a model ϑ' of G', i.e. an assignment with $\vartheta'(s_i) := \vartheta(s_i)$ for $s_1, \ldots, s_m$ which satisfies all the clauses of G'. We extend ϑ' to the block variables by considering all $D_i := \{s_{i1}, \ldots, s_{im_i}\} \in G$ in order. Since ϑ satisfies D_i, there must be a $1 \leq j \leq m_i$ with $\vartheta(s_{ij}) = 1$. If $blockRep(s_{ij}, D_i) = \{s_{ij}\}$, then the compressed sat wedge for D_i contains s_{ij} and is satisfied by ϑ'. If $blockRep(s_{ij}, D_i) = \{b_{ij}\}$, then the defining clause of b_{ij} contains s_{ij}, and is satisfied by ϑ'. This means that the only negative occurrence of $\neg b_{ij}$ becomes irrelevant, so that b_{ij} is a pure literal, allowing us to set $\vartheta'(b_{ij}) := 1$, causing ϑ' to satisfy the compressed sat wedge $\bigcup_{j=1}^{m_i} blockRep(s_{ij}, D_i)$. Having processed all D_i in this way, such that all the compressed sat wedges are satisfied, there might be some blocks B_j whose defining clauses $\{\neg b_j, s_{j1}, \ldots, s_{jn_j}\}$ we have not touched and therefore not satisfied yet. All these can be satisfied by simply defining $\vartheta'(b_j) := 0$.
$\Leftarrow$: Let ϑ' be a model of G'. We show that by restricting ϑ' to the selector variables, we receive a model of G. For every $D_i := \{s_{i1}, \ldots, s_{im_i}\} \in G$, we need to show that $\vartheta'(s_{ij}) = 1$ for some j. We know that ϑ' satisfies $\bigcup_{j=1}^{m_i} blockRep(s_{ij}, D_i)$, i.e. that $\vartheta'(blockRep(s_{ik}, D_i)) = 1$ for some $1 \leq k \leq m_i$. If $blockRep(s_{ik}, D_i) = \{s_{ik}\}$, then we have $\vartheta'(s_{ik}) = 1$, and D_i is satisfied. If $blockRep(s_{ik}, D_i) = \{b_k\}$ for some block variable b_k, we consider the block definition clause $\{\neg b_k, s_{k1}, \ldots, s_{kn_k}\}$. Since $\vartheta'(b_k) = 1$ and ϑ' satisfies the clause, we must have $\vartheta'(s_{kj}) = 1$ for some $C_{kj} \in B_k$. The representability of G in $\mathcal{B}$ gives us $s_{kj} \in D_i$, so that D_i is satisfied in this case as well. □

Theorem 5.2.5. *(**Unit propagation equivalence under meta instance compression**)*
For any consistent assumption set $A := \{l_1, \ldots, l_k\} \subseteq \bigcup_{i=1}^{m} \{s_i, \neg s_i\}$ and a compression G' of a meta instance G for F, every selector variable unit which is derived during unit propagation on $G \cup \{l_1\} \cup \cdots \cup \{l_k\}$ is also derived during unit propagation on $G' \cup \{l_1\} \cup \cdots \cup \{l_k\}$.

Proof. We proceed by induction over the propagation steps. In the base case, the derived unit $\{l\}$ is one of the assumptions l_i. Since the assumption units in both cases are identical, the unit $\{l\}$ will trivially also be derived from unit propagation on the compressed instance. Otherwise, there must have been a meta clause $D_i = \{s_{i1}, \ldots, s_{in_i}\}$ which was reduced to $\{l\}$ by previous propagation steps, i.e. $l = s_{ip}$ for some $1 \leq p \leq n_i$. The induction hypothesis in this case is that the propagated literals $\neg s_{i1}, \ldots, \neg s_{ip-1}, \neg s_{ip+1}, \ldots, \neg s_{in_i}$ have already been derived and propagated during unit propagation on $G' \cup \{l_1\} \cup \cdots \cup \{l_k\}$. Now consider the compressed sat wedge $\bigcup_{j=1}^{n_i} blockRep(s_{ij}, D_i)$ for D_i.
We first show that all elements of each $blockRep(s_{ij}, D_i)$ except those of $blockRep(s_{ip}, D_i)$ were cancelled out by propagating the units given by the induction hypothesis. Consider an arbitrary $j \neq p$. If $blockRep(s_{ij}, D_i) = \{s_{ij}\}$, this single element has trivially been cancelled out while propagating $\neg s_{ij}$. If $blockRep(s_{ij}, D_i) = \{b_q\} \neq blockRep(s_{ip}, D_i)$ for

some block definition variable b_q, consider the block definition clause $\{\neg b_q, s_{q1}, \ldots, s_{qn_q}\}$. By the representability of G in the block partition $\mathcal{B}$, we must have $\{s_{q1}, \ldots, s_{qn_q}\} \subset D_i$, so by the disjointness of blocks and the induction hypothesis, we know that $\{s_{q1}\}, \ldots, \{s_{qn_q}\}$ have been derived and propagated. But this means that the unit $\{\neg b_q\}$ was derived and propagated next, cancelling out $blockRep(s_{ij}, D_i)$ in this case as well.
Finally, consider the only remaining unit $blockRep(s_{ip}, D_i)$ in the compressed sat wedge. If $blockRep(s_{ip}, D_i) = \{s_{ip}\}$, we have already derived the unit clause $\{s_{ip}\} = \{l\}$. If $blockRep(s_{ip}, D_i) = \{b_q\}$ for some block definition variable b_q, $\{b_q\}$ is propagated, reducing the block definition clause $\{\neg b_q, s_{q1}, \ldots, s_{qn_q}\}$ to $\{s_{q1}, \ldots, s_{qn_q}\}$, which now only contains s_{ip} along with other variables, all of which must have been contained in D_i, therefore being cancelled out by the units propagated according to the induction hypothesis. But this means that we have also derived the unit $\{s_{ip}\} = \{l\}$ in this last case.
This concludes our inductive proof. □

Let us conclude the exposition with a simple example. Although we have not yet discussed the issue how a good block partition can be inferred and maintained during interactive reduction, we can already give an example of an intermediary state of the process, see how the definitions and theorems just introduced are reflected in it, and gain a first impression of compression efficiency. In Figure 5.1, there is a meta instance G consisting of five clauses over 15 selector variables. In the compressed variant on the right, the block definition clauses are mentioned last, and the sat wedges represented by each line are identical. Already in this toy example, the space efficiency of the partition-based compression scheme becomes apparent. In real-life examples where each of the uncompressed meta clauses consists of thousands of selector variables, the effect is of course more pronounced, as we shall see in a benchmark at the end of Section 5.2.1.

5.2.1 Algorithm

With the essential theory of meta instance compression by a block partition in place, we now turn to the practical question how the compression and, most importantly, the maintenance and adaptation of a block partition $\mathcal{B}$ in which G remains representable is implemented in practice. To make the definition of the algorithm more concise, we first introduce an auxiliary procedure (Algorithm 6) which splits an element of $\mathcal{B}$ into two subblocks, and adapts the representation of the meta instance accordingly.

With this important helper method defined, it becomes straightforward to write a method $ensureRepresentability(\mathcal{B}, G, D, n_{max})$ for refining the block partition $\mathcal{B}$ in order to maintain n_{max}-representability when an arbitrary clause D of selector variables is added to G. The pseudocode for this method is given as Algorithm 7.

uncompressed meta instance G	compression G'
$\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$	$\{b_1, b_2, b_3, s_{13}\}$
$\{s_1, s_2, s_3, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{14}, s_{15}\}$	$\{b_1, b_2\}$
$\{s_1, s_2, s_3, s_4, s_{14}, s_{15}\}$	$\{b_1, s_4\}$
$\{s_1, s_2, s_3, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{12}, s_{13}, s_{14}, s_{15}\}$	$\{b_1, b_2, b_4\}$
$\{s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{12}\}$	$\{b_2, s_4, s_{12}\}$
	$\{\neg b_1, s_1, s_2, s_3, s_{14}, s_{15}\}$
	$\{\neg b_2, s_5, s_6, s_7, s_8, s_9, s_{10}\}$
	$\{\neg b_3, s_4, s_{11}\}$
	$\{\neg b_4, s_{10}, s_{13}\}$

Figure 5.1: Example of a meta instance, and its compression by a block partition.

Algorithm 6 blockPartitionSplit($\mathcal{B}$,G',B,B')

Input: meta instance G' compressed by block partition $\mathcal{B}$, block $B \in \mathcal{B}$, subblock $B' \subset B$
Output: removes the block B from $\mathcal{B}$, adds new blocks B' and $B \backslash B'$ instead, adapts clauses in G' such that it remains the compression of G by $\mathcal{B}$

1: $\mathcal{B} := \mathcal{B} \backslash \{B\}$ ▷ remove split block from partition
2: $G' := G' \backslash \{\{\neg b\} \cup \{s_i \mid C_i \in B\}\}$ ▷ remove block definition clause for B
3: $B_1 := B'$; $\mathcal{B} := \mathcal{B} \cup \{B_1\}$ ▷ new block, block ID = 1 without loss of generality
4: $G' := G' \cup \{\{\neg b_1\} \cup \{s_i \mid C_i \in B'\}\}$ ▷ add block definition clause for B_1
5: $B_2 := B \backslash B'$; $\mathcal{B} := \mathcal{B} \cup \{B_2\}$ ▷ new block, block ID = 2 without loss of generality
6: $G' := G' \cup \{\{\neg b_2\} \cup \{s_i \mid C_i \in B \backslash B'\}\}$ ▷ add block definition clause for B_2
7: **for** $D' \in G'$ with $b \in D'$ **do** ▷ in all meta clauses with B's definition variable b ...
8: $D' := D' \backslash \{b\} \cup \{b1, b2\}$ ▷ ... replace b with the new representation of $B' \cup B \backslash B'$
9: **end for**

Algorithm 7 ensureRepresentability($\mathcal{B}$,G',D,n_{max})

Input: meta instance G' compressed by $\mathcal{B}$, new meta clause $D = \{s_1, \ldots, s_n\}$
Output: refines $\mathcal{B}$ to ensure n_{max}-representability of D

1: $S := D$ ▷ agenda of selector variables to represent in D'
2: **while** $S \neq \emptyset$ **do**
3: $B_{max} := \arg\max_{B \in \mathcal{B}} |B \cap S|$ ▷ determine block with maximal overlap to S
4: **if** $n_{max} < |B_{max} \cap S| < |B_{max}|$ **then** ▷ B_{max} partially outside S, split needed
5: $blockPartitionSplit(\mathcal{B}, G, B_{max}, B_{max} \cap S)$
6: **end if**
7: $S := S \backslash B_{max}$
8: **end while**
9: **return**

In the beginning, the meta instance G_0 only consists of one clause $\{s_1, \ldots, s_m\}$, expressing the trivial knowledge that the empty clause set (the complement of $F = \{C_1, \ldots, C_m\}$) is satisfiable. We start with the trivial block partition $\mathcal{B}_0$ consisting of a single block $B_0 = F$. According to the definition, the compression of G_0 under $\mathcal{B}_0$ is $G'_0 := \{\{\neg b_0, s_1, \ldots, s_m\}, \{b_0\}\}$. To see that Algorithm 7 does what it is designed to do, we can now formulate and prove the following theorem:

Theorem 5.2.6. ***(Correctness of ensureRepresentability)***
Let $D_1, D_2, \ldots, D_k$ be a series of meta clauses. Starting with $\mathcal{B}_0$ and G_0 as defined above, we sequentially derive $\mathcal{B}_i$ and G'_i by calling ensureRepresentability$(\mathcal{B}_{i-1}, G'_{i-1}, D_i, n_{max})$. Then, the meta problem $G_k := \{D_1, \ldots, D_k\}$ is n_{max}-representable in $\mathcal{B}_k$.

Proof. We proceed by induction over i, the index of the D_i last added.
"$i = 0$": G_0 is obviously n_{max}-representable in $\mathcal{B}_0$, as $\forall 1 \leq i \leq m : blocks(s_i) = \{B_0\} = G_0$.
"$i - 1 \to i$": The step corresponds to calling $ensureRepresentability(\mathcal{B}_{i-1}, G'_{i-1}, D_i, n_{max})$ on a new meta clause $D_i := \{s_{i1}, \ldots, s_{in_i}\}$. We need to show that $G_{i-1} \cup \{D_i\}$ is n_{max}-representable in the changed block partition $\mathcal{B}_i$ that results from executing this call.
By the induction hypothesis, every $D := \{s_1, \ldots, s_k\} \in G_{i-1}$ is n_{max}-representable in $\mathcal{B}_{i-1}$. This will not change during any call to $blockPartitionSplit(\mathcal{B}_{i-1}, G_{i-1}, B_{max}, B_{max} \cap S)$, because an execution of this method can only cause the block assigned to a variable to become smaller, not breaking the condition $blocks(s_j) \subseteq D$ for any j.
To prove that the definition of n_{max}-representability holds for the newly added clause D_i, we consider each $s_{ij} \in D_i$ in turn. In one iteration of the while loop, we must have $s_{ij} \in B_{max}$. Let us refer to this B_{max} by the symbol B_j. We need to distinguish three cases. If $B_j = \{s_{ij}\}$, then trivially $blocks(s_{ij}) = \{s_{ij}\} \subseteq D_i$. If $|blocks(s_{ij}) \cap D_i| \leq n_{max}$, then the representability condition for s_{ij} trivially holds because its antecedent does not.

Otherwise, the condition in line 4 holds because $|D \cap B_j| \geq |S \cap B_j|$. After the execution of *blockPartitionSplit*, s_{ij} will end up in a new block $B_1 := B_j \cap S \subseteq B_j \cap D_i$, so that we have $blocks(s_{ij}) = B_1 \subseteq B_j \cap D_i \subseteq D_i$, fulfilling the condition in this case as well. □

Turning to the question of runtime complexity, we see that it is linear in the size of the added clause, and also linear in the sum of instance and meta instance size, making the maintenance of the data structure affordable also for larger applications:

Theorem 5.2.7. ***(Complexity of ensureRepresentability)***
The worst-case runtime complexity of ensureRepresentability$(\mathcal{B}, G', D, n_{max})$ *is in* $O(l(m+n))$ *for* $l := |D|$, $m := |F|$, *and* $n := |G|$, *i.e. the number of stored sat wedges.*

Proof. The complexity of computing $B \cap S$ is linear in $|B|$. Because of the disjointness of blocks, we have $\sum_{B \in \mathcal{B}} |B| \leq m$. Therefore, the search for the maximum overlap in line 3 can be executed in $O(m)$. In the worst case, the while loop in lines 2-8 is iterated $O(l)$ times, because we can only guarantee that one of the elements of S falls away in each iteration. Turning to the complexity of the calls to *blockPartitionSplit*, we note that all its lines can be executed in $O(1)$ except the for loop in lines 7-9. In the worst case, b is contained in every compressed sat wedge, meaning that the loop can be executed up to $O(n)$ times. On efficient data structures for clause set representation, line 8 can be executed in $O(1)$, giving us a runtime of $O(n)$ for each execution of *blockPartitionSplit*. Altogether, we get the stated runtime complexity of $O(l(m+n))$. □

Algorithm 8 addSatWedge($\mathcal{B}$,G',D)

Input: meta instance G' compressed by $\mathcal{B}$, n_{max}-representable clause $D = \{s_1, \ldots, s_n\}$
Output: changes G' into the compression of $G \cup \{D\}$

```
1:  D' := {}
2:  S := D                                  ▷ agenda of selector variables to represent in D'
3:  while S ≠ ∅ do
4:      B_max := arg max_{B∈B} |B ∩ S|      ▷ determine block with maximal overlap to S
5:      if B_max ∩ S = B_max then
6:          D' := D' ∪ {b_max}              ▷ b_max is the block definition variable for B_max
7:      else if |B_max ∩ S| ≤ n_max then ▷ B_max partially outside S, but overlap acceptable
8:          D' := D' ∪ (B_max ∩ S)
9:      else
10:         print("ERROR: D not n_max-representable!")
11:         return
12:     end if
13:     S := S\B_max
14: end while
15: G' := G' ∪ {D'}
16: return
```

Having developed an algorithm for ensuring the n_{max}-representability of a new sat wedge $D = \{s_1, \ldots, s_n\}$, it remains to define a method $addSatWedge(\mathcal{B}, G', D)$ which computes the compressed sat wedge D' out of D in such a way that $G' \cup \{D'\}$ remains the compression of $G \cup \{D\}$ under $\mathcal{B}$. This is achieved by the pseudocode given in Algorithm 8, as the following theorem shows:

Theorem 5.2.8. ***(Correctness of addSatWedge)***
Let G' be the compression of a meta instance G under a block partition $\mathcal{B}$. For a new meta clause $D := \{s_1, \ldots, s_n\}$, let $G \cup \{D\}$ be n_{max}-representable in $\mathcal{B}$. After the function call $addSatWedge(\mathcal{B}, G', D)$, G' is the compression of $G \cup \{D\}$ under $\mathcal{B}$.

Proof. The call to *addSatWedge* does not change anything about the block declarations in G', so we only have to show that the generated D' is equal to $\bigcup_{j=1}^{n} blockRep(s_j, D)$, i.e. that for each s_j, D' contains b_j if $\{B_j\} := blocks(s_j) \subseteq D$, and s_j itself otherwise. In some iteration, we have $B_{max} = B_j = blocks(b_j)$. The n_{max}-representability of D gives us either $B_j \subseteq D$, which is covered by the condition of line 5, or $|B_j \cap D| \leq n_{max}$, which is covered by that in line 7. In the first case, by the disjointness of blocks we have $B_j \subseteq S$, i.e. $B_j \cup S = B_j$, such that b_j is added to D' in line 6. In the second case, we get $D' := D' \cup \{B_j \cap S\}$, which by the disjointness of blocks means $s_i \in D'$. □

Theorem 5.2.9. ***(Complexity of addSatWedge)***
The worst-case runtime complexity of $addSatWedge(\mathcal{B}, G', D)$ *is in* $O(l \cdot m)$*, where* $l := |D|$ *is again the size of the sat wedge, and* $m := |F|$ *the size of the original instance.*

Proof. Again, the search for the maximum overlap in line 4 can be executed in $O(m)$, all the other operations within the while loop are dominated by this, and we cannot guarantee anything better than $O(l)$ iterations of the while loop in the worst case. □

In a final step, we now consider how the *ensureRepresentability* and *addSatWedge* methods can be combined to infer a block partition $\mathcal{B}$ and maintain the corresponding compression G' while incrementally adding sat wedges to G. The correctness results show that this could be achieved by simply setting $n_{max} := 1$ and calling *ensureRepresentability* before each call to *addSatWedge*. In the context of interactive reduction, we can do better if we exploit that typical unsuccessful reductions will always lead to sat wedges that contain one-variable extensions to the positive selector literals representing some US in the reduction graph. This means that it suffices to call *ensureRepresentability* with $n_{max} := 0$ on the US representation of the new US in the case of a successful reduction to achieve the necessary 1-representability for following calls to *addSatWedge* with $n_{max} := 1$ after each unsuccessful reduction attempt. This is exactly how block partition inference was implemented in the prototype. For simultaneous reductions of multiple clauses and other operations such as autarky reduction which cause larger overlaps, an additional call to *ensureRepresentability* remains necessary.

