Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Combining Clause Learning and Resolution for Multimodal Reasoning

Daniella Angelos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Cláudia Nalon

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Combining Clause Learning and Resolution for Multimodal Reasoning

Daniella Angelos

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof.a Dr.a Cláudia Nalon (Orientadora)
CIC/UnB

Prof. Dr. Bruno Lopes Vieira
UFF

Prof. Dr. Thiago Paulo Faleiros
CIC/UnB

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 13 de agosto de 2019

Dedicatória

Dedico este trabalho à minha mãe, Ilma Vitorino, por toda a dedicação e proteção direcionados à criação de suas filhas, que possibilitaram que eu me tornasse uma mulher forte, assim como ela. Todo o meu amor incondicional é seu, mãe. Às minhas irmãs Iasmin e Aline que viveram e vivem comigo todos os momentos, fáceis ou difíceis, também dedico este trabalho.

Dedico também ás minhas amigas de infância, Fyama e Jéssica, a família que eu escolhi. Pessoas incríveis e cheias de luz que estão ao meu lado a muitos anos. Mesmo que por vezes não encontremos tempo ou meios para nos encontrar, sempre damos um jeito de estar presente na vida uma das outras.

Por fim, dedico este trabalho ao meu melhor amigo e namorado Pedro Lima. Sua bondade, generosidade e senso de justiça me inspiram muito. Sou muito grata por termos escolhido entrelaçar nossos dias com amor e carinho mútuos e pelo tanto que nos acrescentamos desde então.

Agradecimentos

Gostaria de agradeceder à professora Cláudia Nalon pela excelente orientação, sua atenção aos detalhes e seu toque humano, que fizeram total diferença ao longo de todos esses anos trabalhando juntas. Obrigada pela compreensão durante todos os meus dias difíceis. Sinto que fiz uma parceria para a vida.

Agradeço também à CAPES, a agência de fomento que subsidiou meus estudos ao longo de todo o mestrado. A bolsa que recebi foi essencial para a minha permanência na universidade.

Adiciono um agradecimento especial aos meus colegas Lucas Amaral e Felipe Rodopoulos. Obrigada pela parceria ao longo do mestrado, os incentivos mútuos e a companhia durante os congressos.

Resumo

Vários aspectos de sistemas computacionais complexos podem ser modelados utilizando linguagens lógicas, que permitem a caracterização de noções como probabilidades, possibilidades, noções temporais, conhecimento e crenças [32, 37, 38, 64]. Lógicas modais, mais especificamente, têm sido amplamente estudadas em Ciência da Computação, pois possibilitam modelar de forma natural, por exemplo, noções de conhecimento e crença em sistemas multiagentes [16, 32, 64] e aspectos temporais na verificação formal de problemas relacionados a sistemas concorrentes e distribuídos [37, 38]. Apesar de sua simplicidade, linguagens modais são expressivas para raciocinar sobre relações entre diferentes contextos ou interpretações [15].

Uma vez que um sistema é especificado em uma linguagem lógica, é possível usar métodos de prova para verificar as propriedades desse sistema. Em geral, se $\mathcal{I}$ é o conjunto de fórmulas que representam a implementação e $\mathcal{S}$ é o conjunto que caracteriza a especificação, o processo de verificação consiste em mostrar que é possível *derivar* os aspectos especificados em $\mathcal{S}$ a partir de $\mathcal{I}$. Cada método de prova corresponde a um conjunto de regras de inferência acompanhado de uma metodologia de como lidar com as fórmulas. Geralmente, a literatura sobre um sistema de prova contém análises de melhor e pior casos. Também pode-se encontrar na literatura comparações entre diferentes métodos de prova para uma linguagem lógica específica. Para lógicas modais, por exemplo, uma análise empírica de desempenho de quatro sistemas de prova diferentes pode ser encontrada em [42]. Ainda é possível combinar métodos de prova para se beneficiar do melhor cenário de cada método.

Frequentemente, métodos de prova são projetados para serem implementados como provadores de teoremas automatizados, ou seja, aspectos relacionados à busca por uma prova também são considerados no projeto de um método de prova específico. Na literatura, existem vários provadores de teoremas para lógicas modais [4, 67, 76, 77]. Neste trabalho, o foco é o provador de teoremas para a lógica multimodal básica K_n: KSP [61], que implementa o método baseado em resolução proposto em [60]. K_n é a linguagem que estende a lógica proposicional clássica com a adição dos operadores modais de necessidade, denotado por $\boxed{a}$, e de possibilidade, denotado por $\diamondsuit$, ambos indexados por um agente a.

A semântica dos operatores é dada por "é necessário do ponto de vista do agente a" e "é possível do ponto de vista do agente a", respectivamente.

O KSP traduz as fórmulas de entrada para uma forma normal clausal na qual cláusulas são rotuladas pelo nível modal em que ocorrem, ajudando a restringir aplicações desnecessários das regras de inferência do cálculo que o KSP implementa. Para reduzir o espaço de busca para uma prova, vários refinamentos e estratégias de simplificação são implementadas como parte do provador KSP. Para obter o melhor desempenho para uma fórmula específica, ou classe de fórmulas, é importante escolher as estratégias e otimizações mais adequadas.

De acordo com [61], o KSP apresenta um ótimo desempenho se o conjunto de símbolos proposicionais for uniformemente distribuído pelos níveis modais. No entanto, quando há um grande número de símbolos proposicionais em apenas um nível específico, a eficiência diminui. A razão é que a tradução para a forma normal usada sempre gera conjuntos satisfatíveis de cláusulas proposicionais, ou seja, cláusulas sem operadores modais. Como resolução depende da saturação do conjunto de cláusulas, isso pode custar muito tempo. Este trabalho contribui para o KSP com a implementação de uma opção adicional à lista de estratégias disponíveis do provador. O objetivo dessa nova opção é tentar reduzir o tempo que o KSP gasta saturando o conjunto de cláusulas, e, possivelmente, aumentar a velocidade na qual as regras de inferência que lidam com o raciocínio modal podem ser aplicadas. Portanto, a contribuição deste trabalho consiste, principalmente, na adição de duas novas regras ao cálculo implementado pelo KSP. Um dos resultados apresentados é a demonstração de que as regras são corretas e a adição de ambas não interfere na completude do provador.

Nossa implementação utiliza um provador de satisfatibilidade booleana baseado em aprendizagem de cláusulas conduzida por análise de conflitos (CDCL SAT *solver*). O problema de satisfatibilidade booleana (SAT) é o problema de determinar se existe alguma valoração para as variáveis de uma fórmula proposicional que torne esta fórmula verdadeira. Provadores para este problema (SAT *solvers*) são em geral conhecidos por serem muito eficientes. Esses provadores comumente são capazes de resolver problemas considerados difíceis, com mais de um milhão de variáveis e milhões de restrições [35]. Uma das principais razões para o amplo uso de SAT em muitas aplicações é que os CDCL SAT *solvers* são muito eficientes na prática [35]. A principal idéia por trás da aprendizagem de cláusulas implementada nesses *solvers* é adicionar as causas de um dado conflito, derivado de atribuições parciais de valores de verdade a símbolos proposicionais, como cláusulas aprendidas. Então essas informações são usadas para podar a árvore de busca por uma valoração satisfatível em diferentes partes.

Competições anuais também influenciaram o desenvolvimento de implementações inteligentes e eficientes dos provadores baseados em SAT, como Chaff [56], MiniSat [73] e Glucose [5]. Essas competições permitem que muitas técnicas sejam exploradas e criam uma extensa coleção de instâncias de problemas do mundo real e problemas projetados como desafiadores, formando um grande banco referência de casos de teste [35]. Além da análise de conflitos, os provadores modernos incluem técnicas como estruturas de dados com avaliação preguiçosa, reinicializações de busca, heurísticas de ramificação controlada por conflitos e estratégias de exclusão de cláusulas [14]. Em 2005, mais especificamente, Sörensson e Een desenvolveram um provador SAT com minimização de cláusulas de conflito, chamado *MiniSat* [73], que permaneceu por um tempo como o SAT *solver* do estado da arte. O MiniSat é um provador popular, com um desempenho surpreendente, que implementa de forma sucinta muitas heurísticas conhecidas. Ele ganhou vários prêmios na competição SAT 2005 e tem a vantagem de ser código aberto. Mantém até hoje sua importância como uma ferramenta usada como parte integrada de diferentes sistemas [26, 27, 28, 75]. Aproveitamos os esforços teóricos e práticos que foram direcionados em melhorar a eficiência de tais provadores para ajudar a acelerar o processo de saturação dentro dos procedimentos já implementados no KSP.

Nossa implementação modifica o provador KSP para realizar uma chamada externa ao MiniSat. O provador modificado pode ser encontrado em [1]. A idéia principal por trás dessa combinação é criar uma consulta para o SAT *solver* contendo cláusulas especiais criadas a partir de cada nível modal. Se o provador baseado em CDCL aprender uma ou mais cláusulas pelo procedimento de análise de conflitos, adicionamos-as ao conjunto de cláusulas proposicionais da respectiva instância do KSP, pois sabemos que essas cláusulas aprendidas são consequências do conjunto que passamos ao MiniSat. Além disso, se o MiniSat concluir que o conjunto dado como entrada é insatisfatível, também aprendemos uma cláusula que, por construção, deve ser uma das premissas das regras de inferência que lidam com raciocínio modal. Aprendendo esta cláusula, nossa hipótese é que a aplicação dessas regras será possível sem que o KSP dependa da saturação do conjunto de cláusulas proposicionais.

A combinação implementada foi testada com três grandes bancos de problemas clássicos, LWB [11], MQBF [43, 53] e 3CNF [53], em um Ubuntu 18.04 com um processador Intel i7 e 16 GB de memória. Assim como o KSP sozinho, nossa combinação não conseguiu resolver nenhuma instância dos problemas pertecentes ao 3CNF no limite de tempo de 20 minutos. Para os demais bancos, o limite de tempo dado foi de 100 segundos. Os resultados experimentais de nossa implementação executando com os bancos MQBF e LWB não apresentaram, em geral, um ganho em tempo de execução, provavelmente devido à sobrecarga que inserimos com a chamada ao MiniSat.

Entretanto, nossa implementação foi capaz de avançar o KSP na resolução de entradas insatisfatíveis de uma das famílias mais difíceis do banco LWB. Essa família corresponde ao princípio da casa dos pombos estabelecido como fórmulas em K_n. Essas fórmulas são projetadas com um grande número de símbolos proposicionais distribuídos em um máximo de dois níveis modais, o que representa exatamente o cenário para o qual desejávamos melhorar a eficiência do KSP. O provador sozinho é capaz de resolver cinco instâncias dessa família, enquanto nossa combinação conseguiu resolver até a nona entrada. Outros provadores na literatura também apresentam grande dificuldade em resolver instâncias dessa família [61]. Além disso, a execução do KSP combinado com o MiniSat para essas entradas em particular mostrou que o uso do MiniSat permitiu uma diminuição significativa no número de inferências necessárias para que o KSP encontrasse uma prova. Isso é relevante quando consideramos uso de memória como parâmetro de eficiência. Este último resultado indica que nossa combinação teve um impacto positivo na execução do KSP. Acreditamos que um refinamento nas mudanças implementadas no provador, buscando um meio termo entre a sobrecarga da chamada ao MiniSat e minimizar o número de inferências feitas, poderia levar a resultados ainda melhores.

Palavras-chave: lógica modal, resolução, provador baseado em SAT, aprendizagem de cláusulas, combinação de métodos de prova

Abstract

Modal logics have been widely studied in Computer Science for allowing the characterisation of complex systems that express notions in terms of knowledge, belief etc. In this work, we focus on the theorem prover for the basic multimodal language K_n: KSP, which implements a clausal resolution method. Clauses are labelled by the modal level at which they occur, helping to restrict unnecessary applications of the resolution inference rules. KSP performs well if the set of propositional symbols are uniformly distributed over the modal levels. However, when there is a high number of variables in just one particular level, the performance deteriorates. One reason is that the specific normal form we use always generates satisfiable sets of propositional clauses. As resolution relies on saturation, this can be very time consuming. Our work contributes to KSP with the implementation of an additional option to the list of available strategies. This option attempts to reduce the time KSP spends during saturation, or even to increase the rate in which inference rules that deal with modal reasoning can be applied. We modified KSP to externally call MiniSat, a popular SAT solver based on clause learning. The output from MiniSat, as well as the clauses it might learn in the proof search, gives us important information about the set of clauses at each modal level. Although the proposed combination did not improve KSP performance in average, it allowed KSP to solve more instances of the `ph` family than before. This is one of the hardest families of the LWB benchmark, which corresponds to the *pigeonhole principle* established as a formula in K_n.

Keywords: modal logics, resolution, SAT solver, clause learning, combining proof methods

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Several aspects of complex computational systems can be modelled by means of logic languages, allowing for the characterisation of notions such as probabilities, possibilities, time, knowledge and beliefs [32, 37, 38, 64]. Modal logics, more specifically, have been widely studied in Computer Science as they can naturally model, for instance, notions of knowledge and belief in multiagent systems [16, 32, 64], and temporal aspects in the formal verification of problems related to concurrent and distributed systems [37, 38]. Despite their simplicity, modal languages are expressive languages for reasoning over relations between different contexts or interpretations [15].

Once a system has been specified in a logic language, it is possible to use proof methods to verify properties of such system. In general, if $\mathcal{I}$ is the set of formulae representing the implementation, and $\mathcal{S}$ is the set that characterises the specification, the verification process consists in showing that it is possible to *derive* the specified aspects in $\mathcal{S}$ from $\mathcal{I}$. Each proof method corresponds to a set of inference rules accompanied with a methodology of how to deal with formulae. Commonly, the literature about a proof system contains descriptions of best and worst-case scenarios. We can also find in the literature comparisons between different proof methods for a specific logic language. For modal logics, for instance, an empirical performance analysis of four different proof systems can be found in [42]. It is also possible to combine proof methods in order to benefit from each method's best-case.

Often, proof methods are designed to be implemented as an automated theorem prover, that is, aspects related to search for a proof are also addressed in the design of a particular proof method. In the literature, there are several theorem provers for modal logics [4, 67, 76, 77]. In this work, we focus on KSP [61], a theorem prover for the modal language K_n, which implements the clausal resolution method proposed in [60]. Clauses are labelled by the modal level at which they occur, helping to restrict unnecessary applications of the resolution inference rules. In order to reduce the search space for a proof, several

refinements and simplification techniques are implemented as part of KSP. To get the best performance for a particular formula, or class of formulae, it is important to choose the right strategy and optimisations.

According to [61], KSP performs well if the set of propositional symbols are uniformly distributed over the modal levels. However, when there is a high number of propositional symbols in just one particular level, the performance deteriorates. One reason is that the specific normal form used always generates satisfiable sets of propositional clauses (i.e. clauses without modal operators). As resolution relies on saturation, this can be very time consuming. Our work contributes to KSP with the implementation of an additional option to the list of available strategies. This option attempts to reduce the time KSP spends during saturation and to increase the rate in which inference rules that deal with modal reasoning can be applied.

Our implementation uses a Boolean Satisfiability Solver based on Conflict-Driven Clause Learning (CDCL), or a CDCL SAT solver. SAT solvers are well known to be very efficient in the proof search for propositional logic. They can often solve hard structured problems with over a million variables and several million constraints [35]. One of the main reasons for the widespread use of SAT in many applications is that solvers based on clause learning are very effective in practice [35]. The main idea behind Conflict-Driven Clause Learning is to catch the causes of a conflict, derived from partial assignments to propositional symbols, as learned clauses, and to use this information to prune the search in a different part of the search space.

Annual competitions have also influenced the development of intelligent and efficient implementations of SAT solvers, like Chaff [56], MiniSat [73] and Glucose [5], allowing many techniques to be explored and creating an extensive collection of real-world instances as well as challenging hand-crafted benchmarks problems [35]. Apart from conflict analysis, modern solvers include techniques like lazy data structures, search restarts, conflict-driven branching heuristics and clause deletion strategies [14]. In 2005, more specifically, Sörensson and Een developed a SAT solver with conflict clause minimisation, named *MiniSat* [73], which stayed for a while as the state-of-the-art solver. MiniSat is a popular SAT solver, with an astonishing performance, that implements many well-known heuristics in a succinct manner. It has won several prizes in the SAT 2005 competition, and has the advantage of being open-source. To this day, MiniSat keeps its importance as a tool used as an integrated part of different systems [26, 27, 28, 75]. We take advantage of the theoretical and practical efforts that have been directed in improving the efficiency of such solvers for helping to speed the saturation process within the procedures already implemented in KSP.

Our implementation modifies KSP to externally call MiniSat. The modified prover

can be found in [1]. The main idea behind this combination is to build a query for the SAT solver containing special clauses built from each modal level. If the CDCL solver learns one or more clauses by the conflict analysis procedure, we add all them back to the set of propositional clauses of KSP, as we know that these learnt clauses are consequences of the set we passed to MiniSat. Additionally, if the solver concludes that our input is unsatisfiable, we also learn a clause that, by construction, must be one of the premises of the inference rules that deal with modal reasoning. By learning this clause, our hypothesis is that the application of these rules will be possible without KSP relying on saturation.

Although experimental results of our implementation running on classical benchmarks did not present, in general, a gain in run-time parameters, probably due to the overhead we inserted calling MiniSat, we were able to advance KSP in solving unsatisfiable entries of one of the hardest families of the LWB benchmark. This family corresponds to the pigeonhole principle established as formulae in K_n. These formulae are formatted with a large number of propositional symbols distributed over a maximum of two modal levels, which represents exactly the scenario that we wish to improve the efficiency of KSP for. The prover alone is able to solve five instances of this family while our combination managed to solve until the ninth entry. Besides, the execution of KSP combined with MiniSat for these particular entries shows that the use of MiniSat allowed an overall decrease in the number of required inferences on the search for a proof. This is noteworthy when considering both time and memory issues.

The first chapters of this work set the ground theory needed to understand the scenario in which our work fits. Chapter 2 formally introduces the syntax and semantics of the propositional modal language which is the main focus of this work: K_n. In Chapter 3 we briefly introduce clausal resolution, the base of the modal-layered calculus behind KSP. We also present the calculus KSP implements, and give an overview of how the proof search of KSP, fully presented in [61], is structured.

In Chapter 4, we discuss the basic architecture often implemented in SAT solvers, and the role of clause learning in the successful, widespread use of them. In Chapter 5 we see how we can use this solvers as a new strategy option implemented in KSP. This chapter presents the additional rules needed for the combination, as well as their soundness proofs. We also present the experimental results running KSP combined with MiniSat in Chapter 5. Finally, Chapter 6 brings a general overview of the whole work and leaves suggestions for future work.

Chapter 2

Modal Logics

This chapter formally introduces K_n, a *propositional modal logic language*, semantically determined by an account of necessity and possibility [58].

In classical logic, propositions or sentences are evaluated to either *true* or *false*, in any model [44]. However, in natural language, we often distinguish between various modalities of truth, such as *necessarily* true, *known to be* true, *believed to be* true or yet true *at some time in the future*, for example.

Modal logics extend classical logic by adding operators, known as *modal operators*, to express one or more of these different modes of truth. Different modalities define different languages [15]. The essence in modal logics is that, in opposition to propositional logic, for instance, all the information available is not seen as from outside. Instead, the relational structure between given contexts is examined so formulae can be evaluated from inside these structures, i.e., at a particular state [15]. In this sense, the key concept behind the modal operators is to allow us to access the information held at different contexts or interpretations, an abstraction that here we may think as *possible worlds*.

A propositional modal language is the well known classical propositional logic language augmented by a collection of modal operators [15]. The purpose of these operators is to allow the information that holds at other worlds to be examined — but, crucially, only at worlds visible or accessible from the current one via an accessibility relation [15]. Then, the evaluation of a modal formula depends on the set of possible worlds and the accessibility relations defined over these worlds. This idea will be made precise in Section 2.2, when we define the *satisfiability* of formulae. It is possible to define several accessibility relations between worlds, and different modal logics are defined by restrictions on relations.

The modal language which is the focus of this work is the extension of the classical propositional logic plus the unary operators $\boxed{a}$ and $\diamondsuit_a$, whose readings are "is necessary from the point of view of an agent a" and "is possible from the point of view of an agent a", respectively. This language is known as K_n, where n is the number of possible agents.

The satisfiability and validity of a formula in K_n depend on a structure known as a *Kripke model*, a structure proposed by Saul Kripke to semantically analyse modal logics [46]. A set of worlds, the accessibility relations over these worlds, and a valuation function define a Kripke model.

K_n is characterised by the schema $\boxed{a}(\varphi \Rightarrow \psi) \Rightarrow (\boxed{a}\varphi \Rightarrow \boxed{a}\psi)$ (axiom K), where a is an index from a finite, fixed set of *agents*, and φ, ψ are well-formed formulae as defined in Defition 2.1.1. The addition of other axioms defines different systems of modal logics and it imposes restrictions on the class of models where formulae are valid [19]. For instance, if we add the formula $\boxed{a}\varphi \Rightarrow \varphi$ (axiom T) as an axiom, the evaluation of formulae can be restricted to models whose accessibility relations, for an agent a, are reflexive.

In the following, we will formally define the modal language. The syntax and semantics of K_n are given in Sections 2.1 and 2.2, respectively, following the presentation in [58].

2.1 Syntax

The language of K_n is equivalent to its set of *well-formed formulae*, denoted by $\mathsf{WFF}_{\mathsf{K}_n}$, which is constructed from a denumerable set of *propositional symbols* or *variables* $\mathcal{P} = \{p, q, r, \ldots\}$, the negation symbol $\neg$, the disjunction symbol $\vee$, the conjunction symbol $\wedge$, the implication symbol $\Rightarrow$, the double-implication symbol $\Leftrightarrow$, and the modal connectives $\boxed{a}$ and $\diamondsuit_a$, that express the notion of necessity and possibilitiy, for each a in a finite, non-empty fixed set of indexes $\mathcal{A} = \{1, \ldots, n\}$, where $n \in \mathbb{N}$.

