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

IMPLEMENTAÇÃO DE UM PROVADOR DE TEOREMAS POR RESOLUÇÃO PARA LÓGICAS MODAIS NORMAIS

George Bezerra Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.ª Dr.ª Cláudia Nalon

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenadora: Prof.ª Dr.ª Maristela Terto de Holanda

Banca examinadora composta por:

Prof.ª Dr.ª Cláudia Nalon (Orientadora) — CIC/UnB
Prof. Dr. Mauricio Ayala-Rincón — CIC/UnB
Prof. Dr. Flávio Leonardo Cavalcanti de Moura — CIC/UnB

CIP — Catalogação Internacional na Publicação

Silva, George Bezerra.

IMPLEMENTAÇÃO DE UM PROVADOR DE TEOREMAS POR RESOLUÇÃO PARA LÓGICAS MODAIS NORMAIS / George Bezerra Silva. Brasília : UnB, 2013.
115 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. Lógica modal, 2. Resolução, 3. Prova automática de teoremas

CDU 004.42:510.6

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

IMPLEMENTAÇÃO DE UM PROVADOR DE TEOREMAS POR RESOLUÇÃO PARA LÓGICAS MODAIS NORMAIS

George Bezerra Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

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

Prof. Dr. Mauricio Ayala-Rincón
CIC/UnB

Prof. Dr. Flávio Leonardo Cavalcanti de Moura
CIC/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 23 de dezembro de 2013

Agradecimentos

Aos meus pais, Geraldo e Rose, por terem me criado e ensinado a perseverar diante dos desafios. Aos meus irmãos Leonardo, pelas palavras amigas e conselhos nos momentos de dificuldade, e Eduardo, por estar sempre disponível para ajudar. À minha namorada, Karina, por toda a compreensão durante o período de produção deste trabalho e pelo amor, carinho e companheirismo que sempre teve comigo.

À Prof.ª Dr.ª Cláudia Nalon por aceitar me oferecer orientação neste trabalho de graduação, mesmo em condições adversas, por confiar na minha capacidade e por me auxiliar incondicionalmente em cada etapa. À Prof.ª Dr.ª Maristela Terto de Holanda pelo grande apoio em semestres anteriores.

Aos meus amigos pelo incentivo e apoio, especialmente aos grandes companheiros Anayran Pinheiro e Rodrigo Von-Grapp.

Resumo

Neste trabalho apresentamos a implementação de um provador de teoremas para sistemas da lógica modal normal para múltiplos agentes, baseado na técnica de resolução modal publicada por Nalon and Dixon [2007]. Ao contrário das demais abordagens de prova por resolução na lógica modal, onde as regras de inferência são elaboradas para uso em um sistema em particular (por exemplo, K, S4, S5), o método de resolução usado como base para a implementação apresentada se concentra nas restrições impostas às relações de acessibilidade entre mundos para cada sistema modal. Portanto, temos regras de inferência específicas para sistemas seriais, reflexivos, simétricos, transitivos e euclidianos, além de um conjunto de regras de inferência válidas em toda a classe de sistemas modais normais. Desta forma, a implementação torna-se abrangente, por conseguir lidar com diversos sistemas modais através da combinação das regras de inferência mencionadas, e expansível, pois basta implementar novas regras de inferência que capturem outra restrição nas relações de acessibilidade para aumentarmos a abrangência do cálculo, sem prejudicar a correção e completude dos resultados para os sistemas que já estavam cobertos. O programa apresentado, escrito na linguagem C++, efetua o cálculo de prova para os quinze sistemas modais normais definidos pelas diferentes combinações dos axiomas $\mathbf{D}_i$, $\mathbf{T}_i$, $\mathbf{B}_i$, $\mathbf{4}_i$ e $\mathbf{5}_i$, incluindo sistemas como $\mathrm{K}_{(n)}$, $\mathrm{D}_{(n)}$, $\mathrm{T}_{(n)}$, $\mathrm{B}_{(n)}$, $\mathrm{S4}_{(n)}$ e $\mathrm{S5}_{(n)}$, onde o índice (n) representa a versão multi-agente de um sistema.

Palavras-chave: Lógica modal, Resolução, Prova automática de teoremas

Abstract

In this work we present an implementation of a theorem prover for multi-agent normal modal logic systems, based on the modal resolution technique published by Nalon and Dixon [2007]. Differently from other approaches for resolution proof systems or methods in modal logic, where inference rules are developed for use in a particular system e.g. K, S4 or S5, the resolution method used as a basis for the implementation presented here focuses on the restrictions imposed on the accessibility relations between worlds for each modal system. Thus, we have specific inference rules for serial, reflexive, symmetric, transitive and Euclidean systems, as well as a set of inference rules for every normal modal system. Therefore, the implementation becomes comprehensive, since it can deal with several modal systems by combination of the inference rules mentioned above; and extensible, since we only need to implement new inference rules which capture another restriction in the accessibility relations to increase the range of systems included in the calculus, without compromising the soundness and completeness of the results for modal systems already covered before. The presented program, coded in the C++ language, automatically produces proofs for the fifteen distinct normal modal systems defined by the different combinations of the axioms $\mathbf{D}_i$, $\mathbf{T}_i$, $\mathbf{B}_i$, $\mathbf{4}_i$ and $\mathbf{5}_i$, including systems like $\mathrm{K}_{(n)}$, $\mathrm{D}_{(n)}$, $\mathrm{T}_{(n)}$, $\mathrm{B}_{(n)}$, $\mathrm{S4}_{(n)}$ and $\mathrm{S5}_{(n)}$, where the (n) index indicates a multi-agent version of a system.

Keywords: Modal logic, Resolution, Automated theorem proving

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A lógica modal é um método de raciocínio extremamente útil em diferentes áreas da ciência da computação, tais como inteligência artificial, teoria de bancos de dados, sistemas distribuídos, verificação de programas e teoria da criptografia. Por exemplo, a lógica epistêmica, que trata do conhecimento de agentes, pode ser modelada através da lógica modal e possui aplicações em sistemas distribuídos e inteligência artificial; formas particulares da lógica epistêmica são também utilizadas para raciocinar sobre provas de conhecimento-zero, que é uma parte da teoria de criptografia [Lambert et al., 2004].

Dada uma especificação lógica, um provador de teoremas pode então ser utilizado para verificar propriedades do sistema. Para poder modelar diferentes aspectos de uma situação em particular, pode ser necessário combinar diferentes linguagens lógicas. Uma das combinações mais simples e frequentemente usadas é fusão de lógicas, onde os operadores modais de diferentes componentes não interagem entre si, sendo então independentemente axiomatizáveis. Isso significa que o cálculo de prova para a lógica combinada pode ser potencialmente obtida pela combinação do cálculo de cada linguagem. Porém, é necessário cuidado para assegurar que a informação aplicável a vários componentes é corretamente tratada e trocada entre os componentes. Por exemplo, ao combinar diferentes lógicas modais proposicionais devemos assegurar que fórmulas proposicionais suficientes são passadas entre os sistemas.

Nalon and Dixon [2007] apresentaram em seu artigo um método uniforme para provar teoremas para várias *lógicas modais normais proposicionais* para múltiplos agentes, baseado em resolução. Ao contrário das demais abordagens de prova por resolução na lógica modal, onde as regras de inferência são elaboradas para um sistema em particular (por exemplo, K, S4, S5), o método de resolução usado como base para a implementação aqui apresentada se concentra nas restrições impostas às relações de acessibilidade entre mundos para cada sistema modal. No artigo mencionado são apresentadas regras de inferência específicas para sistemas seriais, reflexivos, simétricos, transitivos e euclidianos, além de um conjunto de regras de inferência válidas em toda a classe de sistemas modais normais. Portanto, é possível representar sistemas modais cujas relações de acessibilidade possuam qualquer combinação dentre as cinco propriedades supramencionadas, ou nenhuma delas. Conforme mostrado em [Chellas, 1980], existem exatamente 15 sistemas modais normais que podem ser definidos através de combinações dessas propriedades — o método apresentado em [Nalon and Dixon, 2007] consegue provar teoremas, de forma correta e completa, para os 15 sistemas, através da combinação das regras de inferência

apresentadas no artigo. Tal método de prova é adequado para implementação em computadores por ser baseado na técnica de resolução, um método de prova muito utilizado na lógica clássica e facilmente automatizável. Apesar disso, ainda não existia qualquer implementação desse método.

Neste trabalho apresentamos a implementação de um provador de teoremas, escrita na linguagem C++, que utiliza o método de prova mencionado acima. No provador de teoremas que apresentamos foram implementadas as regras de inferência comuns para todos os sistemas modais normais, além das regras para sistemas seriais, reflexivos, simétricos, transitivos e euclidianos. Desta forma, o programa consegue provar teoremas em quinze sistemas modais normais: $\mathrm{K}_{(n)}$, $\mathrm{D}_{(n)}$, $\mathrm{T}_{(n)}$, $\mathrm{KB}_{(n)}$, $\mathrm{K4}_{(n)}$, $\mathrm{K5}_{(n)}$, $\mathrm{KDB}_{(n)}$, $\mathrm{KD4}_{(n)}$, $\mathrm{KD5}_{(n)}$, $\mathrm{KD45}_{(n)}$, $\mathrm{K45}_{(n)}$, $\mathrm{B}_{(n)}$, $\mathrm{KB4}_{(n)}$, $\mathrm{S4}_{(n)}$ e $\mathrm{S5}_{(n)}$, onde o índice (n) representa a versão multi-agente de um sistema. Posteriormente o programa poderá ser estendido determinando e implementando regras de inferência que capturem outras restrições nas relações de acessibilidade.

A escolha de implementação do método citado acima foi feita devido à capacidade de extensão deste para outras lógicas modais. Por exemplo, a implementação pode ser estendida para abranger outras lógicas modais normais, não apenas as 15 descritas em [Chellas, 1980], através da introdução de novas regras de inferência. Além disso, seguindo as ideias expostas por [Nalon and Dixon, 2007], é possível combinar a resolução implementada neste trabalho com o cálculo de resolução para lógicas temporais, por exemplo para a lógica temporal linear proposicional (PTL) ou para a lógica temporal em árvore (CTL), para implementar um provador de teoremas para combinações (fusões) de todas as lógicas modais normais com as lógicas temporais PTL ou CTL.

1.1 Estrutura

No Capítulo 2 deste trabalho mostramos a base teórica para o entendimento da Lógica Modal para vários agentes (também chamada de *lógica multimodal*). Apresentamos a sintaxe e semântica das lógicas modais normais, onde o axioma $\mathbf{K_i}$ (a versão multimodal do axioma $\mathbf{K}$) é válido. Mostramos também os axiomas $\mathbf{D_i}$, $\mathbf{T_i}$, $\mathbf{B_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$, juntamente com as implicações da validade desses axiomas em um sistema modal — a validade de cada um desses axiomas gera sistemas *seriais*, *reflexivos*, *simétricos*, *transitivos* e *reflexivos*, respectivamente. Por fim, apresentamos os 15 sistemas modais distintos que podem ser obtidos através da combinação entre os axiomas mencionados.

O Capítulo 3 inicia com uma breve introdução ao método de prova por resolução para a lógica proposicional clássica, descrito originalmente por Robinson [1965], descrevendo o processo de transformação de uma fórmula para a *forma normal conjuntiva* utilizada no procedimento de prova por resolução, para então descrever o método de prova propriamente dito. Após isso, mostramos o método de prova para lógicas modais normais, baseado em resolução, proposto por Nalon and Dixon [2007]. Primeiramente mostramos a tradução de uma fórmula para a forma normal utilizada pelo método, chamada de *Forma Normal Separada para Sistemas Normais* (FNS_K). Depois disso tratamos das sete regras de inferência para resolução em toda a classe de sistemas modais normais, e por fim mostramos as regras de inferência específicas para sistemas seriais, reflexivos, simétricos, transitivos e euclidianos. Não entramos nos detalhes da prova de correção,

completude e terminação de cada regra, pois estas podem ser vistas no artigo original, mas apresentamos os dados necessários para o entendimento de cada uma das regras.