Another issue that we have not yet touched upon is the compatibility of block partition handling and subsumption checking as we implemented it for the uncompressed case. The problem is best explained in an example. Assume that we are learning a sat wedge $\{s_1\}$ which subsumes a block definition clause $\{\neg b_1, s_1, s_2\}$. By default, backward subsumption checking would remove the block definition clause when adding the sat wedge to the meta instance, breaking the partition structure in an uncontrollable way. To avoid such effects, block definition clauses must be exempt from backward subsumption checking.

In Figure 5.2, we return to our small example from the last chapter to illustrate what the inference algorithm thus defined does. We start with a reduction graph consisting of two nodes. The successful reduction step between these nodes has split the instance into two blocks, one containing the clauses $\{C_1, C_2, C_3, C_4, C_7\}$ which fell away during that operation, the other containing the clauses of unknown status. Executing another reduction attempt for the reduction candidate C_8, we arrive at a new US of size 9. From this successful reduction attempt, we want to learn the sat wedge $\{C_7, C_8\}$. We show the results of the call to *ensureRepresentability* with $n_{max} := 0$. Since the elements of this sat wedge come from different blocks, these elements are removed from their respective blocks. Both C_7 and C_8 end up as singleton blocks as a result of the algorithm.

With block partition inference implemented in the prototype, we can now say more about the degree of compression we can achieve by using it. While proving theoretical results about the goodness of the compression is beyond the scope of this thesis, we can get an impression of rates of reduction in meta instance size by running three reduction agents on

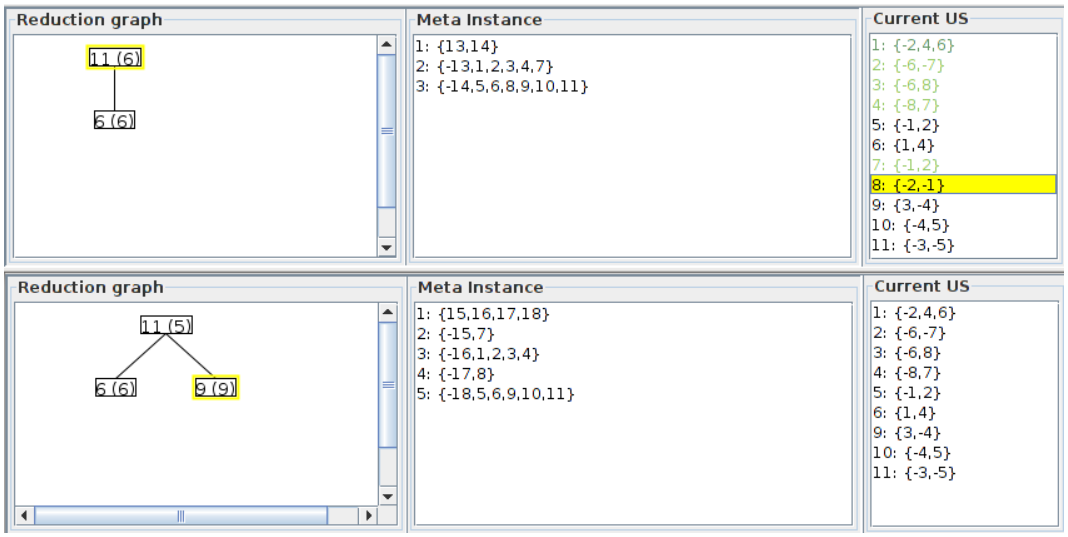


Figure 5.2: Example of reduction step while inferring a block partition.

five random unsatisfiable instances from two standard test sets for MUS extraction. For the experiment, the already mentioned Daimler testset for automotive product configuration [32] and another instance set from the test data for the SAT competition 2011 [10] were chosen. The results for these instances are summarized in Figure 5.3. The computed values are an approximation to the memory size needed for the original meta instance and the compressed instance after running three reduction agents with descending, ascending and centered relevance heuristics until each of them found a MUS. The learnt sat wedges were added to the meta instance using the procedure just sketched. In the experience of the author, the very high compression rates achieved here are typical.

5.2.2 Interactive Visualization

So far, we have mainly viewed blocks as a means of compressing the meta instance. We now turn to the structure-revealing aspect of the inferred block partition. To understand why the inferred blocks reflect valuable information about clause conspiracies, consider what happens whenever there is a successful reduction followed by clause set refinement which leads us from some US S to another US $S' \subset S$. As soon as we learn that some clause $C_i \in S'$ is critical,

Test set and instance name	$\lvert F\rvert$	$\lvert G\rvert$	$\lvert G'\rvert$	$\sum_{D\in G} \lvert D\rvert$	$\sum_{D'\in G'} \lvert D'\rvert$
SAT11-Competition: `bf1355-127.cnf`	7,306	439	441	3,143,824	8,186
SAT11-Competition: `bf1355-462.cnf`	7,305	469	472	3,353,061	8,493
SAT11-Competition: `bf1355-530.cnf`	7,305	253	255	1,827,249	7,813
SAT11-Competition: `bf1355-666.cnf`	7,305	343	345	2,466,969	7,993
SAT11-Competition: `bf1355-741.cnf`	7,307	247	249	1,784,903	7,803
Daimler: `C168_FW_SZ_66.cnf`	5,425	280	284	1,493,047	6,355
Daimler: `C202_FW_SZ_103.cnf`	10,283	428	437	4,252,915	12,193
Daimler: `C208_FA_SZ_121.cnf`	5,278	97	99	508,990	5,474
Daimler: `C210_FW_RZ_57.cnf`	7,405	76	78	560,980	7,559
Daimler: `C220_FV_SZ_55.cnf`	5,753	916	923	4,989,549	10,133

Figure 5.3: Benchmark results for meta instance compression by block partition.

we will learn a meta clause which contains all the selector variables for $F \backslash S'$. Often, we will also have some criticality information about S in the form of meta clauses which contain all the selector variables for $F \backslash S$. If the new meta clause is added to the compressed instance, the block inference algorithm will in most cases produce a new block containing all the selector variables for $S \backslash S'$. This means that blocks will very often represent groups of clauses which fell away together during a successful reduction step followed by clause set refinement.

As mentioned in Section 5.1.2, there is a strong connection between such conspiracies and subtrees of refutation proofs. This gives us a semantic way to interpret the inferred block partition. Blocks are connected to fragments of refutation proofs, and the ways in which blocks can be combined into unsatisfiable subsets reveal a lot about necessary and optional contributions to the unsatisfiability. Thoroughly investigating the connections between block partitions and possible refutation proofs goes beyond the scope of this thesis, but already at this highly informal level we can see that a block partition along with dependencies between the blocks tells us more than individual MUSes could about the structure of the unfeasibility we are analyzing.

To make the block information directly and intuitively accessible to the user, the obvious choice was to extend the prototype's graphical user interface by a **block display** component. The most direct and scalable way to implement such a view was found to be another custom variant of Swing's `JList` component, which by default is placed between the reduction graph visualization and the current US view. Each list entry in the block display represents one block of positive selector literals by a list of associated clause IDs. Like in the US view, the block display uses font colours to mark the status of different blocks with respect to the current US. The colour coding makes it possible to understand at a glance how the current US is composed, and which parts of it are already known to be critical or unnecessary. Since the colours are updated whenever a different US is selected, the block display is also an important tool for understanding how much different USes have in common. The visual and colour-coded representation of the relevant structural features makes it easy to spot overlaps. The default block colours and their semantics are listed in Figure 5.4.

Color	**Explanation**
grey	the entire block is outside the US
dark red	the entire block is critical in the US
light red	some clauses in the block are critical in the US
dark green	the entire block was successfully removed from the US
light green	some clauses in the block were successfully removed from the US
black	unkown status, none of the other conditions apply

Figure 5.4: The default colour schema for partition block status encoding.

In Figure 5.5, we see an example of the block view in the context of the other views during a reduction process for our simple example instance. The currently selected US of size 10 is displayed in the current US view. Note that the reduction steps we have executed so far have led the system to infer a block partition with three blocks of size 1, which is very typical behaviour in constructed example instances, but much less common in large instances derived from applications. The reader will notice that the blocks are coloured according to the colour schema defined in Figure 5.4. For instance, the block $\{6, 9, 10, 11\}$ is coloured in light red to indicate that some of its members have already found to be critical, so that we can be sure we will not be able to remove the entire block. The singleton block $\{5\}$ is coloured in grey because it is not part of the current US. Note that already in the case of this small instance which is still possible to comprehend as a clause list, the grouping of clauses

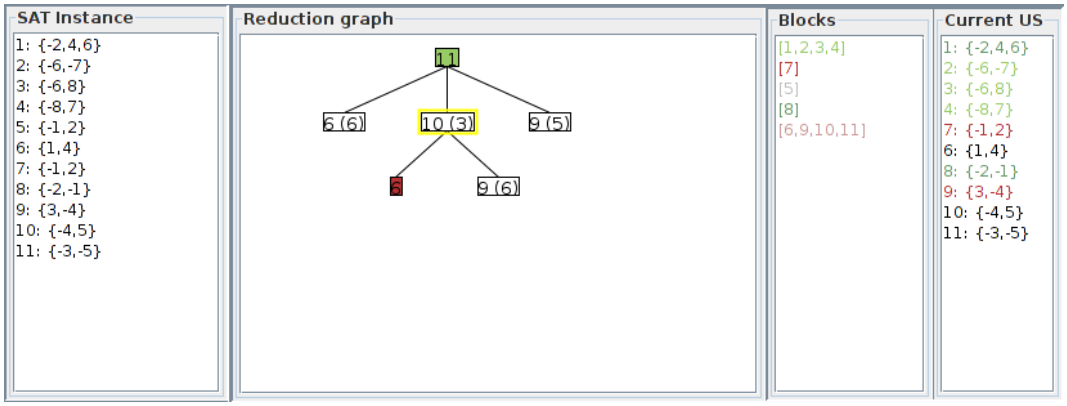


Figure 5.5: An example of the block display during a reduction process.

defined by the inferred blocks tells us a lot more about the structure of the problem. For instance, we know that the conspiring clauses with IDs 6, 10, and 11 are likely to be critical clauses just like their block mate 9. Inspection of the other USes would also allow us to see at a glance how far the two USes of size 6 overlap by comparing the blocks' status colours.

The block display can be used for purposes beyond mere visualization of conspiring clauses and overlaps between MUSes. We can extend the notion of interactive reduction to not only work on single clauses (or selected subsets) in a MUS, but on inferred blocks. Any maximal group of clauses that have always fallen away together will form a block. It is clear that in order to explore new parts of the reduction graph, it seems a reasonable move to try out whether the clauses of a block of unknown status can again all be removed in one step. This suggests an additional interaction pattern between the block display and the US display where the block display allows us to select a block, whose members are then selected in the US display. Since by far the most common action a user will want to execute on a block is the attempt to remove it, workflow efficiency can further be increased by making this action accessible via a double click on a block. In practice, a fallback option in case the entire block could not be removed at once has turned out to be convenient. In the current implementation, the fallback mechanism attempts to reduce single clauses in the block until either all of them are known to be critical, or a successful reduction is performed, causing the block to be split. With this mechanism, it becomes possible to perform manual MUS extraction via double clicks in the block display alone. In the resulting workflow, the user breaks down the large MUS instance into ever smaller blocks, throwing away parts of blocks until all the remaining blocks are critical.

5.2.3 Application to GMUS Extraction

Blocks can not only be implicitly defined by an inference algorithm, but there is also the possibility to define and integrate blocks a priori. An obvious application for this is to implement group MUS extraction based on the block partition approach we have just developed.

Manual GMUS extraction can very straightforwardly be emulated in the extended prototype by adding a pre-defined block for each group to the meta instance when loading the group CNF instance. The only necessary changes to the interface are to deactivate the fallback procedure in case of unsuccessful block removal, to switch off clause set refinement, and to disallow the selection and reduction of individual clauses by means of the US view.

This leaves the user only with the option of removing entire blocks which exactly correspond to the groups defined in the GMUS instance. The selector variables for don't-care clauses can simply be left unassigned, and their selector variables be set to true. This will cause the variables to not be available in the block view.

Apart from the block definition clauses, the resulting meta problem will only contain sat wedges which are expressible by a disjunction of block definition variables, giving these variables exactly the role selector variables played in Nadel's approach to GMUS extraction (see Section 2.3.2). For automated GMUS extraction, the reduction agents would also need to be adapted to not use the forbidden options and techniques, and to recognize a US as a GMUS as soon as all remaining blocks are known to be critical.

The close analogy between block partitions and GMUS instances can also be viewed from another angle. Block partition inference is essentially a method for automatically turning any SAT instance into a GMUS instance by grouping together clauses which are closely connected. In essence, we have thus developed a scheme for splitting up any unsatisfiable SAT instance into interesting components. In Section 6.5, we will investigate the question in how far the resulting components are meaningful in an application.

5.3 Block Trees

Our second approach to maintaining a block structure over the selector literals of the meta instance allows blocks to contain other blocks, inferring a recursive block structure over the clauses in the CNF instance. We will define a **block tree** as a tuple $\mathcal{BT} := (V, E) := (\{B_0, B_1, \ldots, B_k\}, \{(B_i, B_j) \mid B_i \supset B_j,\ \neg\exists h\colon B_i \supset B_h \supset B_j\})$, where again we are building on blocks $B_j \subseteq F$. The non-overlap condition is weakened to allow for inclusion, we merely postulate that for all nodes B_h, B_i, B_j, we have $(B_h, B_i), (B_h, B_j) \in E \Rightarrow B_i \cap B_j = \emptyset$, i.e. child blocks need to be disjoint subsets of the parent block. In addition, we impose the condition $\bigcup_{(B_i,B_j)\in E} B_j = B_i$, i.e. every clause in a non-leaf block must be contained in one of its child blocks, which implies that every clause is assigned to one of the leaves of the block tree. Finally, we postulate that there is a **top block** $B_0 := F$ containing all clauses, which completes the axioms ensuring that the block tree as defined above is indeed a tree. This time, the **trivial block tree** $\mathcal{BT}_0 := (\{B_0\}, \{\}) := (\{F\}, \{\})$ will be the starting point for our inference algorithm.

The set of **leaf blocks** $L(\mathcal{BT}) := \{B_i \in V \mid \neg\exists B_j \in V\colon (B_i, B_j) \in E\}$ obviously constitutes a block partition as defined in the last section. This important observation allows us to see a block tree as a block partition enhanced by an additional structure of blocks that group together smaller blocks, in which we can express and infer connections between larger groups of blocks instead of only between individual blocks. Note that the values of the function $blocks(s_i) := \{b_j \mid C_i \in B_j\}$ are no longer singleton sets, but always contain the members of a chain of blocks $B_k \subseteq B_{k-1} \subseteq \cdots \subseteq B_0$ from the $B_k \in L(\mathcal{BT})$ that s_i is assigned to up to the top block B_0. This allows us to define a function $\boldsymbol{leafBlock(s_i)}$ that returns the associated leaf block B_k, and the function $\boldsymbol{maxBlock(s, D)} := \begin{cases} leafBlock(s) & if\ \neg\exists B \in V \mid B \subseteq D \\ \max\{B_i \in blocks(s) | B_i \subseteq D\} & otherwise \end{cases}$ for retrieving the maximal block containing s that is completely contained in D, defaulting to $leafBlock(s)$ if no such block exists.

Using $L(\mathcal{BT})$, the definition of n_{max}-representability for block partitions can be generalized to block trees by using $leafBlock(s_i)$ instead of $blocks(s_i)$ in the definition. With the new notational machinery, we can now formally define the compression of a meta instance G by a block tree $\mathcal{BT}$ in the following way:

Definition 5.3.1. *(**Meta instance compression by a block tree**)*
*Let $G := \{D_1, \dots, D_n\}$ be a meta instance for the CNF instance $F := \{C_1, \dots, C_m\}$, and let $\mathcal{BT} := \{\{B_0, B_1, \dots, B_k\}, E\}$ be a block tree over F in which G is representable. We define the **tree compression** G^T of G by $\mathcal{BT}$ as containing*
1) For each $B_i = \{C_{i1}, \dots, C_{im_i}\} \in L(\mathcal{BT})$, the clause $\{\neg b_i, s_{i1}, \dots, s_{im_i}\}$,
2) for each $B_j \in V \backslash L(\mathcal{BT})$, the clause $\{\neg b_j, b_{j1}, \dots, b_{jk_j}\}$, where the corresponding blocks $B_{j1}, \dots, B_{jk_j}$ are the children of B_j in the block tree, and
3) a clause $\bigcup_{j=1}^{n_i} maxBlockRep(s_{ij}, D_i)$ for each meta clause $D_i := \{s_{i1}, \dots, s_{in_i}\}$, where we define $\boldsymbol{maxBlockRep(s, D)} := \begin{cases} maxBlock(s, D) & if\ leafBlock(s) \subseteq D \\ \{s\} & else \end{cases}$.