Definition 2.1.1 The set of well-formed formulae, $\mathsf{WFF}_{\mathsf{K}_n}$, is the least set such that:

1. $p \in \mathsf{WFF}_{\mathsf{K}_n}$, for all $p \in \mathcal{P}$
2. if $\varphi, \psi \in \mathsf{WFF}_{\mathsf{K}_n}$, then so are $\neg\varphi, (\varphi \vee \psi), (\varphi \wedge \psi), (\varphi \Rightarrow \psi), (\varphi \Leftrightarrow \psi), \diamondsuit_a\varphi$ and $\boxed{a}\varphi$, for each $a \in \mathcal{A}$
3. $\mathbf{false}, \mathbf{true} \in \mathsf{WFF}_{\mathsf{K}_n}$

We have chosen to define all connectives as primitive instead of using a minimal set of completely expressive set of connectives, as most of those connectives occur in the normal form we introduce in Section 3.2. The usual abbreviations are listed below. The logical constants **false** and **true** have the same semantic value under every interpretation. They are better known as abbreviation to the more complex formulae:

- $\mathbf{false} = \varphi \wedge \neg\varphi$ (*falsum*)

- **true** $= \neg$**false** (*verum*)

The operator $\diamondsuit$ is the dual of $\boxed{a}$, for each $a \in \mathcal{A}$, that is, $\diamondsuit\varphi$ can also be seen as an abbreviation for $\neg\boxed{a}\neg\varphi$, with $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$. The usual abbreviations for classical connectives are:

- $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$
- $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$
- $\varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

In the following, we use these abbreviations to shorten definitions as we can derive the whole set of connectives from a smaller, but still expressive, subset. The use of this subset can reduce the size of a few proofs as they rely on induction on the set of connectives. Furthermore, parentheses may be omitted if the reading is not ambiguous. Additionally, when $n = 1$, we may omit the index in the modal operators, i.e. we just write $\Box\varphi$ and $\Diamond\varphi$, for a well-formed formula φ.

We define a *literal* as a propositional symbol $p \in \mathcal{P}$ or its negation $\neg p$, and denote by $\mathcal{L}$ the set of all literals. A *modal literal* is a formula of the form $\boxed{a}l$ or $\diamondsuit l$, with $l \in \mathcal{L}$ and $a \in \mathcal{A}$. If l is a literal, we call $\neg l$ its complement and say that l and $\neg l$ form, in either order, a *complementary pair*.

The *modal depth* of a formula is recursively defined as follows:

Definition 2.1.2 Let φ and ψ be well-formed formulae. We define the modal depth of a formula as $mdepth : \mathsf{WFF}_{\mathsf{K}_n} \longrightarrow \mathbb{N}$ inductively as:

1. $mdepth(p) = 0$, for $p \in \mathcal{P}$
2. $mdepth(\neg\varphi) = mdepth(\varphi)$
3. $mdepth(\varphi \vee \psi) = \max\{mdepth(\varphi), mdepth(\psi)\}$
4. $mdepth(\boxed{a}\varphi) = mdepth(\varphi) + 1$

This function represents the maximal number of nested modal operators in a formula. For instance, if $\varphi = \Box\Diamond p \vee \Diamond q$, then $mdepth(\varphi) = 2$.

The *modal level* of a formula (or a subformula), that is, the number of modal operators in the scope of which the (sub)formula occurs, is given relative to its position in an *annotated syntactic tree*.

Definition 2.1.3 Let Σ be the alphabet $\{1, 2, .\}$ and Σ^* the set of all finite sequences over Σ. We define $\tau : \mathsf{WFF}_{\mathsf{K}_n} \times \Sigma^* \times \mathbb{N} \longrightarrow \mathscr{P}(\mathsf{WFF}_{\mathsf{K}_n} \times \Sigma^* \times \mathbb{N})$ as the partial function inductively defined as follows:

1. $\tau(p, \lambda, ml) = \{(p, \lambda, ml)\}$
2. $\tau(\neg\varphi, \lambda, ml) = \{(\neg\varphi, \lambda, ml)\} \cup \tau(\varphi, \lambda.1, ml)$
3. $\tau(\boxed{a}\varphi, \lambda, ml) = \{(\boxed{a}\varphi, \lambda, ml)\} \cup \tau(\varphi, \lambda.1, ml + 1)$
4. $\tau(\varphi \vee \psi, \lambda, ml) = \{(\varphi \vee \psi, \lambda, ml)\} \cup \tau(\varphi, \lambda.1, ml) \cup \tau(\psi, \lambda.2, ml)$

with $p \in \mathcal{P}, \lambda \in \Sigma^*, ml \in \mathbb{N}$ and $\varphi, \psi \in \mathsf{WFF}_{\mathsf{K}_n}$.

The function τ applied to $(\varphi, 1, 0)$ returns the annotated syntatic tree for φ, where each node is uniquely identified by a subformula, its path order (or its position) in the tree, and its modal level, defined below.

Definition 2.1.4 Let φ be a formula and let $\tau(\varphi, 1, 0)$ be its annotated syntactic tree. We define the modal level of a subformula φ' of φ at the position λ as $mlevel : \mathsf{WFF}_{\mathsf{K}_n} \times \mathsf{WFF}_{\mathsf{K}_n} \times \Sigma^* \longrightarrow \mathbb{N}$, and if $(\varphi', \lambda, ml) \in \tau(\varphi, 1, 0)$ then $mlevel(\varphi, \varphi', \lambda) = ml$. Otherwise, it is undefined.

Example 2.1.1 If $\varphi = \boxed{a}\boxed{a}(p \vee \boxed{a}p)$, the application of τ to φ results in the following annotated syntatic tree, also illustrated in Figure 2.1. The nodes are rotulated by formulae, their positions are given on the left, and their modal level on the right.

$$
\begin{aligned}
\tau(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0) &= \{(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0), \tau(\boxed{a}(p \vee \boxed{a}p), 1.1, 1)\} \\
&= \{(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0), (\boxed{a}(p \vee \boxed{a}p), 1.1, 1), (\tau(p \vee \boxed{a}p), 1.1.1, 2)\} \\
&= \{(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0), (\boxed{a}(p \vee \boxed{a}p), 1.1, 1), (p \vee \boxed{a}p, 1.1.1, 2), (\tau(p), 1.1.1.1, 2), \\
&\quad (\tau(\boxed{a}p), 1.1.1.2, 2)\} \\
&= \{(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0), (\boxed{a}(p \vee \boxed{a}p), 1.1, 1), (p \vee \boxed{a}p, 1.1.1, 2), (p, 1.1.1.1, 2), \\
&\quad (\boxed{a}p, 1.1.1.2, 2), (\tau(p), 1.1.1.2.1, 3)\} \\
&= \{(\boxed{a}\boxed{a}(p \vee \boxed{a}p), 1, 0), (\boxed{a}(p \vee \boxed{a}p), 1.1, 1), (p \vee \boxed{a}p, 1.1.1, 2), (p, 1.1.1.1, 2), \\
&\quad (\boxed{a}p, 1.1.1.2, 2), (p, 1.1.1.2.1, 3)\}
\end{aligned}
$$

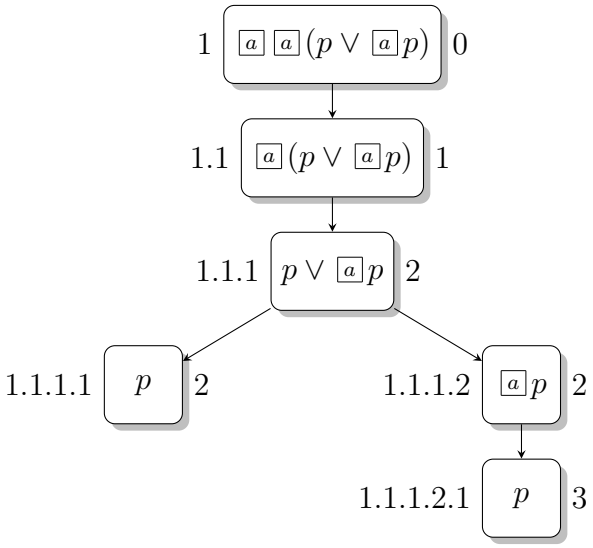


Figure 2.1: Annotated tree for $\boxed{a}\boxed{a}(p \vee \boxed{a}p)$.

2.2 Semantics

The semantics of K_n is presented in terms of Kripke structures.

Definition 2.2.1 A Kripke model for the set of propositional variables $\mathcal{P}$ and the finite set of agents $\mathcal{A} = \{1, \ldots, n\}$ is given by the tuple

$$\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$$

where W is a non-empty set of possible worlds; each R_a, $a \in \mathcal{A}$, is a binary relation on W, that is, $R_a \subseteq W \times W$, and $\pi : W \times \mathcal{P} \longrightarrow \{\mathit{false}, \mathit{true}\}$ is the valuation function that associates to each world $w \in W$ a truth-assignment to propositional symbols.

As a remark on our notation, the truth values are written in italic (e.g. *false*) whereas the constants are written in bold (e.g. **false**).

We write $R_a wv$ to denote that v is accessible from w through the accessibility relation R_a, that is $(w, v) \in R_a$, and $R_a^+ wv$, to mean that v is reachable from w through a finite number of steps, that is, there exists a sequence $(w_1, \ldots, w_k)$ of worlds such that $R_a w_i w_{i+1}$, for all i, $1 \leq i < k$, where $w_1 = w$ and $w_k = v$, with $a \in \mathcal{A}$, $w, v, w_i \in W$ and $i, k \in \mathbb{N}$. Note that R_a^+ is the *transitive closure* of R_a, the least transitive relation that contains all elements of R_a.

Satisfiability and *validity* of a formula are defined in terms of the *satisfiability relation*.

Definition 2.2.2 Let $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ be a Kripke model, $w \in W$ and $\varphi, \psi \in \mathsf{WFF}_{\mathsf{K}_n}$. The *satisfiability relation*, denoted by $\langle \mathfrak{M}, w \rangle \models \varphi$, between a world w and a formula φ, is inductively defined by:

1. $\langle \mathfrak{M}, w \rangle \models p$ if, and only if, $\pi(w, p) = \mathit{true}$, for $p \in \mathcal{P}$;
2. $\langle \mathfrak{M}, w \rangle \models \neg\varphi$ if, and only if, $\langle \mathfrak{M}, w \rangle \not\models \varphi$;
3. $\langle \mathfrak{M}, w \rangle \models \varphi \vee \psi$ if, and only if, $\langle \mathfrak{M}, w \rangle \models \varphi$ or $\langle \mathfrak{M}, w \rangle \models \psi$;
4. $\langle \mathfrak{M}, w \rangle \models \boxed{a}\varphi$ if, and only if, for all $t \in W$, $(w, t) \in R_a$ implies $\langle \mathfrak{M}, t \rangle \models \varphi$.

We can just write $w \models \varphi$ to denote that w satisfies φ when the model is clear from the context. Notice that the evaluation of the $\boxed{a}$ operator quantifies over all reachable worlds in the Kripke structure, characterising it as universal. The dual operator, $\diamondsuit_a$, therefore has an existential characterisation, forcing a world satisfying the modal literal to exist.

A formula $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$ is said to be *locally satisfiable* if there exists a Kripke model $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ such that $\langle \mathfrak{M}, w \rangle \models \varphi$ for some $w \in W$. In this case we simply write $\mathfrak{M} \models_L \varphi$ to mean that $\mathfrak{M}$ locally satisfies φ. A model $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ is said to *globally satisfy* a formula φ, denoted $\mathfrak{M} \models_G \varphi$, if for all $w \in W$, we have $\langle \mathfrak{M}, w \rangle \models \varphi$. A formula φ is said to be *globally satisfiable* if there is a model $\mathfrak{M}$ such that $\mathfrak{M}$ globally satisfies φ. We say that a set $\mathcal{F}$ of formulae is locally satisfiable if the conjunction of every $\varphi \in \mathcal{F}$ is locally satisfiable. Global satisfiability of sets is defined analogously. A formula is said to be *valid* if it is satisfiable in all models.

The *local satisfiability problem* for K_n corresponds to determining the existence of a model in which a formula is locally satisfied, while the *global satisfiability problem* corresponds to determining the existence of a model in which a formula is globally satisfied.

Example 2.2.1 Let $\mathfrak{M}$ be the model illustrated in Figure 2.2. Take $\mathfrak{M} = (W, R, \pi)$, for $p, q \in \mathcal{P}$ and $\mathcal{A} = \{1\}$, where

(*i*) $W = \{w_0, w_1, w_2\}$

(*ii*) $R = \{(w_1, w_1), (w_2, w_2), (w_0, w_1), (w_0, w_2)\}$

(*iii*) $\pi(w, p) = \begin{cases} \mathit{true} & \text{if } w = w_1 \\ \mathit{false} & \text{otherwise} \end{cases}$

(*iv*) $\pi(w, q) = \mathit{true}$ for all $w \in W$

Note that w_0 satisfies both $\Diamond p$ and $\Diamond \neg p$ in $\mathfrak{M}$. This is a rather simple example to illustrate that, even though some sentence evaluates to true at some world, one can see the same sentence occurring with the opposite valuation through an accessibility relation. This kind of reasoning is not possible in classical propositional logic. Other examples of formulae satisfied by this model at w_0 are: $q \vee p$ and $\Diamond \Box p \wedge \Diamond \Box \neg p$. Also note that q is globally satisfied in $\mathfrak{M}$ since it is true at every world of the model.

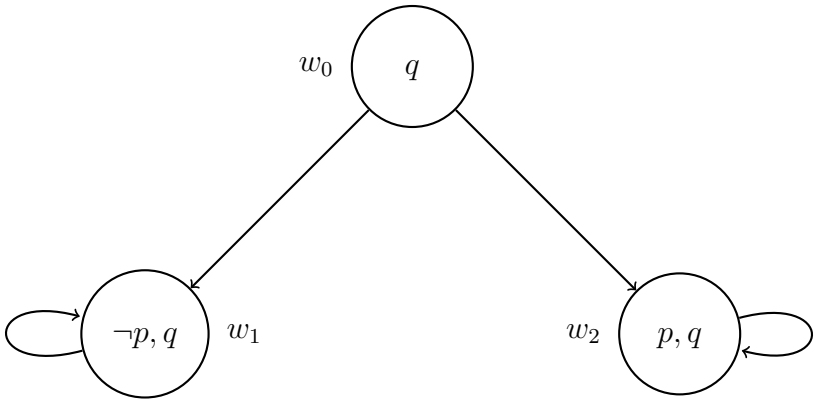


Figure 2.2: Example of a Kripke model for K_n

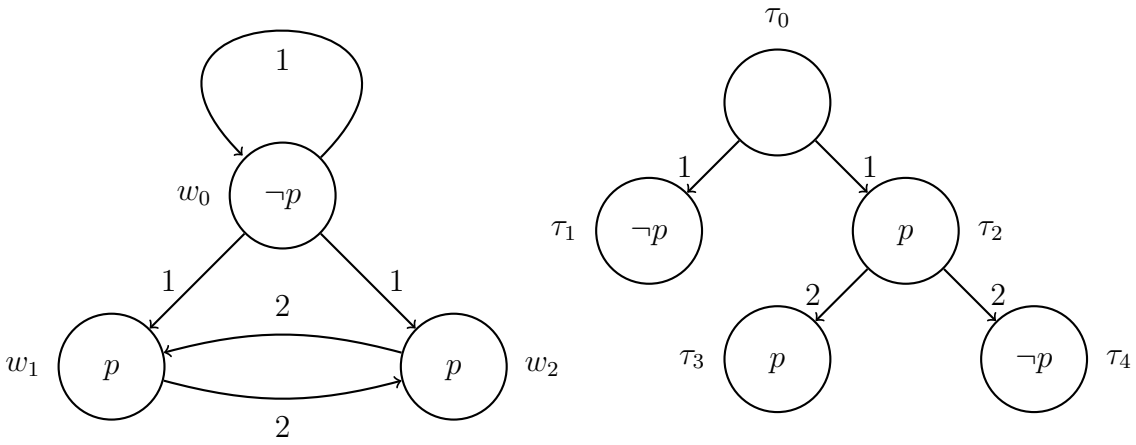
Example 2.2.2 (*Tree-like* model) Consider the formula $\varphi = \boxed{1}(p \Rightarrow \Diamond p)$. The Figure 2.3 contains two examples of models, Figure 2.3a and Figure 2.3b, that satisfy φ at w_0 and τ_0, respectively, hence, φ is locally satisfiable. Note that the model from the Figure 2.3b has a graphical representation equivalent to a tree.

As trees play an important role in future definitions, we will take this opportunity to define them formally. By a tree $\mathfrak{T}$ we mean a relational structure (T, S) where T is a set of nodes and S is a binary relation over these nodes. T contains a unique node $r_0 \in T$ (called the *root*) such that all other nodes in T are reachable from r_0, that is, for all $t \in T$, we have $S^+ r_0 t$. Also, every element of T, distinct from r_0, has a unique S-predecessor, and S is acyclic, that is, for all $t \in T$, it is not the case that $S^+ tt$ [3].

Figure 2.3b is a *tree model*. By a tree model we mean a Kripke model (W, R, π), with $\mathcal{A} = \{1\}$, where (W, R) is a tree with a distinguish world $w_0 \in W$ as its root. A *tree-like model* for K_n is a model $(W, R_1, \ldots, R_n, \pi)$, with $\mathcal{A} = \{1, \ldots, n\}$, such that $(W, \cup_{i \in \mathcal{A}} R_i)$ is a tree, with $w_0 \in W$ as the root.

Let $\mathfrak{M} = (W, R_1, \ldots, R_a, \pi)$ be a tree-like model for K_n with $w_0 \in W$ as a root. We define the *depth* : $W \longrightarrow \mathbb{N}$ of a world $w \in W$, as the length of the path from w_0 to w through the union of the relations in $\mathfrak{M}$. We sometimes say *depth* of $\mathfrak{M}$ to mean the longest path from the root to any world in W.

The following theorems have been adapted from the ones presented in [3].



(a) Kripke model (b) Tree model

Figure 2.3: Models that satisfy φ of Example 2.2.2

Theorem 2.2.1. *Let $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$ be a formula and $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ be a model. Then $\mathfrak{M} \models_L \varphi$ if and only if there is a tree-like model $\mathfrak{M}'$ such that $\mathfrak{M}' \models_L \varphi$. Moreover, $\mathfrak{M}'$ is finite and its depth is bounded by $mdepth(\varphi)$.*

The proof of Theorem 2.2.1 presented in [15] constructs a tree-like model $\mathfrak{M}'$ as a *generated submodel* of $\mathfrak{M}$, that is, it restricts both the set of worlds and the binary relations to a subset of W which is relevant to determining the satisfiability of a given formula. Then, using the satisfiability invariance of generated submodels, also proved in [15], which states that a formula is satisfiable in a model if, and only if, it is satisfiable in its generated submodels, the proof becomes trivial.

Theorem 2.2.2. *Let $\varphi, \varphi' \in \mathsf{WFF}_{\mathsf{K}_n}$ and $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ be a tree-like model such that $\mathfrak{M} \models_L \varphi$. If $(\varphi', \lambda', ml) \in \tau(\varphi, 1, 0)$ and φ' is satisfied in $\mathfrak{M}$, then there is $w \in W$, with $depth(w) = ml$, such that $\langle \mathfrak{M}, w \rangle \models \varphi'$. Moreover, the subtree rooted at w has height equal to $mdepth(\varphi')$.*

The proof of Theorem 2.2.2 is by induction on the structure of the formula and shows that a subformula φ' of φ is satisfied at a node with distance ml of the root of the tree-like model. As determining the satisfiability of a formula φ depends only on its subformulae φ', only the subtrees of height $mdepth(\varphi')$ starting at level ml need to be checked. The bound on the height of the subtrees follows from Theorem 2.2.1 [60].

The global satisfiability problem for a modal logic is equivalent to the local satisfiability problem of the logic obtained by adding the universal modality, $\boxed{*}$, to the original modal language [36]. Let K_n^* be the logic obtained by adding $\boxed{*}$ to K_n. A model $\mathfrak{M}^*$ for K_n^* is the pair $(\mathfrak{M}, R_*)$, where $\mathfrak{M} = (W, R_1, \ldots, R_n, \pi)$ is a tree-like model for K_n and

$R_* = W \times W$. A formula $\boxed{*}\varphi$ is satisfied at the world $w \in W$, in the model $\mathfrak{M}^*$, written $\langle \mathfrak{M}^*, w \rangle \models \boxed{*}\varphi$, if, and only if, for all w' such that $(w, w') \in R_*$, we have that $\langle \mathfrak{M}^*, w' \rangle \models \varphi$. As the relation R_* is total, it is easy to see that this corresponds to the global satisfiability problem as defined before. In [36], the following result is established and proved, for first-order definable modal logics.

Theorem 2.2.3. *$\mathfrak{M} \models_G \varphi$, if, and only if, $\mathfrak{M}^* \models_L \boxed{*}\varphi$, for a formula φ.*

The satisfiability problems for K_n are proven to be PSPACE-complete [74], for local satisfiability, and EXPTIME-complete, for global satisfiability [74]. These might seem as discouraging results and one may wonder whether automated computation for such logics can be viable in practice. However, the kinds of formulae that give rise to the worst case scenarios seem to be hardly encountered in real applications [41]. This has allowed for the successful utilisation of modal logics in, for instance, multiagent systems [16, 32, 64] and distributed systems [37, 38], due to the implementation of automated reasoning systems that perform very well for these logics.

Chapter 3

Modal-Layered Resolution

Modal languages, in general, can be used in Computer Science to represent properties of complex systems. Once a system is described by a logical language, we also want to *reason* about its properties and we do this through a *proof method*, also called *proof system* or *calculus*. This chapter formally introduces a calculus for the modal language defined on Chapter 2. But first, we will go through some basic notions about proof systems.

The *proof* of theorems, or the *deduction* of consequences from assumptions, in Mathematics, is typically characterised by a conclusion that follows from a (possibly empty) set of formulae [44].

Formally, a proof is a finite object built in conformity to a fixed set of syntactic rules [33]. The set of syntactic rules that define proofs, called *inference rules*, are said to specify a proof system. These rules allow the derivation of formulae from sets of formulae through strict symbol manipulation [31]. Besides the set of syntactic rules, a proof method may also be characterised by a set of statements that are assumed to be true, called *axioms*. Axioms may serve as premises or starting points for further reasoning.

A proof system for a particular logic is said to be *sound* if any formula that has a proof is a valid formula of the logic, and it is *complete* for a particular logic if any valid formula has a proof [33]. Therefore, a sound and complete calculus allows us to produce a proof for a formula if and only if this formula is valid.

There are many kinds of proof systems. One way of categorising proof systems involves the chaining in which the rules are applied to form a line of reasoning. If the chaining starts from a set of conditions and moves toward some (possibly remote) conclusion, the system is called *forward chaining*. *Backward* systems, on the other hand, start from what is needed to be proved and moves towards the axioms. Proof search by backward chaining is a standard technique in automated reasoning [39].

One of the most common examples of a backward system is a *tableau system*. Tableau systems exist in many varieties and for several logics, but they all share a few aspects [33].

For example, they are all *refutational systems*, i.e., to prove a formula, we begin by negating it, then analysing the consequences of doing so. If it is the case that all the consequences turn out to be contradictory, we conclude that the original formula has been proved [33]. Refutational systems are based on the fact that a formula is valid if, and only if, its negation is unsatisfiable.