No Capítulo 4 descrevemos o provador de teoremas em si. Mostramos quais regras de inferência foram implementadas, como executar o programa e damos uma visão geral do seu fluxo de execução. Após isso, mostramos as estruturas de dados utilizadas para representar os símbolos proposicionais, literais e cláusulas em cada estágio da execução. Por fim, detalhamos cada um dos módulos do programa, mostrando quais as funcionalidades providas por cada módulo e como estas foram implementadas, de forma geral.

Finalmente, no Capítulo 5 fazemos um comparativo entre os objetivos do trabalho e os resultados alcançados, além de expor as dificuldades encontradas e mencionar o que se pretende implementar nas futuras versões do programa.

Capítulo 2

Lógica Modal

A *lógica modal* estuda as proposições modais e as relações lógicas que elas possuem umas com as outras. Ela é uma extensão da lógica proposicional clássica com a inclusão de *operadores modais*, que quando aplicados a uma proposição especificam uma maneira ou *modo* pelo qual se obtém uma interpretação para esta proposição [Zalta, 1995].

A interpretação mais conhecida da lógica modal, e também aquela que iniciou os estudos nessa área, é a que considera as *modalidades aléticas* — as modalidades da necessidade e da possibilidade. Nessa interpretação, o operador modal '$\Box$' significa "é necessário". É comum o uso de um segundo operador modal, '$\Diamond$', definido como o dual de '$\Box$' (ou seja, '$\Diamond p$' é equivalente a '$\neg\Box\neg p$'), que possui o significado de "é possível". Deste modo, as fórmulas '$\Box p$' e '$\Diamond p$' devem ser lidas como "p é necessário" e "p é possível", respectivamente.

Para outras modalidades são utilizadas diferentes interpretações desses operadores. Por exemplo, na lógica demonstrativa '$\Box p$' significa "é *demonstrável* que p". Na lógica epistêmica, que lida com o conhecimento, é comum se utilizar o operador 'K_i' ao invés de '$\Box$' para representar que um agente i conhece algo — desta forma, '$K_i\varphi$' deve ser lido como "o agente i conhece φ" ou "o agente i sabe que φ". O uso de lógicas modais com diversos agentes é tratado com mais detalhes na Seção 2.1.1. Aplicações mais recentes da lógica modal incluem linguagens que lidam com diversas estruturas relacionais, tais como sistemas de transição de estado para programas de computador, redes semânticas para a representação de conhecimento ou estruturas de valores de atributos na linguística [Gabbay et al., 2003].

No decorrer deste trabalho será utilizada frequentemente a interpretação usual, onde os símbolos unários de modalidade '$\Box$' e '$\Diamond$' representam *necessidade* e *possibilidade*, respectivamente. A escolha visa apenas auxiliar na clareza ao tratar do tópico apresentado e a modalidade alética foi escolhida por ser a interpretação mais recorrente. Porém, é possível estender o método de prova apresentado neste trabalho — o qual é detalhado no Capítulo 3 — para outras lógicas modais normais, independente da interpretação dos operadores (a definição formal de lógica modal normal é dada na Seção 2.3).

2.1 Sintaxe

A lógica modal é uma extensão da lógica proposicional clássica, com a adição de operadores modais. Neste trabalho o foco são as lógicas multimodais, que consideram diversos

agentes. Portanto, apresentamos esse conceito para então podermos definir um sistema modal normal para múltiplos agentes, o qual chamamos de $\mathrm{K}_{(n)}$

2.1.1 Lógicas multimodais

Como mencionado no início deste capítulo, existem diversas aplicações onde dois ou mais operadores modais são necessários. Por exemplo, para representar o conhecimento de n agentes se desenvolvendo no tempo podemos utilizar $n+1$ operadores $\Box$ — um para tratar tempo e um para cada agente, para representar seu conhecimento. Uma das possibilidades para a representação de lógicas multimodais é estender os conceitos e resultados da lógica modal para a representação de problemas que envolvem vários agentes.

Na abordagem utilizada neste trabalho, uma lógica modal pode ter diversos *agentes*. Cada agente possui suas próprias definições de quais proposições são necessárias e quais são possíveis em cada caso — a forma como essas relações modais são construídas para cada agente é detalhada na Seção 2.2, que trata sobre semântica. No que se refere à sintaxe, ao invés de utilizarmos apenas um operador $\Box$, definimos n operadores $\boxed{1}, \ldots, \boxed{n}$, onde n é a quantidade de agentes considerados.

Note que a lógica modal com apenas um agente pode ser vista como um subconjunto da lógica multimodal, onde $n = 1$. Desta forma, qualquer resultado demonstrado para a lógica multimodal definida acima pode ser facilmente aplicado na lógica modal com apenas um agente.

2.1.2 Sistema modal $\mathbf{K}_{(n)}$

O mais fraco dos sistemas modais normais, conhecido como $\mathrm{K}_{(n)}$, é a extensão da lógica proposicional clássica com os operadores $\boxed{i}$, $1 \leq i \leq n$, no qual o axioma modal $\mathbf{K_i}$ é válido, como veremos à frente ao definirmos a axiomatização para $\mathrm{K}_{(n)}$. No sistema $\mathrm{K}_{(n)}$ não há restrições nas relações de acessibilidade entre os mundos. Como o índice n sugere, consideramos a versão na lógica multimodal, isto é, a que considera vários agentes.

Linguagem de $\mathbf{K}_{(n)}$

Conforme mostrado em [Nalon and Dixon, 2007], as fórmulas em $\mathrm{K}_{(n)}$ são construídas a partir de um conjunto enumerável[1] de *símbolos proposicionais*, também chamados de *fórmulas atômicas*:

$$\mathcal{P} = \{p_0, p_1, p_2, \ldots\},$$

além de um conjunto finito de agentes:

$$\mathcal{A} = \{1, \ldots, n\}.$$

As outras fórmulas são formadas a partir dos símbolos proposicionais, dos operadores proposicionais '$\top$', '$\neg$' e '$\wedge$' e de um conjunto de operadores modais unários $\boxed{1}, \ldots, \boxed{n}$, um

[1]Um conjunto *enumerável* é aquele onde podemos designar um número natural diferente para cada elemento, podendo o conjunto ser infinito ou não.

para cada agente, onde '$\boxed{i}\varphi$' é lido como "o agente i considera φ necessário". Definimos então a linguagem de $\mathrm{K}_{(n)}$.

O conjunto de fórmulas bem-formadas de $\mathrm{K}_{(n)}$, $\mathrm{FBF}_{\mathrm{K}_{(n)}}$, é dado por:

- os símbolos proposicionais estão em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$;
- $\top$ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$;
- se φ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$, então $\neg\varphi$ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$;
- se φ e ψ estão em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$, então $(\varphi \wedge \psi)$ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$;
- se φ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$, então $\boxed{i}\varphi$ está em $\mathrm{FBF}_{\mathrm{K}_{(n)}}$, $\forall i \in \mathcal{A}$.

Normalmente definimos os operadores '$\bot$', '$\vee$', '$\rightarrow$', '$\leftrightarrow$' e '$\diamondsuit$' a partir dos operadores mencionados anteriormente, da seguinte forma:

- $\bot = \neg\top$;
- $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$;
- $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$;
- $(\varphi \leftrightarrow \psi) = ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$; e
- $\diamondsuit\varphi = \neg\boxed{i}\neg\varphi$.

A fórmula '$\diamondsuit\varphi$' é lida como "o agente i considera φ possível". O operador '$\top$' é chamado de *verum* ou ***verdadeiro***, enquanto o operador '$\bot$' é chamado de *falsum* ou ***falso***.

Quando há apenas um agente, ou seja, $n = 1$, podemos omitir o índice do operador modal, isto é, $\Box\varphi = \boxed{1}\varphi$.

Esquemas axiomáticos

Um *esquema axiomático*, ou simplesmente *axioma*, é um conjunto de fórmulas com um formato em particular [Chellas, 1980]. Por exemplo, o axioma 5, dado por:

5. $\Diamond\varphi \rightarrow \Box\Diamond\varphi$

representa todas as fórmulas deste formato — uma implicação, onde à esquerda da implicação temos uma possibilidade de uma fórmula bem-formada qualquer, enquanto à direita temos a necessidade da possibilidade da mesma fórmula.

Utilizando outro exemplo, no esquema K, dado por:

K. $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$

as fórmulas φ e ψ podem ser quaisquer fórmulas bem-formadas, inclusive fórmulas idênticas.

Axiomatização para $\mathbf{K}_{(n)}$

O sistema modal $\mathrm{K}_{(n)}$ contém as tautologias da lógica proposicional[2], além do axioma $\mathbf{K_i}$ e das regras de inferência *modus ponens* e *regra da necessidade*:

i) **Axioma $\mathbf{K_i}$:** $\boxed{i}(\varphi \to \psi) \to (\boxed{i}\varphi \to \boxed{i}\psi), \forall i \in \mathcal{A}$.

ii) **Modus Ponens:** $\dfrac{\varphi \qquad \varphi \to \psi}{\psi}$;

iii) **Regra da Necessidade:** $\dfrac{\varphi}{\boxed{i}\varphi}, \forall i \in \mathcal{A}$; e

Relembrando o conceito de regra de inferência, o entendimento apropriado para modus ponens é: se φ e $\varphi \to \psi$ são teoremas, então ψ também é um teorema. Da mesma forma, para a regra da necessidade temos que se φ é um teorema, então $\boxed{i}\varphi$ também é um teorema, para todo $i \in \mathcal{A}$.[3]

2.2 Semântica

Na Lógica, a semântica analisa a interpretação das fórmulas de linguagens formais. Sendo assim, para definirmos a semântica da Lógica Modal é necessário definir como interpretar os operadores modais. Um conceito bastante utilizado para avaliar o significado das fórmulas modais é o de *mundos possíveis*, apresentado a seguir.

2.2.1 Mundos possíveis

A avaliação de uma fórmula modal depende de um conjunto de interpretações, também chamado conjunto de *mundos possíveis* ou *estados possíveis*. Para entender o conceito de mundos possíveis e sua relação com a lógica modal, considere um conjunto de mundos — incluindo o nosso próprio, o mundo real — onde alguns mundos sejam alternativos a outros (por exemplo, uma alternativa ao nosso mundo seria outro onde o golpe militar de 1964 no Brasil não tenha ocorrido). Denotamos então uma *relação de acessibilidade* entre mundos por $\mathcal{R}$, de forma que se '$(x, y) \in \mathcal{R}$', então "y é um mundo possível dada a informação em x", ou que "y é um mundo que pode ser acessado a partir de x" (outras formas de explicar '$(x, y) \in \mathcal{R}$' utilizadas neste trabalho são "y é acessível a partir de x" e "x pode acessar y").

Como mostrado em [Gabbay et al., 2003], na interpretação de mundos possíveis da lógica modal, cada mundo x se encontra sob as leis da lógica clássica: cada símbolo proposicional é ou verdadeiro ou falso em cada mundo e os valores de verdade das proposições não-modais compostas são determinadas por tabelas-verdade Booleanas.

Dada uma proposição φ, uma fórmula modal '$\Box\varphi$' é verdadeira em x se φ é verdadeira em todos os mundos acessíveis a partir de x; '$\Diamond\varphi$' é verdadeira em x se φ é verdadeira em pelo menos um mundo y acessível a partir de x [Nalon and Dixon, 2007].

[2] As tautologias referidas tratam-se tanto daquelas pertencentes à lógica proposicional clássica quanto de tautologias com modalidades — por exemplo, $\Box q \vee \neg\Box q$ é uma tautologia, pois ela tem a mesma forma de $p \vee \neg p$ [Blackburn et al., 2002].

[3] Um teorema é qualquer fórmula que possa ser provada com bases nos axiomas e regras de inferência (axiomas são automaticamente teoremas) [Chellas, 1980].

2.2.2 Estruturas e modelos de Kripke

Uma vez que temos definido o conceito de mundos possíveis, podemos conceituar as estruturas e modelos da lógica modal que utilizaremos neste trabalho.

Definição 2.1. Uma *estrutura de Kripke para n agentes sobre* $\mathcal{P}$ é uma tupla

$$\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle,$$

onde $\mathcal{S}$ é um conjunto de *mundos* possíveis com um mundo distinto s_0 e cada um dos $\mathcal{R}_i \subseteq \mathcal{S} \times \mathcal{S}$ é uma relação binária em $\mathcal{S}$.