The first type of clause, encoding an implication $b_i \rightarrow (s_{i1} \vee \dots \vee s_{im_i})$, is again called a **block definition clause**, whereas the second type of clause, encoding $b_j \rightarrow (b_{j1} \vee \dots \vee b_{jk_j})$, is called a **superblock declaration**. The third type of clause is called a **tree-compressed sat wedge**. Next, we show that the new compression scheme still has the desirable properties we proved in Section 5.2 for the block partition:

Theorem 5.3.2. *(**Equisatisfiability of tree compression**)*
For a CNF instance F and a block tree $\mathcal{BT}$ over F, an arbitrary meta instance G that is n_{max}-representable in $\mathcal{BT}$ is satisfiable iff the tree compression G^T of G by $\mathcal{BT}$ is satisfiable.

Proof. We take the compression G' of G by $L(\mathcal{BT})$. By Theorem 5.2.4, we know that G' is equisatisfiable to G. Therefore, we only need to show that G^T is satisfiable iff G' is.
$\Rightarrow$: Let ϑ^T be a model of G^T. We show that by restricting ϑ^T to selector variables and the block variables for $L(\mathcal{BT})$, we receive a model of G'. The block definition clauses we do not need to consider, since they are identical in G^T and G'. Therefore, it suffices to show that the compressed sat wedge $\bigcup_{j=1}^{n_i} blockRep(s_{ij}, D_i)$ is satisfied for each $D_i = \{s_{i1}, \dots, s_{in_i}\}$. Consider the corresponding tree-compressed sat wedge $\bigcup_{j=1}^{n_i} maxBlockRep(s_{ij}, D_i)$. We know that at least one $maxBlockRep(s_{ip}, D_i)$ is satisfied by ϑ^T. If $leafBlock(s_{ip}) \cup D_i \neq D_i$, then we must have $\vartheta'(s_{ip}) = \vartheta^T(s_{ip}) = 1$, so that $blockRep(s_{ip}, D_i)$ is satisfied, too. Otherwise, we only know that $\vartheta^T(maxBlock(s_{ip}, D_i)) = 1$, and must inductively traverse the block tree downward, starting with $b_q := maxBlock(s_{ip}, D_i)$. In each step, we consider the superblock declaration $\{\neg b_q, b_{q1}, \dots, b_{qk_q}\}$, which needs to be satisfied. Since $\vartheta^T(b_q) = 1$, one of the child block definition variables $\{b_{q1}, \dots, b_{qk_q}\}$ must be set to true, becoming the new b_q. This consideration is applied inductively until eventually, we have $b_q = leafBlock(s_{ip})$. The block definition clause $\{\neg b_q, s_{q1}, \dots, s_{qk_q}\}$ needs to be satisfied by ϑ^T, so for some s_{qj} we must have $\vartheta'(s_{qj}) = \vartheta^T(s_{qj}) = 1$. Because $B_j \subseteq D_i$, we have $s_{qj} \in blockRep(s_{ij}, D_i)$, so that the compressed sat wedge is satisfied.
$\Leftarrow$: Let ϑ' be a model of G'. We show that we can extend ϑ' to the non-leaf block variables to produce a model ϑ^T of G^T. For each D_i, we must show the satisfiability of $\bigcup_{j=1}^{n_i} maxBlockRep(s_{ij}, D_i)$. We know that $\bigcup_{j=1}^{n_i} blockRep(s_{ij}, D_i)$ is satisfiable, i.e. we have $\vartheta'(blockRep(s_{ip}, D_i)) = 1$ for some p. If $blockRep(s_{ip}, D_i) = s_{ip}$, then $leafBlock(s_{ip}) \cup D_i \neq D_i$, so that $maxBlockRep(s_{ip}, D_i) = s_{ip}$ as well, satisfying the tree-compressed wedge.
Otherwise, we set $\vartheta^T(b_q) := 1$ for $B_q := leafBlock(s_{ip})$. But this allows us to also set $\vartheta^T(b_r) := 1$ for the parent block B_r, because the superblock declaration for B_r is already satisfied by $\vartheta^T(b_q) := 1$. Inductively continuing with this, we arrive at $B_r = maxBlock(s_{ip}, D)$ with $\vartheta^T(b_r) = 1$, thereby satisfying $\bigcup_{j=1}^{n_i} maxBlockRep(s_{ij}, D_i)$ as desired. After processing all D_i in this manner, there might be some untouched blocks B_j just like in Theorem 5.2.4. For such a block, we can simply set $\vartheta^T(b_j) := 0$, satisfying the corresponding block definition or superblock declaration clause as well. After this, all clauses in G^T are satisfied by ϑ^T, so ϑ^T is a model of G^T. □

Theorem 5.3.3. *(**Unit propagation equivalence under tree compression**) For any consistent assumption set $A := \{l_1, \ldots, l_k\} \subseteq \bigcup_{i=1}^{m} \{s_i, \neg s_i\}$ and a tree compression G^T of a meta instance G for F, every selector variable unit which is derived during unit propagation on $G \cup \{l_1\} \cup \cdots \cup \{l_k\}$ is also derived during unit propagation on $G^T \cup \{l_1\} \cup \cdots \cup \{l_k\}$.*

Proof. We only need to extend the proof of Theorem 5.2.5 to account for the indirection introduced by superblock declarations. The base case for our induction over the propagation steps was already covered there. We have $l = s_{ip}$ for some $1 \leq p \leq n_i$, and the induction hypothesis says that the literals $\neg s_{i1}, \ldots, \neg s_{ip-1}, \neg s_{ip+1}, \ldots, \neg s_{in_i}$ have already been derived and propagated. Consider the tree-compressed sat wedge $\bigcup_{j=1}^{n_i} maxBlockRep(s_{ij}, D_i)$ for D_i. We first show that all literals in each $maxBlockRep(s_{ij}, D_i)$ except those of $maxBlockRep(s_{ip}, D_i)$ were cancelled out by propagating the units given by the induction hypothesis. Consider an arbitrary $j \neq p$. If $maxBlockRep(s_{ij}, D_i) = \{s_{ij}\}$, this single element has trivially been cancelled out while propagating $\neg s_{ij}$. If $maxBlockRep(s_{ij}, D_i) = \{b_q\} \neq maxBlockRep(s_{ip}, D_i)$ for some block definition variable b_q, we already considered the case that $B_q \in L(\mathcal{BT})$ in the proof of Theorem 5.2.5, where we saw that $\{\neg b_q\}$ is derived and propagated, cancelling out $blockRep(s_{ij}, D_i) = maxBlockRep(s_{ij}, D_i)$ in this case, too. Otherwise, we consider the superblock declaration $\{\neg b_q, b_{q1}, \ldots, b_{qk_q}\}$. To show that $\{\neg b_q\}$ is still propagated, we need to show that all the $\neg b_{qj}$ are propagated. This argument can inductively be applied until B_q is a leaf node, which is again covered by the proof of Theorem 5.2.5 because $leafBlock(s_i) \subseteq maxBlock(s_i, D_i)$ for every D_i.
Finally, we turn to the only remaining unit $maxBlockRep(s_{ip}, D_i)$ in the tree-compressed sat wedge. If $maxBlockRep(s_{ip}, D_i) = \{s_{ip}\}$, we have already derived the unit clause $\{s_{ip}\} = \{l\}$. If $maxBlockRep(s_{ip}, D_i) = \{b_q\}$ for some block definition variable b_q, $\{b_q\}$ is propagated. If B_q is a leaf, the last part of the proof of Theorem 5.2.5 shows that the unit $\{s_{ip}\} = \{l\}$ is still derived. Otherwise, we again need to traverse superblock declarations until we arrive at a leaf block. Again, we know that all the units in all leaves under D_q must have been cancelled out because $leafBlock(s_i) \subseteq maxBlock(s_i, D_i)$ for every D_i. □

In Figure 5.6, we revisit the compression example from Figure 5.1, this time displaying the original meta instance next to a near-optimal tree compression by a very small block tree. The block definition clauses and the superblock declaration clause are mentioned last, so that each tree-compressed sat wedge corresponds to the sat wedge on the same line.

5.3.1 Algorithm

As already in the case of block partition inference, we separate the algorithm for block tree construction into three parts. The first method is called *blockLeafSplit* (Algorithm 9), and fulfills the task *blockPartitionSplit* does for block partitions. Unlike *blockPartitionSplit*,

uncompressed meta instance G	tree compression G'
$\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$	$\{b_3, b_5, s_{13}\}$
$\{s_1, s_2, s_3, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{14}, s_{15}\}$	$\{b_5\}$
$\{s_1, s_2, s_3, s_4, s_{14}, s_{15}\}$	$\{b_1, s_4\}$
$\{s_1, s_2, s_3, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{12}, s_{13}, s_{14}, s_{15}\}$	$\{b_5, b_4\}$
$\{s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{12}\}$	$\{b_2, s_4, s_{12}\}$
	$\{\neg b_1, s_1, s_2, s_3, s_{14}, s_{15}\}$
	$\{\neg b_2, s_5, s_6, s_7, s_8, s_9, s_{10}\}$
	$\{\neg b_3, s_4, s_{11}\}$
	$\{\neg b_4, s_{10}, s_{13}\}$
	$\{\neg b_5, b_1, b_2\}$

Figure 5.6: Example of a meta instance, and its compression by a block tree.

however, given the necessary hash-based indexing structures, it can be executed in constant time, because we do not need to replace the block variable b by a new representation. The old block B will still be part of the block tree, even though it is not a leaf any longer. Note that the block trees we can construct in this way are rather restricted compared to the freedom that the definition would allow. In essence, we are confined to binary trees, and we cannot revert splitting decisions.

Algorithm 9 blockLeafSplit($\mathcal{BT}$,G^T,B,B')

Input: meta instance G^T compressed by block tree $\mathcal{BT} = (\mathcal{V}, \mathcal{E})$, leaf block $B \in V$, $B' \subset B$
Output: splits leaf block B into new blocks B' and $B \backslash B'$; adapts clauses in G^T so that is remains the tree compression of G under the changed block tree $\mathcal{BT}$

$B_1 := B'$; $B_2 := B \backslash B'$ ▷ new blocks, block IDs = 1,2 without loss of generality
$V := V \cup \{B_1, B_2\}$
$E := E \cup \{(B, B_1), (B, B_2)\}$
$G^T := G^T \cup \{\{\neg b_1\} \cup \{s_i \mid C_i \in B'\}\} \cup \{\{\neg b_2\} \cup \{s_i \mid C_i \in B \backslash B'\}\}$ ▷ block def clauses
$G^T := G^T \backslash \{\{\neg b\} \cup \{s_i \mid C_i \in B\}\}$ ▷ remove the old block definition clause for B
$G^T := G^T \cup \{\{\neg b, b_1, b_2\}\}$ ▷ add the superblock declaration for B

However, this restricted method for block tree inference has the advantage of making it straightforward to write variants of *ensureRepresentability* and *addSatWedge* for block trees. Because they now operate on a tree structure, both are much easier to define recursively. The *ensureTreeRepresentability* method we define as Algorithm 10 operates on a clause set D which is to be represented in the tree under a block B_i, splits up the task along the child block boundaries and recursively calls itself on the child blocks with their respective portions of D. If it reaches the base case of a leaf, *blockLeafSplit* is called if necessary. On the topmost level, *ensureTreeRepresentability* is of course always called with $B_i := B_0$, so that processing starts at the top node. The implementation of *addSatWedge* for block trees, as presented in Algorithm 11, follows the same basic layout, but it collects literals in a partial representation D^T of D which is handed on through the recursion, and finally added to G^T. With the following theorems, we again establish the correctness of both methods, and give rough estimates of their runtime complexity. This time, we will perform inductive proofs over sequences of block trees $\mathcal{BT}_i$ and tree compressions G_i^T, where G_0^T is the compression of G by the trivial block tree $\mathcal{BT}_0$. Because the trivial block tree only consists of a single leaf, we have $G_0^T = G'_0$.

Algorithm 10 ensureTreeRepresentability($\mathcal{BT} = (V, E)$,$B_i \in V$,D,n_{max})

Input: node B_i in block tree $\mathcal{BT}$, new meta clause $D = \{s_1, \ldots, s_n\}$, threshold n_{max}
Output: refines $\mathcal{BT}$ (and changes G^T accordingly) to ensure n_{max}-representability of D

if $\emptyset \neq B_i \neq D$ **then** ▷ D is not empty, and does not cover B_i entirely
  if $B_i \in L(\mathcal{BT})$ **then**
    if $|D| > n_{max}$ **then** ▷ overlap with leaf B_i too large
      $blockLeafSplit(\mathcal{BT}, G^T, B_j, D)$
    end if
  else
    for $(B_i, B_j) \in E$ **do** ▷ distribute the task to the child blocks
      $ensureTreeRepresentability(\mathcal{BT}, B_j, D \cap B_j, n_{max})$
    end for
  end if
end if
return

Algorithm 11 addSatWedge($\mathcal{BT}$,G^T,D)

Input: meta instance G^T tree-compressed by $\mathcal{B}$, n_{max}-representable $D = \{s_1, \ldots, s_n\}$
Output: changes G^T into the tree compression of $G \cup \{D\}$

```
1: D^T := buildRepresentation(BT, B_0, D, {})
2: G^T := G^T ∪ {D^T}
3: return
```

Procedure buildRepresentation($\mathcal{BT} = (V, E)$,$B_i \in V$,D,D^T)

```
1: if B_i = D then
2:     D^T := D^T ∪ {b_i}
3: else
4:     if B_i ∈ L(BT) then
5:         if |D| ≤ n_max then
6:             D^T := D^T ∪ D
7:         else
8:             print("ERROR: D not n_max-representable!")
9:             return
10:        end if
11:    else
12:        for (B_i, B_j) ∈ E do
13:            D^T := D^T ∪ buildRepresentation(BT, B_j, D ∩ B_j, D^T)
14:        end for
15:    end if
16: end if
17: return D^T
```

Theorem 5.3.4. ***(Correctness of ensureTreeRepresentability)***
Let $D_1, D_2, \ldots, D_k$ be a series of meta clauses. Starting with $\mathcal{BT}_0$ and G_0^T, we sequentially derive $\mathcal{BT}_i$ and G_i^T by calling ensureTreeRepresentability$(\mathcal{BT}_{i-1}, G_{i-1}^T, D_i, n_{max})$. Then, the meta problem $G_k := \{D_1, \ldots, D_k\}$ is n_{max}-representable in $\mathcal{BT}_k$.

Proof. Again we proceed by induction over i, the index of the D_i last added.
"$i = 0$": G_0 is obviously n_{max}-representable in $\mathcal{BT}_0$, since by definition we have
$\forall 1 \leq i \leq m : \ maxBlocks(s_i, \{s_1, \ldots, s_m\}) = \{F\} = G_0$.
"$i{-}1 \rightarrow i$": This corresponds to calling $ensureTreeRepresentability(\mathcal{BT}_{i-1}, G_{i-1}^T, D_i, n_{max})$ on the new meta clause $D_i := \{s_{i1}, \ldots, s_{in_i}\}$. We need to show that $G_{i-1} \cup \{D_i\}$ is n_{max}-representable in the changed block tree $\mathcal{BT}_i$ that results from executing this call.
By the induction hypothesis, every $D = \{s_1, \ldots, s_m\} \in G_{i-1}$ is n_{max}-representable in $\mathcal{BT}_{i-1}$. A call to $blockPartitionSplit(\mathcal{B}_{i-1}, G_{i-1}, B, D)$ will not change that, because only smaller blocks are added to the tree, and no block is removed, so that $maxBlock(s_j, D) \subseteq D$ still holds in $\mathcal{BT}_i$ for any j. To show that the n_{max}-representability holds in $\mathcal{BT}_i$ for the newly added clause D_i, we need to consider each $s_{ij} \in D_i$ in turn, showing that $maxBlock(s_{ij}, D_i) \subseteq D_i$ if $|maxBlock(s_{ij}, D_i) \cap D_i| > n_{max}$. The only case in which $maxBlock(s_{ij}, D_i) \subseteq D_i$ can be wrong at all is when $maxBlock(s_{ij}, D_i) = leafBlock(s_{ij}) \supset D_i$, i.e. if we have arrived at a leaf block that is a strict superset of D_i. The case $|leafBlock(s_{ij}) \cap D_i| \leq n_{max}$ we do not need to consider, because then the antecedent of the representability condition does not hold. Otherwise, the condition $|leafBlock(s_{ij}) \cap D_i| = |D_i| > n_{max}$ in line 3 holds, causing a call $blockLeafSplit(\mathcal{BT}, G^T, leafBlock(s_{ij}), D_i)$. This call splits $leafBlock(s_{ij})$ into D_i and $leafBlock(s_{ij}) \backslash D_i$. As a result, in $\mathcal{BT}_i$ we have $maxBlock(s_{ij}, D_i) = D_i$, fulfilling the representability condition for j. □

Theorem 5.3.5. *(**Correctness of addSatWedge**)*
Let G^T be the compression of a meta instance G under a block tree $\mathcal{BT}$. For a new meta clause $D := \{s_1, \ldots, s_n\}$, let $G \cup \{D\}$ be n_{max}-representable in $\mathcal{BT}$. After the function call $addSatWedge(\mathcal{BT}, G^T, D)$, G^T is the compression of $G \cup \{D\}$ under $\mathcal{BT}$.