Other aspect common to tableau systems is that they are *analytic*, that is, they usually perform the search for a proof trough breaking down (or branching) the initial formula, according to certain inference rules, until a rule for closing each branch is used. Hence, proofs in tableau systems in general have a graphical representation similar to a tree [23]. Each branch of a tableau proof can be considered to be a partial description of a model on the respective logic. A tableau proof system for modal logics usually labels every step of the proof with a prefix that names a possible world[1] in a model [33]. Therefore, to deal with modal reasoning, whenever a formula $\diamondsuit\varphi$ holds in a specific world labelled by a prefix, the tableau proof creates a new world, whose accessibility from the previous one is made explicit trough the new prefix, that satisfies the formula φ in the scope of the modal operator. Then, following closely the semantics of the $\boxed{a}$ operator, if $\boxed{a}\varphi'$ has the same prefix of $\diamondsuit\varphi$, the formula φ' is *propagated* to the newly created world. Whenever we insert a contradiction in any world, we close the respective branch. If all branches happen to be closed, we can conclude that the original negated formula is unsatisfiable, hence, the original formula is valid.

Aside from tableaux, backward systems may also take the form of a version of Robinson's *resolution rule* [39]. Calculi based on this rule are often referred as *resolution based systems*. These are also refutational systems.

Numerous resolution based systems for propositional logic have been proposed in the literature [10]. The standard system has only one inference rule [66], showed below as the RES rule, which takes two *clauses* with literals that form a complementary pair and generates a *resolvent*, where each clause, denoted by $\mathcal{C}$, is a disjunction of literals. The resolution rule takes, then, the form:

$$[\text{RES}]\ \frac{\mathcal{C}_i \vee l \qquad \neg l \vee \mathcal{C}_j}{\mathcal{C}_i \vee \mathcal{C}_j}$$

where $\mathcal{C}_i$ and $\mathcal{C}_j$ are (possibly empty) clauses, called *parent clauses*, l is a literal and $\mathcal{C}_i \vee \mathcal{C}_j$ is the *resolvent*.

[1]Here, we regard a *world* as a *set of formulae*, following the correspondence given by the syntactic construction of a tableau proof and the construction of a model, as given in the completeness proofs for such proof methods.

We use the constant **false** to denote the *empty clause*, i.e., the clause that contains no literals. Due to the associativity and commutativity properties of disjunction, one may see a clause as a set of literals. In this work we might abuse of set notation, writing $l \in \mathcal{C}$, when l is a literal of $\mathcal{C}$, and $\mathcal{C}_i \subseteq \mathcal{C}_j$, when all literals in $\mathcal{C}_i$ are also literals in $\mathcal{C}_j$.

Resolution-based provers are widely implemented and tested [10]. Besides reliable implementations of the main mechanism for propositional logic, presented in Section 3.1, we can also profit from several complete strategies. For instance, *linear resolution* [45, 50], *set of support strategy* [78], *ordered resolution* [65] and *selection-based resolution* [45]. All those strategies restrict the clauses that can be chosen as premises for the application of the resolution rule whilst retaining completeness. In other words, those strategies try to avoid unnecessary applications of the rule in order to find a proof. Provers for multimodal logics can take advantage of such strategies, as they require pruning of the search space for a proof, in order to deal with the inherent intractability of the satisfiability problem for such logics.

The proof method we will discuss in this chapter is a resolution-based calculus, it was proposed in [60] for both local and global reasoning, and it is thoroughly presented in Section 3.2. The calculus is proved to be sound and complete for K_n in [60]. It was developed with special concerning towards computer implementations and it uses formulae translated into a special normal form. This normal form divides the clause set according to the modal level at which each clause occurs to prevent unecessary applications of inference rules.

3.1 Clausal Resolution

Resolution appeared in the early 1960's through investigations on performance improvements of refutational systems based on the *Herbrand's Theorem*, which allows a kind of reduction of first-order logic to propositional logic [18]. J. A. Robinson incorporated the concept of unification on demand on a refutational system, creating what is now known as the resolution method for first-order logic [66].

Clausal resolution is a simple and adaptable proof system for classical logics. It was claimed to be suitable to be performed by a computer, as, for propositional logic, it has only one inference rule that is applied to the set of clauses. Robinson emphasises that, from the theoretical point of view, an inference rule need only to be sound and effective, that is, it allows only valid consequences to be deduced from premises and it must be algorithmically decidable whether an alleged application of the rule is indeed an application of it.

The single inference rule of this system of logic, the RES rule, entails the *resolution principle*, namely: *From any two clauses $\mathcal{C}_i$ and $\mathcal{C}_j$ which contain a complementary pair of literals, one may infer a resolvent of $\mathcal{C}_i$ and $\mathcal{C}_j$* [66].

Resolution, as proposed by Robinson, is a refutationally complete theorem proving method for first-order logic [66]. In his paper, Robinson presents a formalisation of the calculus for first-order logic, designed to be used as the basic theoretical instrument of the proposed computer theorem-proving program. For the purpose of this work, we are only interested in the calculus for propositional logic. Therefore, we will neither discuss the theory behind the Herbrand's Theorem nor the definition of the unification procedure, but, if curious, the reader can refer to [66] for more details. In the sense of this last remark, the only representational formalism needed is propositional logic.

Resolution relies on saturation. A clause set is said to be *saturated* when further relevant information cannot be generated, regardless of the rule (or set of rules) that is applied [25]. That means that the application of any rule to a saturated set of clauses only generates tautologies or repeated clauses. Saturation, up to redundancy, of the initial clause set is a quite useful criterion when we are thinking in terms of the termination proof and completeness proof of a calculus [33].

As resolution has only the RES rule, to show that a formula φ is valid, $\neg\varphi$ is translated into a normal form, then the inference rule is applied until either no new resolvents can be generated or the empty clause is obtained [10]. The contradiction implies that $\neg\varphi$ is unsatisfiable and hence, that φ is valid.

Example 3.1.1 Consider the set of clauses $\mathcal{S} = \{(\neg p \vee q), (\neg p \vee r), (\neg r \vee q), (\neg q \vee r), \neg r, p\}$. Figure 3.1, taken from [57], shows an example of naive derivation of the empty clause for $\mathcal{S}$, proving that this set is unsatisfiable.

Despite its simplicity, naive resolution is hard to implement efficiently due to the difficulty of finding good choices of clauses to resolve [35]. As one can infer from this example, a raw implementation of resolution-based systems typically implies considerable storage requirements. Even with such a short set of clauses (only 6 initially), the application of the resolution rule, without any restriction, yields the generation of repeated clauses, Clauses 7 and 8, and tautologies, Clauses 9 and 10. That is, useless information is generated and possibly stored until the contradiction can be found.

Robinson established two principles, namely *purity principle* and *subsumption principle*, to discuss the development of efficient resolution systems. Such principles are called *search principles* in [66]. A third principle, named *replacement principle*, is presented in

1.	$\neg p \vee q$		
2.	$\neg p \vee r$		
3.	$\neg r \vee q$		
4.	$\neg q \vee r$		
5.	$\neg r$		
6.	p		
7.	$\neg p \vee q$	[RES,3,2]	$= 1$
8.	$\neg p \vee r$	[RES,4,1]	$= 2$
9.	$\neg q \vee q$	[RES,4,3]	$=$ **true**
10.	$\neg r \vee r$	[RES,4,3]	$=$ **true**
11.	$\neg p$	[RES,5,2]	
12.	$\neg q$	[RES,5,4]	
13.	q	[RES,6,1]	
14.	r	[RES,6,2]	
15.	**false**	[RES,11,6]	contradiction

Figure 3.1: An example of derivation for the set $\mathcal{S}$ in Example 3.1.1

terms of replacement using the subsumption principle.

Definition 3.1.1 If $\mathcal{S}$ is any finite set of clauses, $\mathcal{C}$ is a clause in $\mathcal{S}$ and l a literal in $\mathcal{C}$ with the property that no literal in any other clause in $\mathcal{S}$ form a complementary pair with l, then l is said to be *pure* in $\mathcal{S}$.

The purity principle is then based on Theorem 3.1.1.

Theorem 3.1.1. *If $\mathcal{S}$ is any finite set of clauses, and a clause $\mathcal{C} \in \mathcal{S}$ contains a literal l which occurs pure in $\mathcal{S}$, then $\mathcal{S}$ is satisfiable if and only if $\mathcal{S} \setminus \{\mathcal{C}\}$ is satisfiable.*

Definition 3.1.2 If $\mathcal{C}_i$ and $\mathcal{C}_j$ are two distinct nonempty clauses, we say that $\mathcal{C}_i$ *subsumes* $\mathcal{C}_j$ in the case that $\mathcal{C}_i \subseteq \mathcal{C}_j$.

Theorem 3.1.2 establishes the basic property of subsumption.

Theorem 3.1.2. *If $\mathcal{S}$ is any finite set of clauses, and $\mathcal{C}_j$ is any clause in $\mathcal{S}$ which is subsumed by some clause in $\mathcal{S} \setminus \{\mathcal{C}_j\}$, then $\mathcal{S}$ is satisfiable if and only if $\mathcal{S} \setminus \{\mathcal{C}_j\}$ is satisfiable.*

Theorems 3.1.1 and 3.1.2 are both proved in [66].

Another specially useful search principle derives from the subsumption principle. Suppose that a resolvent $\mathcal{C}_R$ of $\mathcal{C}_i$ and $\mathcal{C}_j$ subsumes $\mathcal{C}_i$. Then, in adding $\mathcal{C}_R$ as a result of resolving $\mathcal{C}_i$ and $\mathcal{C}_j$, we may simultaneously remove, by the subsumption principle, $\mathcal{C}_i$.

(a) Initial clauses

1. $\neg q \vee \neg r$
2. $\neg q \vee r$
3. $u \vee \neg p$
4. $\neg p \vee q \vee \neg r$
5. $\neg p \vee q$
6. p

(b) Purity principle

1. $\neg q \vee \neg r$
2. $\neg q \vee r$
4. $\neg p \vee q \vee \neg r$
5. $\neg p \vee q$
6. p

(c) Subsumption

1. $\neg q \vee \neg r$
2. $\neg q \vee r$
5. $\neg p \vee q$
6. p

(d) Replacement

5. $\neg p \vee q$
6. p
7. $\neg q$ [RES,1,2]
8. q [RES,5,6]
9. **false** [RES,7,8]

Figure 3.2: Derivation schemes for Example 3.1.2

This combined operation leads to the replacement of $\mathcal{C}_i$ by $\mathcal{C}_R$. Accordingly, this principle is named as the replacement principle [66].

The application, to a finite set $\mathcal{S}$ of clauses, of any of the three search principles described, produces a set $\mathcal{S}'$ which either has fewer clauses than $\mathcal{S}$, or has one or more shorter clauses [66]. An evident way of profit from these principles in a refutation procedure is therefore to delay the application of the RES rule to the set of clauses, until the search principles are no longer fulfilled.

Example 3.1.2 Let $\mathcal{S}$ be the set of clauses showed in Figure 3.2a. This time, prior to the exaustive application of the RES rule, let's investigate opportunities to apply the search principles, in order to reduce the number and size of clauses.

In this example, the third clause can be eliminated from the set of clauses through an application of the purity principle, as expressed in Figure 3.2b, because there is no other clause that has the complementary pair of the literal u. Furthermore, as the Clause 5 subsumes the Clause 4, this last one can also be eliminated from the set of clauses, leaving only the clauses in Figure 3.2c. The application of RES to Clauses 1 and 2 generates the resolvent $\neg q$, which subsumes both clauses, hence, by the replacement principle, both may be replaced by Clause 7, as showed in Figure 3.2d. Finally, the application of RES to Clauses 5 and 6, generates the Clause 8 as a resolvent, and then resolving this one with the Clause 7 will generate the empty clause, a contradiction.

There are further principles of the same general sort, possibly less simple than the ones presented earlier, which Robinson considers to be merely a brief view of the possible approaches to the efficiency problem of resolution systems. Details can be found in [66]. Further on this work, we see a practical application for the purity principle, but we refer to it as *pure literal elimination* following more recent literature. We also present *unit propagation*, which can be thought as another search principle. Both will be brought back in Chapter 4, where we discuss the role of SAT Solvers for reasoning in propositional logic.

3.2 Modal-Layered Resolution Calculus for K_n

With the resolution ground for propositional logic all set, now we will present a resolution-based proof system for K_n, the modal language discussed in Chapter 2.

The calculus presented in this section requires a translation into a more expressive modal language, presented in Section 3.2.1, where labels are used to express semantic properties of a formula. This calculus uses labelled resolution in order to avoid unnecessary applications of the inference rules [60]. For instance, we do not apply resolution to clauses at different modal levels, since, as we have seen, they are not, in fact, contradictory.

3.2.1 Separated Normal Form with Modal Levels

Formulae translated into a normal form have a specific, normalised structure, possibly resulting in less operators to handle with, which may imply in a smaller number of rules for a proof system. Hence, a calculus that is planned to be implemented in a computer may take great advantage of normal forms, since the smaller number of inference rules reduces the chances of implementation errors, for example.

The normal form used in this work for K_n is a layered normal form called *Separated Normal Form with Modal Levels*, denoted by SNF_{ml}, proposed in [60], hence, all the definitions in this section are adapted from [60]. A formula in SNF_{ml} is a conjunction of *clauses* where the modal level in which they occur is made explicit in the syntax as a label.

We write $ml : \varphi$ to denote that φ occurs at modal level $ml \in \mathbb{N} \cup \{*\}$. By $* : \varphi$ we mean that φ is true at all modal levels. Formally, let $\mathrm{WFF}^{ml}_{\mathsf{K}_n}$ denote the set of formulae with the modal level annotation, $ml : \varphi$, such that $ml \in \mathbb{N} \cup \{*\}$ and $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$. Let $\mathfrak{M}^* = (W, w_0, R_1, \ldots, R_n, R_*, \pi)$ be a tree-like model and take $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$.

Definition 3.2.1 Satisfiability of labelled formulae is given by:

1. $\mathfrak{M}^* \models_L ml : \varphi$ if, and only if, for all worlds $w \in W$ such that $depth(w) = ml$, we have $\langle \mathfrak{M}^*, w \rangle \models \varphi$;
2. $\mathfrak{M}^* \models_L * : \varphi$ if, and only if, $\mathfrak{M}^* \models \boxed{*}\varphi$.

Clauses in SNF_{ml} are defined as follows.

Definition 3.2.2 Clauses in SNF_{ml} are in one of the following forms:

1. Literal clause $\quad ml : \bigvee_{i=1}^{r} l_i$

2. Positive a-clause $\quad ml : l \Rightarrow \boxed{a}m$

3. Negative a-clause $\quad ml : l \Rightarrow \diamondsuit\!\!\!\!\!\!\diamondsuit_a m$

where $r, i \in \mathbb{N}, ml \in \mathbb{N} \cup \{*\}$, $l, l_i, m \in \mathcal{L}$ and $a \in \mathcal{A} = \{1, \ldots, n\}$.

Positive and negative a-clauses are together known as *modal a-clauses*. The index a can be omitted if it is clear from the context.

Definition 3.2.3 Let $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$. We say that φ is in *Negation Normal Form* (NNF) if it contains only the operators $\neg, \vee, \wedge, \boxed{a}$ and $\diamondsuit\!\!\!\!\!\!\diamondsuit_a$. Also, only propositions are allowed in the scope of negations.

The transformation of a formula $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$ into SNF_{ml} is achieved by first transforming φ into its Negation Normal Form, and then, recursively applying rewriting and renaming of complex formulae [63], until all subformulae are written in one of the forms presented by Definition 3.2.2.

Formally, let φ be a formula in NNF. Take t as a propositional symbol not occurring in φ. The translation of φ into SNF_{ml} is then given by $0 : t \wedge \rho(0 : t \Rightarrow \varphi)$ for local satisfiability — for global satisfiability, the translation is given by $* : t \wedge \rho(* : t \Rightarrow \varphi)$ — where ρ is the *translation function* defined by Definition 3.2.4. We refer as *initial clauses* the literals clauses at the modal level 0.

Definition 3.2.4 The translation function $\rho : \mathrm{WFF}^{ml}_{\mathsf{K}_n} \longrightarrow \mathrm{WFF}^{ml}_{\mathsf{K}_n}$ is defined as follows:

$$
\begin{aligned}
\rho(ml : t \Rightarrow \varphi \wedge \psi) &= \rho(ml : t \Rightarrow \varphi) \wedge \rho(ml : t \Rightarrow \psi) \\
\rho(ml : t \Rightarrow \boxed{a}\varphi) &= (ml : t \Rightarrow \boxed{a}\varphi), \text{ if } \varphi \text{ is a literal} \\
&= (ml : t \Rightarrow \boxed{a}t') \wedge \rho(ml + 1 : t' \Rightarrow \varphi), \text{ otherwise} \\
\rho(ml : t \Rightarrow \diamondsuit\!\!\!\!\!\!\diamondsuit_a\varphi) &= (ml : t \Rightarrow \diamondsuit\!\!\!\!\!\!\diamondsuit_a\varphi), \text{ if } \varphi \text{ is a literal} \\
&= (ml : t \Rightarrow \diamondsuit\!\!\!\!\!\!\diamondsuit_a t') \wedge \rho(ml + 1 : t' \Rightarrow \varphi), \text{ otherwise} \\
\rho(ml : t \Rightarrow \varphi \vee \psi) &= (ml : \neg t \vee \varphi \vee \psi), \text{ if } \varphi \vee \psi \text{ is a disjunction of literals} \\
&= \rho(ml : t \Rightarrow \varphi \vee t') \wedge \rho(ml : t' \Rightarrow \psi), \\
&\quad \text{if } \psi \text{ is not a a disjunction of literals}
\end{aligned}
$$

Where $t, t' \in \mathcal{L}$, $\varphi, \psi \in \mathsf{WFF}_{\mathsf{K}_n}$, $a \in \mathcal{A}$ and $ml \in \mathbb{N} \cup \{*\}$, with $* + 1 = *$. Also, whenever t' renames a subformula φ, it is required that t' is a new propositional symbol (i.e. it does not occur in the original formula).

As the conjunction operator is commutative, associative and idempotent, we will commonly refer to a formula in SNF_{ml} as a set of clauses.

Example 3.2.1 Let φ be the formula $\Box(a \Rightarrow b) \Rightarrow (\Box a \Rightarrow \Box b)$. We show how to translate φ into its normal form, considering the local satisfiability problem.

First, we anchor the NNF of φ to the initial state:

$$0 : t_0 \wedge \rho(0 : t_0 \Rightarrow \Diamond(a \wedge \neg b) \vee \Diamond \neg a \vee \Box b) \tag{3.1}$$

The Equation 3.1 is used to anchor the meaning of φ to the initial state, where the formula is evaluated. The function ρ proceeds with the translation, replacing complex formulae inside the scope of the $\Box$ and $\Diamond$ operators, by means of renaming.

So the translation proceeds as follows:

$$\begin{aligned}
&\rho(0 : t_0 \Rightarrow \Diamond(a \wedge \neg b) \vee \Diamond \neg a \vee \Box b) \\
&= \rho(0 : t_0 \Rightarrow \Diamond(a \wedge \neg b) \vee t_1) \wedge \rho(0 : t_1 \Rightarrow \Diamond \neg a \vee \Box b) \\
&= \rho(0 : t_0 \Rightarrow t_2 \vee t_1) \wedge \rho(0 : t_2 \Rightarrow \Diamond(a \wedge \neg b)) \wedge \rho(0 : t_1 \Rightarrow \Diamond \neg a \vee t_3) \wedge \rho(0 : t_3 \Rightarrow \Box b) \\
&= (0 : \neg t_0 \vee t_2 \vee t_1) \wedge (0 : t_2 \Rightarrow \Diamond t_4) \wedge \rho(1 : t_4 \Rightarrow a \wedge \neg b) \wedge \rho(0 : t_1 \Rightarrow t_5 \vee t_3) \wedge \\
&(0 : t_5 \Rightarrow \Diamond \neg a) \wedge (0 : t_3 \Rightarrow \Box b) \\
&= (0 : \neg t_0 \vee t_2 \vee t_1) \wedge (0 : t_2 \Rightarrow \Diamond t_4) \wedge \rho(1 : t_4 \Rightarrow a) \wedge \rho(1 : t_4 \Rightarrow \neg b) \wedge \\
&(0 : \neg t_1 \vee t_5 \vee t_3) \wedge (0 : t_5 \Rightarrow \Diamond \neg a) \wedge (0 : t_3 \Rightarrow \Box b) \\
&= (0 : \neg t_0 \vee t_2 \vee t_1) \wedge (0 : t_2 \Rightarrow \Diamond t_4) \wedge (1 : \neg t_4 \vee a) \wedge (1 : \neg t_4 \vee \neg b) \wedge \\
&(0 : \neg t_1 \vee t_5 \vee t_3) \wedge (0 : t_5 \Rightarrow \Diamond \neg a) \wedge (0 : t_3 \Rightarrow \Box b)
\end{aligned}$$

The translation generates eight clauses, two of them at the subsequent modal level ($1 : \neg t_4 \vee a$ and $1 : \neg t_4 \vee \neg b$). Note that, from the six clauses at the initial modal level, we have three literal clauses ($0 : t_0$, $0 : \neg t_0 \vee t_2 \vee t_1$ and $0 : \neg t_1 \vee t_5 \vee t_3$), known as the initial clauses, two negative clauses ($0 : t_2 \Rightarrow \Diamond t_4$ and $0 : t_5 \Rightarrow \Diamond \neg a$) and one positive clause ($0 : t_3 \Rightarrow \Box b$).

The set of clauses obtained by translating the formula φ into SNF_{ml} is locally satisfiable if, and only if, φ is locally satisfiable, as given by the next lemma, taken from [60].

Lemma 3.2.1. *Let $\varphi \in \mathsf{WFF}_{\mathsf{K}_n}$ be a formula and let t be a propositional symbol not occurring in φ. Then:*

(i) φ is locally satisfiable if, and only if, $0 : t \wedge \rho(0 : t \Rightarrow \varphi)$ is satisfiable;

(ii) φ is globally satisfiable if, and only if, $* : t \wedge \rho(* : t \Rightarrow \varphi)$ is satisfiable.

3.2.2 Calculus

The motivation for the use of the labelled clausal normal form in the calculus is that inference rules can then be guided by the syntactic information given by the labels and applied to smaller sets of clauses, reducing the number of unnecessary applications of the inference rules, and therefore improving the efficiency of the proof procedure [61].

The modal-layered resolution calculus, proposed in [60], comprises a set of inference rules, individually explained below, for dealing with propositional and modal reasoning.

We denote by σ the result of unifying the labels in the premises for each rule. Formally presented in Definition 3.2.5, the unification function imposes restrictions on the application of the inference rules. We have seen, for instance, that contradictory clauses at different modal levels are not contradictory. Hence, a resolvent should not be deduced.