A relação binária $\mathcal{R}_i$ captura a relação de acessibilidade de acordo com o agente i: um par (s, t) está em $\mathcal{R}_i$ se o agente i considera o mundo t possível, dada a informação no mundo s; ou seja, o agente i considera que t é acessível a partir de s.

Definição 2.2. Um *modelo de Kripke para n agentes sobre* $\mathcal{P}$ é uma tupla

$$M = \langle \mathcal{F}, \pi \rangle,$$

onde $\mathcal{F}$ é uma estrutura de Kripke para n agentes e a função $\pi : \mathcal{S} \times \mathcal{P} \to \{\mathrm{V}, \mathrm{F}\}$, é uma interpretação que associa a cada mundo em $\mathcal{S}$ uma valoração para cada símbolo proposicional.

Uma notação equivalente para um modelo de Kripke para n agentes é

$$M = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n, \pi \rangle,$$

onde os elementos de $\mathcal{F}$ são evidenciados sem ser necessária a sua prévia definição.

Neste texto nos referiremos às estruturas de Kripke para n agentes simplesmente como *estruturas de Kripke*. Da mesma forma, a denominação *modelos* será usada para referir-se aos modelos de Kripke para n agentes.

Seja M um modelo e s um mundo em M. Denotamos por $(M, s) \models \varphi$ o fato que a fórmula φ é satisfeita em s. Similarmente, $(M, s) \not\models \varphi$ denota que a fórmula φ não é satisfeita em s.

Valor de verdade de uma fórmula

O valor de verdade de uma fórmula em um mundo s de um modelo M é dado da seguinte forma:

- $(M, s) \models \top$;
- $(M, s) \models p$ se e somente se $\pi(s)(p) =$ verdadeiro, onde $p \in \mathcal{P}$;
- $(M, s) \models \neg\varphi$ se e somente se $(M, s) \not\models \varphi$;
- $(M, s) \models (\varphi \wedge \psi)$ se e somente se $(M, s) \models \varphi$ e $(M, s) \models \psi$;
- $(M, s) \models \boxed{i}\varphi$ se e somente se, para todo t tal que $(s, t) \in \mathcal{R}_i$, $(M, t) \models \varphi$.

As fórmulas são interpretadas relativamente ao mundo distinto s_0. Intuitivamente, s_0 é o mundo inicial, de onde se começa o raciocínio.

Satisfatibilidade e validade

A partir dos conceitos de estrutura de Kripke e modelo, podemos definir *satisfatibilidade* e *validade* de uma fórmula.

Definição 2.3. Uma fórmula φ é dita *satisfatível* se existe um modelo M tal que $(M, s_0) \models \varphi$.

Definição 2.4. Uma fórmula φ é dita *satisfatível em um modelo* M se $(M, s_0) \models \varphi$.

Definição 2.5. Uma fórmula φ é dita *válida* se ela é satisfatível em todos os modelos M.

Definição 2.6. Uma fórmula φ é dita *válida em um modelo* M se, para todos os mundos s em M, $(M, s) \models \varphi$.

Definição 2.7. Uma fórmula φ é dita *válida em uma estrutura de Kripke* $\mathcal{F}$ se, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$, φ é válido em M.

2.3 Outros Sistemas Modais Normais

O método de prova que apresentaremos no Capítulo 3 deste trabalho utiliza um conjunto de sistemas lógicos aos quais chamamos de *sistemas modais normais*. Um sistema modal normal é aquele que possui, no mínimo, a linguagem e axiomatização para $\mathrm{K}_{(n)}$ definida na Seção 2.1.2 — ou seja, o axioma $\mathbf{K_i}$ é válido naquele sistema, ele possui a regra da necessidade, modus ponens e as tautologias da lógica proposicional.

Nesta seção apresentaremos outros sistemas modais normais que estendem $\mathrm{K}_{(n)}$ pela adição, à axiomatização, de outros esquemas axiomáticos.

2.3.1 Axiomas $\mathbf{D_i}$, $\mathbf{T_i}$, $\mathbf{B_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$

Como visto na Seção 2.1.2, nos sistemas modais normais o esquema axiomático $\boxed{i}(\varphi \to \psi) \to (\boxed{i}\varphi \to \boxed{i}\psi)$ (o axioma $\mathbf{K_i}$) é sempre válido, onde φ e ψ são fórmulas bem-formadas. Porém, existem diversos outros axiomas que podem ser inclusos para gerar diferentes sistemas modais normais.

Como mostraremos adiante, os axiomas descritos nesta seção, quando válidos em uma estrutura de Kripke, implicam em determinadas propriedades das suas relações de acessibilidade. Portanto, relembramos aqui algumas das propriedades de uma relação binária $\mathcal{R}$ sobre um conjunto $\mathcal{S}$, que serão utilizadas mais adiante.

Definição 2.8. $\mathcal{R}$ é *serial* se, $\forall s \in \mathcal{S}, \exists t \in \mathcal{S}$ tal que $(s, t) \in \mathcal{R}$.

Definição 2.9. $\mathcal{R}$ é *reflexiva* se, $\forall s \in \mathcal{S}$, temos que $(s, s) \in \mathcal{R}$.

Definição 2.10. $\mathcal{R}$ é *simétrica* se para todo $s, t \in \mathcal{S}$, se $(s, t) \in \mathcal{R}$, então $(t, s) \in \mathcal{R}$.

Definição 2.11. $\mathcal{R}$ é *transitiva* se para todo $s, t, u \in \mathcal{S}$, se $(s, t) \in \mathcal{R}$ e $(t, u) \in \mathcal{R}$, então $(s, u) \in \mathcal{R}$.

Definição 2.12. $\mathcal{R}$ é *euclidiana* se para todo $s, t, u \in \mathcal{S}$, se $(s, t) \in \mathcal{R}$ e $(s, u) \in \mathcal{R}$, então $(t, u) \in \mathcal{R}$.

No método apresentado no Capítulo 3 são considerados sistemas que podem conter qualquer combinação dos cinco axiomas abaixo, além de $\mathbf{K}_i$:

$\mathbf{D}_i$: $\boxed{i}\varphi \to \Diamond_i\varphi$ $\mathbf{B}_i$: $\varphi \to \boxed{i}\Diamond_i\varphi$ $\mathbf{5}_i$: $\Diamond_i\varphi \to \boxed{i}\Diamond_i\varphi$,

$\mathbf{T}_i$: $\boxed{i}\varphi \to \varphi$ $\mathbf{4}_i$: $\boxed{i}\varphi \to \boxed{i}\boxed{i}\varphi$

onde todos os axiomas acima valem para todo $i \in \mathcal{A}$.

O sistema modal mínimo $\mathrm{K}_{(n)}$ foi apresentado na Seção 2.1.2. A adição dos axiomas acima a $\mathrm{K}_{(n)}$ impõe algumas propriedades nas relações de acessibilidade das estruturas de Kripke, restringindo a classe dos modelos onde as fórmulas são válidas [Nalon and Dixon, 2007]. Mostraremos a seguir que cada um dos axiomas mencionados implica em uma propriedade diferente nas relações de acessibilidade.

Axioma $\mathbf{D}_i$

O axioma $\mathbf{D}_i$ é dado por:

$$\mathbf{D}_i : \quad \boxed{i}\varphi \to \Diamond_i\varphi, \forall i \in \mathcal{A}.$$

A inclusão desse axioma restringe a classe de modelos apenas aos modelos *seriais*. Para demonstrar isso, provamos que o axioma $\mathbf{D}_i$ é válido em uma estrutura de Kripke se e somente se tal estrutura é serial.

Teorema 2.1. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se $\mathcal{F}$ é serial, então o axioma $\mathbf{D}_i$ é válido em $\mathcal{F}$.*

Demonstração. Queremos provar que, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$ em uma estrutura de Kripke serial $\mathcal{F}$, temos $(M, s) \models \boxed{i}\varphi \to \Diamond_i\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$.

Para cada agente $i \in \mathcal{A}$ em cada mundo $s \in \mathcal{S}$ existem dois casos possíveis:

i) Se $(M, s) \not\models \boxed{i}\varphi$ então, pela semântica da implicação, $(M, s) \models \boxed{i}\varphi \to \Diamond_i\varphi$.

ii) Se (a) $(M, s) \models \boxed{i}\varphi$, pela semântica de $\boxed{i}$ temos que $\forall s' \in \mathcal{S}$ tal que $(s, s') \in \mathcal{R}_i$, $(M, s') \models \varphi$. Como $\mathcal{F}$ é serial, então $\mathcal{R}_i$ é serial. Portanto, pela definição de relação serial, $\exists t \in \mathcal{S}$ tal que $(s, t) \in \mathcal{R}_i$. Logo, pelo resultado anterior, $(M, t) \models \varphi$. Pela semântica de $\Diamond_i$, (b) $(M, s) \models \Diamond_i\varphi$. Por fim, por (a) e (b) e pela definição de implicação, obtemos $(M, s) \models \boxed{i}\varphi \to \Diamond_i\varphi$.

Assim, temos que $(M, s) \models \boxed{i}\varphi \to \Diamond_i\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como M é um modelo arbitrário em $\mathcal{F}$, conclui-se que o axioma $\mathbf{D}_i$ é válido em $\mathcal{F}$. $\square$

Teorema 2.2. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se o axioma $\mathbf{D}_i$ é válido em $\mathcal{F}$, então $\mathcal{F}$ é serial.*

Demonstração. Seja $\mathcal{F}$ uma estrutura de Kripke onde o axioma $\mathbf{D}_i$ é válido. Vamos demonstrar por contradição que, neste caso, $\mathcal{F}$ é serial.

Suponha que $\mathcal{F}$ não é serial. Pela suposição, para algum agente $i \in \mathcal{A}$, $\mathcal{R}_i$ não é serial. Portanto, pela negação da definição de serialidade, $\exists s \in \mathcal{S}$ tal que $\forall s' \in \mathcal{S}, (s, s') \notin \mathcal{R}_i$.

Seja $M = \langle \mathcal{F}, \pi \rangle$ um modelo qualquer em $\mathcal{F}$ e s um mundo em M tal que $\forall s' \in \mathcal{S}, (s, s') \notin \mathcal{R}_i$, para algum $i \in \mathcal{A}$. Pela semântica do operador $\boxed{i}$, $(M, s) \models \boxed{i}\varphi$; afinal,

como s não tem acesso a qualquer mundo, $\boxed{i}\varphi$ é trivialmente verdadeiro em s. Além disso, pela semântica de $\diamondsuit$ temos que $(M, s) \not\models \diamondsuit\varphi$.

Porém, o axioma $\mathbf{D_i}$ é válido em $\mathcal{F}$. Pela definição de validade em uma estrutura de Kripke e considerando que M é um modelo em $\mathcal{F}$, obtemos $(M, s) \models \boxed{i}\varphi \rightarrow \diamondsuit\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como $(M, s) \models \boxed{i}\varphi$, pela semântica da implicação temos que $(M, s) \models \diamondsuit\varphi$. Porém, já foi verificado que $(M, s) \not\models \diamondsuit\varphi$, uma contradição.

Portanto, nossa suposição está errada e conclui-se que $\mathcal{F}$ é serial. □

Axioma $\mathbf{T_i}$

O axioma $\mathbf{T_i}$ é dado por:

$$\mathbf{T_i}: \quad \boxed{i}\varphi \rightarrow \varphi, \forall i \in \mathcal{A}.$$

A inclusão desse axioma restringe a classe de modelos apenas aos modelos *reflexivos*. Para demonstrar isso, efetuamos procedimento semelhante ao utilizado para $\mathbf{D_i}$: provamos que o axioma $\mathbf{T_i}$ é válido em uma estrutura de Kripke se e somente se tal estrutura é reflexiva.

Teorema 2.3. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se $\mathcal{F}$ é reflexiva, então o axioma $\mathbf{T_i}$ é válido em $\mathcal{F}$.*

Demonstração. Queremos provar que, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$ em uma estrutura de Kripke reflexiva $\mathcal{F}$, temos $(M, s) \models \boxed{i}\varphi \rightarrow \varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$.