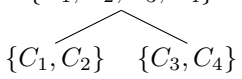
Proof. The proof for Theorem 5.2.8, i.e. the corresponding result for block partition inference, can be adapted with some minor changes. Again, we only need to show that the D^T generated and added to G^T by the call to $addSatWedge(\mathcal{BT}, G^T, D)$ is equal to $\bigcup_{j=1}^{n} maxBlockRep(s_j, D)$ for $D = \{s_1, \ldots, s_n\}$, i.e. that for each $1 \leq j \leq n$, D^T exclusively contains b_j for $B_j := maxBlock(s_j, D)$ if $leafBlock(s_j) \subseteq D$, and s_j otherwise.
Consider the case $leafBlock(s_j) \subseteq D$. By definition, $leafBlock(s_j) \subseteq maxBlock(s_j, D) \subseteq D$. We show that during our recursive traversal of the block tree, for some recursive call to *buildRepresentation* we will have $B_i = maxBlock(s_j, D)$. The only case where block traversal is stopped before reaching a leaf block is if the condition $B_i = D$ in line 1 of *buildRepresentation* holds. Given the way *buildRepresentation* was recursively called, this would imply $D \cap B_i = B_i$, i.e. $B_i \subseteq D$. The recursion could thus be stopped before $maxBlock(s_j, D)$ if we had $maxBlock(s_j, D) \subset B_i$. But we know that $s_j \in maxBlock(s_j, D)$ and thereby $B_i \in blocks(s_j)$, contradicting with $B_i \subseteq D$ the definition of $maxBlock(s_j, D)$.
Consider this call to *buildRepresentation* with $B_i = maxBlock(s_j, D)$. Given the arguments that *buildRepresentation* is recursively called with, we must then have $maxBlock(s_j, D) \cap D = D$, which together with $maxBlock(s_j, D) \subseteq D$ implies $B_i = maxBlock(s_j, D) = D$, causing the condition in line 1 to hold. This means that $\{b_j\}$ fo $B_j = maxBlock(s_j, D)$ is added, proving the theorem in this case.
If $leafBlock(s_j) \supset D$, the n_{max}-representability of D gives us $|leafBlock(s_j) \cap D| = |D| \leq n_{max}$. For this case, we show that for some recursive call to *buildRepresentation* we will have $B_i = leafBlock(s_j)$. The only case where block traversal is stopped before reaching a leaf block is if the condition $B_i = D$ in line 1 of *buildRepresentation* holds. Given the way *buildRepresentation* was recursively called, this would imply $D \cap B_i = B_i$, i.e. $B_i \subseteq D$. The recursion could thus be stopped before $leafBlock(s_j)$ if we had $leafBlock(s_j, D) \subset B_i$. But in this case, we would have an edge $(B_i, leafBlock(s_j, D)) \in E$ in the block tree, so that the recursive call in line 13 would occur, contradicting our assumption that the recursion stopped. Consider now this call to *buildRepresentation* with $B_i = leafBlock(s_j)$. Our case $|B_i \cap D| = |D| \leq n_{max}$ is covered by the condition in line 5. We therefore get $D^T := D^T \cup D$, which in our case means $s_j \in D^T$ because of $s_j \in leafBlock(s_j) \cap D$.
Obviously, in the recursive case of *buildRepresentation* nothing is added, and no further recursion happens if anything was added. Since we determined where the recursion stopped in all cases, nothing else than the variables covered is added to D^T. □

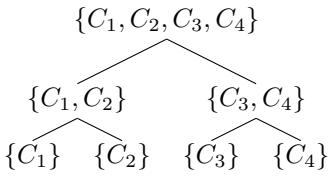
As we will see now, the worst-case runtime for both *ensureTreeRepresentability* and *addSatWedge* is surprisingly long, due to a notorious worst case. Runtimes in practice are much better than one would expect from this (almost linear in m), but this cannot be shown without extensive formal analysis of the distributional properties of learnt sat wedges. Suffice it to say that the number of blocks is typically much lower than the number of clauses in the instance, and that while the representation for a block is generated, we virtually never need to traverse the entire block tree in practice.

Theorem 5.3.6. *(**Complexity of ensureTreeRepresentability and addSatWedge**)*
The worst-case runtime complexity of both $ensureTreeRepresentability(\mathcal{BT}, G', D, n_{max})$ and $addSatWedge(\mathcal{BT}, G^T, D)$ is in $O(m \log m)$, where $m := |F|$.

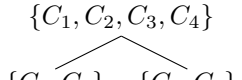
Proof. In the worst case, the block tree distributes the m selector variables over m leaves, and we only have binary branches. If we now add the sat wedge $D = \{s_1, \ldots, s_m\}$, the computation of $D \cap B_j$ for each recursive call in line 8 (or 13, respectively) takes $O(|B_j|)$. We therefore have $\sum_{B \in \mathcal{B}} |B| \leq m$ once on each of the $\log m$ layers of the tree, leading to runtime $O(m \log m)$ for the intersections alone. □

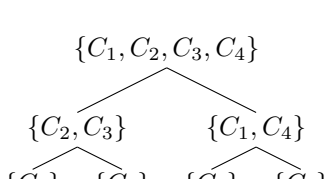
The methods we just discussed are combined into the inference algorithm in exactly the same way as it was described for block partitions in Section 5.2. An interesting observation that is specific to block trees should nevertheless be mentioned, namely that the block tree structure inferred by the algorithm largely depends on the order in which the sat wedges were added. To see this, consider the minimal example of $F = \{C_1, C_2, C_3, C_4\}$ with $\mathcal{BT}_0 = \{(\{C_1, C_2, C_3, C_4\}), \{\}\}$, in which we want the sat wedges $D_1 = \{s_1, s_2\}$ and $D_2 = \{s_2, s_3\}$ to be n_{max}-representable for $n_{max} := 0$. If we add D_1 before D_2, we get

$\mathcal{BT}_1 =$ 

$\mathcal{BT}_2 =$ 

On the other hand, adding D_2 before D_1 leads to a different block structure:

$\mathcal{BT}_1 =$ 

$\mathcal{BT}_2 =$ 

While the block partition $L(\mathcal{BT})$ is identical in both variants, the leaf blocks have been grouped together differently into larger blocks. This observation essentially remains true for instances of any size. To better understand the constraints on the producable block structures over any given unsatisfiable instance, it might be more fruitful to ask which binary block trees over $L(\mathcal{BT})$ can not be produced by any sequence of reduction operations.

5.3.2 Interactive Visualization

Just like in the case of a block partition, exposing the inferred block tree to the user opens up a variety of new information retrieval and control possibilities. The implementation was therefore extended by a **block tree display** component based on Kahina's tree visualization classes. Each node in the tree represents a block. By default, the node caption consists of the ID of the respective block followed by the block size in brackets. The default colouring scheme for the leaf nodes is the same as the one defined for the block partition, but the colour of a non-leaf block is determined in an intuitive way by the colours of its sub-blocks. For example, if all of the child blocks are coloured in dark red to mark their criticality, their parent block will also receive that colour. If only one child block is known to be critical, and all other child blocks are of unknown state, the parent block is coloured in light red.

In Figure 5.7, we return to the block partition example from Figure 5.5 to showcase the block tree display. Note that the leaves of the tree correspond to the block partition we saw there. The order in which the interactive deduction steps were performed has caused the block tree visualized in the view to be inferred. As can be seen in the reduction graph, our currently selected US is minimal. The current US view also shows all clauses as being critical. In the block tree view, we see that all the leaf nodes are coloured either in dark red or in grey. All the leaves coloured in red represent the blocks the MUS is composed of, whereas the grey blocks are those which fell away during the reduction process. Another MUS will exhibit a different pattern of red and grey leaves, making it very easy to visually assess the overlap and the differences between different MUSes.

As we saw in the previous section, the block tree grows whenever a new sat wedge cannot be represented in the old tree. The growth of the block tree by leaf splitting mirrors a

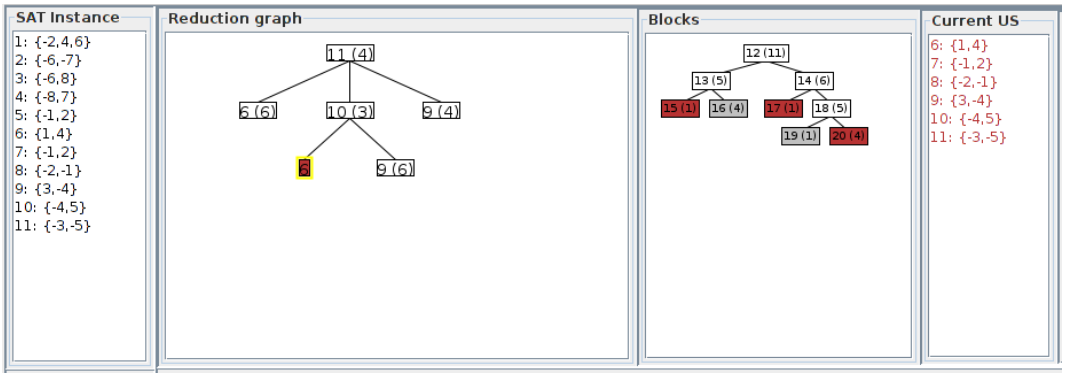


Figure 5.7: An example of the block tree display during a reduction process.

continuous refinement of the block partition represented by the leaf nodes. It is attractive to assume that the block tree reveals some hidden knowledge about the inner structure of the unsatisfiable SAT instance. The problem is that, as already demonstrated, the block trees can differ a lot depending on the order in which the sat wedges were added. While the partition defined by the literals will come out more or less identically (with the n_{max} threshold adding some variety), the inferred tree structure over this partition can be vastly different. If the block tree reveals hidden structure beyond what can be concluded from the underlying partition, this information would just as well depend on the operations executed to derive it. This subject is certainly worthy of closer investigation in the future. In Chapter 6, we will take a first stab at exploring this issue when we assess the usefulness of interactive MUS extraction as a paradigm.

Just like the block partition display, the block tree display is valuable as an additional layer for defining useful clause selections. Moreover, it is again an attractive option to allow US reduction steps to be executed on any block in the tree. Just like for the block display, the default reaction to a double-click on one of the blocks in the block tree is to attempt a simultaneous reduction of all the clauses in the block, followed by the fallback procedure of reducing clauses individually up to the first success if simultaneous reduction is unsuccessful.

In Figure 5.8, we see the effect of a user-defined block reduction operation on a different block tree. The simultaneous reduction of all 144 clauses in block 6082 fails, causing the fallback option to be executed. Already the first reduction in the row is successful, leading to a sat wedge that is not representable in the current tree. On the right side, we see the consequences of the call to *ensureTreeRepresentability*. Block 6082 is not a leaf block any longer, but it was split into a new block of 113 clauses which fell away during clause set refinement, and a core of 31 clauses of unknown criticality. The mechanism demonstrated here makes it possible to find MUSes by means of interaction with the block tree alone, splitting leaf nodes until all leaves are coloured either in dark red or in grey, just as in our first block tree example in Figure 5.7.

5.3.3 Application to Non-CNF Instances

Just as in the case of block partitions, there is a natural application of block trees to a more general form of minimal core extraction. This becomes apparent when we look at the Tseitin encoding of a non-CNF formula in negation normal form (see Definition 2.1.6).

For this application, we need to slightly generalize our notion of a block tree by dropping the condition $\bigcup_{(B_i,B_j)\in E} B_j = B_i$, so that not all the selector variables in a block need to

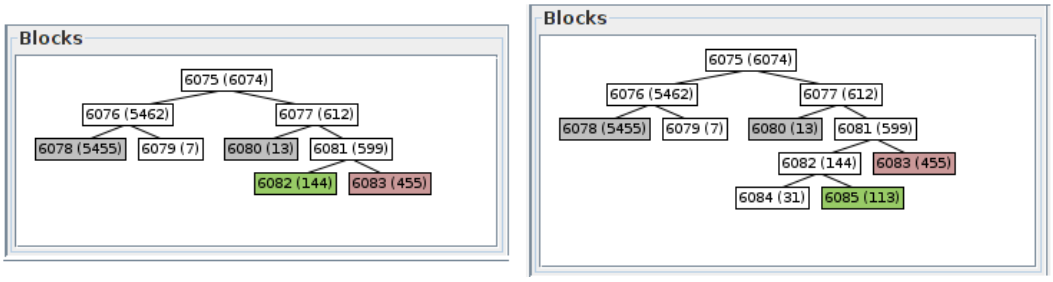


Figure 5.8: An example of interactive block tree node expansion.

be covered by its children. We did not consider this definition of block trees before because it would have made the block inference algorithm much more complicated. In this section, we will again work with an a priori block structure, so that an inference algorithm is not needed. The necessary proofs of equi-satisfiability and propagation completeness can easily be adapted to this more general notion of block tree.

We can now map the formula structure of an NNF formula to a block tree by grouping together the clauses from the Tseitin encoding that together represent a formula subtree. In Figure 5.9, we see an example of an NNF formula tree, the corresponding Tseitin encoding extended by selector variables, and the block tree over these selector variables that models the formula structure.

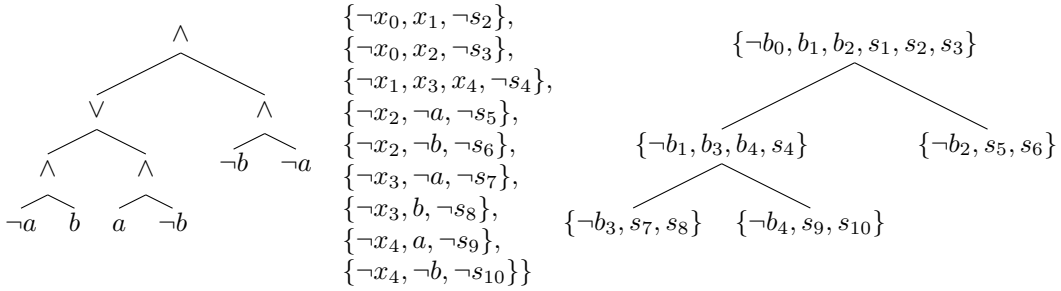


Figure 5.9: Example of NNF formula, its Tseitin encoding, and corresponding block tree

The crucial observation now is that by removing blocks of clauses from the Tseitin encoding, we arrive at Tseitin encodings of subformulae. Consider the case where we want to remove the entire subtree $(\neg a \wedge b) \vee (a \wedge \neg b)$ from the formula. During interactive reduction, a click on the corresponding block B_1 would cause the selector variables $s_4, s_7, s_8, s_9, s_{10}$ to be set to false. F would be reduced to $\{\{x_0\}, \{\neg x_0, x_1\}, \{\neg x_0, x_2\}, \{\neg x_2, \neg a\}, \{\neg x_2, \neg b\}\}$, where the clause $\{\neg x_0, x_1\}$ has become unnecessary, but does not destroy equi-satisfiability because we can now simply set x_1 to true.

The naturalness of this approach to minimal unsatisfiable subformula extraction lies in the fact that block variables in the meta instance now stand for the removal of subtrees, and are used to express dependencies between removals of different subtrees, just like we used them for clause groups in the block partition case. For our example reduction, the resulting F is satisfiable, so we would learn the sat wedge $\{s_4, s_7, s_8, s_9, s_{10}\}$, which would be compressed to $\{b_1\}$, stating in a nicely concise way that the subformula $(\neg a \wedge b) \vee (a \wedge \neg b)$ is critical.

When defining minimal unsatisfiable subformulae in Section 2.2.3, we saw that we only get a reasonable definition if we only allow or-subtrees to be removed. For interactive reduction, this means that we have to differentiate between blocks which correspond to or-subtrees and may therefore be reduced, and other blocks which are not reducible. As in the case of GMUS extraction, we also have to deactivate the possibility of removing individual clauses, and the fallback option of sequential reduction when a block reduction failed.

5.4 Conclusion

In this chapter, we have extended the meta learning approach by blocks of selector variables, and seen how these blocks can be inferred automatically both by refining a flat partition and by building up and maintaining a recursive hierarchy of blocks. Algorithms for doing this were described and to some degree analysed, and implementations of both block schemes were added to the prototype system developed in previous chapters.

We have seen how both block structures can be visualized to expose additional information about the internal structure of unsatisfiable SAT instances. The visualizations solved the problem of making the overlaps between different MUSes transparent, and provide an interface for giving the user visual hints which clauses or clause groups might be worthwhile to try to remove from the current US.

To demonstrate the relevance of the concept of block-based meta instances, we have shown how both GMUS extraction and the extraction of minimal unsatisfiable subformulae from non-CNF instances can be described and emulated very naturally in terms of blocks. Using these techniques, the prototype can easily be turned into a system for interactive extraction of unsatisfiable cores not only of the usual CNF instances, but also of GMUS instances and non-CNF formulae.

Taken together, these innovations complete our exploration of interactive MUS extraction and the features of its current prototype implementation. In the following and final chapter of this thesis, some of the new techniques will be evaluated by using the prototype on a set of instances intended to approximate the real-world instances that users will want to use interactive MUS extraction tools on.

Chapter 6

Evaluating Interactive MUS Extraction

Having built up an architecture and a prototype of a system for interactive MUS extraction, we can now turn to the issue of evaluating the usefulness of the paradigm by testing it in an application context. This endeavour was hampered by the lack of available test data, a problem which is described in Section 6.1. To compensate for this lack, Section 6.2 then develops a novel approach to SAT-based parsing of context-free grammars for natural language processing. A test set of 120 interesting instances is generated, whose properties with respect to minimal unsatisfiable subsets we analyse in Section 6.3. We then turn to the question how unsatisfiable subsets in these instances can be interpreted with respect to grammar debugging tasks. In preparation, we define an application-specific clause relevance filter in Section 6.4. In Section 6.5, we then implement further assistance functionality for displaying this symbolic information, observe the structure of typical conflict sets, and analyse what they can tell us with respect to the grammar. A concluding section wraps up the results of the experiment, and draws a few conclusions concerning the applicability of interactive MUS extraction in general.

6.1 General Issues of Evaluation

While it is possible to use interactive MUS extraction as a tool for exploring the search space for any SAT instance, it can only develop its full potential if the user knows something about the semantics of variables, clauses, and clause blocks in the context of the application in question. In most cases, this presupposes some understanding of the connection between the input SAT instance and the problem it encodes.

From the user's part, this requires domain knowledge in the respective field. For the SAT encoding, it means that at least some variables need to represent problem information in a fashion that is transparent to or at least interpretable by the user. Normally, this would be done by encoding the semantic contribution of variables in their symbols. Here we face the problem that the publicly available test sets for MUS extraction are obfuscated in the sense that they do not contain any symbolic information of this kind.

For comparing the performance of general-purpose SAT solvers, the DIMACS format is the standard exchange format. By default, this format only stores numerical IDs to represent variable names, and no symbolic information is contained in the various test instances which are publicly available. SAT solving technology is thus being evaluated on pure structural information where the symbols have been stripped of all their semantic content. Some general information about the respective application is often given in accompanying papers, but never enough to restore any variable symbols.

The main reason for this unfortunate situation is that the test sets are usually compiled by researchers who are working on an industrial application. If these researchers are not themselves employees bound by non-disclosure agreements, they will have contracts with industry partners which forbid the proliferation of sensible data that could give away trade secrets. Quite understandably, such researchers therefore tend to be very careful in giving away data which could yield any information about application scenarios and the problem encodings. As a tendency, this also extends to academic researchers without any direct industry affiliation, since they have generally invested many work hours into developing efficient encodings for their respective applications, and only very few people are ready to leave the fruits of this work to their competitors in such an active and thriving field.