Definition 3.2.5 Unification is given by a function $\sigma : \mathscr{P}(\mathbb{N} \cup \{*\}) \longrightarrow \mathbb{N} \cup \{*\}$, where $\sigma(\{ml, *\}) = ml$ and $\sigma(\{ml\}) = ml$. Otherwise, σ is undefined.

The inference rules showed in Figure 3.3 can only be applied if the unification of their labels is defined (where $* - 1 = *$). Note that for GEN1 and GEN3, if the modal clauses occur at the modal level ml, then the literal clause occurs at the next modal level, $ml + 1$. The rules that deal with modal reasoning are applied only to clauses with modal operators indexed by the same agent.

$$\text{[LRES]}\quad \frac{\begin{array}{rl} ml_1: & D \vee l \\ ml_2: & D' \vee \neg l \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2\}): & D \vee D' \end{array}} \qquad \text{[MRES]}\quad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} m \\ ml_2: & l_2 \Rightarrow \diamondsuit_a \neg m \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2\}): & \neg l_1 \vee \neg l_2 \end{array}}$$

$$\text{[GEN2]}\quad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} m_1 \\ ml_2: & l_2 \Rightarrow \boxed{a} \neg m_1 \\ ml_3: & l_3 \Rightarrow \diamondsuit_a m_2 \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2, ml_3\}): & \neg l_1 \vee \neg l_2 \vee \neg l_3 \end{array}}$$

$$\text{[GEN1]}\quad \frac{\begin{array}{r} ml_1 : l_1 \Rightarrow \boxed{a} \neg m_1 \\ \vdots \\ ml_r : l_r \Rightarrow \boxed{a} \neg m_r \\ ml_{r+1} : l \Rightarrow \diamondsuit_a \neg m \\ ml_{r+2} : m_1 \vee \ldots \vee m_r \vee m \end{array}}{ml : \neg l_1 \vee \ldots \vee \neg l_r \vee \neg l} \quad \text{where } ml = \sigma(\{ml_1, \ldots, ml_{r+1}, ml_{r+2} - 1\})$$

$$\text{[GEN3]}\quad \frac{\begin{array}{r} ml_1 : l_1 \Rightarrow \boxed{a} \neg m_1 \\ \vdots \\ ml_r : l_r \Rightarrow \boxed{a} \neg m_r \\ ml_{r+1} : l \Rightarrow \diamondsuit_a m \\ ml_{r+2} : m_1 \vee \ldots \vee m_r \end{array}}{ml : \neg l_1 \vee \ldots \vee \neg l_r \vee \neg l} \quad \text{where } ml = \sigma(\{ml_1, \ldots, ml_{r+1}, ml_{r+2} - 1\})$$

Figure 3.3: Inference rules

We will shortly add some intuition about each rule. In the following, consider D to be a disjunction of literals, $l_i, m_i \in \mathcal{L}, ml \in \mathbb{N} \cup \{*\}$ and σ the unification function given in Definition 3.2.5.

- *Literal resolution*: The LRES rule is classical resolution which deals with the propositional portion of the logic. This rule is applied whenever we have a complementary pair of literals on clauses at modal levels for which the unification function is defined.

$$[\mathsf{LRES}] \qquad \frac{\begin{array}{rl} ml_1: & D \vee l \\ ml_2: & D' \vee \neg l \end{array}}{\sigma(\{ml_1, ml_2\}): \quad D \vee D'}$$

- *Modal resolution*: The MRES rule resembles classical resolution in the sense that a formula and its negation cannot be true at the same modal level.

$$[\mathsf{MRES}] \qquad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} m \\ ml_2: & l_2 \Rightarrow \diamondsuit \neg m \end{array}}{\sigma(\{ml_1, ml_2\}): \quad \neg l_1 \vee \neg l_2}$$

  The GEN1 rule corresponds to several applications of classical resolution. This rule establishes what happens when we have a contradiction between a literal clause at a certain modal level and a number of positive modal literals plus a negative modal literal, at the preceding level. This contradiction implies that it is not possible to satisfy all these modal literals at once at the appropriate modal level. Hence, the disjunction of the negated left-hand side of all the modal clauses is inferred.

$$[\mathsf{GEN1}] \qquad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} \neg m_1 \\ & \vdots \\ ml_r: & l_r \Rightarrow \boxed{a} \neg m_r \\ ml_{r+1}: & l \Rightarrow \diamondsuit \neg m \\ ml_{r+2}: & m_1 \vee \ldots \vee m_r \vee m \end{array}}{\begin{array}{rl} ml: & \neg l_1 \vee \ldots \vee \neg l_r \vee \neg l \end{array}}$$

$$\text{where } ml = \sigma(\{ml_1, \ldots, ml_{r+1}, ml_{r+2} - 1\})$$

The GEN2 rule is a special case of GEN1, with the parent clauses themselves holding the conflict.

$$\text{[GEN2]}\qquad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} m_1 \\ ml_2: & l_2 \Rightarrow \boxed{a} \neg m_1 \\ ml_3: & l_3 \Rightarrow \diamondsuit_{\alpha} m_2 \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2, ml_3\}): & \neg l_1 \vee \neg l_2 \vee \neg l_3 \end{array}}$$

The inference rule GEN3 is similar to GEN1 but the contradiction occurs only between right-hand side of the positive a-clauses and the literal clause.

$$\text{[GEN3]}\qquad \frac{\begin{array}{rl} ml_1: & l_1 \Rightarrow \boxed{a} \neg m_1 \\ & \vdots \\ ml_r: & l_r \Rightarrow \boxed{a} \neg m_r \\ ml_{r+1}: & l \Rightarrow \diamondsuit_{\alpha} m \\ ml_{r+2}: & m_1 \vee \ldots \vee m_r \end{array}}{\begin{array}{rl} ml: & \neg l_1 \vee \ldots \vee \neg l_r \vee \neg l \end{array}}$$

where $ml = \sigma(\{ml_1, \ldots, ml_{r+1}, ml_{r+2} - 1\})$

Definition 3.2.6 Let $\mathcal{S}$ be a set of clauses in SNF_{ml}. A *derivation* from $\mathcal{S}$ is a sequence of sets $\mathcal{S}_0, \mathcal{S}_1, \ldots$ where $\mathcal{S}_0 = \mathcal{S}$ and, for each $i > 0$, $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{R\}$, where R is the resolvent obtained from $\mathcal{S}_i$ by an application of either LRES, MRES, GEN1, GEN2 or GEN3. A *local refutation* for $\mathcal{S}$ is a finite derivation that contains the empty clause at the initial modal level, that is 0 : **false**. A *global refutation* for $\mathcal{S}$ is a finite derivation that contains the empty clause at any modal level [60].

Each resolvent R is assumed to be in simplified form, that is, it should not contain repeated literals nor constants. It is also required that R does not already occur in the set of clauses. Finally, R cannot be a tautology.

The proofs for termination, soundness and completeness of this calculus can be found in [60]. The completeness proof heavily relies on the fact the binary resolution calculus for propositional logic is consequence complete [48]. Given a set of clauses $\mathcal{F}$, a clause $\mathcal{C}$ is a *prime consequence* of $\mathcal{F}$ if, and only if, $\mathcal{C}$ is implied by $\mathcal{F}$ and there exists no other clause $\mathcal{D}$ implied by $\mathcal{F}$ such that $\mathcal{C}$ is implied by $\mathcal{D}$ [72]. Given a set of clauses $\mathcal{F}$, a calculus is *consequence complete* if any prime consequence $\mathcal{C}$ of $\mathcal{F}$ is derivable.

Lemma 3.2.2 ([48]). *Let $\mathcal{F}$ be a set of propositional clauses. If a clause $\mathcal{C}$ is a prime consequence of $\mathcal{F}$, then there is a clause $\mathcal{D}$ which is derived by binary propositional resolution from $\mathcal{F}$ and $\mathcal{D}$ subsumes $\mathcal{C}$.*

All the details about proof construction, in the case of unsatisfiable set of clauses, and the model construction, in the case of a satisfiable set of clauses can be found, as mentioned, in [60]. Here, we only point out the relevant part of the proof that we will use later in Chapter 5. Let $\mathbb{C}^w_{a,ml}$ in $\mathcal{F}$ be the set of positive a-clauses corresponding to agent a, that is, the clauses of the form $ml : l' \Rightarrow \boxed{a}m'$, where l' and m' are literals, whose left-hand side are satisfied at the world w at modal level ml. Let $POS_{a,ml}$ be the set of literals in the scope of $\boxed{a}$ on the right-hand side from the clauses in $\mathbb{C}^w_{a,ml}$, that is, if $ml : l' \Rightarrow \boxed{a}m' \in \mathbb{C}^w_{a,ml}$, then $m' \in POS_{a,ml}$. Now, from the proof and model constructions for a set of clauses, given a clause $ml : l \Rightarrow \diamondsuit m$, if $w \models l$ and there is no world satisfying m which is a-related to w, then m, $POS_{a,ml}$, and the literal clauses at the level $ml + 1$ must be contradictory. As m alone is not contradictory and because the case where the literal clauses are contradictory by themselves relies on the completeness of resolution, there are five cases to consider for the application of the modal resolution rules:

1. Assume that $POS_{a,ml}$ itself is contradictory. This means there must be clauses of the form $ml : l_1 \Rightarrow \boxed{a}m', ml : l_2 \Rightarrow \boxed{a}\neg m' \in \mathbb{C}^w_{a,ml}$, where $w \models l_1$ and $w \models l_2$. Thus we can apply GEN2 to these clauses and the negative modal clause $ml : l \Rightarrow \diamondsuit m$ deriving $ml : \neg l_1 \vee \neg l_2 \vee \neg l$. Hence the addition of this resolvent means that w is not part of the model construction.

2. Assume that m and $POS_{a,ml}$ is contradictory. Then, $\mathbb{C}^w_{a,ml}$ in $\mathcal{F}$ contains a clause as $ml : l_1 \Rightarrow \boxed{a}\neg m$ where, from the definition of $\mathbb{C}^w_{a,ml}$, $w \models l_1$. Thus, by an application of MRES to this clause and $ml : l \Rightarrow \diamondsuit m$, we derive $ml : \neg l_1 \vee \neg l$ and w cannot be part of the model construction.

3. Assume that m and the literal clauses at the modal level $ml + 1$ are contradictory. By consequence completeness of binary resolution [48], applications of LRES to the set of literal clauses generates $ml + 1 : \neg m$, which can be used together with $ml : l \Rightarrow \diamondsuit m$ to apply GEN1 and generate $ml : \neg l$. This resolvent deletes w from the model construction as required. Note that this is a special case where the set of positive a-clauses in the premise of GEN1 is empty.

4. Assume that $POS_{a,ml}$ and the literal clauses at the modal level $ml+1$ all contribute to the contradiction (but not m), by the results in [48], applications of LRES will generate the relevant clause to which we can apply GEN3 and delete w as required.

Note that this also happens in the special case where the set of positive clauses in the premise of GEN3 is empty. In this case, the literal clause in the premises is of the form $ml : \mathcal{C}$ where $\mathcal{C}$ is the empty disjunction, which by consequence completeness will also be produced during a derivation.

5. Assume that m, $POS_{a,ml}$ and the literal clauses all contribute to the contradiction. Thus, similarly to the above, applying LRES generates the relevant literal clause to which GEN1 can be applied. This means that w is, again, not part of the model construction.

Note that, for GEN1 and GEN3, whenever a set of modal clauses at a given modal level ml is contradicting with the set of literal clauses at the next level, the relevant literal clauses used in the premises are generated by consequence completeness of the binary resolution calculus (which, in our case, corresponds to applications of LRES).

We finish this section by illustrating the application of the resolution calculus.

Example 3.2.2 Consider the set of clauses in SNF_{ml} of Example 3.2.1, generated for $\varphi = \Box(a \Rightarrow b) \Rightarrow (\Box a \Rightarrow \Box b)$.

Figure 3.4 shows a derivation from the eight clauses obtained by the transformation, once no other rule can be applied and no refutation is generated. Thus, as there is no refutation, φ is a satisfiable formula, as expected.

1.	$0 : t_0$	8.	$1 : \neg t_4 \vee \neg b$	
2.	$0 : \neg t_0 \vee t_2 \vee t_1$	9.	$0 : t_2 \vee t_1$	[LRES, 1, 2, t_0]
3.	$0 : \neg t_1 \vee t_5 \vee t_3$	10.	$0 : t_2 \vee t_5 \vee t_3$	[LRES, 9, 3, t_1]
4.	$0 : t_2 \Rightarrow \Diamond t_4$	11.	$0 : \neg t_2$	[GEN1, 4, 7, 8, t_4]
5.	$0 : t_5 \Rightarrow \Diamond \neg a$	12.	$0 : t_5 \vee t_3$	[LRES, 10, 11, t_2]
6.	$0 : t_3 \Rightarrow \Box b$	13.	$0 : \neg t_3$	[GEN3, 6, 5, 8, b, a]
7.	$1 : \neg t_4 \vee a$	14.	$0 : t_5$	[LRES, 13, 14, t_3]

Figure 3.4: Satisfiability proof of the formula φ from Example 3.2.1

Example 3.2.3 Adapted from [2, 60]. Consider the clauses in Figure 3.5. Clauses 1 and 2 state that a dog is either big or small. Clauses 3 and 4, only serving the purpose of illustration, say that good dogs have calm puppies. The particular situation of Toto, denoted here by t_0, is given in the following clauses: Clauses 5, 6 and 7 say that Toto's big puppies are all good. Clauses 8 and 9 say that a grandpuppy of Toto is not a calm dog. We want to prove that Toto has a small puppy, which appears negated in Clause 10. Otherwise, Toto would only have calm grandpuppies. The refutation is given in Figure 3.5.

1.	$* : big \vee small$	9.	$1 : t_3 \Rightarrow \Diamond_{\forall} \neg calm$	
2.	$* : \neg big \vee \neg small$	10.	$0 : t_0 \Rightarrow \boxed{p} \neg small$	
3.	$* : \neg good \vee t_1$	11.	$1 : \neg t_1 \vee \neg t_3$	[MRES, 9, 4, *calm*]
4.	$* : t_1 \Rightarrow \boxed{p}\, calm$	12.	$1 : \neg good \vee \neg t_3$	[LRES, 11, 3, t_1]
5.	$0 : t_0$	13.	$1 : \neg t_3 \vee \neg t_2 \vee \neg big$	[LRES, 7, 12, *good*]
6.	$0 : t_0 \Rightarrow \boxed{p}\, t_2$	14.	$1 : small \vee \neg t_3 \vee \neg t_2$	[LRES, 13, 1, *good*]
7.	$1 : \neg t_2 \vee \neg big \vee good$	15.	$0 : \neg t_0$	[GEN1, 10, 6, 8, 14, $small, t_2, t_3$]
8.	$0 : t_0 \Rightarrow \Diamond_{\forall} t_3$	16.	$0 :$ **false**	[LRES, 15, 5, t_0]

Figure 3.5: An example of a refutation

3.2.3 KSP

In this section, we briefly introduce KSP, the theorem prover presented in [61] for the basic multimodal logic K_n, which implements a variation of the set of support strategy [78] for the modal resolution-based calculus described in Section 3.2. KSP is written in C. The sources, as well as examples of input, benchmarks and instructions for installing and running, are available for download in [62].

KSP was designed to support experimentation with different combinations of refinements of its basic calculus. Refinements and options for preprocessing and processing the input are coded as independently as possible in order to allow for the easy addition and testing of new features. Although this may not lead to optimal performance, since techniques need to be applied one after another whereas most tools would apply them all together, it might help evaluating how the different options independently contribute to achieve efficiency [61].

KSP accepts inputs coming either from a file or from the command line. A configuration file can also be given. The proof search procedure for local satisfiability is shown in Algorithm 1. The outputs of the input processing procedure, Line 1, are a double-linked annotated abstract tree, representing the input formulae, and a symbol table, which contains information about the propositional symbols, constants, and modal operators occurring in the formula.

Line 2 represents the translation into the normal form. By default, formulae are transformed into SNF_{ml}, as showed in Section 3.2.1. But, some of the refinements implemented in KSP require further transformation of the set of clauses [61]. There are four different options that specify the normal form used: SNF^{+}_{ml}, SNF^{++}_{ml}, $\mathsf{SNF}^{\text{-}}_{ml}$ and SNF^{-}_{ml}. Each of them require the exhaustive application of more specific rewriting rules in addition to the rules in Definition 3.2.4 [61].

The preprocessing of clauses procedure, Line 4, comprises several tasks responsible for applying options provided by the user. For instance, propagation of a literal in the scope of the operator $\Diamond_{\forall}$, applied whenever the option `propdia` is on, is applied at this stage.

The propagation rule is given by

$$\frac{ml: \quad l \Rightarrow \Diamond_a m}{ml+1: \quad m}$$

for literals l and m, modal level ml, and agent a, with the condition that there is only one negative modal clause in the set of clauses at the given modal level. As this inference rule generates a unit clause at the propositional set of formulae, unit propagation and subsumption can be applied, possibly reducing the number of literals and clauses at the modal level $ml + 1$.

Algorithm 1: KSP-Proof-Search

```
1  input_processing
2  snf_transformation
3  clause_preprocessing
4  Γ^lit ← ⋃ Γ^lit_ml
5  while (Γ^lit ≠ ∅) do
6  |  for (all modal levels ml) do
7  |  |  clause ← given(ml)
8  |  |  if (not redundant(clause)) then
9  |  |  |  GEN1(clause, ml, ml − 1)
10 |  |  |  GEN3(clause, ml, ml − 1)
11 |  |  |  LRES(clause, ml, ml)
12 |  |  |  Λ^lit_ml ← Λ^lit_ml ∪ {clause}
13 |  |  Γ^lit_ml ← Γ^lit_ml \ {clause}
14 |  |  if (0 : false ∈ Γ^lit_0) then return unsatisfiable
15 |  Γ^lit ← ⋃ Γ^lit_ml
16 return satisfiable
```

The main loop of KSP, Lines 5-15, is based on the given-clause algorithm implemented in Otter [54], a variation of the set of support strategy [78], a refinement that imposes a restriction on the set of choices of clauses for a step of derivation. In the classical case, the set of clauses $\mathcal{F}$ is partitioned into two sets Γ (the set of support) and $\Lambda = \mathcal{F} \backslash \Gamma$ (the usable set), where Λ must be satisfiable for completeness. For the modal calculus considered here, the set of clauses is partitioned by modal level and, also, separated into literal and modal clauses. Formally, we have $\Gamma = \Gamma^{lit}_{ml}$ and $\Lambda = \Lambda^{lit}_{ml} \cup \Lambda^{mod}_{ml}$ for all modal levels, where the subscript indicates the modal level and the superscript indicates if the set is of literal (lit) or modal clauses (mod). As the calculus does not contain any rules that generate new modal clauses and because the set of modal clauses by itself is satisfiable, there is no need for a set for unprocessed modal clauses [61].

The set Γ is the *set of support* (or the sos set). On the other hand, Λ is called the *usable set*. The given clause is chosen from Γ to resolve with clauses in Λ, and moved from Γ to Λ. Resolvents are then added to Γ. In more detail, let Γ^{lit} be the union of the sets of literal clauses occurring at any modal level, Line 4 of the algorithm. A *cycle* corresponds to one iteration of the outer loop. This loop is executed while the set of unprocessed clauses is not empty. In the inner loop, Lines 6-14, for every modal level ml, a literal clause is returned by the function *given*. There are also several options for selecting the modal level and the literal clause at this stage, they can be found in detail in [61]. Once a clause is selected, it is tested for redundancy, that is if it can be deleted without affecting the satisfiability of the clause set. If the given clause is not deleted by this last procedure, then it is processed against all usable modal clauses in the previous modal level, Lines 9 and 10, which means that KSP tries to apply GEN1 and GEN3 with the literal clause at the modal level ml with the modal clauses at the modal level $ml - 1$. Then, KSP tries to apply the LRES rule with the given clause and all literal clauses in the usable at the same modal level. Note that, as no modal clauses are generated during the main loop and the inference rules MRES and GEN2 only deal with this kind of clauses, they both can be applied during the preprocessing of clauses (Line 3).

Once the inference rules are applied, the chosen clause is moved to the set of processed clauses and removed from the set of unprocessed clauses, Lines 12 and 13. The possible outputs are illustrated by Lines 14 and 16. If the empty clause $0 : \mathbf{false}$ is generated at any point of the execution, then the procedure returns that the set of clauses is unsatisfiable. If the prover is set for global satisfiability, that condition changes to finding the clause $* : \mathbf{false}$. If the empty clause is not found, KSP returns satisfiable.

The results presented in [61] indicate that KSP works well on problems with high modal depth where the separation of modal layers can be exploited to improve the efficiency of reasoning. According to [61], KSP performs well if the set of propositional symbols are uniformly distributed over the modal levels. However, when there is a high number of propositional symbols in just one particular level, the performance deteriorates. One reason is that the specific normal form used always generates satisfiable sets of propositional clauses. As resolution relies on saturation, this can be very time consuming.

In the next chapter we present some techniques proposed to deal with propositional reasoning. These techniques lead to efficient implementations of SAT solvers, which determine whether a propositional formula is satisfiable or not. In Chapter 5, we will show how to combine both modal resolution and the techniques from SAT in order to speed up the time KSP spends dealing with propositional reasoning.

Chapter 4

Satisfiability Solvers

The problem of determining whether a formula in classical propositional logic is satisfiable has the historical honour of being the first problem ever shown to be NP-Complete [21]. Great efforts have been directed in improving the efficiency of solvers for this problem, known as *Boolean Satisfiability Solvers*, or just *SAT solvers*. Despite the worst-case deterministic exponential run time of all the algorithms known, satisfiability solvers are increasingly leaving their mark as a general purpose tool in the most diverse areas [35]. In essence, SAT solvers provide a generic combinatorial reasoning and search platform for such problems. Besides, the source code of many implementations of such solvers is freely available and can be used as a basis for the development of decision procedures for more expressive logics [34].

In the context of SAT solvers for propositional provers, the underlying representation formalism is propositional logic [35]. We are interested in formulae in *Conjunctive Normal Form* (CNF): a formula φ is in CNF if it is a conjunction of clauses. For example, $\varphi = (p \vee \neg q) \wedge (\neg p \vee r \vee s) \wedge (q \vee r)$ is a CNF formula with four variables and three clauses. A clause with only one literal is referred to as a *unit clause*.