Para cada agente $i \in \mathcal{A}$ em cada mundo $s \in \mathcal{S}$ existem dois casos possíveis:

i) Se $(M, s) \not\models \boxed{i}\varphi$ então, pela semântica da implicação, $(M, s) \models \boxed{i}\varphi \rightarrow \varphi$.

ii) Se $(M, s) \models \boxed{i}\varphi$, como $\mathcal{F}$ é reflexiva, então $\mathcal{R}_i$ é reflexiva. Pela definição de relação reflexiva, $(s, s) \in \mathcal{R}_i$. Logo, considerando que $(M, s) \models \boxed{i}\varphi$, pela semântica de $\boxed{i}$ temos $(M, s) \models \varphi$. Por fim, pela definição de implicação, $(M, s) \models \boxed{i}\varphi \rightarrow \varphi$.

Assim, temos que $(M, s) \models \boxed{i}\varphi \rightarrow \varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como M é um modelo arbitrário em $\mathcal{F}$, conclui-se que o axioma $\mathbf{T_i}$ é válido em $\mathcal{F}$. □

Teorema 2.4. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se o axioma $\mathbf{T_i}$ é válido em $\mathcal{F}$, então $\mathcal{F}$ é reflexiva.*

Demonstração. Demonstraremos o teorema por contraposição, ou seja, mostraremos que se $\mathcal{F}$ não é reflexiva, então o axioma $\mathbf{T_i}$ não é válido.

Assuma que $\mathcal{F}$ não é reflexiva. Então, para algum agente $i \in \mathcal{A}$, $\mathcal{R}_i$ não é reflexiva. Portanto, pela negação da definição de reflexividade, $\exists s \in \mathcal{S}$ tal que $(s, s) \notin \mathcal{R}_i$.

Seja $M = \langle \mathcal{F}, \pi \rangle$ um modelo em $\mathcal{F}$ e s um mundo em M tal que $(s, s) \notin \mathcal{R}_i$, para algum $i \in \mathcal{A}$, onde para algum símbolo proposicional p temos $\pi(s)(p) = \text{F}$ e $\pi(s')(p) = \text{V}$, para todo $s' \in \mathcal{S}, s' \neq s$. Assim, $(M, s') \models p$ para todo $s' \in \mathcal{S}, s' \neq s$. Como $(s, s) \notin \mathcal{R}_i$, temos que $(M, s'') \models p$ para todos os mundos s'' tais que $(s, s'') \in \mathcal{R}_i$. Portanto, pela semântica de $\boxed{i}$ obtemos (i) $(M, s) \models \boxed{i}p$. Além disso, pela definição de valor de verdade de uma fórmula, obtemos (ii) $(M, s) \not\models p$. A partir de (i) e (ii) e pela semântica da implicação, obtemos então $(M, s) \not\models \boxed{i}p \rightarrow p$. Portanto, $\mathbf{T_i}$ não é válido no modelo M.

Como M é um modelo de $\mathcal{F}$ e $\mathbf{T_i}$ não é válido em M, concluímos que $\mathbf{T_i}$ não é válido em $\mathcal{F}$. □

Axioma $\mathbf{B_i}$

O axioma $\mathbf{B_i}$ é dado por:

$$\mathbf{B_i}: \quad \varphi \rightarrow \boxed{i}\Diamond\varphi, \forall i \in \mathcal{A}.$$

A inclusão do axioma $\mathbf{B_i}$ restringe a classe de modelos apenas aos modelos *simétricos*. A prova é feita de forma similar às provas para os axiomas anteriores.

Teorema 2.5. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se $\mathcal{F}$ é simétrica, então o axioma $\boldsymbol{B_i}$ é válido em $\mathcal{F}$.*

Demonstração. Queremos provar que, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$ em uma estrutura de Kripke simétrica $\mathcal{F}$, temos $(M, s) \models \varphi \rightarrow \boxed{i}\Diamond\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Suponha então que $\mathcal{F}$ é simétrica; portanto, todas as relações de acessibilidade em M são simétricas, pois M foi construído a partir de $\mathcal{F}$.

Através do raciocínio proposicional e da definição que $\boxed{i}\psi = \neg\Diamond\neg\psi$, podemos reescrever o axioma $\mathbf{B_i}$ como $\Diamond\boxed{i}p \rightarrow p$. Para cada agente $i \in \mathcal{A}$ em cada mundo $s \in \mathcal{S}$, vamos considerar os dois casos.

i) Se $(M, s) \not\models \Diamond\boxed{i}\varphi$, então pela semântica da implicação temos $(M, s) \models \Diamond\boxed{i}\varphi \rightarrow \varphi$.

ii) Se $(M, s) \models \Diamond\boxed{i}\varphi$ então, pela semântica de $\Diamond$, para algum $t \in \mathcal{S}$ tal que $(s, t), \in \mathcal{R}_i$, temos que $(M, t) \models \boxed{i}\varphi$. Como as relações de acessibilidade em M são simétricas, pela definição de relação simétrica temos que se $(s, t) \in \mathcal{R}_i$, então $(t, s) \in \mathcal{R}_i$. Portanto, $(t, s) \in \mathcal{R}_i$. Assim, pela semântica de $\boxed{i}$, se $(M, t) \models \boxed{i}\varphi$ então $(M, s) \models \varphi$. Por fim, pela semântica da implicação obtemos $(M, s) \models \Diamond\boxed{i}\varphi \rightarrow \varphi$.

Assim, temos que $(M, s) \models \Diamond\boxed{i}\varphi \rightarrow \varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como M é um modelo arbitrário em $\mathcal{F}$, conclui-se que o axioma $\mathbf{B_i}$ é válido em $\mathcal{F}$. □

Teorema 2.6. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se o axioma $\boldsymbol{B_i}$ é válido em $\mathcal{F}$, então $\mathcal{F}$ é simétrica.*

Demonstração. Demonstraremos o teorema por contraposição, ou seja, mostraremos que se $\mathcal{F}$ não é simétrica, então o axioma $\mathbf{B_i}$ não é válido.

Assuma que $\mathcal{F}$ não é simétrica. Então, para algum agente $i \in \mathcal{A}$, $\mathcal{R}_i$ não é simétrica. Portanto, pela negação da definição de simetria, $\exists s, t \in \mathcal{S}$ tais que $(s, t) \in \mathcal{R}_i$ e $(t, s) \notin \mathcal{R}_i$.

Seja $M = \langle \mathcal{F}, \pi \rangle$ um modelo em $\mathcal{F}$ e s, t mundos em M tais que $(s, t) \in \mathcal{R}_i$ e $(t, s) \notin \mathcal{R}_i$, para algum $i \in \mathcal{A}$, onde para algum símbolo proposicional p temos $\pi(s)(p) = \text{F}$ e $\pi(s')(p) = \text{V}$, para todo $s' \in \mathcal{S}, s' \neq s$. Assim, $(M, s') \models p$ para todo $s' \in \mathcal{S}, s' \neq s$. Como $(t, s) \notin \mathcal{R}_i$, temos que $(M, t') \models p$ para todos os mundos t' tais que $(t, t') \in \mathcal{R}_i$. Portanto, pela semântica de $\boxed{i}$ temos que $(M, t) \models \boxed{i}p$. Como $(s, t) \in \mathcal{R}_i$, pela semântica de $\Diamond$ obtemos (i) $(M, s) \models \Diamond\boxed{i}p$. A partir da valoração dada, temos também que (ii) $(M, s) \not\models p$. Considerando (i) e (ii) e pela semântica da implicação, obtemos então $(M, s) \not\models \Diamond\boxed{i}p \rightarrow p$. Portanto, $\mathbf{B_i}$ não é válido no modelo M.

Como M é um modelo de $\mathcal{F}$ e $\mathbf{B_i}$ não é válido em M, concluímos que $\mathbf{B_i}$ não é válido em $\mathcal{F}$. □

Axioma $\mathbf{4_i}$

O axioma $\mathbf{4_i}$ é dado por:

$$\mathbf{4_i}: \quad \boxed{i}\varphi \rightarrow \boxed{i}\boxed{i}\varphi, \forall i \in \mathcal{A}.$$

A inclusão do axioma $\mathbf{4_i}$ restringe a classe de modelos apenas aos modelos *transitivos*. A prova é feita de maneira similar às provas para os axiomas anteriores.

Teorema 2.7. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se $\mathcal{F}$ é transitiva, então o axioma $\mathbf{4_i}$ é válido em $\mathcal{F}$.*

Demonstração. Queremos provar que, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$ em uma estrutura de Kripke transitiva $\mathcal{F}$, temos $(M, s) \models \boxed{i}\varphi \rightarrow \boxed{i}\boxed{i}\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Suponha então que $\mathcal{F}$ é transitiva; portanto, todas as relações de acessibilidade em M são transitivas, pois M foi construído a partir de $\mathcal{F}$.

Através do raciocínio proposicional e da definição que $\boxed{i}\psi = \neg\diamondsuit\neg\psi$, podemos reescrever o axioma $\mathbf{4_i}$ como $\diamondsuit\diamondsuit\varphi \rightarrow \diamondsuit\varphi$. Para cada agente $i \in \mathcal{A}$ em cada mundo $s \in \mathcal{S}$, vamos considerar os dois casos.

i) Se $(M, s) \not\models \diamondsuit\diamondsuit\varphi$, então pela semântica da implicação temos $(M, s) \models \diamondsuit\diamondsuit\varphi \rightarrow \diamondsuit\varphi$.

ii) Se $(M, s) \models \diamondsuit\diamondsuit\varphi$ então, pela semântica de $\diamondsuit$, para algum $t \in \mathcal{S}$ tal que $(s, t), \in \mathcal{R}_i$, temos que $(M, t) \models \diamondsuit\varphi$. Pela semântica de $\diamondsuit$, para algum $u \in \mathcal{S}$ tal que $(t, u), \in \mathcal{R}_i$, temos que $(M, u) \models \varphi$. Como as relações de acessibilidade em M são transitivas, pela definição de relação transitiva temos que se $(s, t) \in \mathcal{R}_i$ e $(t, u) \in \mathcal{R}_i$, então $(s, u) \in \mathcal{R}_i$. Portanto, $(s, u) \in \mathcal{R}_i$. A partir disto e como $(M, u) \models \varphi$, então pela semântica de $\diamondsuit$ temos que $(M, s) \models \diamondsuit\varphi$. Por fim, pela semântica da implicação obtemos $(M, s) \models \diamondsuit\diamondsuit\varphi \rightarrow \diamondsuit\varphi$.

Assim, temos que $(M, s) \models \diamondsuit\diamondsuit\varphi \rightarrow \diamondsuit\varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como M é um modelo arbitrário em $\mathcal{F}$, conclui-se que o axioma $\mathbf{B_i}$ é válido em $\mathcal{F}$. □

Teorema 2.8. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se o axioma $\mathbf{4_i}$ é válido em $\mathcal{F}$, então $\mathcal{F}$ é transitiva.*

Demonstração. Demonstraremos o teorema por contraposição, ou seja, mostraremos que se $\mathcal{F}$ não é transitiva, então o axioma $\mathbf{4_i}$ não é válido.

Assuma que $\mathcal{F}$ não é transitiva. Então, para algum agente $i \in \mathcal{A}$, $\mathcal{R}_i$ não é transitiva. Portanto, pela negação da definição de transitividade, $\exists s, t, u \in \mathcal{S}$ tal que $(s, t), (t, u) \in \mathcal{R}_i$ e $(s, u) \notin \mathcal{R}_i$.

Seja $M = \langle \mathcal{F}, \pi \rangle$ um modelo em $\mathcal{F}$ e s, t, u mundos em M tais que $(s, t), (t, u) \in \mathcal{R}_i$ e $(s, u) \notin \mathcal{R}_i$, para algum $i \in \mathcal{A}$, onde para algum símbolo proposicional p temos $\pi(u)(p) =$ V e $\pi(s')(p) =$ F, para todo $s' \in \mathcal{S}, s' \neq u$. Assim, $(M, u) \models p$. Como $(t, u) \in \mathcal{R}_i$, pela semântica de $\diamondsuit$ temos $(M, t) \models \diamondsuit p$. Como $(s, t) \in \mathcal{R}_i$, pela semântica de $\diamondsuit$ temos (i) $(M, s) \models \diamondsuit\diamondsuit p$. Como o símbolo proposicional p é satisfeito apenas no mundo u e pelo fato de $(s, u) \notin \mathcal{R}_i$, então pela semântica de $\diamondsuit$ obtemos (ii) $(M, s) \not\models \diamondsuit p$. A partir de (i) e (ii) e pela semântica da implicação, obtemos então $(M, s) \not\models \diamondsuit\diamondsuit p \rightarrow \diamondsuit p$. Portanto, $\mathbf{4_i}$ não é válido no modelo M.