But even if it were possible to procure a test set with associated symbol tables for the variables, his lack of the relevant domain knowledge (e.g. in hardware design) would have made it very hard for the author to assess what a specialist in the field could gain from an interactive MUS extraction tool. For these reasons, it became necessary to find or develop a SAT encoding for a problem from a domain the author is sufficiently familiar with.

6.2 The Test Case: CFG Parsing for NLP

The mentioned problems in acquiring meaningful instances from industrial applications led the author to resort to the domain of **natural language processing (NLP)** as the one he is most familiar with, and to look for interesting SAT encodings of problems in this field. Surprisingly, an extensive search of the literature revealed that only very few applications of SAT solving in NLP exist, and no literature at all was found for one of the core areas of NLP, the syntactic analysis of natural language. In the author's opinion, the lack of work on SAT-based methods in NLP is due to the reason that during the past two decades, work in mainstream NLP has largely shifted to purely statistical methods, whereas during the heyday of symbolic NLP and artifical intelligence in the 1980s, the field of SAT solving had not yet developed the performant and mature tools available today, causing most of this older work to take place in the framework of logic programming instead.

The only work which was found of potential relevance is the planning-based approach based on answer set programming (ASP) presented by Yulia Lierler and Peter Schüller [38], which we already encountered in Section 3.3.2. The example of a reduction graph we saw there was based on a SAT instance created from an instance of this ASP encoding by means of the ASP grounder Gringo [39]. Originally, the author intended to use existing tools to reduce the entire encoding to SAT, and to evaluate interactive MUS extraction on the resulting unsatisfiable SAT instances. However, it turned out that this is made impossible by the many reduction techniques which Gringo and other ASP grounders perform in order to keep answer set programming feasible. These techniques guarantee that no answer sets (the ASP equivalent of models) are lost, but in the unsatisfiable case, the inconsistencies often get reduced to an empty clause already at this stage, so that most SAT instances produced by grounding and post-processing are already trivially unsatisfiable. This mirrors the situation in logic programming, where a result set is given in the satisfiable case, but for efficiency reasons no explanation is generated in case a predicate call fails. The logic programming and ASP paradigms are not designed to provide explanations for failure, so we will not be able to extract useful test instances from any approach to syntactic parsing which takes place in these paradigms.

The lack of SAT-based previous work leaves a computational linguist with many opportunities for developing novel SAT-based approaches to classical problems in NLP. As a result, the test case for SAT solving that will be developed in this chapter seems to be completely

new. While it can only be considered a first exploratory step towards MUS-based debugging of formal grammars in symbolic NLP, the general ideas we will develop apply just as well to state-of-the-art systems which use much more complex formalisms. But to understand the application and its encoding in SAT, we first have to cover some basic background knowledge on linguistics and NLP.

6.2.1 Parsing Natural Language

The syntactic analysis of natural language builds on the theories and notions of **syntax** as a branch of linguistics, a field with a long tradition of competing paradigms. For our exposition here, we stay in the mainstream tradition of the English-speaking world by working with a generic variant of **phrase structure grammar**. Unlike in other paradigms of syntax which focus on functional connections between words, the phrase structure grammar approach to syntax is primarily concerned with describing what is called the **constituent structure** of natural language sentences. Constituents are words or groups of words which function as units within a hierarchical structure assigned to a sentence, which is usually written in the form of a **syntactic tree**. Most linguists agree on a set of constituency tests which can decide for most sequences of words whether they should be considered constituents.

Vast differences between different linguistic theories concern the question how constituents are labeled, and how their recombination possibilities with other constituents to form larger structures should be modeled. A useful degree of agreement about the labels can be reached for a core set of syntactic **categories**, which divide into word classes and phrase types. Individual words are labeled by **word classes**, examples being traditional notions like that of a noun, a verb, or an adjective. The notion of a **phrase** is again theory-dependent, but the notion may informally be stated as a constituent around one designated head word, where the head word determines the outward behaviour of the constituent. The type of the phrase is usually in some way derived from the class of the head word. To make all these notions more concrete, Figure 6.1 gives a table of all the syntactic categories that are used in the example grammar of this chapter, along with their shorthands, and examples giving a rough impression of what type of constituents these labels apply to.

A	adjective	red, green, dark, intelligent
ADV	adverb	very, well, badly, tomorrow, again
AP	adjective phrase	very old, blue or green, slightly complicated
C	complementizer	that, whether
CP	complementizer phrase	that we have arrived, whether he comes
D	determiner	the, every, some, each, his, that
DP	determiner phrase	the man, every black car, his work
N	noun	work, book, computer, algorithm
NP	noun phrase	short chapter, very dense content
P	preposition	in, on, at, with
PN	proper noun	John, Mary, London, Tübingen
PP	preposition phrase	in London, on the slow bus, at the seaside
PRN	pronoun	he, she, we, that
S	sentence	we know it, this is interesting
VI	intransitive verb	sleeps, walks, snore
VP	verb phrase	sees him again, gives his wife the keys
VT	transitive verb	likes, sees, hear

Figure 6.1: Table of syntactic categories used in the example grammar.

The exact role of these categories in linguistic theory is of no concern to us here, but the list should contain enough information for the non-linguist reader to comprehend the examples in this chapter. For a more complete understanding, it is recommended to consult the first chapters of any standard syntax textbook like the one by Andrew Carnie [47]. In order to put some flesh on the notion of a syntactic tree, a first example of such a structure for the sentence "John saw a small cat on the tree" is visualized in Figure 6.2. Note that the leaves are labelled with word classes, and the non-leaves with phrase types.

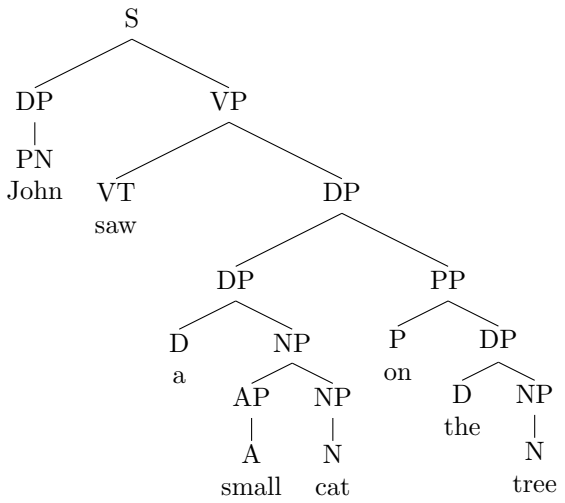


Figure 6.2: Example of a syntactic tree over the example grammar.

One of the central challenges of natural language processing is to automatically determine the syntactic trees assigned to a sentence according to some formal specification of a linguistic theory, a **grammar** in the more narrow sense we will use here. The entire process is called **parsing**, and there are obvious parallels to the use of the word in computer science, when an expression of a programming language is parsed into a syntax tree in the first stage of a compilation process. Parsing for programming languages is (or should be) deterministic, assigning only one structure to each symbol sequence that is a valid expression. In contrast, natural language utterances are very often **ambiguous**, meaning that we can most often assign more than one structure to a sentence, which often reflect different **readings**, i.e. different semantic interpretations. Consider a sentence like "she thinks about the shadows at night", where the question is the position of the phrase "at night" in the syntax tree. If we attach it to the noun phrase "shadows", we get a reading where the thinking might occur by day, and it is only about the shadows appearing at night. On the other hand, if we attach "at night" to the entire verb phrase "thinks about the shadows", then the thinking also occurs at night. In most cases, a human listener will be able to resolve the ambiguity given the context of an utterance, but a formal system which does not have any of this contextual knowledge needs to take the possibility of multiple solutions into account.

For a parser implementation, we need a formal description of the language fragment to be covered. This specification needs to be expressed in some **grammar formalism** that the parser understands. In most general terms, grammar formalisms are systems for defining formal languages, but there is not much more that all formalisms used in NLP have in common. Each grammar formalism is necessarily a compromise between expressivity, tractability, and, perhaps most importantly, the elegance with which certain linguistic phenomena (often specific to one language or a group of languages) can be described.

6.2.2 Context-Free Grammars as a Grammar Formalism

A particularly simple approach to modelling a phrase structure grammar mathematically is to use **context-free grammars (CFGs)** as known from formal language theory as a simple grammar formalism. While context-free grammars are demonstrably not powerful enough to describe all syntactic phenomena which occur in natural languages [48], and although their practical usefulness is limited if they are not enhanced by a unification-based feature logic [49] or by a probabilistic model [50], CFGs have for a long time been the de-facto standard for theory-neutral introductory textbook examples in the field of computational linguistics, and for teaching the basics of syntactic parsing. CFG as a grammar formalism is also a good test case because compared to the formalisms used in practice, they are much more accessible to the non-specialist, while still having enough expressive power for defining models of non-trivial natural language fragments. Another advantage is that most more complex grammar formalisms can be seen as extensions of context-free grammars, so that many of the techniques developed and presented here will also be applicable to these formalisms.

Concerning the basics of context-free grammars and related concepts of formal language theory, any introductory textbook of theoretical computer science should contain everything needed to understand the exposition here. The notation adopted in this chapter is taken from the very popular undergraduate textbook by Michael Sipser [51]. Consequently, we will write a grammar in the CFG formalism as a quadruple $G = (V, \Sigma, R, S)$, where V contains the phrase labels or **non-terminals**, and Σ the words or **terminals**. R is the set of **grammar rules**, and the start variable $S \in V$ aptly coincides with the sentence label. A grammar can be specified completely by only enumerating the elements of R, since V and Σ are implicitly defined by the symbols used in the rules. Using the usual yield and derivation relations between strings over $V \cup \Sigma$, the language accepted by G can be written as $L(G) := \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. A sentence $w \in L(G)$ is called **licensed** or **generated** by G in linguistic terminology. We will use $T(w, G)$ to denote the set of **parse trees** for the sentence w given the grammar G, which is empty iff w is not licensed by G.

The development of symbolic grammars for (fragments of) natural languages is commonly called **grammar engineering**. Like that of a programmer, the workflow of a grammar engineer consists of grammar writing and **grammar debugging**. Three types of problems occur especially often during the process of grammar debugging. The first is the **no-parse** problem, which occurs when for some sentence $w \in \Sigma^*$ which the grammar engineer would want G to license, he gets $w \notin L(G)$. In a programming language, this is equivalent to a syntax error. However, the non-determinism of CFGs for natural languages makes it much harder than for programming languages to provide useful error feedback which answers the question why the sentence was not licensed. The **missing-parse** problem is the situation where we have $T \notin T(w, G)$ for some tree T that we would like to see as a parse tree. Note that in the non-deterministic case, this does not imply wanting that $T(w, G) = \{T\}$. It is well possible that none of the trees in $T(w, G)$ is wrong, but that we are missing some reading. The **spurious-parse** problem is dual to the missing-parse problem in that we have one parse too much, i.e. there is a parse tree $T \in T(w, G)$ which is undesired. Note that this also covers the case where a sentence is licensed by the grammar although it shouldn't.

The non-determinism involved in production steps makes all of these problems challenging to track down and resolve. Often, a good intuitive understanding of possible sources of error is needed to understand where the cause of a problem may be. This makes it difficult for multiple grammar engineers to collaboratively work on a common grammar, or to extend grammars which have not been under active development. In the next section, we will develop a SAT encoding of CFG parsing which allows both the no-parse problem and the missing-parse problem to be encoded in unsatisfiable SAT instances, making them accessible to analysis attempts on the basis of minimal unsatisfiable subsets.

Figure 6.3 shows the small example grammar we will be using throughout this chapter. It forms the basis for the MUS instances we will generate, and we will illustrate the practical solution of the no-parse and missing-parse problems by means of interactive MUS extraction via examples that build on this grammar. Rules are notated compactly in the form A → B, where | is used as a shorthand for disjunctive rules.

S → DP VP	A → old | young | small | tall	ADV → very
DP → D NP | PN | PRN	PRN → everybody | he | she	C → that
VP → VI | VB CP | VP PP | VT DP	P → on | in | under	
NP → NP PP | AP NP | N	PN → john | mary	
AP → A | ADV A	VB → believes | thinks	
PP → P | DP	VI → sleeps | walks	
CP → C | S	VT → sees | likes	

N → man | woman | dog | cat | street | house | tree

Figure 6.3: The example context-free grammar used to generate the test instances.

6.2.3 Encoding CFG Parsing in SAT for Grammar Debugging

In this section, a new encoding of CFG parsing as a SAT problem will be developed. There are of course much more efficient parsing algorithms for CFG, but these approaches do not provide usable conflict information for grammar debugging. The two principal ideas of the encoding scheme are that models correspond to parses, and that variables talk about the assignments of symbols and application of rules on ranges in the input string. Since there are more possible ranges over longer inputs, the size of the encoded grammar grows with the maximum input length. The encoding is best read in a top-down fashion, i.e. by viewing the clauses as implications where the antecedents enforce the consequents.

We will now define the SAT encoding of a grammar $G = (V, \Sigma, R, S)$ for parsing strings of length n. As a first step, we introduce variables $a_{[i,j]}$ (written `a[i,j]` in plain text, e.g. in user interfaces) for encoding statements about the assignment of symbols A to input ranges $[i, j]$ with $0 \leq i < j \leq n$. Symbols which are not connected by a unary rule exclude each other, which we enforce by generating **exclusivity constraints**

$$\{\neg a_{[i,j]}, \neg b_{[i,j]}\} \; \forall \; 0 \leq i < j \leq n, \; \forall A, B \in V : A \overset{*}{\not\Rightarrow} B$$

For reasons of efficiency, we are not introducing variables for the terminals Σ, but only for the word class labels immediately above the words. We are thus parsing sequences of word classes or **pre-terminals** instead of terminals. During parsing, the conversion from words to word classes is then performed in a dictionary-based preprocessing step.

To encode the application of a rule to some range, for each rule $A \to B_1, \ldots, B_m$ and each range $[i, j]$ with $0 \leq i < j \leq n$, we define the variable $a_{b_1,\ldots,b_m[i,j]}$ (`a->b1,...,bm[i,j]`) to encode that the rule $A \to B_1, \ldots, B_m$ was applied to the range $[i, j]$. The presence of a constituent $a_{[i,j]}$ where A is on the left-hand side of some rule requires the application of one of these rules, but only if the rule has as most as many symbols on the right hand side as the range $[i, j]$ is wide:

$$\forall \; 0 \leq i < j \leq n, \; \forall A \in V : \; \{\neg a_{[i,j]}\} \cup \bigcup_{\substack{A \to B_1,\ldots,B_m \in R \\ m \leq j-i}} \{a_{b_1,\ldots,b_m[i,j]}\}$$

Note that if all the right-hand sides for A have too many symbols, we receive a unit clause $\{\neg a_{[i,j]}\}$, expressing the fact that no constituent of type A can then range over $[i, j]$.

Next, we model the connection between rule applications and the child constituents they create. A rule execution $a_{b_1,\ldots,b_m[i,j]}$ can be seen as distributing the range $[i, j]$ over m child constituents. For $m = 1$ (a **unary rule**), the only child must span the entire range:

$$\forall\ 0 \leq i < j \leq n,\ \forall A \to B \in R:\ \{\neg a_{b[i,j]}, b_{[i,j]}\}$$

For each rule $A \to B_1, \ldots, B_m$ with $m > 1$, we introduce a new variable $a_{b_1,\ldots,b_m[i,j]:i_1:\cdots:i_{m-1}}$ (`a->b1,...,bm[i,j]:i1:...:im-1`) for each possible splitting $i < i_1 < \cdots < i_{m-1} < j$. Since rules with $m > 2$ are almost never necessary, the number of splittings does not become untractably large for typical $n < 20$ (sentences which are not composed of smaller sentences only very rarely have 20 or more words). Each application of a rule $A \to B_1, \ldots, B_m$ implies that it is applied using one of the possible splittings:

$$\forall\ 0 \leq i < j \leq n,\ \forall A \to B_1, \ldots, B_m \in R,\ m > 1:$$
$$\{\neg a_{b_1,\ldots,b_m[i,j]}\} \cup \bigcup_{i<i_1<\cdots<i_{m-1}<j} \{a_{b_1,\ldots,b_m[i,j]:i_1:\cdots:i_{m-1}}\}$$

Next, we express that each splitting variant enforces the presence of all the corresponding child constituents at the ranges defined by the splitting:

$$\forall\ 0 \leq i < j \leq n \forall A \to B_1, \ldots, B_m \in R,\ m > 1,\ \forall i = i_0 < i_1 < \cdots < i_{m-1} < i_m = j:$$
$$\bigwedge_{k=1}^{m} \{\neg a_{b_1,\ldots,b_m[i,j]:i_1:\cdots:i_{m-1}}, b_{k[i_{k-1},i_k]}\}$$

Finally, we need to forbid pre-terminals from spanning more than one input position, because for these categories there are no rule encodings that would enforce the presence of any child constituents, which would make it possible for a single pre-terminal to span arbitrary ranges without any check against the input string if we do not explicitly forbid that:

$$\forall A \in V\ where\ \exists w \in \Sigma : \{A \to w\} \in R,\ \forall\ 0 \leq i, j \leq 1\ where\ j - i > 1 : \{\neg a_{[i,j]}\}$$

The encoding of a CFG grammar that we have just introduced can be used to analyse sentences up to length n by solving it with only a few additional assumptions representing the input. However, the encoding of the grammar alone is trivially satisfiable just by assuming that all the antecedents in the implication encodings are false. Only by adding unit assumptions $\{a_{[i,j]}\}$ do we get non-trivial constraints into the problem.