A propositional formula φ takes a value in the set $\{false, true\}$. In algorithms for SAT, variables can be *assigned* a logical value in this same set and, alternatively, variables may also be *unassigned*. A *truth assignment* (or just an assignment) to the set of variables $\mathcal{P}$, is the valuation function π as defined in Definition 2.2.1. As in propositional logic we have a singleton set as the set W of possible worlds, we can omit this set from the function signature and just write $\pi : \mathcal{P} \longrightarrow \{false, true\}$, for simplicity. A *satisfying assignment* for φ is an assignment π such that φ evaluates to *true* under π. A *partial assignment* for a formula φ is a truth assignment to a subset of the variables in φ. For a partial assignment ρ for a CNF formula φ, $\varphi|_\rho$ denotes the simplified formula obtained by replacing the variables appearing in ρ with their specified values, removing all clauses with at least one *true* literal, and deleting all occurrences of *false* literals from the remaining clauses [35].

Therefore, the *Boolean Satisfiability Problem* (SAT) can be expressed as: Given a CNF formula φ, does φ have a satisfying assignment? If this is the case, φ is said to be *satisfiable*, otherwise, φ is *unsatisfiable*. One can be interested not only in the answer of this decision problem, but also in finding the actual assignment that satisfies the formula, when it exists. All practical SAT solvers do produce such an assignment [22].

4.1 The DPLL Procedure

A *complete* solution method for the SAT problem is one that, given the input formula φ, either produces a satisfying assignment for φ or proves that it is unsatisfiable [35]. One of the most surprising aspects of the relatively recent practical progress of SAT solvers is that the best complete methods remain variants of a process introduced in the early 1960's: the Davis-Putnam-Logemann-Loveland, or DPLL, procedure [24], which describes a backtracking algorithm to the search problem of finding a satisfying assignment for a formula in the space of partial assignments. A key feature of DPLL is the efficient pruning of the search space based on falsified clauses. Since its introduction, the main improvements to DPLL have been smart branch selection heuristics, extensions like clause learning and randomised restarts, and well-crafted data structures such as lazy implementations and watched literals for fast unit propagation [35].

Algorithm 2, DPLL-recursive(φ, ρ), shows the basic DPLL procedure on CNF formulae [24], where ρ corresponds to a partial assignment of the CNF formula φ. The main idea is to repeatedly select an unassigned literal l in the input formula and recursively search for a satisfying assignment for $\varphi|_l$ and $\varphi|_{\neg l}$. The step where such an l is chosen is called a *branching step* and l is referred as a *decision variable*. Setting the decision variable to *true* or *false* when making a recursive call is referred to as a *decision*, which is associated with a *decision level* and it is equal to the recursion depth at that stage of the procedure. If the current partial assignment does not satisfy φ, the end of each recursive call takes φ back to fewer assigned literals. This last step is called the *backtracking step*.

A partial assignment ρ is maintained during the search and it is output if the formula turns out to be satisfiable. To increase efficiency, a key procedure in SAT solvers is the *unit propagation procedure* [14], where unit clauses are immediately set to *true* as outlined in Algorithm 2. In most implementations of DPLL, logical inferences can be derived with unit propagation. Thus, this procedure is used for identifying variables which must be assigned a specific value. If $\varphi|_\rho$ contains the empty clause, a *conflict* condition is declared, the corresponding clause of φ from which it came is said to be *violated* by ρ, and the algorithm backtracks.

Algorithm 2: DPLL-recursive(φ, ρ)

```
1  (φ, ρ) ← UnitPropagate(φ, ρ)
2  if φ contains the empty clause then
3  |  return unsatisfiable
4  end
5  if φ has no clauses left then
6  |  Output ρ
7  |  return satisfiable
8  end
9  l ← a literal not assigned by ρ
10 if DPLL-recursive(φ|_l, ρ ∪ {l}) = satisfiable then
11 |  return satisfiable
12 end
13 return DPLL-recursive(φ|_¬l, ρ ∪ {¬l})
```

```
1  sub UnitPropagate(φ, ρ)
2  |  while φ contains no empty clause but has a unit clause C do
3  |  |  l ← the literal in C not assigned by ρ
4  |  |  φ ← φ|_l
5  |  |  ρ ← ρ ∪ {l}
6  |  end
7  |  return (φ, ρ)
```

The literals whose negation do not appear in the formula, called *pure literals*, are also set to *true* as a preprocessing step and, in some implementations, during the simplification process after every branching. We mentioned pure literal elimination in Chapter 3 as the purity principle proposed by Robinson.

Variants of the DPLL algorithm form the most widely used family of complete algorithms for the SAT problem. They are frequently implemented in an iterative manner, instead of using recursion, resulting in significantly reduced memory usage. The efficiency of state-of-the-art SAT solvers relies heavily on various features that have been developed, analysed and tested over the last two decades. These include fast unit propagation using watched literals, deterministic and randomised restart strategies, effective clause deletion mechanisms, smart static and dynamic branching heuristics and learning mechanisms. We will discuss learning mechanisms in the next section and refer the reader to [35] for more details about other search strategies.

4.2 Conflict-Driven Clause Learning

One of the main reasons for the widespread use of SAT in many applications is that solvers based on clause learning are very efficient [35]. The main idea behind *Conflict-*

Driven Clause Learning (CDCL) is to catch the causes of a conflict as learned clauses, and utilise this information to prune the search in a different part of the search space.

Since their inception in the mid-90s, CDCL SAT solvers have been used in practice in several applications. Some successful use cases include a tool for managing learnt clauses among a parallel, memory shared solver [8], an implementation basis for hybrid logics solvers [68], and as plugins for integrated development environments [47]. Additionally, they keep showing outstanding results in SAT competitions [7, 13].

The architecture of CDCL SAT solvers is primarily inspired by the DPLL procedure. The notation used in this section is based on the one in [14]. In CDCL SAT solvers, each variable p is characterised by a number of properties. Mainly, we need to maintain its *value*, its *antecedent* and its decision level. A variable's value is denoted by $\nu(p)$, and defined in the set $\{false, true, u\}$, where $\nu(p) = u$ means that p is still unassigned.

A variable p that is assigned a value as the result of unit propagation is said to be *implied*. The unit clause $\mathcal{C}$ used for implying this value is said to be the antecedent of p, and it is denoted by $\alpha(p) = \mathcal{C}$. For decision variables or unassigned variables, the antecedent is nil. Ergo, antecedents are only defined for variables whose value is implied by other assignments.

The decision level of a variable p, written $\delta(p)$, denotes the depth of the decision tree at which the variables are assigned a value in $\{true, false\}$ or $\delta(p) = -1$ if the value of p is still unassigned, therefore, $\delta(p) \in \{-1, 0, 1, \ldots, |\mathcal{P}_\varphi|\}$, where $\mathcal{P}_\varphi$ denotes the set of all variables that appear in the initial formula φ. The decision level associated with variables used for branching steps is specified by the search process, and denotes the current depth of the *decision stack*. Hence, a variable p associated with a decision assignment is characterised by having $\alpha(p) = nil$ and $\delta(p) > 0$. On the other hand, the decision level of p with antecedent $\mathcal{C}$ is given by $\delta(p) = \max(\{0\} \cup \{\delta(p') \mid p' \in \mathcal{C} \wedge p' \neq p\})$, i.e., the highest decision level of the literals appearing in the antecedent clause.

The notation $p = v@d$ is used to denote that $\nu(p) = v$ and $\delta(p) = d$. Moreover, the decision level of a literal is defined as the decision level of its variable, that is, $\delta(l) = \delta(p)$ if $l = p$ or $l = \neg p$.

Example 4.2.1 Consider the formula

$$\begin{aligned}\varphi &= \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \\ &= (p \vee \neg s) \wedge (p \vee r) \wedge (\neg r \vee q \vee s)\end{aligned}$$

Assume that the decision assignment is $s = false@1$. Unit propagation yields no additional implied assignments. Assume that the second decision is $p = false@2$. Unit propagation yields the implied assignments $r = true@2$ and $q = true@2$. Therefore,

$\pi = \{(s, \mathit{false}), (p, \mathit{false}), (r, \mathit{true}), (q, \mathit{true})\}$ is a satisfying assignment for φ, since π makes φ true. Moreover, $\alpha(r) = \mathcal{C}_2$ and $\alpha(q) = \mathcal{C}_3$.

During the execution of a DPLL based SAT solver, assigned variables, as well as their antecedents, define a directed acyclic graph $G_{IMP} = (V, E)$ referred to as the *implication graph* [70]. The vertices of this graph are assigned variables or the special node **false**, the empty clause which represents a contradiction, that is, $V \subseteq \mathcal{P} \cup \{\mathbf{false}\}$. If unit propagation yields an unsatisfiable clause $\mathcal{C}_i$, then this special vertex is used to represent it. In this case, the antecedent of **false** is defined by $\alpha(\mathbf{false}) = \mathcal{C}_i$. Remember that the constant **false**, representing empty clauses, is different from the value *false* attributed to variables.

The edges in the implication graph are obtained from the antecedent of each implied variable. If $\alpha(p) = \mathcal{C}$ then there is a directed edge, labelled by $\mathcal{C}$, from each variable in $\mathcal{C}$, other than p, to p.

Example 4.2.2 (Implication graph without conflict). Consider the CNF formula:

$$\begin{aligned}\varphi &= \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \\ &= (p \vee t \vee \neg q) \wedge (p \vee \neg r) \wedge (q \vee r \vee s)\end{aligned}$$

Assume the decision assignment $t = \mathit{false}@1$ has been taken. Moreover, assume that the current decision assignment is $p = \mathit{false}@2$. Unit propagation yields the implied assignments $q = \mathit{false}@2$, $r = \mathit{false}@2$ and $s = \mathit{true}@2$. The resulting implication graph is shown in Figure 4.1. As all variables have been assigned a value and the implication graph does not contain the vertex **false**, this set of decision assignments forms a satisfying assignment for φ.

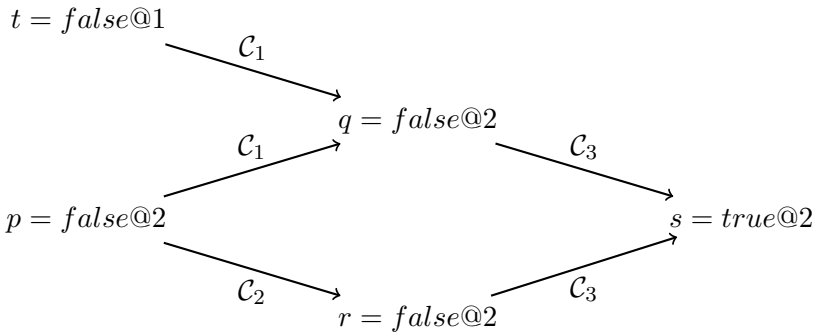


Figure 4.1: Implication graph for Example 4.2.2

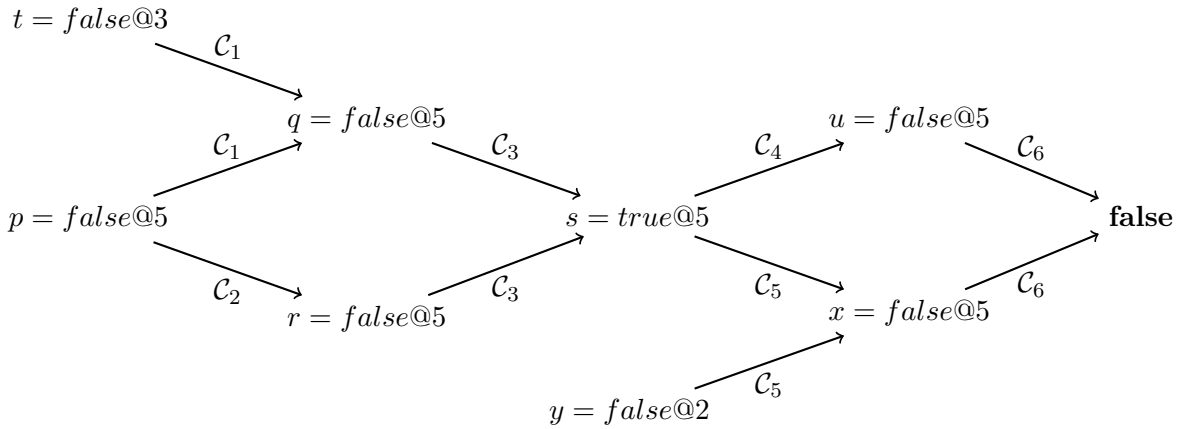


Figure 4.2: Implication graph for Example 4.2.3

Example 4.2.3 (Implication graph with conflict). Consider the CNF formula:

$$\begin{aligned}\psi &= \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3 \wedge \mathcal{C}_4 \wedge \mathcal{C}_5 \wedge \mathcal{C}_6 \\ &= (p \vee t \vee \neg q) \wedge (p \vee \neg r) \wedge (q \vee r \vee s) \wedge (\neg s \vee \neg u) \wedge (y \vee \neg s \vee \neg x) \wedge (u \vee x)\end{aligned}$$

Assume the decision assignments $y = \mathit{false}@2$ and $t = \mathit{false}@3$. Moreover, assume the current decision assignment $p = \mathit{false}@5$. Unit propagation yields the implied assignments $q = \mathit{false}@2$, $r = \mathit{false}@2$, $s = \mathit{true}@2$, $u = \mathit{false}@5$ and $x = false@5$. These last two assignments generate a conflict as the clause $\mathcal{C}_6$ can no longer be satisfied. Therefore, the resulting implication graph has the conflict vertex, as shown in Figure 4.2, with $\alpha(\textbf{false}) = \mathcal{C}_6$. Hence, this set of decision assignments does not satisfy ψ.

Algorithm 3, taken from [14], shows the standard structure of a CDCL SAT solver, which essentially follows the one from DPLL. With respect to DPLL, the main differences are the call to function `ConflictAnalysis` each time a conflict is identified, and the call to `Backtrack` when backtracking takes place. Moreover, the `Backtrack` procedure allows for backtracking prior to the most recent decision step .

In addition to the main CDCL function, the following auxiliary functions are used:

- `UnitPropagate`: same as in DPLL, consists of the iterated application of the unit propagation procedure. If an unsatisfiable clause is identified, then a conflict indication is returned.
- `AllVariablesAssigned`: tests whether all variables have been assigned, in which case the algorithm terminates indicating that the CNF formula is satisfiable.
- `PickBranchingVariable`: consists of selecting a variable to assign and deciding its value.

Algorithm 3: CDCL(φ, ρ)

```
1  if UnitPropagate(φ, ρ) yields a conflict then
2      return unsatisfiable
3  dl ← 0
4  while ¬AllVariablesAssigned(φ, ρ) do
5      l ← PickBranchingVariable(φ, ρ)
6      dl ← dl + 1
7      ρ ← ρ ∪ {l}
8      if UnitPropagate(φ, ρ) yields a conflict then
9          β ← ConflictAnalysis(φ, ρ)
10         if β < 0 then
11             return unsatisfiable
12         else
13             Backtrack(φ,ρ,β)
14             dl ← β
```

- `ConflictAnalysis`: consists of analysing the most recent conflict and learning a new clause from the conflict. The architecture of this procedure is described in Section 4.2.1.
- `Backtrack`: backtracks to the decision level computed by `ConflictAnalysis`.

The typical CDCL algorithm, shown in Algorithm 3, does not account for a few often used techniques, as for instance, search restarts and deletion policies. Search restarts cause the algorithm to restart itself, but keeping the learnt clauses. Clause deletion policies are used to choose learnt clauses that can be deleted, which allows the memory usage of the SAT solver to be kept under control. If interested, the reader can refer to [14] and [35] for more information about these techniques.

4.2.1 Conflict Analysis

Each time the CDCL SAT solver identifies a conflict due to unit propagation, the conflict analysis procedure is invoked. As a result, one or more new clauses are learnt, and a backtracking decision level is computed. This procedure analyses the structure of unit propagation and decides which literals to include in the learnt clause.

The decision levels associated with assigned variables define a partial order over the variables. Starting from a given unsatisfiable clause (represented in the implication graph as the vertex **false**), the conflict analysis procedure visits variables implied at the most recent decision level. This means that this procedure will visit the graph vertex which represents the variables assigned at the current largest decision level. Then, it identifies

the antecedents of these variables and keeps from the antecedents the literals assigned at decision levels less than the one being considered. This process is repeated until the most recent decision variable is visited.

Let d be the current decision level and p the current decision variable such that $\delta(p) = d$. Let $\nu(p) = v$ be the decision assignment and let $\mathcal{C}$ be an unsatisfiable clause identified with unit propagation. In terms of the implication graph, the conflict vertex **false** is such that $\alpha(\textbf{false}) = \mathcal{C}$. Moreover, take $RES(\mathcal{C}_i, \mathcal{C}_j)$ to represent the process of applying the binary resolution rule for two clauses $\mathcal{C}_i$ and $\mathcal{C}_j$ containing contradictory literals, as defined in Chapter 3.

The clause learning procedure used in SAT solvers can be defined by a sequence of selective resolution operations [12, 71], that at each step yields a *new temporary clause*.

Definition 4.2.1 Let l be a literal of a clause $\mathcal{C}$, and d the current decision level. Then, consider the following predicate:

$$\xi(\mathcal{C}, l, d) = \begin{cases} 1 & \text{if } l \in \mathcal{C} \wedge \delta(l) = d \wedge \alpha(l) \neq nil \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Then, using the predicate defined by Equation 4.1, let $\mathcal{C}^{d,i}$, with $i = 0, 1, \ldots$, be a intermediate clause obtained as follows:

$$\mathcal{C}^{d,i} = \begin{cases} \alpha(\textbf{false}) & \text{if } i = 0 \\ RES(\mathcal{C}^{d,i-1}, \alpha(l)) & \text{if } i \neq 0 \wedge \xi(\mathcal{C}^{d,i-1}, l, d) = 1 \\ \mathcal{C}^{d,i-1} & \text{if } i \neq 0 \wedge \forall\, l\ \xi(\mathcal{C}^{d,i-1}, l, d) = 0 \end{cases} \tag{4.2}$$

The predicate ξ holds if, and only if, the implied literal l of $\mathcal{C}$ is assigned a value at the current decision level d.

Equation 4.2 can be used for formalising the clause learning procedure. The first condition, $i = 0$, denotes the initialisation step given after **false** is generated in G_{IMP}, where all literals in the unsatisfiable clause are added to the first intermediate clause. Afterwards, at each step i, a literal l assigned at the current decision level d is selected and the intermediate clause $\mathcal{C}^{d,i-1}$ is resolved with the antecedent of l. Therefore, $\mathcal{C}^{d,i}$ denotes the clause obtained after i resolution operations.

An iteration i such that $\mathcal{C}^{d,i} = \mathcal{C}^{d,i-1}$ represents a reached *fixed point*, and $\mathcal{C}_L \stackrel{\text{def}}{=} \mathcal{C}^{d,i}$ denotes the new learnt clause. Observe that the number of resolution operations represented by Equation 4.2 is no greater than $|\mathcal{P}_\varphi|$.

Example 4.2.4 (Clause learning) Consider Example 4.2.3. The application of clause learning to this example results in the intermediate clauses shown in Table 4.1. The

Table 4.1: Resolution steps during clause learning

$\mathcal{C}^{5,0} = \{u, x\}$	Literals in $\alpha(\mathbf{false})$
$\mathcal{C}^{5,1} = \{\neg s, x\}$	Resolve with $\alpha(u) = \mathcal{C}_4$
$\mathcal{C}^{5,2} = \{\neg s, y\}$	Resolve with $\alpha(x) = \mathcal{C}_5$
$\mathcal{C}^{5,3} = \{q, r, y\}$	Resolve with $\alpha(s) = \mathcal{C}_3$
$\mathcal{C}^{5,4} = \{p, t, r, y\}$	Resolve with $\alpha(q) = \mathcal{C}_1$
$\mathcal{C}^{5,5} = \{p, t, y\}$	Resolve with $\alpha(r) = \mathcal{C}_2$
$\mathcal{C}^{5,6} = \mathcal{C}_L = \{p, t, y\}$	No more resolution operations given

resulting learnt clause is $\mathcal{C}_L = \mathcal{C}^{5,6} = (p \vee t \vee y)$. Alternatively, this clause can be obtained by inspecting the graph in Figure 4.2 and selecting the negation of the literals assigned at decision levels less than the current decision level 5, i.e. $t = \mathit{false}@3$ and $y = \mathit{false}@2$, and by selecting the current decision assignment $p = \mathit{false}@5$.

Modern SAT solvers implement an additional refinement of Definition 4.2.1, by further exploiting the structure of implied assignments induced by unit propagation, which is a key aspect of the clause learning procedure [70]. This aspect was further explored with *Unit Implication Points* (UIPs) [70]. A UIP is a vertex u in the implication graph, such that every path from a decision vertex p to the conflict vertex **false** contains u. It represents an alternative decision assignment at the current decision level that results in the same conflict. The main motivation for identifying UIPs is to reduce the size of learnt clauses.

In the implication graph, there is a UIP at decision level d when the number of literals in $\mathcal{C}_L$ assigned at decision level d is 1 [14].

Definition 4.2.2 Let $\phi(\mathcal{C}, d)$ be the number of literals in $\mathcal{C}$ assigned at decision level d:

$$\phi(\mathcal{C}, d) = |\{l \in \mathcal{C} \mid \delta(l) = d\}| \tag{4.3}$$

As a result, the clause learning procedure with UIPs is given by:

$$\mathcal{C}^{d,i} = \begin{cases} \alpha(\mathbf{false}) & \text{if } i = 0 \\ RES(\mathcal{C}^{d,i-1}, \alpha(l)) & \text{if } i \neq 0 \wedge \xi(\mathcal{C}^{d,i-1}, l, d) = 1 \\ \mathcal{C}^{d,i-1} & \text{if } i \neq 0 \wedge \phi(\mathcal{C}^{d,i-1}, d) = 1 \end{cases} \tag{4.4}$$

Equation 4.4 allows creating a clause containing literals from the learnt clause until the first UIP is identified. It is simple to develop equations for learning clauses for each additional UIP. However, this is unnecessary in practice, since the most effective CDCL SAT solvers stop clause learning when the first one is reached. Moreover, clause learning

could potentially stop at any UIP. But, considering the largest decision level of the literals of the clause learnt at each UIP, the clause learnt at the first UIP is guaranteed to contain the smallest one. What follows from this is that choosing the clause learnt at the first UIP implies the highest backtrack jump in the search tree [14].

Example 4.2.5 Consider again Example 4.2.3. The default clause learning procedure would learn the clause $(p \vee t \vee y)$ (see Example 4.2.4). However, by taking into consideration that $s = \mathit{true}@5$ is a UIP, applying the clause learning of Equation 4.4 yields the resulting clause $(\neg s \vee y)$. One advantage of this new clause is that it has one less literal than the learnt clause from Example 4.2.3.

Clause learning with conflict analysis does not affect soundness or completeness [14]. Conflict analysis identifies new clauses trough applications of the resolution rule. Therefore, each learnt clause can be inferred from the original clauses and other learnt clauses by a sequence of resolution steps [14]. Let us establish this last remark as a Theorem for future reference.