Como M é um modelo de $\mathcal{F}$ e $\mathbf{4_i}$ não é válido em M, concluímos que $\mathbf{4_i}$ não é válido em $\mathcal{F}$. □

Axioma 5_i

Por último, o axioma $\mathbf{5_i}$ é dado por:

$$\mathbf{5_i}: \quad \Diamond_i \varphi \rightarrow \Box_i \Diamond_i \varphi$$

A inclusão do axioma $\mathbf{5_i}$ restringe a classe de modelos apenas aos modelos *euclidianos*. A prova é feita de forma similar às provas para os axiomas anteriores.

Teorema 2.9. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se $\mathcal{F}$ é euclidiana, então o axioma $\mathbf{5_i}$ é válido em $\mathcal{F}$.*

Demonstração. Queremos provar que, para todo modelo $M = \langle \mathcal{F}, \pi \rangle$ em uma estrutura de Kripke euclidiana $\mathcal{F}$, temos $(M, s) \models \Diamond_i \varphi \rightarrow \Box_i \Diamond_i \varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$.

Para cada agente $i \in \mathcal{A}$ em cada mundo $s \in \mathcal{S}$ existem dois casos possíveis:

i) Se $(M, s) \not\models \Diamond_i \varphi$ então, pela semântica da implicação, $(M, s) \models \Diamond_i \varphi \rightarrow \Box_i \Diamond_i \varphi$.

ii) Se $(M, s) \models \Diamond_i \varphi$, pela semântica de $\Diamond_i$ podemos afirmar que existe um mundo $t \in \mathcal{S}$ tal que $(s, t) \in \mathcal{R}_i$ e $(M, t) \models \varphi$. Como $\mathcal{F}$ é euclidiana, então $\mathcal{R}_i$ é euclidiana. Se $(s, s') \in \mathcal{R}_i$, então pela definição de relação euclidiana e considerando que $(s, t) \in \mathcal{R}_i$, temos que $(s', t) \in \mathcal{R}_i$, onde $s' \in \mathcal{S}$, pois $\mathcal{R}_i$ é euclidiana.

   Através da semântica de $\Diamond_i$, se $(M, t) \models \varphi$ então $(M, s') \models \Diamond_i \varphi, \forall s'$ tal que $(s, s') \in \mathcal{R}_i$. Considerando então que para todos os mundos s' acessíveis a partir de s temos $(M, s') \models \Diamond_i \varphi$, podemos afirmar pela semântica de $\Box_i$ que $(M, s) \models \Box_i \Diamond_i \varphi$. Por fim, pela definição de implicação, $(M, s) \models \Diamond_i \varphi \rightarrow \Box_i \Diamond_i \varphi$

Assim, $(M, s) \models \Diamond_i \varphi \rightarrow \Box_i \Diamond_i \varphi, \forall s \in \mathcal{S}, \forall i \in \mathcal{A}$. Como M é um modelo arbitrário em $\mathcal{F}$, conclui-se que o axioma $\mathbf{5_i}$ é válido em $\mathcal{F}$. □

Teorema 2.10. *Seja $\mathcal{F} = \langle \mathcal{S}, s_0, \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle$ uma estrutura de Kripke. Se o axioma $\mathbf{5_i}$ é válido em $\mathcal{F}$, então $\mathcal{F}$ é euclidiana.*

Demonstração. Demonstraremos o teorema por contraposição, ou seja, mostraremos que se $\mathcal{F}$ não é euclidiana, então o axioma $\mathbf{5_i}$ não é válido.

Assuma que $\mathcal{F}$ não é euclidiana. Então, para algum agente $i \in \mathcal{A}$, $\mathcal{R}_i$ não é euclidiana. Portanto, pela negação da definição de relação euclidiana, $\exists s, t, u \in \mathcal{S}$ tal que $(s, t), (s, u) \in \mathcal{R}_i$ e $(t, u) \notin \mathcal{R}_i$.

Através do raciocínio proposicional e da definição que $\Box_i \psi = \neg \Diamond_i \neg \psi$, podemos reescrever o axioma $\mathbf{5_i}$ como $\Diamond_i \Box_i \varphi \rightarrow \Box_i \varphi$.

Seja $M = \langle \mathcal{F}, \pi \rangle$ um modelo em $\mathcal{F}$ e s, t, u mundos em M tais que $(s, t), (s, u) \in \mathcal{R}_i$ e $(t, u) \notin \mathcal{R}_i$, para algum $i \in \mathcal{A}$, onde para algum símbolo proposicional p temos $\pi(u)(p) = \mathrm{F}$ e $\pi(s')(p) = \mathrm{V}$, para todo $s' \in \mathcal{S}, s' \neq u$. Como o símbolo proposicional p não é satisfeito apenas no mundo u e pelo fato de $(t, u) \notin \mathcal{R}_i$, pela semântica de $\Box_i$ temos que $(M, t) \models \Box_i p$. Como $(s, t) \in \mathcal{R}_i$, pela semântica de $\Diamond_i$ obtemos (i) $(M, s) \models \Diamond_i \Box_i p$. Como $(s, u) \in \mathcal{R}_i$, então pela semântica de $\Box_i$ temos $(M, s) \not\models \Box_i p$. A partir de (i) e (ii) e pela semântica da implicação, obtemos então $(M, s) \not\models \Diamond_i \Box_i p \rightarrow \Box_i p$. Portanto, $\mathbf{5_i}$ não é válido no modelo M.

Como M é um modelo de $\mathcal{F}$ e $\mathbf{5_i}$ não é válido em M, concluímos que $\mathbf{5_i}$ não é válido em $\mathcal{F}$. □

2.3.2 Relacionamentos entre os axiomas

Em alguns casos, devido às propriedades das relações binárias, a validade de um ou mais dos cinco axiomas estudados na seção anterior implica automaticamente na validade de outro. Nesta seção mostraremos os relacionamentos entre esses axiomas, listando quais são os casos onde ocorre essa validação automática de axiomas através da combinação de outros. Tal resultado é importante ao determinar equivalências entre os possíveis sistemas modais gerados pela inclusão desses axiomas.

Teorema 2.11. *Se $\boldsymbol{T_i}$ é válido em uma estrutura de Kripke $\mathcal{F}$, então $\boldsymbol{D_i}$ também é válido em $\mathcal{F}$.*

Demonstração. Como observado no Teorema 2.4, se o axioma $\mathbf{T_i}$ é válido em uma estrutura de Kripke $\mathcal{F}$, então $\mathcal{F}$ é reflexiva.

É trivial observar que toda relação reflexiva é serial: pela definição, se uma relação binária $\mathcal{R}$ em $\mathcal{S}$ é reflexiva, então $(s, s) \in \mathcal{R}, \forall s \in \mathcal{S}$. Portanto, $\exists s' \in \mathcal{S}$ tal que $(s, s') \in \mathcal{R}, \forall s \in \mathcal{S}$; no caso, $s' = s$. Logo, $\mathcal{R}$ também é serial.

Assim, se todas as relações em $\mathcal{F}$ são reflexivas, elas também são todas seriais e $\mathcal{F}$ por sua vez é serial. Por fim, pelo Teorema 2.1, se uma estrutura de Kripke $\mathcal{F}$ é serial, então o axioma $\mathbf{D_i}$ é válido em $\mathcal{F}$. □

Lema 2.12. *Se uma relação binária é reflexiva e euclidiana, ela também é simétrica.*

Demonstração. Seja uma relação binária $\mathcal{R}$ em $\mathcal{S}$, onde $\mathcal{R}$ é reflexiva e euclidiana. Sejam $s, t \in \mathcal{S}$ tais que $(s, t) \in \mathcal{R}$. Pela definição de reflexividade, $(s, s) \in \mathcal{R}$. Como $(s, t), (s, s) \in \mathcal{R}$, pela definição de relação euclidiana temos que $(t, s) \in \mathcal{R}$.

Portanto, se $(s, t) \in \mathcal{R}$ então $(t, s) \in \mathcal{R}$. Logo, conclui-se que $\mathcal{R}$ é simétrica. □

Teorema 2.13. *Se $\boldsymbol{T_i}$ e $\boldsymbol{5_i}$ são válidos em uma estrutura de Kripke $\mathcal{F}$, $\boldsymbol{B_i}$ também é válido em $\mathcal{F}$.*

Demonstração. Seja $\mathcal{F}$ uma estrutura de Kripke onde os axiomas $\mathbf{T_i}$ e $\mathbf{5_i}$ sejam válidos. Pelos Teoremas 2.4 e 2.10, $\mathcal{F}$ é reflexiva e euclidiana. Portanto, todas as suas relações de acessibilidade contêm as duas propriedades mencionadas.

Pelo Lema 2.12, se todas as relações de acessibilidade de $\mathcal{F}$ são reflexivas e euclidianas, elas também são simétricas. Portanto $\mathcal{F}$ é simétrica. Por fim, pelo Teorema 2.5, se uma estrutura de Kripke $\mathcal{F}$ é simétrica, então o axioma $\mathbf{B_i}$ é válido em $\mathcal{F}$. □

Lema 2.14. *Se uma relação binária é simétrica e euclidiana, ela também é transitiva.*

Demonstração. Seja uma relação binária $\mathcal{R}$ em $\mathcal{S}$, onde $\mathcal{R}$ é simétrica e euclidiana. Sejam $s, t, u \in \mathcal{S}$ tais que $(s, t), (t, u) \in \mathcal{R}$. Pela definição de simetria, $(t, s) \in \mathcal{R}$. Como $(t, s), (t, u) \in \mathcal{R}$, pela definição de relação euclidiana temos que $(s, u) \in \mathcal{R}$.

Portanto, se $(s, t), (t, u) \in \mathcal{R}$ então $(s, u) \in \mathcal{R}$. Logo, conclui-se que $\mathcal{R}$ é transitiva. □

Teorema 2.15. *Se $\boldsymbol{B_i}$ e $\boldsymbol{5_i}$ são válidos em uma estrutura de Kripke $\mathcal{F}$, $\boldsymbol{4_i}$ também é válido em $\mathcal{F}$.*

Demonstração. Seja $\mathcal{F}$ uma estrutura de Kripke onde os axiomas $\mathbf{B_i}$ e $\mathbf{5_i}$ sejam válidos. Pelos Teoremas 2.6 e 2.10, $\mathcal{F}$ é simétrica e euclidiana. Portanto, todas as suas relações de acessibilidade contêm as duas propriedades mencionadas.

Pelo Lema 2.14, se todas as relações de acessibilidade de $\mathcal{F}$ são simétricas e euclidianas, elas também são transitivas. Portanto $\mathcal{F}$ é transitiva. Por fim, pelo Teorema 2.7, se uma estrutura de Kripke $\mathcal{F}$ é transitiva, então o axioma $\mathbf{4_i}$ é válido em $\mathcal{F}$. □

Lema 2.16. *Se uma relação binária é simétrica e transitiva, ela também é euclidiana.*

Demonstração. Seja uma relação binária $\mathcal{R}$ em $\mathcal{S}$, onde $\mathcal{R}$ é simétrica e transitiva. Sejam $s, t, u \in \mathcal{S}$ tais que $(s,t),(s,u) \in \mathcal{R}$. Pela definição de simetria, $(t,s) \in \mathcal{R}$. Como $(t,s),(s,u) \in \mathcal{R}$, pela definição de transitividade temos que $(t,u) \in \mathcal{R}$.

Portanto, se $(s,t),(s,u) \in \mathcal{R}$ então $(t,u) \in \mathcal{R}$. Logo, conclui-se que $\mathcal{R}$ é euclidiana. □

Teorema 2.17. *Se $\boldsymbol{B_i}$ e $\boldsymbol{4_i}$ são válidos em uma estrutura de Kripke $\mathcal{F}$, $\boldsymbol{5_i}$ também é válido em $\mathcal{F}$.*

Demonstração. Seja $\mathcal{F}$ uma estrutura de Kripke onde os axiomas $\mathbf{B_i}$ e $\mathbf{4_i}$ sejam válidos. Pelos Teoremas 2.6 e 2.8, $\mathcal{F}$ é simétrica e transitiva. Portanto, todas as suas relações de acessibilidade contêm as duas propriedades mencionadas.