We shall see that a variety of problems can be expressed by simply adding different sets of unit assumptions to the grammar encoding. The simplest form allows us to use a SAT solver as a parser. This works by postulating $\{s_{[0,n]}\}$ along with unit clauses that specify which categories can be found at which positions in the input. Let us illustrate the procedure on a simple example sentence which is licensed by the grammar. We parse the sentence "john sleeps" by generating the encoding of our example grammar for inputs of size 2, and adding the unit clauses $\{s_{[0,2]}\}, \{pn_{[0,1]}\}, \{vi_{[1,2]}\}$ in a preprocessing step. Instead of formally proving the correctness of our SAT encoding, we will go through a solver run for this example step by step, which should contribute to a complete understanding of the way in which the encoding works. We write down the relevant steps of the solver run:

1. $\{s_{[0,2]}\}$ is propagated, reducing $\{\neg s_{[0,2]}, s_{dp,vp,[0,2]}\}$ to $\{s_{dp,vp,[0,2]}\}$
2. $\{s_{dp,vp,[0,2]}\}$ is propagated, reducing $\{\neg s_{dp,vp,[0,2]}, s_{dp,vp,[0,2]:1}\}$ to $\{s_{dp,vp,[0,2]:1}\}$

3. $\{s_{dp,vp,[0,2]:1}\}$ is propagated, reducing $\{\neg s_{dp,vp,[0,2]:1}, dp_{[0,1]}\}$ to $\{dp_{[0,1]}\}$, and $\{\neg s_{dp,vp,[0,2]:1}, vp_{[1,2]}\}$ to $\{vp_{[1,2]}\}$
4. $\{dp_{[0,1]}\}$ is propagated, reducing $\{\neg dp_{[0,1]}, dp_{d,np[0,1]}, dp_{pn[0,1]}, dp_{prn[0,1]}\}$ to $\{dp_{d,np[0,1]}, dp_{pn[0,1]}, dp_{prn[0,1]}\}$
5. $\{vp_{[1,2]}\}$ is propagated, reducing $\{\neg vp_{[1,2]}, vp_{vi[1,2]}, vp_{vb,cp[1,2]}, vp_{vt,dp[1,2]}\}$ to $\{vp_{vi[1,2]}, vp_{vb,cp[1,2]}, vp_{vt,dp[1,2]}\}$
6. assume $\{dp_{dp,pn[0,1]}\}$, clash with $\{\neg dp_{dp,pn[0,1]}\}$, backtracking
7. assume $\{dp_{pn[0,1]}\}$, reducing $\{\neg dp_{pn[0,1]}, pn_{[0,1]}\}$ to $\{pn_{[0,1]}\}$, no clash
8. assume $\{vp_{vi[1,2]}\}$, reducing $\{\neg vp_{vi[1,2]}, vi_{[1,2]}\}$ to $\{vi_{[1,2]}\}$, no clash

At this point, we have found a model where the following variables are set to true, and which exactly corresponds to the model MiniSat generates when run on the instance:

$$s_{[0,2]}, s_{dp,vp[0,2]}, s_{dp,vp[0,2]:1}, dp_{[0,1]}, dp_{pn[0,1]}, pn_{[0,1]}, vp_{[1,2]}, vp_{vi[1,2]}, vi_{[1,2]}$$

This set of facts represents the expected parse tree, with "john" being analysed as a DP and "sleeps" as a VP. Adding the clause $\{\neg dp_{[0,1]}, \neg vp_{[1,2]}\}$, we find that the SAT encoding of the parse becomes unsatisfiable, showing that no further parse exists.

A parser based on this encoding, which allows the user to specify a CFG in a textual format and a sentence to be parsed as arguments to a command-line tool, was implemented as a testing environment. Given the straightforward nature of the encoding, it is surprising to see that the parser thus constructed is actually usable for sentence lengths up to 25. In the author's opinion, this says nothing about the quality of the encoding, but more about the maturity and efficiency of current SAT solving technology.

We now turn to the question how the no-parse and missing-parse problems are encoded. The no-parse problem is simply the situation where we encode a parsing problem as above, but the resulting SAT instance is unsatisfiable. In this situation, we can analyse the infeasibility by extracting and inspecting MUSes. The difference in the missing-parse case is that we can postulate constituent labels for ranges in addition to $s_{[0,n]}$ and the input encoding. Any expected parse can simply be expressed as a set of unit assumptions about consituent labels. The MUSes in such an instance can give us very specific hints about why our expectation was wrong.

At this point, it will be helpful to introduce another example. Assume that we have parsed the sentence "old mary sleeps" for our example grammar, and the SAT-based parsing system has told us that the problem is unsatisfiable. If a linguist now wants to find out why the sentence was not licensed, she will in most cases be able to ask a more specific question. This could be an entire parse tree that was expected, but for some reason not generated as a model. For the example sentence, this could just be the expectation that "old mary" should be parsed as a DP. Both entire parse trees and such isolated assumptions can straightforwardly be stated as additional unit clauses. For our example, let us add the already mentioned assumption $dp_{[0,2]}$.

We will let the sentence "old mary sleeps" be our running example for interactive MUS extraction in the rest of this chapter. For now, we only quote and discuss a refutation proof generated by MiniSat for this instance, with the purpose of further deepening the reader's familiarity with the SAT encoding of CFG parsing. The proof as visualized by a small ad-hoc helper tool implemented in the Kahina framework is displayed in Figure 6.4. The resolution steps need to be read in a bottom-up fashion, resulting in the empty clause at

the top. In the proof we can read how all the three possibilities of establishing $d_{[0,2]}$ fail. In the right branch, the options $d_{pn[0,2]}$ and $d_{prn[0,2]}$ fail because PN and PRN are word classes which cannot range over more than one word. In the left branch, we see how the option $d_{d,np[0,2]}$ fails because it would require "mary" to be parsed as an NP, which is impossible for a proper noun according to the grammar.

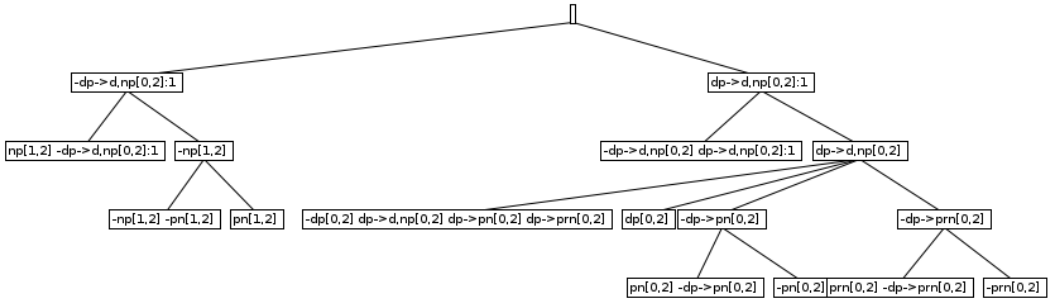


Figure 6.4: The smallest refutation proof for the missing-parse example.

With a natural SAT encoding of the parsing problem at our disposal that can easily be extended by additional assumptions to formalize no-parse and missing-parse problems, we should ask ourselves whether explanations for the third problem type of grammar engineering, the spurious-parse problem, cannot be generated in this way, too. Regrettably, the answer seems to be no. Without any formal notion of what counts as an explanation, we cannot give a formal proof of this. However, given sufficient understanding of the SAT encoding, an informal argument should suffice to establish this point. Consider what happens if we try to suppress a parse by adding negated constituent range variables to the problem. If we arrive at an unsatisfiable instance in this way, the refutation proofs generated (and also inspection of MUSes) will not explain to us why the spurious parse was generated, but could only be read as answering the question why there are no other parses.

6.2.4 Generating Test Instances

In order to have a wide array of different grammar engineering problems available for our exploration of interactive MUS extraction in practice, 120 test instances were generated from the example grammar in Figure 6.3. 100 of these instances encode the parsing problem for different sentences, of which one half is licensed and the other rejected by the grammar. The 50 unsatisfiable instances can also be seen as instances of the no-parse problem in grammar engineering. Care was taken to make use of all the grammar rules and words defined in the grammar, and multiple sentences of every length between 1 and 12 were chosen, with two outliers giving an impression of larger instances.

Concerning the sentence types, the 50 no-parse problems are a balanced mix of sentences which are grammatical in English, but not licensed by the example grammar ("he believes in Mary"), sentences which are slightly ungrammatical because one word is missing ("every man likes cat") or the order between two words is incorrect ("a young very woman likes an old man"), sentences which contain multiple errors ("sleeps cat the tree under"), and almost random sequences of words which are not analysable even by a human reader with maximum error tolerance ("every every man man","tree under sees"). The instances derived from these sentences differ in internal structure, since they encode all types of infeasibilities from a single local problem to utter chaos.

The other 20 instances are encodings of missing-parse problems, all of them derived from grammatical sentences which have no parse, but for which expectations can be expressed. To build instances of the other mentioned type of missing-parse problem, where we expect multiple readings but one of them is missing, we would need a slightly larger and much more ambiguous example grammar. Altogether, the test set attempts to sample the full range of instances as they might arise from grammar debugging systems, in robust processing of sentences produced by non-native speakers, and in educational settings where the basics of natural language processing are taught.

Because no commercial interests are involved in an experimental encoding of context-free grammar engineering, the resulting set of test instances could be released to the public with all the symbolic information attached. The author hopes that this test set will allow other researchers to develop and explore new semantically motivated ideas in SAT solving without having to either rely on benchmark sets that are stripped of any semantics, or being confined to their specific application. The symbols are defined in the DIMACS files via comments of the form `c [varID] [symbol]` between the header and the clause declarations. The full test set will be made available on the author's webpage.

6.3 Properties of the Test Instances

As for any new test set, it is of interest to characterize the instances in terms of a few basic measures. In this section, the test set developed in Section 6.2.4 is first analysed in terms of the number of clauses, the number of variables, and average clause size. Then, we use the CAMUS tool [20] for finding all MUSes to analyse the search space for as many instances as possible, condensing the acquired knowledge into statistics about average number of MUSes and their sizes. Finally, we compute statistics for the two measures proposed by Kullmann which are easy to determine, i.e. the sizes of the lean kernel and of the intersection of all MUSes. All these numbers are summarized in Figure 6.5 for both the no-parse and the missing-parse problems. In Section 6.3.4, the computed measures are compared to the corresponding values for common benchmark sets as cited in publications, helping us to evaluate the representativeness of our test case for applications of MUS extraction.

6.3.1 Problem Sizes and Basic Measures

Given the nature of the encoding, it is not surprising that the no-parse problems for sentences of the same length have exactly the same number of clauses and variables. In fact, the only difference between the instances of one length is in the unit clauses that describe the input. The first block of lines in the no-parse table of Figure 6.5 can therefore be read as exact numbers, there was no need to compute average values. The possibly most interesting fact to observe about these basic measures is how they grow with input length. Considering that the encoding introduces separate variables for every partition of every possible range for every constituent and every rule, the number of variables in the test instances is still manageable. A semi-formal consideration of the combinatorics quickly explains why this is so. The number of possible ranges is roughly $\binom{n}{2}/2 = \frac{n(n-1)}{4}$. Since most grammar rules are binary (not only in our example grammar, but in linguistic theory in general), only $m-1$ different splittings need to be considered for a range of size m. Altogether, the total number of range splittings, and thereby of variables for constant grammar size, is in $O(n^3)$.

An important question that is not answered by the computed numbers is how much the size of the encoding grows with grammar size. After all, while in a grammar engineering system we will never want to debug sentences of more than 15 words (problems in longer sentences can be reduced to minimal examples by leaving out constituents), the size of the example grammar used in this chapter is a lot smaller than what one would use in practice. A CFG

for the syntax of a natural language with reasonable coverage tends to contain not less than a few hundred rules, and uses about 60 different constituent labels. Another glimpse at the definitions of the encoding tells us that the number of constituents introduces another square factor (because of the exclusivity clauses), but that the number of rule declaration and distribution clauses is only linear in the number of rules. Still, this means that the SAT encoding of a realistic grammar will contain millions of clauses, and that one will need to confine oneself to sub-grammars to achieve acceptable response times during interactive MUS extraction for grammar debugging. Extracting and debugging sub-grammars in isolation is already a well-established technique in grammar engineering, so that this problem does not immediately detract from the viability of our SAT encoding for applications.

An interesting observation about the computed measures is that the average clause size is always very close to 2, but continually grows with the maximum input length n. The closeness to 2 is due to the large number of exclusivity constraints which make up most of the clauses. The increasing tendency is due to the growing number of splittings, which has no bearing on the number of exclusivity constraints, but produces more clauses of size 3.

6.3.2 Number and Size of MUSes

For computing the statistics about MUS sizes and the number of MUSes in the test instances, the CAMUS tool [20] for enumerating all MUSes was used. As can be seen in the second block of the results in Figure 6.5, the number of MUSes explodes with the input size. Even for $n = 4$, the enumeration of all MUSes was not feasible any longer, not terminating after half an hour for even a single instance and filling gigabytes of disk space with lists of clause IDs. Using CAMUS's option to define timeouts for both phases of the algorithm, it was possible to at least generate a sample of all MUSes for inputs of length 4, so that approximate values for the minimum and maximum MUS sizes could be determined. The values computed in this way are marked with $\geq$ or $\leq$, depending on the direction in which the real results could deviate from the ones that were feasible to compute. For $n > 4$, CAMUS does not even begin to output MUSes within 30 minutes of runtime, because already its first phase of computing maximal satisfiable subsets takes too long. Fortunately, CAMUS offers the option to perform a branch-and-bound search in order to find the smallest MUS, which is of course much more efficient than an attempt to enumerate all MUSes. Using this option, it was possible to find the smallest MUS in all instances up to $n = 5$, at which point the runtime for a single instance amounted to several hours.

For $n > 4$, the entries in the tables except for minimum MUS size are therefore only very rough estimates, which were determined by walking through the reduction graph of each instance on ten different paths created by a specialized $findAnotherRandomMUS$ heuristic that always selects a deletion candidate from the clauses of unknown status in the current US until a MUS is found. This heuristic has the advantage of disrupting previously found proofs by attempting to remove clauses involved in them, which leads to a much greater fan-out into different MUSes than what would happen for completely random deletion candidates. The table entries are the largest and smallest sizes of a MUS found in this way, and the average is only computed over the size of the MUSes encountered for each instance.

The most interesting point these data illustrate is that there is usually a considerable difference in size between the smallest and the largest MUS. This indicates that simply computing some MUS will not be enough for a grammar debugging application, since for a sentence of length 4, a MUS of size 19 will obviously be a lot easier to analyse and understand than one of size 105. This means that interactive MUS extraction promises to be of use for our application, provided that the variable symbols will give a grammar engineer useful hints which deletion candidates potentially lead to significant reductions.

no-parse: **length**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**14**	**18**
#instances	5	5	5	5	5	5	5	5	5	2	2	2	1	1
clauses	318	982	2013	3431	5256	7508	10207	13373	17026	21186	25873	31107	43296	75118
variables	33	109	238	430	695	1043	1484	2028	2685	3465	4378	5434	8015	15333
avg. clause size	1.987	1.990	1.995	2.003	2.010	2.018	2.025	2.034	2.041	2.049	2.056	2.063	2.077	2.102
avg #MUSes	3	6.8	2402	> 1.45M	–	–	–	–	–	–	–	–	–	–
min MUS size	3	6	9	19	28	39	≤ 51	≤ 64	≤ 280	≤ 308	≤ 109	≤ 780	≤ 1290	≤ 2741
max MUS size	3	11	42	≥ 105	≥ 114	≥ 196	≥ 298	≥ 323	≥ 481	≥ 469	≥ 707	≥ 973	≥ 1625	≥ 3063
avg MUS size	3	8.9	30.7	~ 36.3	~ 67.7	~ 100.0	~ 139.7	~ 192.0	~ 372.6	~ 398.1	~ 386.1	~ 848.5	~ 1401.3	~ 2773.4
avg. unusable	95.28	64.68	44.02	34.50	28.06	23.71	20.43	17.83	15.80	18.35	12.68	11.51	9.68	7.17
avg. necessary	0.31	0.43	0.29	0.22	0.16	0.22	0.08	0.14	0.20	0.16	0.09	0.12	0.39	0.08

missing-parse: **length**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
#instances	2	2	2	2	2	2	2	2	2	2
max. clauses	983	2014	3433	5258	7510	10210	13376	17028	21188	25876
variables	109	238	430	695	1043	1484	2028	2685	3465	4378
avg. clause size	1.989	1.995	2.002	2.010	2.018	2.026	2.034	2.041	2.049	2.056
avg #MUSes	30.132	1.521	> 7.04M	–	–	–	–	–	–	–
min MUS size	3	10	2	6	6	≤ 12	≤ 6	≤ 6	≤ 40	≤ 109
max MUS size	16	29	≥ 53	≥ 106	≥ 116	≥ 298	≥ 307	≥ 470	≥ 750	≥ 817
avg MUS size	6.1	24.6	~ 37.7	~ 75.8	~ 57.0	~ 170.7	~ 191.5	~ 241.1	~ 636.4	~ 471.2
avg. unusable	57.83	43.72	34.49	28.25	23.71	20.67	17.99	15.89	14.22	12.67
avg. necessary	0.05	0.32	0.13	0.12	0.05	0.04	0.00	0.00	0.01	0.00

Figure 6.5: Various statistics for the grammar engineering test instances.

6.3.3 Unusable and Necessary Clauses

Recall that according to Kullmann's classification introduced in Section 2.4.2, the unusable clauses form the complement of the lean kernel, i.e. they cannot be used in any refutation proof, making them useless for the purpose of MUS construction. The necessary clauses are the ones which are critical already at the root node of the reduction graph, which is equivalent to the intersection of all MUSes. The numbers in Figure 6.5 give the percentage of unusable and necessary clauses, respectively. Both measures were computed using the author's own implementations of autarky pruning and stack-based reduction of individual clauses in reduction graph nodes, both based on Zielke's version of MiniSat.

The percentage of necessary clauses is a good measure of the overlap between MUSes, which turns out to vary strongly depending on the structure of the error in the input. In a no-parse problem, $s_{[0,n]}$ must always be present in all conflict sets, but there are cases where two isolated infeasibilities (e.g. two missing words) do not interact at all except that they both depend on $s_{[0,n]}$, making this the only necessary clause. On the other hand, if there is only one local problem (such as one missing word) in the sentence, the number of necessary clauses can come close to the minimum MUS size. This is why the benchmark values show no clear dependency on sentence length except for very short sentences. The cause of the very low number of necessary clauses for the missing-parse instances is that the MUSes in these problems often do not overlap at all, since each of the other assumptions alone may be enough to produce a conflict.

While the number of unusable clauses is very high for short sentences, their percentage rapidly gets smaller with growing input length. While not all of these clauses can be used for constructing MUSes (see discussion in Section 2.4.2), the numbers show a clear tendency for most clauses to be usable in some way to derive a refutation proof. The high percentage of clauses which are neither necessary nor unusable explains very well where the explosion in the number of MUSes comes from.