Theorem 4.2.1. *If $\mathcal{C}_L$ is a learnt clause from the set $\mathcal{F}$, then $\mathcal{F}$ is satisfiable if and only if $\mathcal{F} \cup \{\mathcal{C}_L\}$ also is.*

Additionally, neither the modified backtracking step affects the correctness of the solver, since backtracking information is obtained from each new learnt clause [14]. Proofs of soundness and completeness for different variations of CDCL SAT procedures can be found in [51, 52, 79].

Clause learning finds other applications besides the key efficiency improvements to CDCL SAT solvers. One example is *clause reuse*. In a large number of applications, clauses learnt from a given CNF formula can often be reused for related CNF formulae. The basic concept is to reuse constraints on the search space which are deduced while checking a previous instance, for speeding up the SAT checking of the current one [69].

Moreover, for unsatisfiable subformulae, the clauses learnt by a CDCL SAT solver, as formalised in Definition 4.2.1, encode a resolution refutation of the original formula. Given the way clauses are learnt in this solvers, each learnt clause can be explained by a number of resolution steps, each of which is a trivial resolution step. As a result, the resolution refutation can be obtained from the learnt clauses in linear time and space on the number of learnt clauses.

For unsatisfiable formulae, the resolution refutations obtained from the clauses learnt by a SAT solver serve as certificate for validating the correctness of the SAT solver [14]. Besides, resolution refutations based on clause learning find practical applications, including hardware model checking [55].

4.3 Modern SAT Solvers

Annual competitions have led to the development of intelligent and efficient implementations of SAT solvers, allowing many techniques to be explored and creating an extensive collection of real-world instances as well as challenging hand-crafted benchmarks problems [35]. Apart from conflict analysis, modern solvers include techniques like lazy data structures, search restarts, conflict-driven branching heuristics and clause deletion strategies [14].

In 2001, researches at Princeton University designed a complete solver, which they named Chaff [56], that, as most solvers, is an instance of the DPLL search procedure with a number of enhancements aiming in efficient applications. Their algorithm was later used as base for numerous well-succeed implementations (as [20] and [73], for instance). Chaff achieved significant performance gains through careful engineering of all aspects of the search — especially a particularly efficient implementation of unit propagation and a novel low overhead decision strategy. It was able to obtain, at the time, one to two orders of magnitude performance improvement on difficult SAT benchmarks [56].

Alongside the optimised unit propagation, Chaff also designed a decision heuristic, which they called *Variable State Independent Decaying Sum (VSIDS)*. Decision assignment consists of the determination of which new variable should be assigned a value, in each decision level. There are several strategies in the literature to try and solve this problem, but by that far, there was a lack of clear statistical evidence supporting one strategy over others, making difficult to determine what differs a good decision from a bad one [56]. With that in mind, and also after some testing with a few available strategies, Chaff's designers thought they could come up with a considerably better heuristic for the range of problems on which Chaff was being tested. The VSIDS strategy is described as follows:

1. Each literal has a counter, initialised to 0.
2. When a clause is added to the clause database, the counter associated with each literal in the clause is incremented.
3. The unassigned literal with the highest counter is chosen at each decision.
4. Ties are broken randomly by default.
5. Periodically, all the counters are divided by a constant.

Overall, this strategy can be viewed as attempting to satisfy the conflict clauses but, particularly, attempting to satisfy *recent* conflict clauses, due to the fifth item. Since difficult problems generate many conflicts, the conflict clauses dominate the problem in terms of literal count, so this approach distinguishes itself in how it favours the information

generated by recent conflict clauses [56]. To this day, CDCL SAT solvers crucially depend on the VSIDS decision heuristic for their performance [49].

A few years later, Sörensson and Een developed a SAT solver with conflict clause minimisation, named *MiniSat* [73], that stayed for a while as the state-of-the-art solver. MiniSat is a popular SAT solver, with an astonishing performance, that implements many well-known heuristics in a succinct manner. It has won several prizes in the SAT 2005 competition, and has the advantage of being open-source.

MiniSat is the implementation of the solver described in [29]. It corresponds to a minimalist Chaff-like SAT solver based on clause learning by conflict analysis. A number of small improvements has been made with respect to the original Chaff. Two important features are the incremental SAT interface, and the support for user defined Boolean constraints. Despite these improvements not being really significant for the SAT competition, they are important when using MiniSat as an integrated part of a bigger system [73]. In this sense, a user application can specify and solve SAT-problems, through MiniSat's external interface, sending its own previously known Boolean constraints.

In 2009, a MiniSat based solver appeared claiming to predict the quality of learnt clauses. *Glucose* [5] is based on a scoring scheme introduced earlier on the same year by Audemard and Simon in [6]. The name of the solver is a contraction of the concept of "glue clauses", a particular kind of clauses that Glucose detects and preserves during the search.

Detecting what would be a good clause to learn in advance is a challenge. Moreover, deleting useful clauses can increase time dramatically [5]. To avoid this unwanted deletion, solvers must let the maximum number of clauses grow exponentially. On very hard benchmarks, CDCL solvers face memory issues and, even if they do not, their greedy learning scheme deteriorates the performance. In [6], Audemard and Simon showed that a very simple static measure on clauses can dramatically improves the performance of the publicly available version of MiniSat of that time.

In their studies, Audemard and Simon observed a strong relationship between the overall decreasing of decision levels, presented by most solvers running on a industrial set of benchmarks, and the performance of them [6]. In this sense, they thought that finding the part of the learning schema that enforces this decreasing rate, could lead to identifying good learnt clauses in advance.

Moreover, they refer as *look-back justification* the estimated number of conflicts before reaching the first decision level, which stops the search. One curious thing they noticed was that the look-back justification is also strong when the solver finds a solution, i.e., a satisfying assignment. This suggested that, on satisfiable instances of problems, the solver does not always correctly guess a value for a literal, but learns that the opposite

value leads to a contradiction.

During the search, each decision is often followed by a large number of unit propagations. All the literals that share the same decision level are called "blocks" of literals. Intuitively, at the semantic level, there is a chance that they are linked with each other by direct dependencies. The idea is that a good learning schema should add explicit links between independent blocks of decision literals. If the solver stays in the same search space, a learnt clause that establishes such link should help reducing the number of next decision levels in the remaining computation [6].

Glucose partitions the literals in a clause with respect to their decision level. The *Literals Blocks Distance (LBD)* of a clause is exactly the number of subsets of its literals, divided accordingly to the current assignment. The LBD score of each learnt clause is computed and stored when the clause is produced. This measure is static, even if updated during the search (LBD score of a clause can be re-computed when the clause is used in unit-propagation).

For instance, consider a learnt clauses of LBD 2. We have seen that this clause is generated until the first UIP is identified, conform Definition 4.2.2. Therefore, it only contains one variable of the most recent decision level which are "glued" to the literals propagated earlier trough the learnt clause, no matter the size of the clause. These clauses are very important during the search, and they receive the special name "glue clauses".

Experimental results lead Audemard and Simon to conclude that this strategy works in practice. In most of the cases, the decreasing rate of decision levels was indeed accelerated.

One can notice that these last decades showed an enormous progress in the performance of SAT solvers. We have seen these modern solvers increasingly leaving their mark as a general-purpose tool, which provides a "black-box" procedure that can often solve hard structured problems with over a million variables and several million constraints [35].

As mentioned in Chapter 3, the KSP prover loses in performance when there is a high number of propositional symbols in just one particular level. We already know that these sets are satisfiable, as the specific normal form used always generates satisfiable sets of propositional clauses. If we feed a SAT solver, like MiniSat or Glucose, with these satisfiable sets, each time a conflict is identified, the learnt clauses may help KSP to increase the rate in which the inference rules that deal with modal reasoning are used.

4.3.1 MiniSat and Glucose

In the Section 4.3, we gave an overview of two of the most efficient SAT solvers known to this date: MiniSat and Glucose. Both solvers are good candidates to be combined with KSP as they both are known to efficiently solve propositional problems. As we mentioned in Section 3.2.3, when there is a large number of variables in one specific modal level,

KSP may spend a considerable amount of time trying to saturate the set of propositional clauses. Our hypothesis is that the techniques developed in solvers as MiniSat and Glucose can help KSP reduce this time, or even to guide the application of inferences rules that deal with modal reasoning. This section discuss the implementations of these solvers.

As we have seen, MiniSat is a minimalist solver, that implements a variation of Chaff's Variable State Independent Decaying Sum. MiniSat stayed for a long as the state-of-the-art solver. To these days, this solver keeps its importance as a tool used as an integrated part of different systems [26, 27, 28, 75]. In this sense, MiniSat's public interface and support for user defined constraints are key features. The source code for MiniSat, which was written in C++, can be found at [30].

It is through MiniSat's interface, taken from [29] and illustrated below, that a user application can specify and solve SAT problems [29].

Interface 4: Public interface

```
1 class Solver
2     var newVar()
3     bool addClause(Vec⟨lit⟩ literals)
4     bool add...(...)
5     bool solve(Vec⟨lit⟩ assumptions)
6     Vec⟨bool⟩ model
7 end
```

The `add...` method should be understood as a place-holder for additional constraints implemented in an extension of MiniSat. For a standard SAT problem, the interface is used in the following way: Variables are introduced by calling `newVar`. From these variables, clauses are built and added by `addClause`. Trivial conflicts, such as two unit clauses $\{p\}$ and $\{\neg p\}$ being added, can be detected by this method, in which case it returns *false*. If no such trivial conflict is detected during the clause insertion phase, `solve` is called with an empty list of assumptions. It returns *false* if the set of clauses added is unsatisfiable, and *true* if it is satisfiable, in which case the model can be read from the public vector "model". If the solver returns satisfiable, new constraints can be added repeatedly to the existing database and `solve` may execute again.

The search procedure of a modern solver is usually the most complex part to implement [29]. Heuristically, variables are picked and assigned values until the propagation detects a conflict. At this point, a conflict clause is built and added to the clauses database. Variables are then unassigned by backtracking until the conflict clause becomes unit, from which point this unit clause is propagated and the search process continues.

The learning procedure of MiniSat starts when a constraint becomes impossible to satisfy under the current assignment. The conflicting constraint yields the search for the

set of variable assignments that make it contradictory. For the conflict clause, for instance, this would be all its variables. Each of the variable assignments returned must be either a decision or a implied variable. For implied variables, we ask for the set of variable assignments that forced the implication to occur, continuing the analysis backwards.

The procedure is repeated until some termination condition is fulfilled, resulting in a set of variable assignments that implied the conflict. A clause prohibiting that particular assignment is added to the clause database. This learnt clause must always, by construction, be implied by the original problem constraints.

Learnt clauses serve two purposes: they drive the backtracking (as showed in Section 4.2) and they speed up future conflicts by caching the reason for the conflict.

The main mechanism in which MiniSat runs, taken from [29], is illustrated bellow.

Algorithm 5: Main mechanism of MiniSat

```
1  loop
2  |  propagate()
3  |  if not conflict then
4  |  |  if all variables assigned then return SAT
5  |  |  else decide()
6  |  else
7  |  |  analyse()
8  |  |  if top-level conflict then return UNSAT
9  |  |  else backtrack()
10 |  end
11 end
```

On the other hand, we have Glucose, which improved the performance of MiniSat [5], the solver that it was based on, claiming to predict the quality of learnt clauses. As we are interested in the clauses a solver will learn from the set we feed it, this may highlight Glucose from other SAT solvers. The question we face is to either go with the minimalism of MiniSat, or to bet on Glucose's robust search for a good learnt clause.

Glucose has also led its mark on SAT competitions, it won prizes pretty much annually since it was first introduced in 2009 until 2015. It is also written in C++, its source code is available at [9]. Glucose share the same public interface as MiniSat.

Audemard and Simon, the authors of Glucose, saw that one of the key factors for the performance of the solver is not only based on the identification of good clauses, but also on the removing of bad ones. However, as a side effect, aggressively deleting clauses affects the CDCL completeness [9].

Based on this last remark, remembering that we are more interested in sound and complete proof methods, this work focus on integrating KSP with MiniSat, instead of Glucose.

Chapter 5

Combining KSP and MiniSat

Now that all the theoretical ground is set, we can start talking about the contribution of this work, which is the combination of the calculus presented in Section 3.2 with the CDCL procedure presented in Section 4.2. The final purpose of the combination is to modify KSP, the prover that implements this calculus, to externally invoke MiniSat, a CDCL SAT solver. The modified prover, including sources, benchmarks, configuration files, examples, instructions for building and executing can be found in [1].

The main idea behind the combination, is to use the clauses from the satisfiable sets of literal clauses, generated by the translation into the normal form at each modal level, with a special set of unit clauses built from the literals appearing in each positive a-clause and one literal from some negative a-clause, all from the previous modal level in relation to the literal clauses. If the CDCL solver learns one or more clauses by the conflict analysis procedure we add it back to the set of propositional clauses. Additionally, if the solver concludes that the set of clauses is unsatisfiable, as we know that the set of literal clauses is by itself satisfiable, we know that the insertion of the unit clauses formed by the literals from modal clauses at the prior modal level caused the contradiction. This gives us sufficient information to learn the disjunction of all negated literals from these modal clauses as a new propositional clause at the current modal level. By learning this clause, we expect to be able to apply the GEN1 or GEN3 rules without relying on saturation of the set of propositional clauses.

Section 5.1 establishes our combination as two new rules that we add to the calculus in Section 3.2, additionally, it presents the correctness proof for both rules. Section 5.2 discusses the modifications made to KSP and MiniSat in order to implement the combination. Finally, Section 5.3 presents experimental results for KSP combined with MiniSat running on established modal benchmarks.

5.1 Combining Rules

Our combination relies on two new rules, LSAT and MSAT, in addition to the ones from the calculus that KSP implements, shown in Figure 3.3. These rules are presented in descriptive form, but fulfilling the same standard structure: list of premises followed by a conclusion. This is due to one of the rules relying on the unsatisfiability of a set of clauses, therefore, the standard way of representing a inference rule does not apply for it. In the following we give the required premises for the new rules.

Let $\mathcal{F}$ be a set of clauses in SNF_{ml}. Let also $POS_{a,ml} = \{m \mid (ml : t \Rightarrow \boxed{a} m) \in \mathcal{F}\}$ be the finite set of unit clauses formed by the literals in all positive a-clauses in $\mathcal{F}$, at a modal level ml. Suppose that $NEG_{a,ml} = \{m\}$ is a singleton formed by the unit clause such that $(ml : t \Rightarrow \diamondsuit m) \in \mathcal{F}$ is a negative a-clause in $\mathcal{F}$, at a modal level ml. Additionally, let $LIT_{ml} = \{D \mid (ml : D = \bigvee_{i=1}^{r} l_i) \in \mathcal{F}\}$ be the set of the disjunctions occurring on the literal clauses in $\mathcal{F}$ at a modal level ml. Note that LIT_{ml} is still a set of clauses, only without the label for modal levels.

The following two rules are applied to the set of clauses formed by the literal clauses at modal level ml_1, with the set of unit clauses built from the literals from the positive clauses, at modal level ml_2, and a literal from a negative clause, at modal level ml_3. Remind the unification function denoted by σ as defined in Definition 3.2.5. For the rules to be applied, the unification for $\sigma(\{ml_1, ml_2 + 1, ml_3 + 1\})$ has to be defined.

The intuition behind this construction is similar to the one behind a tableau proof for modal logics, briefly mentioned in Chapter 3. As a result of the existential and universal characters of the modal operators of possibility and necessity, respectively, we try to build a world that satisfy each literal in the scope of the possibility operator, along side with all the literals in the scope of necessity operators, with the literals that already exist at the corresponding modal level. The new rules work as follows:

- [LSAT] If $\mathcal{C}_L$ is a learnt clause from the conflict analysis of a CDCL procedure for the set of clauses $LIT_{ml_1} \cup POS_{a,ml_2} \cup NEG_{a,ml_3}$, then we can derive $(\sigma(\{ml_1, ml_2 + 1, ml_3 + 1\}) : \neg m \vee \mathcal{C}_L)$, where $m \in NEG_{a,ml_3}$, as a new literal clause at the modal level resulting from unifying $ml_1, ml_2 + 1$ and $ml_3 + 1$. For local reasoning, this means that a new clause is added, at the same level of the literal clauses, built from the disjunction of the learnt clause $\mathcal{C}_L$ with the negated literal in the scope of the possibility operator at the previous modal level. We bind the new learnt clause to the literal in the scope of the possibility operator because contradictory clauses generated from literals from different negative clauses are not, in fact, contradictory, as they can hold at different worlds.

- [MSAT] We know that the translation to SNF_{ml} only generates satisfiable sets of literal clauses at any modal level, thus LIT_{ml_1} is satisfiable. If $LIT_{ml_1} \cup POS_{a,ml_2} \cup NEG_{a,ml_3}$ turns out to be unsatisfiable, we are sure that the addition of the clauses in $POS_{a,ml_2} \cup NEG_{a,ml_3}$ turned the whole set unsatisfiable. From this fact, we can derive that at least one of the literals in the modal clauses must be false at the same modal level as the literal clauses, that is, we can add $(\sigma(\{ml_1, ml_2 + 1, ml_3 + 1\}) : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m)$, where $m_i \in POS_{a,ml_2}$, for every $1 \leq i \leq b$, and $m \in NEG_{a,ml_3}$, to the set of literal clauses at the modal level resulting from unifying $ml_1, ml_2 + 1$ and $ml_3 + 1$.

5.1.1 Correctness Results

This section presents the proofs of soundness for the new inference rules, LSAT and MSAT. Theorems 5.1.1 and 5.1.2 are based on the same premises for the rules. Proofs will be presented for local reasoning, but it is easily extended to consider global reasoning.

Theorem 5.1.1 (LSAT). *Let $\mathcal{C}_L$ be a learnt clause trough CDCL from the set $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$. Then $\mathcal{F}$ is satisfiable if, and only if, $\mathcal{F} \cup \{ml + 1 : \neg m \vee \mathcal{C}_L\}$, where $m \in NEG_{a,ml}$, also is.*

Proof. Suppose that $\mathcal{F}$ is satisfiable. From Theorem 4.2.1 we know that CDCL does not affect correctness, therefore, $\mathcal{C}_L$ can be shown to be a logical consequence of $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$ trough a finite number of resolution steps, according to Definition 4.2.2. As $\mathcal{F}$ is satisfiable, we know that there is a model that satisfies $\mathcal{F}$ (recall how we defined the satisfiability problem in Section 2.2). If this model has a world at modal level $ml + 1$ that satisfies m, we know that it must also satisfy all $m_1, \ldots, m_b$ literals positive clauses, in addition to the clauses in LIT_{ml+1}. Finally, because $\mathcal{C}_L$ is a consequence of $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$, it can be shown that $\mathcal{C}_L$ is also satisfiable in this world.

As the construction of the $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$ set used the literals from modal clauses, we bind the learnt clause to the literal from the negative clause, knowing that every world that satisfy this literal will also satisfy $\mathcal{C}_L$. Therefore, the addition of $\{ml + 1 : \neg m \vee \mathcal{C}_L\}$ to $\mathcal{F}$ is sound and $\mathcal{F} \cup \{ml + 1 : \neg m \vee \mathcal{C}_L\}$ is also satisfiable.

Now, suppose $\mathcal{F} \cup \{ml + 1 : \neg m \vee \mathcal{C}_L\}$ satisfiable. From the satisfiability of sets, we know that $\mathcal{F}$ must be satisfiable and that finishes this proof. □

Theorem 5.1.2 (MSAT). *If $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$ is unsatisfiable, then $\mathcal{F}$ is satisfiable if, and only if, $\mathcal{F} \cup \{ml + 1 : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m\}$, where $m_i \in POS_{a,ml}$ for $0 \leq i \leq b$ and $m \in NEG_{a,ml}$, also is.*

Proof. Suppose that $\mathcal{F}$ is satisfiable and that $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$ is unsatisfiable. As we know that LIT_{ml+1} is satisfiable and m by itself its not contradictory, there are five cases left to consider for the proof of soundness. Recall the few highlighted details, presented in Section 3.2, from the completeness proof for the calculus. The cases we have to consider for this proof follow directly from each item of the presented details.

- If $POS_{a,ml}$ or $POS_{a,ml} \cup NEG_{a,ml}$ are either contradictory, it means that we are trying to add a clause of the form $ml{+}1 : \neg m_1 \vee \ldots \vee m \vee \neg m$, assuming $m \in NEG_{a,ml}$ and $\neg m \in POS_{a,ml}$, without lack of generality. A clause with a tautology $(m \vee \neg m)$ does not affect the satisfiability of the set.

- If $LIT_{ml+1} \cup NEG_{a,ml}$ is contradictory, like mentioned in Item 3 of the details given for the completeness proof, by consequence completeness of binary resolution (Lemma 3.2.2), applications of LRES to the set of literal clauses would generate $ml{+}1 : \neg m$. Therefore the addition of this clause is sound. And by the subsumption principle (Lemma 3.1.2), the addition of any clause that subsumes it also is.

- If $LIT_{ml+1} \cup POS_{a,ml}$ is contradictory, it follows from Item 4 that the clause $ml+1 : \neg m_1 \vee \ldots \vee \neg m_b$ would also be generate by applications of LRES. Therefore, the addition of this clause, or any other that subsumes it, to $\mathcal{F}$ is also sound.

- Finally, the case in which $LIT_{ml+1} \cup POS_{a,ml} \cup NEG_{a,ml}$ also follows from Item 4 (and Item 5). Then we know that the clause $ml + 1 : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m$ would also be generated by applications of LRES.

From the five cases we considered, we know that adding $\{ml + 1 : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m\}$ to $\mathcal{F}$ is sound and, therefore, $\mathcal{F} \cup \{ml + 1 : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m\}$ is also satisfiable.

Now, suppose $\mathcal{F} \cup \{ml + 1 : \neg m_1 \vee \ldots \vee \neg m_b \vee \neg m\}$ satisfiable. Again from the satisfiability of sets, we know that $\mathcal{F}$ must be satisfiable and that finishes this proof. □

Completeness of the calculus is not affected as we only add rules that derive clauses that were already consequences of the initial set of clauses. In other words, they would have been generated at some point by exclusively applying LRES to the set of literal clauses at a given modal level, or at least be subsumed by some of these clauses.

5.2 Implementation

We had two options on how to combine KSP with MiniSat. For the first one, we would have to interact with the public interface of MiniSat as an integrated part of KSP. The second one is to make an external call to MiniSat from KSP, which requires less interference on KSP. The required effort to adapt the source code of MiniSat for both approaches

would be pretty much the same, but we believe that the first approach probably would result in less overall overhead of time, once every external call from KSP to MiniSat requires a translation from the internal representation of clauses of the first to the internal representation of the other, and vice versa. Hence, it would present a better performance.