Pelo Lema 2.16, se todas as relações de acessibilidade de $\mathcal{F}$ são simétricas e transitivas, elas também são euclidianas. Portanto $\mathcal{F}$ é euclidiana. Por fim, pelo Teorema 2.9, se uma estrutura de Kripke $\mathcal{F}$ é euclidiana, então o axioma $\mathbf{5_i}$ é válido em $\mathcal{F}$. □

Lema 2.18. *Se uma relação binária é serial, simétrica e transitiva, ela também é reflexiva.*

Demonstração. Seja uma relação binária $\mathcal{R}$ em $\mathcal{S}$, onde $\mathcal{R}$ é serial, simétrica e transitiva. Seja um elemento $s \in \mathcal{S}$ qualquer. Pela definição de serialidade, $\exists t \in \mathcal{S}$ tal que $(s,t) \in \mathcal{R}$. Pela definição de simetria, $(t,s) \in \mathcal{R}$. Como $(s,t),(t,s) \in \mathcal{R}$, pela definição de transitividade temos que $(s,s) \in \mathcal{R}$.

Portanto, $\forall s \in \mathcal{S}$ temos que $(s,s) \in \mathcal{R}$. Logo, conclui-se que $\mathcal{R}$ é reflexiva. □

Teorema 2.19. *Se $\boldsymbol{D_i}$, $\boldsymbol{B_i}$ e $\boldsymbol{4_i}$ são válidos em uma estrutura de Kripke $\mathcal{F}$, $\boldsymbol{T_i}$ também é válido em $\mathcal{F}$.*

Demonstração. Seja $\mathcal{F}$ uma estrutura de Kripke onde os axiomas $\mathbf{D_i}$, $\mathbf{B_i}$ e $\mathbf{4_i}$ sejam válidos. Pelos Teoremas 2.2, 2.6 e 2.8, $\mathcal{F}$ é serial, simétrica e transitiva. Portanto, todas as suas relações de acessibilidade contêm as três propriedades mencionadas.

Pelo Lema 2.18, se todas as relações de acessibilidade de $\mathcal{F}$ são seriais, simétricas e transitivas, elas também são reflexivas. Portanto $\mathcal{F}$ é reflexiva. Por fim, pelo Teorema 2.3, se uma estrutura de Kripke $\mathcal{F}$ é reflexiva, então o axioma $\mathbf{T_i}$ é válido em $\mathcal{F}$. □

2.3.3 Lista de sistemas modais normais

Como visto no início da Seção 2.1.2, todo sistema modal normal possui, no mínimo, a linguagem e a axiomatização para $\mathrm{K}_{(n)}$, ou seja, as regras da lógica proposicional, o axioma $\mathbf{K_i}$, a *regra da necessidade* e o *modus ponens*. Definimos o sistema modal que possui apenas essa axiomatização como $\mathrm{K}_{(n)}$.

No entanto, conforme visto na Seção 2.3.1, a adição de outros esquemas axiomáticos modais a $\mathrm{K}_{(n)}$ gera sistemas modais diferentes, onde cada um deles possui restrições nas

propriedades das suas relações de acessibilidade. Mais especificamente, para os cinco axiomas apresentados naquela seção:

- o axioma $\mathbf{D_i}$ é válido se e somente se as relações de acessibilidade são *seriais*;
- o axioma $\mathbf{T_i}$ é válido se e somente se as relações de acessibilidade são *reflexivas*;
- o axioma $\mathbf{B_i}$ é válido se e somente se as relações de acessibilidade são *simétricas*;
- o axioma $\mathbf{4_i}$ é válido se e somente se as relações de acessibilidade são *transitivas*;
- o axioma $\mathbf{5_i}$ é válido se e somente se as relações de acessibilidade são *euclidianas*.

Note que o sistema $\mathrm{K}_{(n)}$ gera a lógica modal normal mais fraca (menos restritiva). A inclusão, por exemplo, do axioma $\mathbf{T_i}$ em $\mathrm{K}_{(n)}$ gera um novo sistema modal, ao qual chamamos de $\mathrm{KT}_{(n)}$ (pois os axiomas $\mathbf{K_i}$ e $\mathbf{T_i}$ são válidos nesse novo sistema), que possui apenas os modelos de $\mathrm{K}_{(n)}$ que sejam *reflexivos*. Assim, todos os modelos em $\mathrm{KT}_{(n)}$ também estão em $\mathrm{K}_{(n)}$, porém existem modelos em $\mathrm{K}_{(n)}$ que não estão em $\mathrm{KT}_{(n)}$ — o que torna $\mathrm{KT}_{(n)}$ uma lógica mais restritiva.

É possível combinar diversos axiomas para gerar sistemas modais ainda mais restritivos, nos quais em todos os seus modelos valem as propriedades de cada axioma adicionado. Por exemplo, ao adicionar $\mathbf{D_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$ a $\mathrm{K}_{(n)}$ obtemos o sistema $\mathrm{KD45}_{(n)}$, o qual possui apenas modelos que sejam *seriais*, *transitivos* e *euclidianos*.

Pela análise combinatória é possível verificar que existem 32 combinações do axioma $\mathbf{K_i}$ com os axiomas $\mathbf{D_i}$, $\mathbf{T_i}$, $\mathbf{B_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$. Porém, como foi provado na Seção 2.3.2, a validade de certos axiomas em um sistema implica automaticamente na validade de outros.

Conforme mostrado por Chellas [1980], incluindo o próprio $\mathrm{K}_{(n)}$ existem então apenas 15 sistemas modais distintos que podem ser gerados através dos axiomas mostrados anteriormente. Enumeramos a seguir esses 15 sistemas, incluindo seus equivalentes. Em negrito são mostrados os nomes usuais para alguns sistemas modais:

1. $\mathrm{K}_{(n)}$
2. $\mathbf{D_{(n)}}$: $\mathrm{KD}_{(n)}$
3. $\mathbf{T_{(n)}}$: $\mathrm{KT}_{(n)} = \mathrm{KDT}_{(n)}$
4. $\mathrm{KB}_{(n)}$
5. $\mathrm{K4}_{(n)}$
6. $\mathrm{K5}_{(n)}$
7. $\mathrm{KDB}_{(n)}$
8. $\mathrm{KD4}_{(n)}$
9. $\mathrm{KD5}_{(n)}$
10. $\mathrm{KD45}_{(n)}$
11. $\mathrm{K45}_{(n)}$
12. $\mathbf{B_{(n)}}$: $\mathrm{KTB}_{(n)} = \mathrm{KDTB}_{(n)}$
13. $\mathrm{KB4}_{(n)} = \mathrm{KB5}_{(n)} = \mathrm{KB45}_{(n)}$
14. $\mathbf{S4_{(n)}}$: $\mathrm{KT4}_{(n)} = \mathrm{KDT4}_{(n)}$
15. $\mathbf{S5_{(n)}}$: $\mathrm{KT5}_{(n)} = \mathrm{KDT5}_{(n)} = \mathrm{KDB4}_{(n)} = \mathrm{KDB5}_{(n)} = \mathrm{KTB4}_{(n)} = \mathrm{KTB5}_{(n)} = \mathrm{KT45}_{(n)} = \mathrm{KDTB4}_{(n)} = \mathrm{KDTB5}_{(n)} = \mathrm{KDT45}_{(n)} = \mathrm{KDB45}_{(n)} = \mathrm{KTB45}_{(n)} = \mathrm{KDTB45}_{(n)}$

Perceba que o sistema $\mathrm{S5}_{(n)}$ é o mais restritivo da lista, pois nele os cinco axiomas apresentados na Seção 2.3.1 são válidos. A Figura 2.1 mostra um diagrama com os 15 sistemas mencionados. As inclusões entre os sistemas são marcadas por setas: *extensões* de um sistema seguem na direção de cada seta (por exemplo, $\mathrm{T}_{(n)}$ é uma extensão de $\mathrm{D}_{(n)}$).

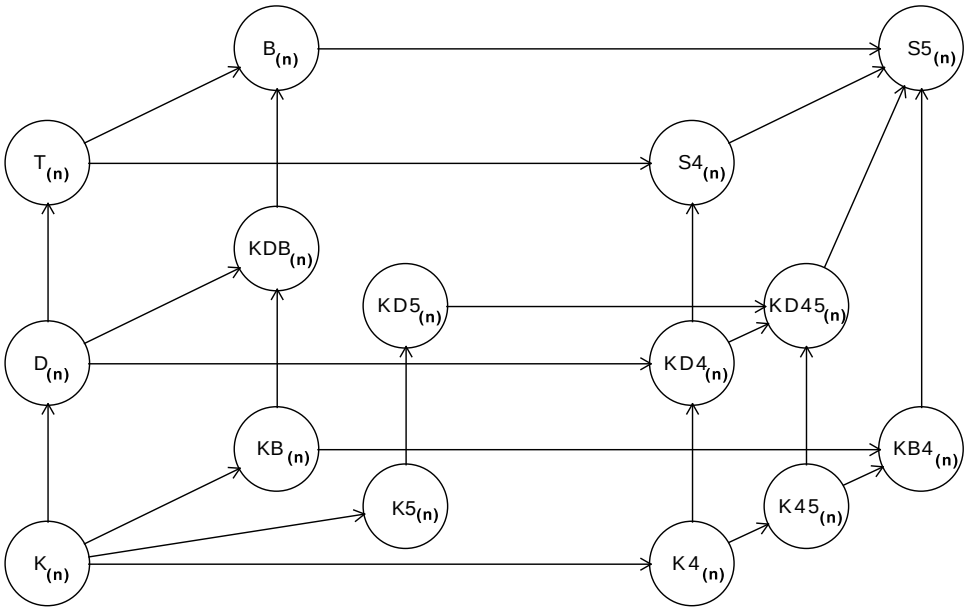

Figura 2.1: Hierarquia entre os sistemas modais normais

Segundo Chellas [1980], historicamente os mais importantes desses sistemas, nas suas formas com apenas um agente, são o **KD**, o **KT**, o **KTB**, o **KT4** e o **KT5**.

Garson [2013] esclarece que na lógica deôntica, que trata sobre o que é obrigatório e o que é permitido, é comum considerar sistemas modais que tenham pelo menos o axioma $\mathbf{D_i}$, para determinar que quando φ é obrigatório, φ é permitido. Devido à importância de **KD** como um sistema básico para a lógica deôntica, ele é usualmente referido simplesmente como **D**. De forma similar, para a lógica que trata sobre necessidade e possibilidade, é comum considerar sistemas modais que tenham pelo menos o axioma $\mathbf{T_i}$, de forma a determinar que se φ é necessário, então é o caso que φ. Desta forma, a importância de **KT** como sistema básico faz com que ele seja usualmente referido apenas como **T**.

Quanto aos outros sistemas, Chellas [1980] comenta que o sistema **KTB** é conhecido como *sistema Brouwersche* (também chamado de **B**), pois o axioma **B** é por vezes chamado de *axioma Brouwersche*, em homenagem a L. E. J. Brouwer. Os outros dois sistemas — **KT4** e **KT5** — são os sistemas **S4** e **S5** apresentados por Lewis em 1932, no seu livro *Symbolic Logic*.

Capítulo 3

Método de Prova

Neste capítulo apresentamos um método de prova de teoremas para sistemas modais normais com vários agentes, desenvolvido por Nalon and Dixon [2007]. Tal método é baseado no procedimento de resolução apresentado por Robinson [1965]. Ao contrário de outras abordagens, onde as regras de inferência são baseadas na sintaxe de um conjunto de axiomas em particular, o método mostrado neste capítulo concentra-se nas restrições impostas nas relações de acessibilidade binárias para cada sistema normal em particular. Estas restrições foram apresentadas nas Seções 2.3.1 a 2.3.3, onde mostramos sistemas modais cujas relações de acessibilidade podem possuir qualquer combinação dentre as seguintes propriedades: serial, reflexiva, simétrica, transitiva e euclidiana.