6.3.4 Comparison to Other Benchmark Sets

If we want to generalize our observations about interactive MUS extraction on the new test set to industrial applications in general, we first need to assess the similarity of our test set to existings benchmarks. In this section, this will be done using the figures given by Oliver Kullmann et al. [27] for the Daimler instances in their investigation of clause classification, and the benchmark data for CAMUS given on Mark Liffiton's webpage [52].

While our test instances are very small compared to the industrial instances which have been used in recent SAT competitions, they are roughly comparable in size to the instances for which global MUS statistics and complete clause classification were found to be still feasible, and which were therefore analysed by Kullmann et al. and Liffiton. A major advantage of our application is that using larger grammars and longer inputs, we could create instances of arbitrary size, so that it will be easy to generate more challenging benchmarks for more efficient future tools.

A core observation concerning the benchmark data is that the number of unusable clauses are very low even compared to the Daimler test set, where the corresponding percentages vary between 43% and 97%. The numbers for other benchmark sets are generally much higher, since the instances from most other benchmark sets only contain a few MUSes, and the Daimler instances are famous for containing a comparatively large number. By contrast, the percentages of necessary clauses are rather low both in our benchmark set and the Daimler instances (all below 0.5% in our case, and all below 1.0% except very few outliers with up to 5.2% in the Daimler case). This figure is of course very different for instances

with only one or a handful of MUSes, even more so in the cases where the MUSes cover a large part of the instance.

While some of the Daimler instances with many MUSes also exhibit moderately large differences between the size of the smallest and the largest MUS (with the maximum absolute difference found by Liffition between sizes 112 and 234, and the maximum relative difference between sizes 10 and 66), the ubiquity of this phenomenon in our test set sets it apart from all the industrial instances considered by Liffiton.

Altogether, the instances we generated are not similar to industrial instances in every respect, especially in that the number of MUSes seems to exceed everything previously seen in benchmark sets, and that the variation in MUS sizes is unusually high. The instances therefore contain some of the most interesting MUS landscapes in available SAT instances, but only in a few aspects approximate the industrial applications we set out to emulate. The only benchmark set for which some conclusions can be drawn is the Daimler test set, which displays at least some variation in MUS sizes, and a high number of MUSes for a few instances. For interactive reduction, the grammar engineering test set turns out to be a particularly attractive test case because it is clear that simply extracting some MUS will in general not lead to useful explanations.

6.4 Defining the Relevant Clauses

Many clauses of the SAT encoding have the purpose of enforcing the semantics of CFG rules, but are not informative from the standpoint of a user who is familiar with the formalism. For instance, a grammar engineer does not need to be told that the categories NP and VP exclude each other, or that applying a binary rule to a range forces one to split the range between the child constituents. The clauses which enforce this common knowledge unnecessarily inflate the MUSes of our instances, making them difficult to interpret because of all the clauses that are essential for the encoding to work, but constitute nothing but visual clutter for a user trying to understand the conflict.

Because of space limitations, we cannot quote a full MUS which would make this problem immediately apparent. In Figure 6.6, we give the minimum MUS for our example instance "old mary sleeps". While the infeasibility encoded in this MUS is relatively straightforward to comprehend, especially with the corresponding refutation proof in Figure 6.4 as a guideline, it is obvious that some clauses such as $\{\neg np_{[1,2]}, \neg pn_{[1,2]}\}$ and $\{\neg dp_{prn[0,2]}, prn_{[0,2]}\}$, although being necessary for the refutation proof, inflate the size of the MUS without adding anything to the explanation that could not be derived immediately from knowledge about categories which exclude each other, or from basic principles of context-free grammar.

6.4.1 A Filter For Don't-Care Clauses

The type of problem described here appears to arise in many application of SAT solving where models or conflict sets need to be interpreted. The standard solution for addressing the issue is the don't-care variable mechanism introduced in Section 2.2.1. We are interested in the MUSes of the missing-parse problem F with respect to a set of relevant clauses

$\{dp_{[0,2]}\}$
$\{pn_{[1,2]}\}$
$\{\neg prn_{[0,2]}\}$
$\{\neg pn_{[0,2]}\}$
$\{\neg dp_{[0,2]}, dp_{d,np[0,2]}, dp_{pn[0,2]}, dp_{prn[0,2]}\}$
$\{\neg dp_{d,np[0,2]}, dp_{d,np[0,2]:1}\}$
$\{\neg dp_{prn[0,2]}, prn_{[0,2]}\}$
$\{\neg dp_{d,np[0,2]:1}, np_{[1,2]}\}$
$\{\neg np_{[1,2]}, \neg pn_{[1,2]}\}$
$\{\neg dp_{pn[0,2]}, pn_{[0,2]}\}$

Figure 6.6: The unfiltered MUS for the missing-parse problem "old mary sleeps".

$R \subset F$. A specialized function for selecting the elements of R needs to be defined for every application, so the don't-care mechanism was implemented in Kahina in the form of an abstract `ClauseFilter` class which has access to the instance, and can be extended for any application by implementing its single method `acceptsClause(int clauseID)`, returning `false` for clauses that are not filtered out (i.e. for $C \in R$), and `true` for $C \in F \backslash R$.

To motivate the `ClauseFilter` implementation `CfgDontCareFilter` which we will use for our further investigations, we reconsider the example MUS in Figure 6.6. As we recognized there, clauses of the type $\{\neg a_{[i,j]}, \neg b_{[i,j]}\}$ are not very informative from the perspective of a grammar engineer, since the fact that most constituent labels exclude each other is intuitively clear, and it is easy enough to spot variable pairs which encode two different constituents for the same range. `acceptsClause(int clauseID)` is therefore implemented to return `true` for these clauses in our `CfgDontCareFilter`. Clauses of type $\{\neg a_{b[i,j]}, b_{[i,j]}\}$ are also not useful, since they merely enforce the semantics of unary rules, which does not need to be made explicit to a grammar engineer. A third kind of clause which does not contribute much to any explanation is the type $\{\neg a_{b_1,\ldots,b_m[i,j]}\} \cup \bigcup_{i<i_1<\cdots<i_{m-1}<j} \{a_{b_1,\ldots,b_m[i,j]:i_1:\cdots:i_{m-1}}\}$ enforcing splits, so we return `true` for these as well.

6.4.2 Structure and Size of Relevant Clause MUSes

Not surprisingly, applying our relevance filter to the grammar engineering instances leads to a significant reduction in size. The MUSes are also reduced by a significant factor, making them much more easily interpretable. Returning to our concrete example, Figure 6.7 shows the filtered version of our minimum MUS for "old mary sleeps". Note that the size of this MUS was reduced from 10 to 6, and it is still just as easily interpretable. Verbalizing the explanation it encodes, the assumption that "old mary" is a DP means that it has to be parsed as a PN, as a PRN, or as a sequence of a D and an NP. It cannot be a PRN because PRNs of length 2 are impossible, and the same is true for the label PN. Parsing it as D NP would require "mary" to be an NP, which is incompatible to its label PN.

$$\begin{array}{ll} \{dp_{[0,2]}\} & \{\neg dp_{[0,2]}, dp_{d,np[0,2]}, dp_{pn[0,2]}, dp_{prn[0,2]}\} \\ \{pn_{[1,2]}\} & \{\neg dp_{d,np[0,2]:1}, np_{[1,2]}\} \\ \{\neg prn_{[0,2]}\} & \{\neg pn_{[0,2]}\} \end{array}$$

Figure 6.7: The filtered MUS for the missing-parse problem "old mary sleeps".

To demonstrate that this reduction in MUS size is the typical behaviour for all instances, Figure 6.8 shows the effects of the relevance filter on instance size and average MUS size (again estimated using ten runs of the $findAnotherRandomMUS$ heuristic used in Section 6.3.2) for five randomly picked sentences from the no-parse test set. We see that average relevant MUS size does not grow as fast as the average unfiltered MUS size, leading to significant improvements in maintainability and interpretability.

Sentence	$\lvert F \rvert$	$\lvert R \rvert$	$\varnothing$ $\lvert$**MUS**$\rvert$	$\varnothing$ $\lvert$**rMUS**$\rvert$
"man the sleeps"	2.013	190	19.7	12.3
"a young man mary likes"	5.256	672	66.0	47.0
"a man sleeps a tree"	5.256	672	85.9	60.6
"very old woman sees dog on street"	10.207	1.626	91.0	51.7
"cat the sleeps under tree very tall a"	13.373	2.330	126.2	69.9

Figure 6.8: Examples of the effects of the relevance filter on MUS sizes.

6.5 Interactive MUS Extraction for CFG Debugging

In this section, we analyse the behaviour of our grammar engineering instances in practice within our interactive MUS extraction system. We configure the system to apply the relevance filter defined in the last section, and collect some experience in applying interactive MUS extraction in order to form an opinion whether the idea has sufficient potential for real applications to warrant further exploration.

6.5.1 Interpreting MUSes for Grammar Engineering

Given the structure of the encoding, the MUSes extracted from grammar engineering problems are best read in top-down fashion, starting with the assumptions. The best way to analyse a conflict set is to follow the implications, taking the most plausible alternative at each disjunctive consequent. This will finally lead either to a clash between a positive and a negative literal of the same variable which gives us a very basic reason why the analysis failed, or to a point where the reason for the clash becomes apparent using the knowledge the user has about the grammar.

If we proceed in this way in order to interpret the filtered MUS for our example "old mary sleeps", our assumption $dp_{[0,2]}$ leads us to the clause $\{\neg dp_{[0,2]}, dp_{d,np[0,2]}, dp_{pn[0,2]}, dp_{prn[0,2]}\}$. Here, we already see that none of the choices makes any sense, since a DP is only allowed to consist of a proper noun, a pronoun, or a determiner followed by a DP. This would tell a grammar engineer that an additional rule needs to be added which allows an adjective and a proper noun to combine into a DP. In fact, with the rule DP $\rightarrow$ A PN added, the problem will become satisfiable, fixing the bug in the grammar.

This way of thinking amounts to reading the MUSes of unsatisfiable grammar engineering problems as suggestions for **minimal repairs** that could be executed on the grammar in order to make it license the desired parse or parse tree. Since each MUS can be considered a different explanation, many different repair suggestions can be generated in this way.

6.5.2 Observations concerning Interactive Reduction

The view of the reduction graph as a space of possible repairs leads to a viable guiding principle for interactively selecting interesting deletion candidates at the root node of the reduction graph. In general, grammar engineers will want to prioritize deletion of those clauses that represent rules which should not be changed according to their respective linguistic theories. Interactive MUS extraction then becomes a way of reasoning about the grammar, and a method for retrieving instant feedback on the effects of possible changes to the grammar.

Altogether, we arrive at a grammar debugging workflow which consists of two different work modes that complement each other. Interactive MUS extraction becomes the method of choice for navigating the space of possible changes to the grammar, while modifying the SAT encoding of the bug-ridden parse by postulating additional constituent labels for some input ranges becomes the primary approach to narrowing down the problem in order to arrive at more concise explanations, just as we did with the assumption $dp_{[0,2]}$ in the case of "old mary sleeps".

6.5.3 Displaying Symbolic Information

To give a user hints about worthwhile reduction attempts, e.g. by informing about the correspondences between CFG rules and clauses of the encoding, the symbolic information associated with the variable IDs needs to be made visible in the MUS extraction system.

This was done by allowing symbol tables to be loaded from the comments of DIMACS files, and by replacing the variable numbers with the corresponding symbols in the instance view and the US view. The format for the symbols in the CFG encoding is the one we already defined alongside the more mathematical notation adopted for the discussion here. Since all the clauses used in the encoding are short, the resulting clause display gives the user a complete overview of the semantic content of smaller MUSes, which can be read with about the same ease as the format we used for the MUSes in Figures 6.6 and 6.7.

For larger MUSes, the higher abstraction layer of a block partition must assist in reducing the conflicting clause set to a smaller number of manageable clause conspiracies. This use of the block inference mechanism is supported by the current prototype by an interface `BlockContentSummarizer` that can implement any function which generates from a block of clauses and the symbol table an application-dependent string representation for the block.

The first version of the `CfgBlockContentSummarizer` implementation of this interface simply retrieves the symbolic representations of the clauses in all blocks of size 1 or 2, and generates a string that shows the resulting set of sets of symbols. While this worked reasonably well for the experiments (see below), once more experience with the typical structure of inferred blocks has been collected, it will be worthwhile to think about a block summary function whose output remains closer to the encoded CFG semantics.

6.5.4 Interpreting Blocks with Respect to the Grammar

The main problem of interactive reduction on the no-parse instances turns out to be that clause set refinement almost always leads to a MUS immediately, which makes the reduction graph very flat, essentially reducing it to a list of MUSes. This is quite a contrast to the instances interactive reduction was developed on, as demonstrated by the much more interesting reduction graphs displayed in previous chapters. The underlying reason for this behaviour seems to be that there is virtually no redundancy in the encoding, making it impossible to find shorter variants of proofs, which blocks the essential mechanism behind a MUS extraction that needs several deletion-based reduction steps. This structural property of the encoding also makes large parts of the huge search spaces unaccessible if we use clause set refinement, which is not necessarily undesirable.

While the reduction graph thus turns out to be of only very limited use for our application, some of the other components provide information that goes beyond a mere list of selectable reduction candidates. The main reason for this is that the block views split up MUSes of a few dozens of clauses into more managable units, indicating repeated parts in different MUSes and making connections between clauses more directly visible.

While it is true that every MUS provides a different explanation of the unfeasibility, the visualization components of the interactive MUS extraction system make it possible to inspect the overlaps and the differences between these explanations. The intersection of all MUSes is often very small (as discussed in Section 6.3.3), and it is interpretable as a core that is common to all the explanations. In each of the MUSes that are discovered via interactive reduction, this core of the conflict information is enhanced by a few small blocks of clauses which together cause a conflict. Being able to compare multiple different ways in which the conflict arises makes it much easier to find the core of an infeasibility.

These observations are illustrated very well by Figure 6.9, where we see a state of the user interface in the middle of an interactive MUS extraction process for our missing-parse example problem. The screenshots show the information displayed for two different MUSes. The MUS of size 6 is the one from Figure 6.7, which we already analysed in Section 6.4.2. The inferred block cuts this MUS into four parts: three blocks of size 1, and one block of

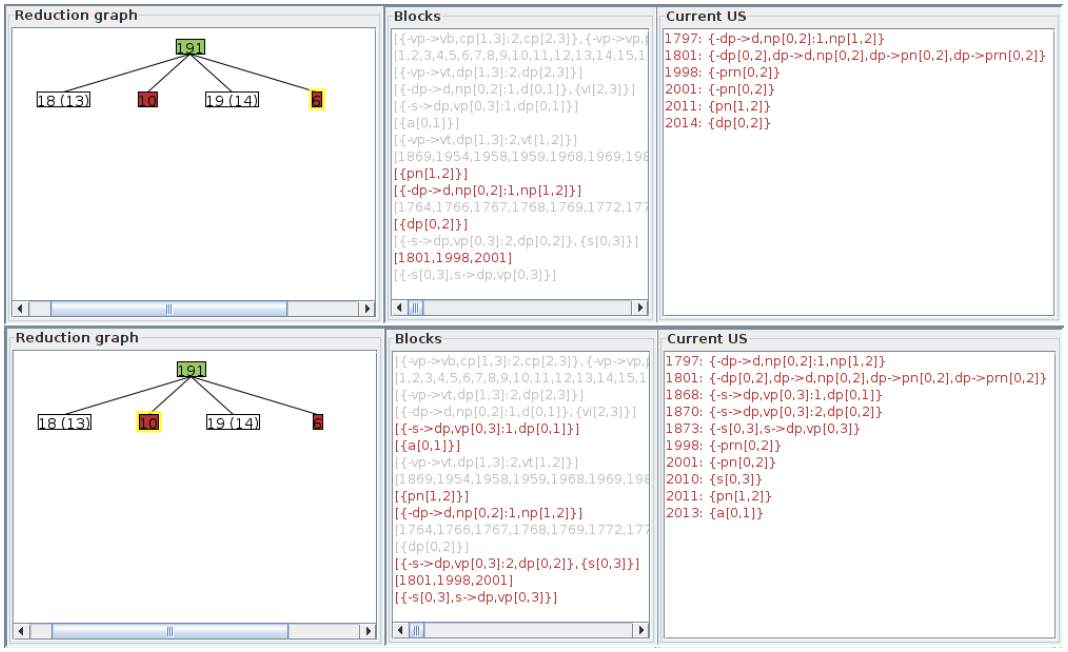


Figure 6.9: Two different MUSes for the missing-parse example in the prototype.

size 3. By comparison to the corresponding display for the MUS of size 10, we see that the assumption $\{d_{[0,2]}\}$ is not part of that MUS, but its removal had to be compensated by three additional blocks of size 1, and the block $\{\{s_{[0,3]}\}, \{\neg s_{dp,vp[0,3]:1}, dp_{[0,2]}\}\}$ of size 2. It is obvious that the clauses of this block contribute to a derivation of $\{d_{[0,2]}\}$, the assumption that was used by the smaller MUS. Considering the three other additional blocks, we see that one of them states that S $\rightarrow$ DP VP is the only rule which can be used to label a range with the sentence symbol S, and the other two show that the splitting alternative $s_{dp,vp[0,3]:1}$ does not work out, completing the derivation of $\{d_{[0,2]}\}$.

To see the merit in making MUS overlaps transparent, note that the visualization shows that the MUSes share the blocks $\{\{\neg dp_{[0,2]}, dp_{d,np[0,2]}, dp_{pn[0,2]}, dp_{prn[0,2]}\}, \{\neg prn_{[0,2]}\}, \{\neg pn_{[0,2]}\}\}$, $\{\{\neg dp_{d,np[0,2]:1}, np_{[1,2]}\}\}$, and $\{\{pn_{[1,2]}\}\}$. While these five clauses do not yet constitute a MUS, it is obvious that they encode the core of the infeasibility, which can be extended to a MUS by adding any set of clauses that in some way derive the additional $\{d_{[0,2]}\}$.