We think, however, that the second approach is more viable, at least at a first moment, as it demands less changes on the source code of KSP, because we are more interested in a proof of concept to verify that our approach does not affect the correctness of the prover. Implementing MiniSat as an integrated part of KSP is left as a possible future work. Thus, our implementation changes KSP to externally invoke a minimally modified version of MiniSat. After a thorough study of MiniSat's source code, we only found two necessary changes. As we are only interested in the final answer, satisfiable or not, and the clauses learned from the proof search procedure, we modified MiniSat to output even less information than the default triggered by the lowest level of verbosity of the solver. Additionally, we had to force MiniSat to output the learnt clauses as soon as they are produced.

Algorithm 6: KSP-Proof-Search-With-SAT-Solver

```
1  input_processing
2  snf_transformation
3  clause_preprocessing
4  MOD_D ← modal_clauses_translation(Λ^mod)
5  sat_skip ← 0
6  while (Γ^lit ≠ ∅) do
7      if (!sat_skip) then
8          LIT_D ← propositional_clauses_translation(Γ^lit ∪ Λ^lit)
9          Γ^lit ← Γ^lit ∪ minisat_call(LIT_D, MOD_D)
10     sat_skip ← (sat_skip + 1) % sat_cycles
11     for (all modal levels ml) do
12         clause ← given(ml)
13         if (not redundant(clause)) then
14             GEN1(clause, ml, ml − 1)
15             GEN3(clause, ml, ml − 1)
16             LRES(clause, ml, ml)
17             Λ^lit_ml ← Λ^lit_ml ∪ {clause}
18         Γ^lit_ml ← Γ^lit_ml \ {clause}
19         if (0 : false ∈ Γ^lit_0) then return unsatisfiable
20     Γ^lit ← ⋃ Γ^lit_ml
21 return satisfiable
```

Algorithm 1 on Section 3.2.3 presented the proof search procedure of KSP. Now, Algorithm 6 shows exactly where, Lines 4, 5 and 7 - 10, we had to modify KSP for our pur-

poses of combination. In the Algorithm 6, let $\Gamma^{lit} = \bigcup \Gamma^{lit}_{ml}$ be the set of support of literal clauses occurring at every modal level. Analogously, let $\Lambda^{lit} = \bigcup \Lambda^{lit}_{ml}$ and $\Lambda^{mod} = \bigcup \Lambda^{mod}_{ml}$ be the usable sets of literal and modal clauses.

Lines 7 and 10 represent an additional option when running KSP combined with MiniSat. For experimental purposes, we let the user define the value of `sat_cycles`, the variable representing the frequency that the SAT solver is called. If the user, for instance, set the `sat_cycles` to 100, MiniSat will be invoked every 100 cycles of the outer loop of KSP. The default value of `sat_cycles` is one, i.e., MiniSat called every iteration of the main loop.

The `modal_clauses_translation` and `literal_clauses_translation` procedures, Lines 4 and 8, respectively, correspond to the translation of variables from the internal representation of KSP, which is a signed integer, to the internal representation of MiniSat, based on Dimacs [17], which is also a signed integer. The `literal_clauses_translation` procedure, shown in Algorithm 7, where $LIT_{D,ml}$ represents all clauses at modal level ml already in Dimacs format, is quite straightforward. It translates all literal clauses at every modal level to their specific Dimacs representation, without the modal level annotation.

Algorithm 7: propositional_clauses_translation($\Gamma^{lit} \cup \Lambda^{lit}$)

```
1 for all modal levels ml do
2 |   LIT_{D,ml} ← ∅
3 |   for all clauses C in Γ^{lit}_{ml} ∪ Λ^{lit}_{ml} do
4 |   |   LIT_{D,ml} ← LIT_{D,ml} ∪ DIMACS(C)
5 |   end
6 end
7 return ⋃ LIT_{D,ml}
```

We know that the sets of literal clauses are satisfiable, thus, we are interested in the clauses MiniSat might learn if it eventually finds a conflict. Additionally, we also want to see the effect of adding to the set of literal clauses, unit clauses with the literals on the scope of modal operators at the previous modal level. The intuition behind this addition was discussed in Section 5.1. This information might lead to an early application of GEN1 or GEN3. For that purpose, the `modal_clauses_translation` builds clauses containing literals in the scope of modal operators, accordingly with Algorithm 8, that will be passed along with the literal clauses to the SAT solver. Algorithm 8 builds, for each negative a-clause, a set of unit clauses with the literal in the scope of the $\diamondsuit_a$ operator and all unit clauses formed by the literals in the scope of the $\boxed{a}$ operator. Then, it translates each of these new unit clauses to Dimacs format. We already know that no modal clauses are generated during the main loop of KSP, that is why we can call this translation prior to the main loop.

Algorithm 8: modal_clauses_translation(Λ^{mod})

```
1  for all modal levels ml do
2  |  MOD_{D,ml} ← ∅
3  |  for all agents a do
4  |  |  POS_{a,ml} ← ∅
5  |  |  for all positive a-clauses ml : l ⇒ [a]m in Λ^{mod}_{ml} do
6  |  |  |  POS_{a,ml} ← POS_{a,ml} ∪ DIMACS({m})
7  |  |  end
8  |  |  for all negative a-clauses ml : l ⇒ ⟐_a m in Λ^{mod}_{ml} do
9  |  |  |  MOD_{D,ml} ← MOD_{D,ml} ∪ {POS_{a,ml} ∪ DIMACS({m})}
10 |  |  end
11 |  end
12 end
13 return ⋃ MOD_{D,ml}
```

The call to MiniSat, illustrated by Algorithm 9, occurs every k iterations of KSP main loop, where k is set by the user trough the `sat_cycles` configuration. The aim is to feed MiniSat with the set of clauses that contains all literal clauses at a specific modal level with each unit clause built from the modal literals at the previous modal level.

Algorithm 9: minisat_call(LIT_D, MOD_D)

```
1  C_new ← ∅
2  for all modal levels ml do
3  |  for all set of unit clauses Δ = {m_1, ..., m_b, m} in MOD_{D,ml−1} do
4  |  |  while MiniSat (LIT_{D,ml} ∪ Δ) not finished do
5  |  |  |  if MiniSat learns a new clause C_L then
6  |  |  |  |  C_new ← C_new ∪ {ml : ¬m ∨ C_L}
7  |  |  |  end
8  |  |  end
9  |  |  if MiniSat returns unsatisfiable then
10 |  |  |  C_new ← C_new ∪ {ml : ¬m_1 ∨ ... ¬m_b ∨ ¬m}
11 |  |  end
12 |  end
13 end
14 return C_new
```

Each set of unit clauses $\Delta = \{m_1, \ldots, m_b, m\} \in MOD_{D,ml-1}$, Line 3, represents the clauses built from a literal m that occurs in a negative clause, at modal level $ml - 1$, and all literals occurring on the positive clauses at the same level. If MiniSat happens to learn a new clause while reasoning over the set of literal clauses with these additional clauses, we add it to the set of literal clauses of KSP, with a binding to the literal in the possibility operator that was passed along. Moreover, if MiniSat decides that the set is unsatisfiable,

we also add a new clause to the literal clauses of KSP. As shown in Section 5.1, the unsatisfiability of the set means that we cannot place all modal literals, from the previous modal level, at the current one, meaning that at least one of them must be false. The derived clause, Line 10, must be a premise for GEN1 and/or GEN3, as the clause only contains literals in the scope of modal operators in modal clauses at the previous level, therefore, we expect that, once learning this clause, one of the two inference rules could be applied right away. At the current state of the implementation, we do not immediately try to apply either rules, as it would have required further changes on KSP main loop. We leave it, though, as a future work. The `minisat_call` procedure returns the set of all clauses that must be added to the literal clauses at each corresponding modal level, as given in Line 14 of Algorithm 9.

Example 5.2.1 Consider the initial set of clauses in SNF_{ml}, Clauses 1-8, from Figure 5.1. The figure shows a refutation of the set using a call to MiniSat. In this example, MiniSat was called with the set of clauses $\{a, \neg a \vee \neg b, a \vee \neg b \vee c, \neg a, b\}$ as input, where $\neg a$ and b are the literals of the modal Clauses 4 and 5, respectively. When MiniSat returns unsatisfiable, we add Clause 9 to the set of literal clauses at modal level one. This clause allows the application of GEN1 instantly, as showed in Clause 10. Without the call to MiniSat, KSP would have to correctly choose Clauses 6 and 8 to apply LRES, generating clause $1 : \neg b$, and then apply GEN3, generating the same Clause 10.

1.	$0 : t_0$	8.	$1 : \neg a \vee \neg b$	
2.	$0 : \neg t_0 \vee t_1$	9.	$1 : \neg b \vee a$	[MSAT, 4, 5, 6, 8]
3.	$0 : \neg t_0 \vee t_2$	10.	$0 : \neg t_1 \vee \neg t_2$	[GEN1, 4, 5, 9, a, b]
4.	$0 : t_1 \Rightarrow \Diamond \neg a$	11.	$0 : \neg t_0 \vee \neg t_2$	[LRES, 2, 10, t_1]
5.	$0 : t_2 \Rightarrow \Box b$	12.	$0 : \neg t_0$	[LRES, 3, 11, t_2]
6.	$1 : a$	13.	$0 :$ **false**	[LRES, 1, 12, t_0]
7.	$1 : a \vee \neg b \vee c$			

Figure 5.1: Example of refutation using KSP combined with MiniSat

5.3 Experimental results

In Section 5.1.1 we proved that the addition of the LSAT and MSAT rules to the calculus that KSP implements does not affect correctness. However, we also need to determine if we did not insert any implementation errors to the KSP source code while adding these rules. In this sense, our experimental tests are interested in the evaluation of the impact and overhead of calling MiniSat from KSP, as well as the answers KSP gives as indications

that the implementation is correct. For these purposes, we compared KSP 0.1.2 running on three benchmarks. The benchmarks used consist of three collections of modal formulae:

- LWB: Basic modal logic benchmark formulae [11], with 378 formulae divided into 9 families. Each family is a set with 21 parameterised formulae. Of the 378 formulae, half are satisfiable and half are unsatisfiable by construction of the benchmark families. The minimum modal depth of formulae in this collection is 1, the maximum 30,004.
- MQBF: This benchmark corresponds to the complete set of TANCS-2000 modalised random QBF (MQBF) formulae [53] complemented by the additional MQBF formulae provided by Kaminski and Tebbi [43]. This collection consists of five families, called qbf, qbfL, qbfS, qbfML, and qbfMS, with 617 formulae that are known to be satisfiable and 399 known to be unsatisfiable, total of 1016 formulae. The minimum modal depth of formulae in this collection is 19, the maximum is 225.
- 3CNF: This collection consists of 135 3CNF formulae over 3 propositional symbols with modal depth 2, 4 or 6, all of which are known to be satisfiable [53].

For purposes of comparison, we executed KSP with five configuration files that differ only on the option regarding the call to MiniSat: `satsolver,k`, where `k` is the number of iterations of the main loop between every call to MiniSat. The remaining used settings were: `prenex`, `early_mlple`, `bnfsimp`, `snf++`, `unit`, `lhs_unit`, `limited_reuse_renaming`, `mlple`, `populate_usable,max_lit_positive`, `ordered`, `propdia`, `mres`, `local`, `shortest`, `sos_subsumption`, `forward`, `backward` and, finally, `full_check_repeated`. The choice for this particular set of options was due to the results on the empirical experimentation considering different configurations presented in [59] and [61]. We will now give a quick overview of these options that were used in our experimentation.

The description of the options was taken from [61], and the reader can refer to it if interested in other configuration options. For the options related to input processing, we used the option `prenex` to translate a formula into prenex. Basically, the prenex normal form corresponds to pushing the modal operators occurring in a formula φ as far as possible outwards the formula. With option `early_mlple`, also related to input processing, pure literal elimination is applied at each modal level, prior the transformation into the clausal form. With option `bnfsimp` simplification is applied to formulae in Box Normal Form (BNF) instead of formulae in NNF. The translation into BNF removes the $\Diamond\!\!\!\!-$ operator, allowing the $\boxed{a}$ operator to occur in the scope of negation. For example, the formulae $\boxed{a}(p \vee q) \wedge \Diamond\!\!\!\!-(\neg p \wedge \neg q)$, in NNF, and $\boxed{a}(p \vee q) \wedge \neg\boxed{a}(p \vee q)$, in BNF, are semantically equivalent, although the contradiction appears explicitly on the second formula.

Regarding to the SNF_{ml} translation, we choose the option `snf++`, that applies the following rewriting rules, in addition to the ones in Definition 3.2.4.

$$\rho(ml : t \Rightarrow \boxed{a} l) = (ml : t \Rightarrow \boxed{a} t') \wedge \rho(ml + 1 : t' \Rightarrow l)$$
$$\rho(ml : t \Rightarrow \diamondsuit_{\!a} l) = (ml : t \Rightarrow \diamondsuit_{\!a} t') \wedge \rho(ml + 1 : t' \Rightarrow l)$$

These additional rules sets the ordering of the new introduced propositional symbols to be lower than the ordering of any other symbol in the clause set. The translation into `snf++` is needed for retaining completeness when ordered resolution is also set (see below). Options `unit` and `lhs_unit` were also set, which allows unit resolution, applying the additional inference rules illustrated in Figure 5.2.

$$\text{[UNIT]}\quad \frac{\begin{array}{rl} ml_1 : & D \vee l \\ ml_2 : & \neg l \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2\}) : & D \end{array}} \qquad \text{[UNIT-GEN1]}\quad \frac{\begin{array}{rl} ml_1 : & t \Rightarrow \diamondsuit_{\!a} m \\ ml_2 : & \neg m \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2 + 1\}) : & \neg t \end{array}}$$

$$\text{[UNIT-GEN3]}\quad \frac{\begin{array}{rl} ml_1 : & t_1 \Rightarrow \boxed{a} m_1 \\ ml_2 : & t_2 \Rightarrow \diamondsuit_{\!a} m_2 \\ ml_3 : & \neg m_1 \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2 + 1\}) : & \neg t_1 \vee \neg t_2 \end{array}}$$

$$\text{[LHS-UNIT-1]}\quad \frac{\begin{array}{rl} ml_1 : & l \\ ml_2 : & l \Rightarrow \diamondsuit_{\!a} m \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2\}) : & \mathbf{true} \Rightarrow \diamondsuit_{\!a} m \end{array}} \qquad \text{[LHS-UNIT-2]}\quad \frac{\begin{array}{rl} ml_1 : & l \\ ml_2 : & l \Rightarrow \boxed{a} m \end{array}}{\begin{array}{rl} \sigma(\{ml_1, ml_2\}) : & \mathbf{true} \Rightarrow \boxed{a} m \end{array}}$$

Figure 5.2: Unit resolution rules

Additionally, with option `limited_reuse_renaming`, we forced the same new propositional symbol to be used for all occurrences of the same subformula being renamed at a particular modal level.

Concerning the preprocessing of clauses procedure, the configuration options used were: `mlple` which allowed pure literal elimination whenever a literal is pure at some modal level. Option `populate_usable,max_lit_positive`, that moves literal clauses whose maximal literal is positive from Γ^{lit}_{ml} to Λ^{lit}_{ml}. Option `ordered`, where clauses can only be resolved on their maximal literals with respect to an ordering chosen by the prover in such a way to preserve completeness, was also set. Propagation of a literal in the scope of possibility operator, as showed in Section 3.2.3, are applied because the option `propdia` is set. Moreover, we forced the MRES rule to be applied even when it is not needed for completeness, because it produces very short resolvents, with the option `mres`. Also, `forward` and `backward` are set, so that deletion of newly generated or old clauses which are subsumed is applied; it also means that self-subsumption is applied during preprocessing of clauses. We also forced subsumption checking in the whole set

of clauses by setting the option `sos_subsumption`: the default is to apply subsumption only against the usable.

Finally, with respect to the main loop of the proof search, we set KSP to only do local reasoning with the `local` option. With option `shortest`, we tell KSP to choose the clause with the smallest size at a particular modal level. With `full_check_repeated`, repetition of clauses is checked against all sets of clauses at the same modal level: the default is to check for repetition only against the set of support.

The option that differed each file was the `satsolver,k` one. The contribution to KSP that this work set out to do was to implement the changes illustrated by the algorithms of Section 5.2, that are called whenever the `satsolver` option is present. In the option, `k` is the value attributed to the variable `sat_cycles`, which corresponds to the number of iterations of KSP's main loop between each call to MiniSat, set as default to one. We choose to run KSP with $\texttt{k} \in \{1, 10, 100, 1000, 10000\}$ and also without the call to MiniSat at all. We expected to see a decay of performance for low values of k, as the call for MiniSat causes a considerable overhead for translating and building clauses and making the external call. For larger values of k, we expected to see a similar performance from KSP without the call to MiniSat, as it only does it in very large intervals, sometimes does not even get a change to call it at all, in instances that KSP is able to solve very quickly.

Benchmarking was performed on a Ubuntu 18.04 with an Intel i7-8565U CPU @1.80 GHz and 16 GB main memory. For each formula and each configuration we have determined the median run time over three runs with a time limit of 100 CPU seconds. The choice for this limit is justified by the fact that KSP alone is able to solve most entries from LWB and MQBF benchmarks within this time constraint.

While KSP performs pretty well for those entrances, for 3CNF it cannot solve any entry, even when we increased the time limit to 1200 CPU seconds. Our expectation was that this would be different for KSP combined with MiniSat. Unfortunately, our expectations were not fulfilled. Each tested configuration of KSP failed to solve every formulae from this benchmark within this large limit. Therefore, in the following, we will only report on the findings of our experiment over the other two sets of formulae used for benchmarking. For both the LWB and MQBF benchmarks, all instances solved within the previous stipulated time limit, with each configuration file, returned the correct answer. This serves as indication that all errors that appeared during the implementation stage were fixed. Therefore, the implementation itself is considered a proof of concept that the addition of the new rules did not change the correctness of KSP.

Table 5.1 shows the number of satisfiable and unsatisfiable instances solved by KSP running the benchmarks for every configuration file. In general, KSP combined with MiniSat did not surpass KSP running alone, for any user defined number of cycles between

each call. As expected, calling the SAT solver every iteration of the main loop generates a large amount of overhead, which decreases as we increase the interval that is called, though it seems to never cross a worthwhile barrier. Large intervals did, also as expected, perform similar to KSP alone. This can also be observed in both graphs of Figure 5.3, which illustrate the amount of instances solved by the time of execution for each benchmark.

Table 5.1: Number of satisfiable and unsatisfiable instances solved

	LWB		**MQBF**	
Configuration	**Satisfiable**	**Unsatisfiable**	**Satisfiable**	**Unsatisfiable**
Total of instances	**189**	**189**	**617**	**399**
`sat-1`	141	157	365	134
`sat-10`	158	160	577	167
`sat-100`	159	166	600	226
`sat-1000`	162	167	603	301
`sat-10000`	163	166	603	309
`without`	163	166	602	338

Figure 5.3 shows the impact on run-time of the different frequencies of calls to MiniSat on the performance of KSP. For the LWB benchmark, `without`, `sat-10000`, `sat-1000` and even `sat-100` were able to solve most entries in under one second. While the line for `sat-10` indicates that, even with some overhead, it was able to solve most entries within the time limit of 100 seconds. But `sat-1` shows a lot more explicitly the impact of the overhead caused by the calls to MiniSat on the run-time. For the MQBF benchmark, we have `sat-10000` and `sat-1000` presenting an almost identical performance, slightly inferior to the one from `without`. Again, `sat-1` presents the worst performance.

Remember that the LWB benchmark has the same number of satisfiable and unsatisfiable formulae. Thus the analysis of the performance of KSP with MiniSat with respect to the satisfiability of formulae is straightforward. Figure 5.4 shows that KSP alone already solves slightly more unsatisfiable entries for this benchmark. This pattern was repeated for all the other settings. All instances of KSP combined with MiniSat were able to resolve pretty much the same number of instances, compared to KSP alone, with the exception of KSP calling MiniSat every iteration of the main loop. It presented a large decrease of performance for satisfiable entries. This might indicate that, besides the known overhead from the excessive number of calls to MiniSat, the clauses we learn trough our implemented procedures favoured the search for a refutation rather than the construction of a satisfying model, for this benchmark. The opposite behaviour is, though, observed for

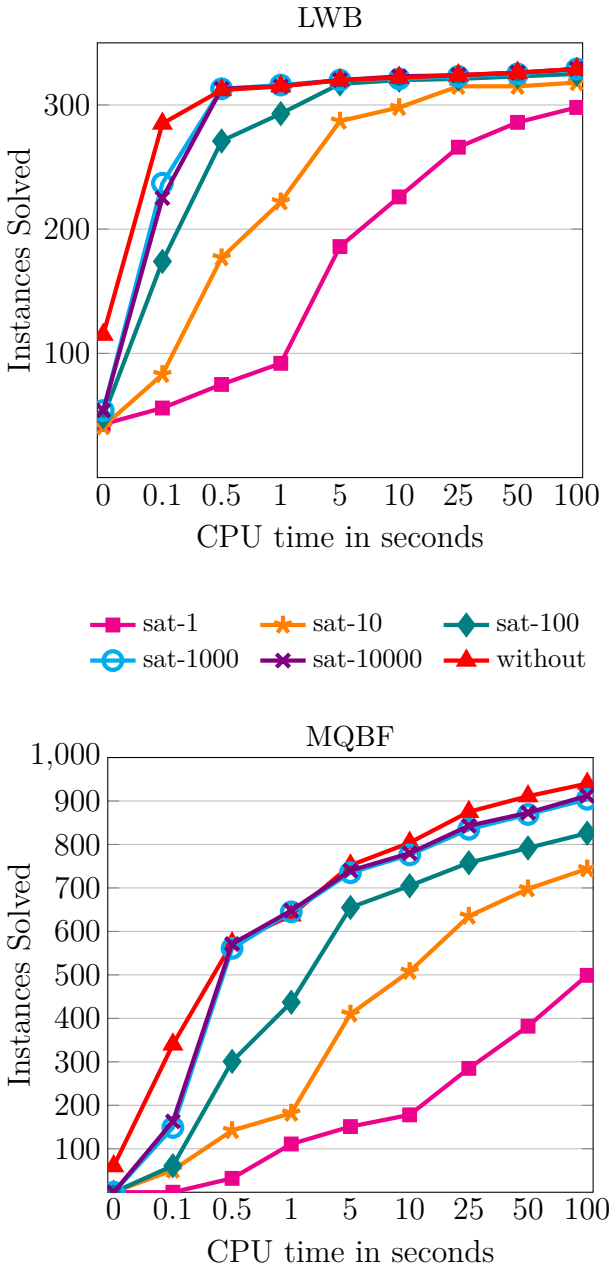


Figure 5.3: Benchmark results for KSP with and without the call for MiniSat

the MQBF benchmark, where `sat-1` only solves about 40% of the total of the unsatisfiable instances solved by KSP alone. While for satisfiable instances, it was able to solve around 60%. The remaining configurations also presented a better rate at solving satisfiable instances for this benchmark. These differences might be explained by the inherent characteristics from each benchmark.