No método apresentado, temos um conjunto de regras de inferência válidas para todos os modais normais. Além disso, Nalon and Dixon [2007] também definem outras regras de inferência para sistemas que possuam outros axiomas além de $\mathbf{K}_i$: como a inclusão de um novo axioma em um sistema adiciona uma restrição nas relações de acessibilidade, então novas regras de inferência são dadas para capturar cada uma dessas restrições. Assim, o método descrito a seguir pode ser utilizado para qualquer um dos 15 sistemas da lógica multimodal mostrados na Seção 2.3.3, bastando adicionar ao cálculo as regras de inferência referentes aos axiomas presentes naquele sistema.

Primeiramente, faremos uma introdução aos métodos para prova automática de teoremas, que são voltados para a implementação em computadores, com foco em um dos métodos utilizados para esse fim, o de *resolução*. Após essa introdução, descreveremos o método de prova para sistemas modais normais utilizado neste trabalho, que é baseado na resolução proposicional.

3.1 Prova de Teoremas por Resolução

De acordo com Fitting [1990], um método de prova de teoremas é um algoritmo, no sentido matemático, utilizado para provar que uma fórmula é válida. Por ser um algoritmo, ele realiza manipulações mecânicas em uma sequência de símbolos formais, sem que haja necessidade de se definir o significado de cada fórmula. Ao desenvolver um método de prova de teoremas, deve-se verificar que ele:

i) só prova fórmulas válidas (ou seja, é correto);

ii) prova todas as fórmulas válidas (ou seja, é completo); e

iii) retorna um resultado em um número finito de passos (ou seja, é terminante).

Dentre os métodos de prova presentes na literatura, alguns são adequados para a implementação em computadores. A implementação de um método de prova em um computador pode ser útil para verificar a validade de fórmulas extensas ou prover resultados de validade de fórmulas rapidamente. No âmbito da prova automática de teoremas, grande parte dos métodos de prova presentes na literatura não é considerado eficiente para a implementação em computadores, como os *sistemas de Hilbert*, a *dedução natural* e o *cálculo de sequentes* de Gentzen, por exemplo [Fitting, 1990].

Dentre os métodos adequados para a implementação em computadores, alguns dos mais conhecidos são a *resolução* e o *tableaux*. Descrições de cada um desses métodos para a lógica proposicional clássica e para a lógica de primeira ordem são fartas na literatura, tendo sido abordadas por diversos autores, como Bibel [1987]; Fitting [1990]. Considerando que o método de prova implementado neste trabalho é baseado na resolução proposicional, daremos uma introdução à resolução proposicional para que tenhamos uma base para tratar do método de prova para sistemas modais.

Não podemos utilizar a resolução diretamente em toda a sintaxe da lógica; ela deve ser aplicada em uma fórmula que esteja em uma *forma normal*. Essas formas normais têm a vantagem de admitir sistemas de inferência mais simples, proporcionando uma maneira mais eficiente de buscar uma prova.

A prova para uma fórmula proposicional φ através de resolução é baseada na refutação da forma normal de $\neg\varphi$, onde o princípio geral é a prova por contradição. Assim, distinguimos duas fases na prova de uma fórmula φ por resolução:

i) transformar $\neg\varphi$ para uma forma normal — no caso da resolução proposicional utilizamos a *forma normal conjuntiva* (FNC); e

ii) aplicar o método de resolução à fórmula na forma normal.

Vemos a seguir a forma normal conjuntiva, para em seguida ver o método de resolução para a lógica proposicional.

3.1.1 Forma normal conjuntiva

Se as fórmulas proposicionais forem colocadas em um formato padrão, é possível que seja mais fácil determinar sua validade ou sua satisfatibilidade. De fato, a maioria das técnicas para prova automatizada de teoremas converte as fórmulas para algum tipo de forma normal como primeiro passo. Apresentamos nesta seção a forma normal utilizada na resolução proposicional, chamada de forma normal conjuntiva. Primeiramente, mostramos algumas definições sobre conjunção e disjunção de fórmulas, conforme visto em [Fitting, 1990].

Definição 3.1. Seja $\varphi_1, \ldots, \varphi_n$ uma lista de fórmulas proposicionais, onde $n \geq 0$. Definimos dois novos tipos de fórmulas proposicionais:
$\varphi_1 \vee \cdots \vee \varphi_n$ é uma *disjunção* de $\varphi_1, \varphi_2, \ldots, \varphi_n$; e
$\varphi_1 \wedge \cdots \wedge \varphi_n$ é uma *conjunção* de $\varphi_1, \varphi_2, \ldots, \varphi_n$.

No caso de uma lista vazia ($n = 0$), consideramos que uma disjunção vazia é equivalente a $\bot$, enquanto uma conjunção vazia é equivalente a $\top$. Definimos agora *literal* e *cláusula*.

Definição 3.2. Um *literal* é uma fórmula nos formatos p ou $\neg p$, onde p é um símbolo proposicional.

Definição 3.3. Uma *cláusula* é uma disjunção na qual cada membro é um literal.

Uma fórmula proposicional está na forma normal conjuntiva (FNC), também chamada de *forma clausal* ou *conjunto de cláusulas*, se ela é uma conjunção onde cada membro é uma cláusula. Por exemplo, $(\varphi \vee \psi) \wedge (\neg\psi \vee \omega) \wedge \bot$ está na forma normal conjuntiva, onde φ, ψ e ω são fórmulas proposicionais e $\bot$ representa uma cláusula vazia.

Como o método de resolução proposicional só pode ser aplicado a fórmulas na FNC, para que ele possa ser utilizado em todas as fórmulas da lógica proposicional é necessário que haja um algoritmo que traduza qualquer fórmula bem-formada da lógica proposicional para a FNC, mantendo todas as propriedades de satisfatibilidade. Mostramos a seguir um algoritmo que preenche os requisitos mencionados.

Algoritmo de tradução para a FNC. Qualquer fórmula proposicional pode ser convertida para a FNC da forma descrita a seguir, onde φ, ψ e ω são fórmulas proposicionais.

1) Elimine todas as implicações e equivalências da fórmula através das substituições:

   - $\varphi \rightarrow \psi = \neg\varphi \vee \psi$;
   - $\varphi \leftrightarrow \psi = (\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$.

   Eventualmente isso removerá todas as instâncias de '$\rightarrow$' e '$\leftrightarrow$', restando apenas os operadores '$\neg$', '$\wedge$' e '$\vee$'.

2) Mova todos os $\neg$ para dentro dos parênteses aplicando repetidamente as Leis de DeMorgan, e elimine todas as duplas negações:

   - $\neg(\varphi \wedge \psi) = (\neg\varphi) \vee (\neg\psi)$;
   - $\neg(\varphi \vee \psi) = (\neg\varphi) \wedge (\neg\psi)$.
   - $\neg\neg\varphi = \varphi$.

3) Por meio de distribuição, substitua repetidamente todos os casos de $\varphi \vee (\psi \wedge \omega)$ por $(\varphi \vee \psi) \wedge (\varphi \vee \omega)$.

Após o término dessas etapas, a fórmula proposicional estará em sua forma normal conjuntiva.

Ao construir formas normais deve-se considerar a correção do algoritmo, isto é, se a fórmula dada como resultado mantém todas as propriedades de satisfatibilidade da fórmula original; e a sua terminação, isto é, se o algoritmo retorna um resultado em um número finito de passos, para qualquer fórmula dada como entrada. Provas de correção e terminação do algoritmo de tradução para a FNC são facilmente encontradas na literatura e consideramos não ser necessário incluí-las neste texto.

3.1.2 Resolução proposicional

O método de prova por resolução para a lógica proposicional utiliza uma regra de inferência chamada de *resolução*, aplicando-a exaustivamente em todas as cláusulas obtidas após a tradução para a FNC, até que se obtenha uma cláusula vazia ou não seja mais possível aplicar a regra de resolução.

A regra da resolução deve ser aplicada tomando como premissas duas cláusulas, onde l_i, l'_i e l são literais, $1 \leq i \leq n$:

$$\text{[Resolução]} \qquad \frac{\begin{array}{c} l_1 \vee \cdots \vee l_m \vee l \\ l'_1 \vee \cdots \vee l'_n \vee \neg l \end{array}}{l_1 \vee \cdots \vee l_m \vee l'_1 \vee \cdots \vee l'_n}$$

Portanto, se as cláusulas $C_1 = l_1 \vee \cdots \vee l_m \vee l$ e $C_2 = l'_1 \vee \cdots \vee l'_n \vee \neg l$ são satisfatíveis, então a cláusula $C = l_1 \vee \cdots \vee l_m \vee l'_1 \vee \cdots \vee l'_n$ também é satisfatível. Dizemos nesse caso que C_1 e C_2 foram resolvidas em l e chamamos a cláusula C resultante da aplicação da resolução de *resolvente*.

Método de prova por resolução para a lógica proposicional

O método de prova por resolução para uma fórmula proposicional ψ é um algoritmo para encontrar uma contradição em ψ. Portanto, se quisermos provar que ψ é válido, precisamos aplicar o método de prova por resolução a $\neg\psi$ a fim de encontrar uma contradição, o que efetivamente prova a validade de ψ. O método é definido a seguir.

1) Seja uma fórmula proposicional φ. Traduza φ para a FNC, obtendo uma conjunção de cláusulas.

2) Aplique repetidamente a regra da resolução a cada par de cláusulas que possua literais complementares, isto é, l em uma cláusula e $\neg l$ na outra (onde l é um literal), adicionando o resolvente ao conjunto de cláusulas se ele já não estiver presente.

   2.1) Se a aplicação da regra da resolução produzir uma cláusula vazia (ou seja, $\bot$), então φ é uma contradição.

   2.2) Se não houver possibilidade de adicionar uma nova cláusula através da regra da resolução, então φ não é uma contradição; portanto, φ é satisfatível (mas não necessariamente válida).

Note que, ao aplicar a regra da resolução a um par de cláusulas, devemos fazer algumas simplificações a fim de garantir a terminação do método. Sejam $l, l_1, \ldots, l_n$ literais. Se o resolvente for da forma $l_1 \vee \cdots \vee l_n \vee l \vee l$, podemos simplificá-lo para $l_1 \vee \cdots \vee l_n \vee l$. De forma similar, se o resolvente for da forma $l_1 \vee \cdots \vee l_n \vee l \vee \neg l$, então ele é uma tautologia e não é necessário adicioná-lo ao conjunto de cláusulas.

Consideramos não ser necessário para o escopo deste trabalho reproduzir as provas de correção, completude e terminação do método de prova por resolução clássica. Caso necessário, tais provas podem ser consultadas em [Robinson, 1965].

3.1.3 Estratégias completas para resolução proposicional

Mostramos nesta seção estratégias que podem ser utilizadas para encontrar uma contradição mais rapidamente, aumentando a eficiência do método, sem alterar a completude do método — ou seja, as estratégias a seguir preservam a satisfatibilidade do conjunto de cláusulas.

Resolução linear

A *resolução linear*, apresentada por Loveland [1970], é uma estratégia popular para reduzir o espaço de busca por uma prova em métodos de resolução. Essa estratégia utiliza uma derivação linear de um conjunto de cláusulas $\mathcal{S}$, ou seja, uma lista de cláusulas $(C_1, \ldots, C_n)$ onde $C_i \in \mathcal{S}$ e cada C_{i+1} é o resolvente de C_i e B, onde $B \in \mathcal{S}$. Se C_n é a cláusula vazia, então $\{C_1, \ldots, C_n\}$ é uma *refutação linear*. Loveland [1970] prova em seu trabalho que, para todo conjunto de cláusulas insatisfatível, é possível encontrar uma refutação linear; portanto, a resolução linear é completa.

A partir do resultado acima, determina-se um método de resolução linear para encontrar uma contradição em um conjunto de cláusulas $\mathcal{S}$ da seguinte forma:

1. Faça uma derivação linear de $\mathcal{S}$.
2. Se nenhuma contradição for encontrada, faça um *backtrack* e escolha outra cláusula para resolver com a última cláusula gerada (gerando um novo ramo na árvore de prova). Faça uma nova derivação linear.
3. Repita até que seja encontrada uma contradição ou não haja mais cláusulas a resolver.

Subsunção

A subsunção, apresentada por Lee [1967], é amplamente utilizada para efetuar simplificações no método de resolução para a lógica clássica. Em seu trabalho, Kowalski [1970] mostra que, para um conjunto de cláusulas da lógica clássica, é possível eliminar todas as cláusulas que sejam *subsumidas* por outras sem alterar a satisfatibilidade do conjunto.