6.6 Conclusion

In this chapter, we have endeavoured to assess the potential of interactive MUS extraction in an application. The unavailability of instances that include symbolic information forced us to develop a novel application of SAT solving, which was done in one of the author's core areas of knowledge, namely the syntactic analysis of natural language. We confined ourselves to context-free grammars as a traditional common ground for the various grammar formalisms currently used in NLP, and developed a SAT encoding of problems which commonly occur during grammar debugging.

A set of 120 test instances was generated from this encoding, including 70 instances which represent typical grammar debugging problems. Our analysis of these instances by means of the available tools showed an explosion in the number of MUSes for non-trivial sentence

lengths. At the same time, the smallest MUSes found all have a managable number of clauses, which could further be reduced by an application-specific relevance filter to about two thirds the size.

Looking at the applicability of interactive MUS extraction for these instances, we saw that the structure of the reduction graph is typically very flat because clause set refinement almost always generates a MUS. However, the possibility to generate new explanations by defining parts of the grammar which are not desired as parts of it enables the user to systematically generate different MUSes which explain the infeasibility from different angles. An adaptation of the partition block view for displaying a semantic summary of the content of smaller blocks was found to be extremely helpful. The block partition proved a powerful tool for comparing different unsatisfiable subsets, making it possible to isolate the core of a grammar bug very quickly.

One problem of the current display of semantic information, which exclusively relies on the symbols generated for the encoding, is that effective interpretation of the conflict sets presupposes a certain familiarity with the inner workings of the encoding. One possible way of improving this aspect would be to order the clauses in the MUS display into a tree structure that is similar to a refutation proof, but does not contain any of the clauses that were rejected by the relevance filter. Another approach would be to develop a scheme for computing semantic block summaries that abstracts away from the encoding details and presents the contained information in a format that is closer to the usual notation of linguistic rules.

Chapter 7

Conclusion and Outlook

This final chapter begins by summarizing and assessing the results of the thesis from both a theoretical and a practical perspective. The second part concerns the current state of the prototype system for interactive MUS extraction, focusing on structural weaknesses as well as various teething troubles that remain to be resolved. The final section presents three promising avenues for future work based on the results of this thesis.

7.1 Interactive MUS Extraction as a Paradigm

The guiding idea for this thesis was to turn deletion-based MUS extraction into an interactive process, with the goal of allowing experts to use their domain knowledge while looking for good explanations of infeasibilities. This general idea has been thoroughly explored and partially evaluated in the four main chapters of the thesis. For a recapitulation of the results, however, it is best not to follow the chronology of chapters. Instead, the theoretical concepts and considerations will be discussed separated from practical aspects of the implementation, which are discussed together with the experimental evidence derived from running the prototype system on a novel benchmark set.

7.1.1 Theoretical Results

Starting from the basic idea of explicitly modelling the search space of deletion-based MUS extraction as a graph of USes, this reduction graph was conceived as a subset lattice. The discussion has shown that many of the core concepts in MUS extraction can be seen from a new angle if they are interpreted as information about the subset lattice.

We have analysed the information that can be gained from successful and unsuccessful reduction attempts in the deletion-based MUS extraction paradigm, and developed a scheme for learning and retrieving this information for a maximum of information reuse during interactive search space exploration. We have shown that a clausal meta instance over the selector variables can store information about all the encountered satisfiable subsets, and this in a way that makes it possible to retrieve the clauses which are implied to be critical in some US by unit propagation of selector variables alone. Dually, redundancy information can be stored and retrieved by maintaining a second meta instance in DNF.

The general idea of expressing connections between clauses as meta constraints over selector variables was shown to have other potential applications such as an axiomatization of the desired MUS size or a generalized variant of group-based MUS extraction.

Block-based representations were then developed mainly in order to derive more compact representations of a clausal meta instance, but they were found to be interesting from a theoretical perspective as well. Two general possibilities for inferring a block structure over selector variables and thereby over clauses were explored in some detail. The block partition structure was shown to be of use as a method for expressing GMUS extraction, whereas the block tree structure was found to be closely related to the Tseitin encoding of formulae in negation normal form.

The final theoretical contribution of this thesis is the development of a SAT encoding of CFG parsing, with the purpose of not only building a parser, but a system that generates feedback information in the form of MUSes if a parse has failed. We have also seen that this SAT encoding can easily be extended by additional assumptions in order to encode missing-parse problems as encountered in symbolic grammar engineering.

7.1.2 Experimental Results

In order to provide a usable prototype implementation of interactive MUS extraction, the preparations for this thesis have also involved extensive implementation work. The core components of the resulting prototype system are an interactive visualization of the reduction graph with color-coded state information in a US inspection view, and a parallel architecture which allows user-defined reduction agents to be used for automated search space exploration. Both automated reduction agents and the user have access to state-of-the-art techniques in MUS extraction, based on fully integrated new implementations of both model rotation and lean kernel extraction.

A central focus of the implementation work was on putting into practice the meta instance approach developed in theory. Only the clausal meta instance for storing and deriving reducibility information was added to the system, since the technique of clause set refinement renders explicit maintenance of fall-away information too cost-inefficient. Both block inference schemes were implemented in the system, and the efficiency of a block partition representation in terms of the compression rate was determined experimentally on a small set of randomly selected benchmark instances. The user interface was extended by experimental display components for direct interaction with the block structures, leading to interesting alternatives to the standard interactive MUS extraction workflow.

One of the central purposes of the implementation was to evaluate the usefulness of the new interactive MUS extraction paradigm in practice. However, the unavailability of test instances that contain the symbolic information necessary for such an evaluation has made it difficult to find an adequate testing environment. By developing a SAT encoding of context-free grammar engineering problems, a major effort was made to address this problem by approaching a task from the author's area of knowledge. However, the benchmark set generated from this application turned out to have properties which, while being interesting in themselves, are not shared with typical benchmark instances from industrial applications. For this reason, our analysis of these instances fell short of providing conclusive evidence for or against the general applicability of interactive MUS extraction. Still, a few useful initial observations indicating the paradigm's practicality could be made.

Nevertheless, the SAT instances thus created have very unusual structural properties concerning the number of MUSes as well as the great size differences between them. These properties make the test set potentially valuable for the SAT community, not only because it is derived from an application that has not been very visible to the automated reasoning community, but especially because the instances can be freely redistributed with all the symbolic information attached.

7.2 Shortcomings of the Prototype Implementation

The interactive MUS extraction system described in this work has a range of shortcomings that still limit its practical usefulness at the moment. Some of the problems are due to the slightly ad-hoc system architecture that leads to performance issues on larger instances, others are due to the immature user interface. In this section, the most prominent problems of both types are discussed along with possible future solutions.

7.2.1 Architectural Limitations and Performance Issues

Some of the view update computations in the current version of the prototype are relatively time-consuming, especially that of custom content summaries for the block displays, and the recomputation of the reduction graph view. If large parts of the search space are explored interactively, the latency introduced by the slow view updates becomes rather noticable. These problems could be alleviated by more efficient implementations of graph layout and list rendering algorithms, but they will never cease to appear on larger instances if the user expects all views to be up to date.

The problem with some view computation times was aggravated by the difficulty to implement a regime which updates interacting views so often that the user is always presented with the current state of the model in all views, without wasting a lot of computation time on unnecessary redraws. In essence, this problem can only be avoided if each component remembers at any time whether it needs to be redrawn. Monitoring this need for redraws again presupposes an implementation of the corresponding mechanism in all the data models. Kahina is beginning to be extended by support for such a mechanism, so it might only be a matter of time until a solution for this problem is provided.

In addition, while the algorithms such as block inference and assumption propagation are implemented well enough to work for instance sizes up to 10,000 clauses, beyond that the long computation times detract from the ease of workflow. It is very probable that these problems can be remedied by more efficient data structures and less naive implementations of the algorithms, but it appears that especially the possibilities for concurrent reduction will need to be restricted to make the system practicable for larger instances.

7.2.2 Weaknesses of the User Interface

The user interface of the prototype is at some places not optimally intuitive. For instance, this concerns the clause selection mechanism which has some potential to confuse users who are not familiar with the subselection workflow. Another case in point is the obscurity of the exact operations that are executed in response to a double click in the block display. At least some feedback should be displayed to explain what is happening under the hood.

Another general problem of the current version is the lack of instantaneous feedback when an action was started that needs some time to compute. It is always possible to find out what the system is doing by inspecting the console output, but the user interface alone leaves the user with incomplete information about the current state. If the user then assumes that an action was not started properly and issues new commands while others are pending, bad interactions including GUI freezes can occur, e.g. if a different node in the reduction graph is selected while a user-executed reduction step is pending. In a release version, these problems must be resolved by making the current state more transparent in the interface, and by blocking options that might cause unpredictable interactions that compromise system stability. It should also be possible to cancel pending operations in case of problems.

7.3 Future Work

In this final section, we discuss three major areas in which further work that builds on the results of this thesis suggests itself. These opportunities arise from both the theoretical and the practical part of the thesis, but also from further exploring the proposed application of SAT technology in grammar engineering systems.

7.3.1 Further Investigation of Meta Constraints

Some open questions about the concepts introduced in this thesis might be worth answering. For instance, the question remains open whether block trees could be given more semantics by analyzing the space of possible block trees derivable by block inference, and connecting properties of this search space to features of SAT instances. Another open question is in how far both block inference algorithms can be applied to the dual case of a meta instance that collects unsat wedges in the form of minterms, and how the implied fall-away information could efficiently be extracted from this DNF instance.

We have only touched the surface of possible applications for the general idea of working with meta constraints over selector variables. The generalization of GMUS extraction and the axiomatization of MUS size discussed at the end of Chapter 4 are merely a few initial ideas in this direction. The possibility of explicitly constraining in arbitrarily complex ways which clauses may occur together in conflict sets, or defining clauses to automatically fall away under certain conditions, results in a framework in which many existing approaches to extracting minimal explanations might be unified. New applications in defining relevant sets of minimal unsatisfiable cores might arise from this as well, a topic area that will need to be discussed with experts from different application areas.

7.3.2 Extensions and Improvements to the Prototype

The most obvious future extensions of the prototype would be the introduction of special modes for GMUS and minimal unsatisfiable subformula extraction that build on the close connections to block structures investigated in Chapter 5, but do not make it necessary for the user to generate the relevant meta constraints for each instance before importing them into the interactive system, as it would need to be done in the current version.

Another major extension would consist in adding expansion agents as the dual mechanism to reduction agents, unifying insertion-based and deletion-based MUS extraction in one search space visualization. This would imply the need to maintain two different meta instances, a CNF instance for storing sat wedges in the form of clauses, and a DNF instance for storing unsat wedges in the form of minsets.

7.3.3 Exploring SAT-based Grammar Engineering

The encoding of grammar engineering problems for context-free grammar is of value to computational linguistics because it shows that SAT encodings might well be a practicable approach to modelling and solving grammar engineering problems. It will therefore be interesting to develop similar encodings for the more complex grammar formalisms that are used in practice. A promising avenue for further research would then be to implement an experimental grammar engineering system around interactive MUS extraction, allowing for interactivity between elements of the grammar definition and the corresponding clauses of the SAT encoding. This could also involve the development of methods for making use of refutation proofs in order to enhance the readability of MUS-based explanations.

Bibliography

[1] Robert Allen Reckhow. *On the lengths of proofs in the propositional calculus.* PhD thesis, 1976.

[2] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[3] David A. Plaisted and Steven Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[4] G. Audemard and L. Simon. Glucose's Home Page. Web, 2012. Access date: 2012-11-16. http://www.lri.fr/~simon/?page=glucose.

[5] Institute for Formal Models and Verification. PrecoSAT. Web, 2012. Access date: 2012-11-16. http://fmv.jku.at/precosat/.

[6] Bioinformatics, and Empirical & Theoretical Algorithmics Laboratory, University of British Columbia. SATzilla: Portfolio-based algorithm selection for SAT. Web, 2012. Access date: 2012-11-16. http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/.

[7] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[8] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[9] João P. Marques Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[10] The SAT association. The international SAT Competitions web page. Web, 2012. Access date: 2012-11-16. http://www.satcompetition.org/.

[11] Niklas Eén and Niklas Sörensson. The MiniSat Page. Web, 2012. Access date: 2012-11-16. http://minisat.se/MiniSat.html.

[12] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing - SAT 2003, 6th International Conference, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[13] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010), Lugano, Switzerland, October 20-23*, pages 221–229, 2010.

[14] Viktor Schuppan. Towards a Notion of Unsatisfiable Cores for LTL. In Farhad Arbab and Marjan Sirjani, editors, *FSEN'09*, pages 57–72. School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran, 2009.

[15] Hans Kleine Büning and Oliver Kullmann. Minimal Unsatisfiability and Autarkies. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 339–401. IOS Press, 2009.

[16] Inês Lynce and João P. Marques Silva. On Computing Minimum Unsatisfiable Cores. In *Theory and Applications of Satisfiability Testing - SAT 2004, 7th International Conference, Vancouver, BC, Canada, 10-13 May 2004, Online Proceedings*, 2004.

[17] Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João P. Marques Silva, and Karem A. Sakallah. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In Bacchus and Walsh [53], pages 467–474.

[18] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding All Minimal Unsatisfiable Subsets. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '03, pages 32–43, New York, NY, USA, 2003. ACM.

[19] James Bailey and Peter J. Stuckey. Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, PADL'05, pages 174–186, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reason.*, 40(1):1–33, January 2008.

[21] Martin Lahl. Beweisbasierte Berechnung von minimal unerfüllbaren Kernen. Diplomarbeit, Universität Tübingen, May 2012.

[22] Hans van Maaren and Siert Wieringa. Finding Guaranteed MUSes Fast. In Büning and Zhao [54], pages 291–304.

[23] Paolo Liberatore. Redundancy in logic I: CNF propositional formulae. *Artificial Intelligence*, 163(2):203–232, April 2005.

[24] João P. Marques-Silva and Inês Lynce. On Improving MUS Extraction Algorithms. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011, 14th International Conference, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings.*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011.

[25] Anton Belov and João Marques-Silva. Accelerating MUS extraction with Recursive Model Rotation. In Per Bjesse and Anna Slobodová, editors, *11th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011), Austin, TX, USA, October 30 - November 02*, pages 37–40, 2011.

[26] Siert Wieringa. Understanding, Improving and Parallelizing MUS Finding Using Model Rotation. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 672–687. Springer, 2012.

[27] Oliver Kullmann, Inês Lynce, and João Marques-Silva. Categorisation of Clauses in Conjunctive Normal Forms: Minimally Unsatisfiable Sub-clause-sets and the Lean Kernel. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 22–35. Springer, 2006.

[28] Oliver Kullmann. On the use of autarkies for satisfiability decision. *Electronic Notes in Discrete Mathematics*, 9:231–253, 2001.

[29] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.

[30] Mark H. Liffiton and Karem A. Sakallah. Searching for Autarkies to Trim Unsatisfiable Clause Sets. In Büning and Zhao [54], pages 182–195.

[31] OKsolver-2002. Oksolver. Web, 2012. Access date: 2012-11-19. http://cs-svr1.swan.ac.uk/~csoliver/OKsolver.html.

[32] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal Methods for the Validation of Automotive Product Configuration Data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, January 2003. Special issue on configuration.

[33] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Tracking MUSes and Strict Inconsistent Covers. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 39–46. IEEE Computer Society, 2006.

[34] Barry O'Sullivan, Alexandre Papadopoulos, Boi Faltings, and Pearl Pu. Representative Explanations for Over-Constrained Problems. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 323–328. AAAI Press, 2007.

[35] Johannes Dellert, Kilian Evang, and Frank Richter. Kahina, a Debugging Framework for Logic Programs and TRALE. *The 17th International Conference on Head-Driven Phrase Structure Grammar*, 2010.

[36] Kilian Evang and Johannes Dellert. Kahina - Trac. Web, 2012. Access date: 2012-11-24. http://www.kahina.org/trac.

[37] Stephan Kottler, Christian Zielke, Paul Seitz, and Michael Kaufmann. CoPAn: Exploring Recurring Patterns in Conflict Analysis of CDCL SAT Solvers - (Tool Presentation). In Cimatti and Sebastiani [55], pages 449–455.

[38] Yuliya Lierler and Peter Schüller. Parsing Combinatory Categorial Grammar via Planning in Answer Set Programming. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning*, volume 7265 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2012.

[39] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* Series 3. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 345–351. Springer, 2011.

[40] Department of Information and Computer Science, Aalto University. ASPTOOLS: A Tool Collection for ASP. Web, 2012. Access date: 2012-11-28. http://www.tcs.hut.fi/Software/asptools/.

[41] João P. Marques Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1999.

[42] Lintao Zhang. On Subsumption Removal and On-the-Fly CNF Simplification. In Bacchus and Walsh [53], pages 482–489.

[43] James M. Crawford and Larry D. Auton. Experimental Results on the Crossover Point in Satisfiability Problems. In Richard Fikes and Wendy G. Lehnert, editors, *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*, pages 21–27. AAAI Press / The MIT Press, 1993.

[44] Hantao Zhang and Mark E. Stickel. An Efficient Algorithm for Unit Propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, pages 166–169, Fort Lauderdale (Florida USA), 1996.

[45] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[46] Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect Hashing and CNF Encodings of Cardinality Constraints. In Cimatti and Sebastiani [55], pages 397–409.

[47] Andrew Carnie. *Syntax: A Generative Introduction.* Blackwell Publishing, 2007.

[48] Stuart M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, 1985.

[49] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, 1992.

[50] Eugene Charniak. Statistical Parsing with a Context-Free Grammar and Word Statistics. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the 14th National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island*, pages 598–603. AAAI Press / The MIT Press, 1997.

[51] Michael Sipser. *Introduction to the Theory of Computation.* Thomson, 2006.

[52] Mark Liffiton. Mark Liffiton - CAMUS. Web, 2013. Access date: 2013-01-22. http://sun.iwu.edu/~mliffito/camus/.

[53] Fahiem Bacchus and Toby Walsh, editors. *Theory and Applications of Satisfiability Testing - SAT 2005, 8th International Conference, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science.* Springer, 2005.

[54] Hans Kleine Büning and Xishun Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science.* Springer, 2008.

[55] Alessandro Cimatti and Roberto Sebastiani, editors. *Theory and Applications of Satisfiability Testing - SAT 2012, 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science.* Springer, 2012.