The prover alone, in general, performed better, closely followed by KSP calling MiniSat every 1000 and 10000 cycles. KSP calling MiniSat every 10 and 100 cycles performances were average when comparing with KSP calling MiniSat every iteration of the main loop.

Therefore, overall, it seems to not pay off the overhead created for externally calling MiniSat, regardless of the interval that is called. However, for unsatisfiable instances of one specific family of the LWB benchmark, the `ph` family, KSP calling MiniSat every 100 cycles was able to solve up until entry number nine in less than 100 CPU seconds, while KSP alone only solves until the fifth entry, as showed in Table 5.2.

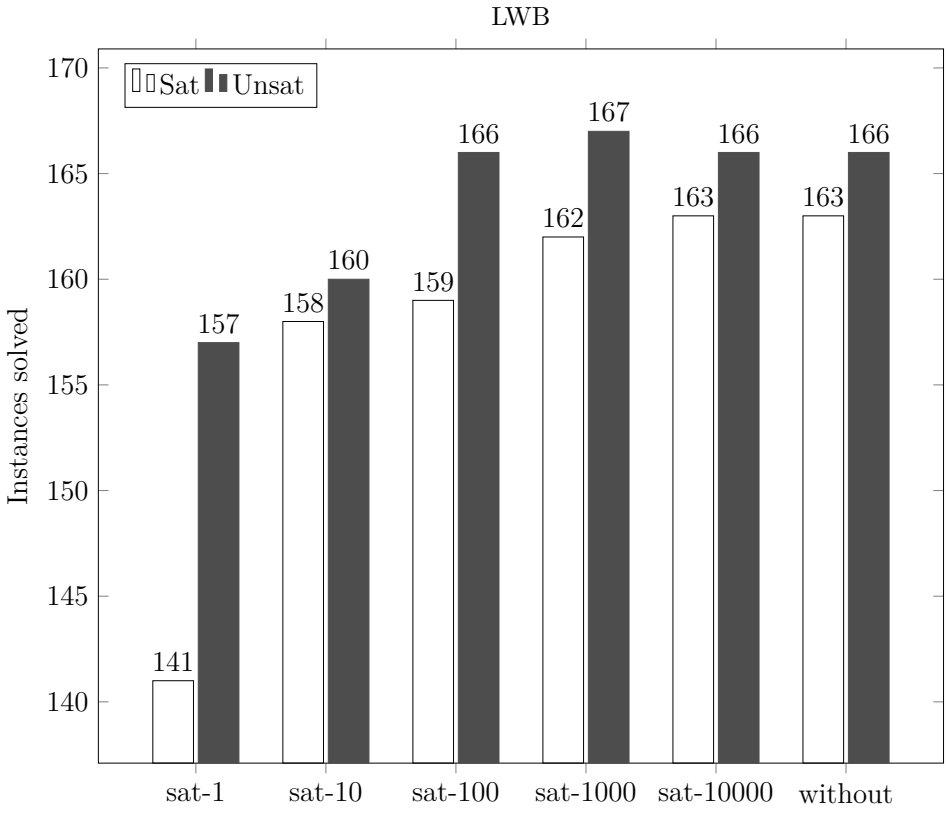


Figure 5.4: Number of satisfiable and unsatisfiable instances solved for KSP with and without the call for MiniSat on benchmark LWB

The `ph` family corresponds to the *pigeonhole principle* established as a formula in K_n. In mathematics, the pigeonhole principle affirms that if n elements are put into m buckets, with $n > m$, then at least one bucket must contain more than one element [40]. This is one of the hardest families of LWB for KSP. Indeed, it is also hard for several other provers as well [61]. Therefore, our contribution to KSP managed to solve four more instances in comparison to before.

According to [11], the `ph` formulae are essentially classical propositional problems with few insertions of modal operators. If k is the entry parameter, the corresponding `ph` unsatisfiable formula has $k^2 + k$ variables and the maximum modal depth of two, as

Table 5.2: Run-time in seconds for each unsatisfiable entry for family `ph`

	Time (seconds) per solved entry							
Configuration	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`sat-1`	0.01	0.26	1.23	4.01	13.67	90.93	-	-
`sat-10`	0.01	0.06	0.18	0.49	1.51	3.88	-	-
`sat-100`	0.01	0.03	0.08	0.07	0.34	0.7	2.42	93.81
`sat-1000`	0.00	0.03	0.12	0.38	1.05	-	-	-
`sat-10000`	0.00	0.03	0.17	2.89	-	-	-	-
`without`	0.01	0.01	0.12	2.83	-	-	-	-

illustrated by Equation 5.1 taken from [11]. The high number of variables distributed over a maximum of two modal levels in these formulae represents exactly the previously described scenario, which was focus of this work that attempt to improve the efficiency of KSP for such entries. Besides having our implementation as proof of concept for the proposed combination of the CDCL procedure implemented in the MiniSat solver with the resolution calculus implemented in KSP, helping KSP to solve more entries from this particular family can also be considered a positive result of this work.

$$\begin{aligned} \mathbf{ph(k)} &= \Diamond left(k) \Rightarrow \Diamond right(k) \\ left(k) &= \textstyle\bigwedge_{i=1}^{k+1} \bigvee_{j=1}^{k} l(i,j) \\ right(k) &= \textstyle\bigvee_{j=1,i_1=1,i_2=i_1+1}^{k,k+1,k+1} (l(i_1,j) \wedge l(i_2,j)) \\ l(i,j) &= \begin{cases} \Box p_{i+j} & i < j \\ p_{i+j} & \text{otherwise} \end{cases} \end{aligned} \tag{5.1}$$

As a last remark on this section, we will take the executions for the `ph` family as an opportunity to start a discussion about the number of inferences made until KSP finds a refutation, for each configuration of KSP. The number of inferences made serves as a parameter of memory usage of KSP.

Table 5.3 shows that the use of MiniSat had a positive impact on the number of new clauses KSP infers for each solved instance of this family. With a special remark to `sat-100`, which was the configuration to solve more instances within the time limit. This table also shows that excessive calls to MiniSat also lead to a larger number of inferences than the required to derive a refutation. This is due to every call to MiniSat possibly adding new clauses that don't necessarily hold essential information, leaving KSP to deal with saturation of the clause set anyway. Even so, `sat-1`, which presented a run-time reasonably superior to KSP calling MiniSat at large intervals or not calling at all, for example, for all the instances they all solved, managed to find the proof with considerably less inferences, in comparison. This indicates that, for the `ph` family, the combination did not required saturation to be able to find a refutation. Further analysis of the relation

between the call to MiniSat and the number of inferences made for different families of formulae is left as future work.

These results indicate that our combination impacted KSP beyond the run-time outcome only. We believe that further polishing of the changes made to the prover related to the call to MiniSat, trying to find a better middle ground between minimising the number of inferences made and the overhead of calls to MiniSat, might lead to even better results. Specially considering the approach of integrating MiniSat into KSP, as we think that it would considerably decrease the overall overhead of our combination.

Table 5.3: Number of inferences for each unsatisfiable entry for family `ph`

	Inferences made per solved entry							
Configuration	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`sat-1`	3	81	456	1870	5903	19510	-	-
`sat-10`	3	34	119	418	1661	6715	-	-
`sat-100`	3	112	195	319	1277	5009	21059	293325
`sat-1000`	3	112	2554	2621	4304	-	-	-
`sat-10000`	3	112	2554	134567	-	-	-	-
`without`	19	150	2724	134418	-	-	-	-

Chapter 6

Conclusion and Future Work

This chapter does an overview of the theoretical knowledge concerned to this work and brings final remarks on our contribution to KSP, as well as suggestions on possible future work. Chapter 2 introduced the propositional modal language which is the main focus of this work: K_n. We have seen that K_n extends classical logic by adding the unary operators $\boxed{a}$ and $\diamondsuit$ to express notions of necessity and possibility, respectively, as different modes of truth. The key goal behind these operators is to allow the reasoning over relations between the abstractions we call possible worlds. Thus, the information holding at worlds accessible from the current one — via an accessibility relation — is available to examination.

In Chapter 3 we briefly introduced clausal resolution, the base of the modal-layered calculus behind KSP. As we have seen, resolution is a simple and adaptable proof system for propositional classical logic, as it has only one inference rule that is thoroughly applied. This rule entails the well known resolution principle. The calculus presented for K_n adds a few more rules, as it also has to deal with modal reasoning. This calculus makes use of labelled resolution in order to avoid unnecessary application of such rules. Thus, it requires a translation of formulae into a more expressive language, where labels are used to express semantic properties. Formulae in K_n are, then, translated into a layered normal form called SNF_{ml}, also presented in Chapter 3. We showed that a formula in SNF_{ml} is a conjunction of clauses where the modal level in which they occur is made explicit in the syntax.

KSP implements this modal labelled resolution calculus. It was designed to support experimentation with different combinations of refinements. KSP presents a great efficiency if the set of propositional symbols are uniformly distributed over the modal levels. However, when there is a high number of variables in just one particular level, the performance deteriorates. One reason is that the specific normal form used always generates satisfiable sets of propositional clauses. As resolution relies on saturation, this can be

very time consuming. Our work investigated alternative paths that KSP can take at this stage. We aimed at strategies that could increase the rate in which we find opportunities for applications of inference rules that deal with modal reasoning.

We saw in Chapter 4 that SAT solvers are very efficient in the proof search for propositional logic. They can often solve hard structured problems with over a million variables and several million constraints. Also in Chapter 4, we discussed the role of clause learning in the successful widespread use of SAT. We saw that the main idea behind Conflict-Driven Clause Learning solvers is to catch clauses of conflict as learned clauses, and to use this information to prune the search in a different part of the search space. The conflict analysis procedure consists of analysing the most recent conflict and learning a new clause from it. Furthermore, we have seen in Chapter 4 that MiniSat is a popular CDCL SAT solver, with an astonishing performance, that implements many well-known heuristics in a succinct manner. It has won several prizes in the SAT 2005 competition, and has the advantage of being open-source. To these days, this solver keeps its importance as a tool used as an integrated part of different systems [26, 27, 28, 75].

Chapter 5 presented the additional inference rules we added to the calculus discussed in Chapter 3, and the implementation of these rules as our contribution to the KSP list of available strategies. We also presented the correctness proofs for the new rules.

Our implementation modifies KSP to externally call MiniSat. The main idea behind this combination, was to build a query for the SAT solver, to find out if the literals from modal clauses, at a specific modal level, can be satisfied in the set of propositional clauses at the next level. We, then, from the main loop of KSP, call MiniSat passing as input the clauses from the satisfiable sets of literal clauses, generated by the translation into the normal form at each modal level, with a special set of unit clauses built from the literals appearing in each $\boxed{a}$ operator and one unit clause with the literal from a $\diamondsuit$ operator as well, all appearing in the previous modal level in relation to the literal clauses. If the CDCL solver learns one or more clauses by the conflict analysis procedure we add it back to the set of propositional clauses, as we know that this learnt clauses are consequences of the set we passed to MiniSat. Additionally, if the solver concludes that the set of clauses is unsatisfiable, we learn the disjunction of all negated literals from the modal clauses as a new propositional clause at the same modal level. By learning this clause, we expected to be able to apply the inference rules that deal with modal reasoning without relying on saturation.

Experimental results on classical benchmarks, as showed in Chapter 5, did not present, in general, a gain in run-time parameters, probably due to the overhead we inserted calling MiniSat. However, getting all correct answers indicates that our implementation of the rules that combine the KSP calculus with a CDCL solver did not introduced errors on

the code, also serving as proof of concept. Additionally, we were able to advance KSP in solving unsatisfiable entries of one of the hardest families of the LWB benchmark. This family corresponds to the pigeonhole principle established as a formula in K_n. These formulae have a structure similar to the scenario for which we hoped to improve the efficiency of KSP: a large number of propositional symbols in a specific modal level. The prover alone is able to solve five instances of this family while our combination managed to solve until the ninth entry. The execution of KSP combined with MiniSat for these entries, also showed that using MiniSat allowed an overall decrease in the number of required inferences on the search for a proof. Therefore, our hypothesis that the combination might decrease the time KSP spends saturating the clause set was confirmed for this particular family.

As future work we leave the following suggestions:

- We consider to implement our combination with MiniSat as an integrated part of KSP, as it provides a nice public interface that we can easily interact with. However, as making an external call to the solver demanded less interference on KSP, we have chosen this option in the current work. The implementation showed interesting results, though our combination did not improve KSP performance in average. But maybe using MiniSat as an internal component of the prover, the overhead might be reduced, and the efficiency of the combination may be increased.

- When the set we pass to MiniSat turns out to be unsatisfiable, we build a clause exactly like the premises of two of the inference rules that deal with modal reasoning. After the construction of these clauses, we could try to apply these rules right away. But this would require further changes on the main loop of KSP. Thus we also left it as a possible future work.

- The results for the `ph` family of the LWB benchmark revealed that our combination can help KSP reduce the number of inferences needed to build a proof. We leave space for further analysis of the relation between the number of inferences made and the time KSP spends in the proof search, and how the combination with MiniSat can impact on this relation.

- Finally, although our approach was designed to deal with either local or global reasoning, we have not experimented the combination of KSP and MiniSat for global satisfiability. The main reason is that there is a lack of established benchmarks for such problems. We intend to do further experimentation on the performance of our implementation when dealing with global satisfiability by designing and/or adapting existing benchmarks.

Bibliography

[1] D. Angelos. KSP +minisat: sources and benchmarks. `http://www.cic.unb.br/~nalon/#software`, 2019. vii, 3, 45

[2] C. Areces, H. De Nivelle, and M. De Rijke. Prefixed resolution: A resolution method for modal and description logics. *Lecture Notes in Computer Science*, 1632:187–201, 1999. 26

[3] C. Areces, R. Gennari, J. Heguiabehere, and M. De Rijke. Tree-based heuristics in modal theorem proving. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 199–203. IOS Press, 2000. 10

[4] C. Areces and J. Heguiabehere. HyLoRes: A hybrid logic prover, Sept. 18 2002. v, 1

[5] G. Audemard and L. Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009. vii, 2, 41, 44

[6] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404, 2009. 41, 42

[7] G. Audemard and L. Simon. Glucose in the sat 2014 competition. *SAT COMPETITION 2014*, page 31, 2014. 33

[8] G. Audemard and L. Simon. Lazy clause exchange policy for parallel sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 197–205. Springer, 2014. 33

[9] G. Audemard and L. Simon. The glucose sat solver. `https://www.labri.fr/perso/lsimon/glucose/`, 2019. 44

[10] L. Bachmair and H. Ganzinger. Resolution theorem proving. *Handbook of automated reasoning*, 1:19–99, 2001. 14, 15, 16

[11] P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, 2000. vii, 53, 58, 59

[12] P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *CoRR*, abs/1107.0044, 2011. 37

[13] A. Biere. Splatz, lingeling, plingeling, treengeling, yalsat entering the sat competition 2016. *Proc. of SAT Competition*, pages 44–45, 2016. 33

[14] A. Biere, M. Heule, H. van Maaren, and T. Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009. vii, 2, 31, 33, 35, 36, 38, 39, 40

[15] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic: Graph. Darst*, volume 53. Cambridge University Press, 2002. v, 1, 4, 11

[16] M. Bratman. *Intention, plans, and practical reason*. Center for the Study of Language and Information, California, USA, 1987. v, 1, 12

[17] G. Caire, S. Shamai, and S. Verdu. Dimacs series in discrete mathematics and theoretical computer science. *American Mathematical Society*, 2004. 50

[18] M. A. Casanova. *Programação em lógica e a linguagem Prolog*. E. Blucher, 1987. 15

[19] B. F. Chellas. *Modal logic — an introduction*. Press Syndicate of the University of Cambridge, London, 1980. 5

[20] W. Chrabakh and R. Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 37–37. IEEE, 2003. 40

[21] S. A. Cook. The complexity of theorem proving procedures. In *stoc71*, pages 151–158, 1971. 30

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. 31

[23] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga. *Handbook of tableau methods*. Springer Science & Business Media, 2013. 14

[24] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. 31

[25] N. Dershowitz and C. Kirchner. Abstract saturation-based inference. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 65–74. IEEE, 2003. 16

[26] N. Een. Cut sweeping. *Cadence Design Systems, Tech. Rep*, 2007. vii, 2, 43, 62

[27] N. Eén, A. Mishchenko, and N. Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. *CoRR*, abs/1008.2021, 2010. vii, 2, 43, 62

[28] N. Een, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2007. vii, 2, 43, 62

[29] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003. 41, 43, 44

[30] N. Eén and N. Sörensson. The minisat page. `http://minisat.se/`, 2019. 43

[31] N. Eisinger and J. Ohlbach. Deduction systems based on resolution. Technical report, European Computer-Industry Research Centre GmbH, München, 1991. 13

[32] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MITpress, Cambridge, MA, USA, 1995. v, 1, 12

[33] M. Fitting and R. L. Mendelsohn. *First-order modal logic*, volume 277. Springer Science & Business Media, 2012. 13, 14, 16

[34] E. Giunchiglia, A. Tacchella, and F. Giunchiglia. Sat-based decision procedures for classical modal logics. *Journal of Automated Reasoning*, 28(2):143–171, 2002. 30

[35] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008. vi, vii, 2, 16, 30, 31, 32, 36, 40, 42

[36] V. Goranko and S. Passy. Using the universal modality: gains and questions. *Journal of Logic and Computation*, 2(1):5–30, 1992. 11, 12

[37] B. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/New York, 1982. v, 1, 12

[38] J. Halpern, Z. Manna, and B. Moszkowski. A Hardware Semantics Based on Temporal Intervals. *Lecture Notes in Computer Science*, 154:278–291, 1983. v, 1, 12

[39] J. A. Harland, D. J. Pym, and M. Winikoff. Forward and backward chaining in linear logic. *Electronic Notes in Theoretical Computer Science*, 37:1–16, 2000. 13, 14

[40] I. N. Herstein. *Topics in Algebra*. Blaisdell, Waltham, MA, 1964. 58

[41] I. Horrocks, U. Hustadt, U. Sattler, and R. Schmidt. 4 computational modal logic. In *Studies in Logic and Practical Reasoning*, volume 3, pages 181–245. Elsevier, 2007. 12

[42] U. Hustadt and R. A. Schmidt. An empirical analysis of modal theorem provers. *Journal of Applied Non-Classical Logics*, 9(4):479–522, 1999. v, 1

[43] M. Kaminski and T. Tebbi. Inkresat: modal reasoning via incremental reduction to sat. In *International Conference on Automated Deduction*, pages 436–442. Springer, 2013. vii, 53

[44] S. C. Kleene. *Mathematical logic*. Courier Corporation, 2002. 4, 13

[45] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3-4):227–260, 1971. 15

[46] S. A. Kripke. Semantical analysis of modal logic I. *Zeitschr. Math. Logik Grund. Math*, 9:67–96, 1963. 5

[47] D. Le Berre and A. Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. 33

[48] R. C. T. Lee. *A completeness theorem and computer program for finding theorems derivable from given axioms*. PhD thesis, Berkeley, 1967. 24, 25

[49] J. H. Liang, V. Ganesh, E. Zulkoski, A. Zaman, and K. Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers, Sept. 14 2015. 41

[50] D. W. Loveland. A linear format for resolution. In *Automation of Reasoning*, pages 399–416. Springer, 1983. 15

[51] J. Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits*. PhD thesis, University of Michigan, 1995. 39

[52] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. 39

[53] F. Massacci and F. M. Donini. Design and results of tancs-2000 non-classical (modal) systems comparison. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 52–56. Springer, 2000. vii, 53

[54] W. W. McCune. Otter 3.0 reference manual and guide. Technical report, Argonne National Lab., IL (United States), 1994. 28

[55] K. L. McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003. 39

[56] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001. vii, 2, 40, 41

[57] C. Nalon. Lógica computacional 1. Notas de Aula, 2015. 16

[58] C. Nalon and C. Dixon. Clausal resolution for normal modal logics. *J. Algorithms*, 62(3-4):117–134, 2007. 4, 5

[59] C. Nalon, C. Dixon, and U. Hustadt. Modal resolution: Proofs, layers, and refinements. *ACM Transactions on Computational Logic (TOCL)*, 20(4):23, 2019. 53

[60] C. Nalon, U. Hustadt, and C. Dixon. A modal-layered resolution calculus for K. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 185–200. Springer, 2015. v, 1, 11, 15, 19, 21, 22, 24, 25, 26

[61] C. Nalon, U. Hustadt, and C. Dixon. KSP a resolution-based theorem prover for K_n: Architecture, refinements, strategies and experiments. *Journal of Automated Reasoning*, Dec 2018. v, vi, viii, 1, 2, 3, 22, 27, 28, 29, 53, 58

[62] C. Nalon, U. Hustadt, and C. Dixon. KSP: sources and benchmarks. `http://www.cic.unb.br/~nalon/#software`, 2019. 27

[63] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986. 20

[64] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484, Cambridge, MA, USA, Apr. 1991. Morgan-Kaufmann. v, 1, 12

[65] R. Reiter. Two results on ordering for resolution with merging and linear format. *Journal of the ACM (JACM)*, 18(4):630–646, 1971. 15

[66] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan. 1965. 14, 15, 16, 17, 18

[67] S. Schulz. The E theorem prover, 2013. `http://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`. v, 1

[68] H. M. Sheini and K. A. Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:165–189, 2006. 33

[69] O. Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 58–70. Springer, 2001. 39

[70] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227. IEEE Computer Society, 1997. 34, 38

[71] J. P. M. Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 675–680, NY, June 5–9 2000. ACM/IEEE. 37

[72] J. R. Slagle, C. L. Chang, and R. C. T. Lee. Completeness theorems for semantic resolution in consequence-finding. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI 1969)*, pages 281–286. William Kaufmann, May 1969. 24

[73] N. Sorensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53), 2005. vii, 2, 40, 41

[74] E. Spaan. *Complexity of Modal Logics*. PhD thesis, University of Amsterdam, 1993. 12

[75] M. Suda. *Resolution-based methods for linear temporal reasoning*. PhD thesis, Saarland University, 2014. vii, 2, 43, 62

[76] The SPASS Team. Automation of logic: Spass, 2010. `http://www.spass-prover.org/`. v, 1

[77] A. Voronkov. Vampire. `http://www.vprover.org/index.cgi`. v, 1

[78] L. Wos, G. A. Robinson, and D. F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM (JACM)*, 12(4):536–541, 1965. 15, 27, 28

[79] L. Zhang and S. Malik. *Searching for truth: techniques for satisfiability of boolean formulas*. PhD thesis, Princeton University Princeton, 2003. 39