Buning and Letterman [1999] dão a seguinte definição para subsunção:

Definição 3.4. Sejam C_1 e C_2 duas cláusulas. C_1 *subsume* C_2 (em símbolos, $C_1 \subseteq C_2$) se e somente se todo literal que ocorre em C_1 também ocorre em C_2.

Conforme demonstrado por diversos autores, como Kowalski [1970], na lógica proposicional podemos usar a *regra da subsunção*: sem afetar a satisfatibilidade de uma fórmula $\varphi = \{C_1, \ldots, C_n\}$, onde $C_1, \ldots, C_n$ são cláusulas, podemos excluir toda cláusula $C_i \in \varphi$ para a qual exista uma cláusula $C_j \in \varphi$ tal que $C_j \subseteq C_i$.

Tal resultado nos leva às duas estratégias a seguir, baseadas em subsunção.

Forward subsumption. Ao aplicar a regra da resolução, não adicionamos o resolvente ao conjunto de cláusulas se ele for subsumido por uma cláusula no conjunto de cláusulas.

Backward subsumption. Após adicionar uma cláusula ao conjunto de cláusulas, removemos todas as cláusulas no conjunto de cláusulas que forem subsumidas pela nova cláusula adicionada.

3.2 Forma Normal para Fórmulas em $\mathrm{K}_{(n)}$

As fórmulas na linguagem de $\mathrm{K}_{(n)}$, descrita na Seção 2.1.2, podem ser transformadas para uma forma normal, denominada *Forma Normal Separada para Sistemas Normais* (FNS_K). A forma normal descrita nesta seção foi descrita originalmente por Nalon and Dixon [2007] e é reproduzida neste texto, adaptada para a notação que utilizamos e com algumas regras adicionais para lidar com o operador $\leftrightarrow$, que foi definido na Seção 2.1.2.

Relembramos que a semântica é dada considerando um modelo com um mundo inicial específico, ou seja, a satisfatibilidade é definida em termos do mundo distinto s_0, conforme as definições feitas na Seção 2.2.2. Portanto, é introduzido um novo conectivo nulário, **início**, para poder representar o mundo a partir do qual iniciamos o raciocínio. Formalmente, temos que

$$(M, s) \models \textbf{início} \text{ se e somente se } s = s_0.$$

Uma fórmula na FNS_K é representada como uma conjunção de cláusulas, que são verdadeiras em todos os mundos alcançáveis; ou seja, ela tem a forma geral

$$\Box^* \bigwedge_i A_i$$

onde A_i é uma cláusula. O operador $\Box^*$ é denominado operador *universal*. A fórmula $\Box^*\varphi$ é verdadeira se e somente se φ é verdadeira no mundo atual e em todos os mundos alcançáveis por ele, onde a definição de mundo alcançável é dada abaixo.

Definição 3.5. Seja M um modelo e s, s' mundos em M. Então s' é *alcançável* por s se e somente se:

i) $(s, s') \in \mathcal{R}_i$ para algum agente $i \in \mathcal{A}$; ou

ii) existe um mundo s'' em M tal que s'' é alcançável por s e s' é alcançável por s''.

O operador universal, que engloba todas as cláusulas, assegura que a tradução de uma fórmula para a FNS_K é verdadeira no mundo atual e em todos os mundos alcançáveis.

Antes de mostrar quais são os possíveis formatos para cláusulas na FNS_K, primeiro apresentaremos a definição de *literal modal*.

Definição 3.6. Um *literal modal* é uma fórmula nos formatos $\boxdot_i l$ ou $\neg\boxdot_i l$, onde l é um literal e $i \in \mathcal{A}$.

Relembramos que a definição de literal foi dada na Seção 3.1.1: um literal é um símbolo proposicional ou sua negação. Tendo esses conceitos definidos, as cláusulas na FNS_K podem ter um dos seguintes formatos:

- Cláusula inicial: $\textbf{início} \rightarrow \bigvee_{b=1}^{r} l_b$
- Cláusula literal: $\top \rightarrow \bigvee_{b=1}^{r} l_b$
- Cláusula i-positiva: $l' \rightarrow \boxdot_i l$

- Cláusula i-negativa: $\quad l' \rightarrow \neg\boxed{i}l$

onde $l, l', l_b \in \mathcal{L}$. Cláusulas i-positivas e i-negativas são denominadas conjuntamente como *cláusulas i-modais*; o índice pode ser omitido se estiver claro a partir do contexto.

A tradução para a FNS_K usa a técnica de renomeação, onde subfórmulas complexas são substituídas por novos símbolos proposicionais e o valor de verdade destes novos símbolos é ligado à fórmula que eles substituíram em todos os mundos. A tradução para a FNS_K de uma fórmula φ na linguagem de $\mathrm{K}_{(n)}$ é dada pelas funções de transformação τ_0 e τ_1, descritas adiante, onde f é um novo símbolo proposicional:

$$\tau_0(\varphi) = \Box^*(\textbf{início} \rightarrow f) \wedge \tau_1\big(\Box^*(f \rightarrow \varphi)\big). \tag{3.1}$$

A função τ_0 é usada para colocar o mundo inicial como referência para o significado de φ, pois é no mundo inicial que a fórmula é avaliada.

A intenção ao final do processo de tradução é ter uma conjunção de cláusulas, onde cada cláusula pode ser inicial, literal, i-positiva ou i-negativa. Para isso é preciso que cada cláusula tenha, no lado direito da implicação, uma disjunção de literais ou um único literal modal. Assim, a função τ_1 transforma operadores proposicionais em conjunções e disjunções, por meio de operações de reescrita clássicas, e substitui fórmulas complexas que estejam dentro do escopo do operador $\boxed{i}$ ou de disjunções, por meio de renomeação.

As regras de transformação a seguir lidam com a dupla negação, com os operadores clássicos '$\leftrightarrow$', '$\rightarrow$', '$\wedge$' e suas negações e com a negação de '$\vee$' (onde φ e ψ são fórmulas):

$$\tau_1\big(\Box^*(x \rightarrow \neg\neg\varphi)\big) = \tau_1\big(\Box^*(x \rightarrow \varphi)\big) \tag{3.2}$$
$$\tau_1\big(\Box^*\big(x \rightarrow (\varphi \leftrightarrow \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \neg\varphi \vee \psi)\big) \wedge \tau_1\big(\Box^*(x \rightarrow \neg\psi \vee \varphi)\big) \tag{3.3}$$
$$\tau_1\big(\Box^*\big(x \rightarrow (\varphi \rightarrow \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \neg\varphi \vee \psi)\big) \tag{3.4}$$
$$\tau_1\big(\Box^*\big(x \rightarrow (\varphi \wedge \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \varphi)\big) \wedge \tau_1\big(\Box^*(x \rightarrow \psi)\big) \tag{3.5}$$
$$\tau_1\big(\Box^*\big(x \rightarrow \neg(\varphi \leftrightarrow \psi)\big)\big) = \tau_1\big(\Box^*\big(x \rightarrow (\varphi \wedge \neg\psi) \vee (\psi \wedge \neg\varphi)\big)\big) \tag{3.6}$$
$$\tau_1\big(\Box^*\big(x \rightarrow \neg(\varphi \rightarrow \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \varphi)\big) \wedge \tau_1\big(\Box^*(x \rightarrow \neg\psi)\big) \tag{3.7}$$
$$\tau_1\big(\Box^*\big(x \rightarrow \neg(\varphi \wedge \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \neg\varphi \vee \neg\psi)\big) \tag{3.8}$$
$$\tau_1\big(\Box^*\big(x \rightarrow \neg(\varphi \vee \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \neg\varphi)\big) \wedge \tau_1\big(\Box^*(x \rightarrow \neg\psi)\big) \tag{3.9}$$

Caso o lado direito da implicação tenha uma disjunção como operador principal e um dos operandos da disjunção seja da forma $\varphi \rightarrow \psi$ ou $\neg(\varphi \wedge \psi)$, onde φ e ψ são fórmulas, o operando é transformado em uma disjunção através das seguintes regras de reescrita (onde δ é uma disjunção de fórmulas):

$$\tau_1\big(\Box^*\big(x \rightarrow \delta \vee (\varphi \rightarrow \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \delta \vee \neg\varphi \vee \psi)\big) \tag{3.10}$$
$$\tau_1\big(\Box^*\big(x \rightarrow \delta \vee \neg(\varphi \wedge \psi)\big)\big) = \tau_1\big(\Box^*(x \rightarrow \delta \vee \neg\varphi \vee \neg\psi)\big) \tag{3.11}$$

Porém, se um dos operandos da disjunção não for um literal e nem uma fórmula em um dos formatos acima, renomeia-se esse operando, substituindo-o por um novo literal e criando uma nova cláusula através da seguinte regra (onde y é um novo símbolo proposicional, δ é uma disjunção de fórmulas e φ não é um literal, nem tem uma implicação ou

a negação de uma conjunção como operador principal):

$$\tau_1\big(\Box^*(x \to \delta \vee \varphi)\big) = \tau_1\big(\Box^*(x \to \delta \vee y)\big) \wedge \tau_1\big(\Box^*(y \to \varphi)\big) \tag{3.12}$$

As seguintes regras lidam com os operadores modais que contenham uma subfórmula complexa dentro de seu escopo, renomeando essa subfórmula e criando uma nova cláusula para a obtenção de um literal modal (onde y é um novo símbolo proposicional e φ não é um literal):

$$\tau_1\big(\Box^*(x \to \boxed{i}\varphi)\big) = \tau_1\big(\Box^*(x \to \boxed{i}y)\big) \wedge \tau_1\big(\Box^*(y \to \varphi)\big) \tag{3.13}$$
$$\tau_1\big(\Box^*(x \to \neg\boxed{i}\varphi)\big) = \tau_1\big(\Box^*(x \to \neg\boxed{i}\neg y)\big) \wedge \tau_1\big(\Box^*(y \to \neg\varphi)\big) \tag{3.14}$$

Após a aplicação exaustiva das regras acima, note que dentro do escopo de τ_1 haverão apenas cláusulas cujo lado direito da implicação é um literal modal ou uma disjunção de literais. Perceba que após a aplicação das regras (3.13) e (3.14) as cláusulas modais já estão no formato de uma cláusula i-positiva ou i-negativa, não necessitando qualquer transformação. Por fim, as fórmulas cujo lado direito seja uma disjunção de literais são reescritas, colocando-as no formato de cláusulas literais.

$$\tau_1\big(\Box^*(x \to \delta)\big) = \begin{cases} \Box^*(\top \to \neg x \vee \delta) & \text{se } \delta \text{ é uma disjunção de literais,} \quad (3.15) \\ \Box^*(x \to \delta) & \text{se } \delta \text{ é um literal modal.} \quad (3.16) \end{cases}$$

Como cada cláusula modal contém apenas um literal modal com um operador $\boxed{i}$, $i \in \mathcal{A}$, os contextos referentes a cada agente já estão separados ao final da tradução. As cláusulas literais, por sua vez, formam as proposições satisfeitas em todos os mundos.

A prova de correção do método de tradução para a FNS_K pode ser vista em [Nalon and Dixon, 2007].

Exemplo 3.1. Seja φ a fórmula $\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)$. Mostramos aqui como traduzir φ para a sua forma normal. Chamaremos os novos símbolos proposicionais gerados durante o processo de tradução de t_i, onde $i \in \mathbb{N}$.

Primeiramente, colocamos o mundo inicial como referência para φ:

$$\tau_0(\varphi) = \Box^*(\textbf{início} \to t_1) \wedge \tau_1\big(\Box^*\big(t_1 \to \boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)\big)\big)$$

e a tradução prossegue da seguinte forma:

$$\begin{aligned}
&\tau_1\big(\Box^*\big(t_1 \to \boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)\big)\big) \\
&\quad= \tau_1\big(\Box^*\big(t_1 \to \neg\boxed{1}(a \to b) \vee (\boxed{1}a \to \boxed{1}b)\big)\big) \\
&\quad= \tau_1\big(\Box^*\big(t_1 \to t_2 \vee (\boxed{1}a \to \boxed{1}b)\big)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to \neg\boxed{1}(a \to b)\big)\big) \\
&\quad= \tau_1\big(\Box^*(t_1 \to t_2 \vee \neg\boxed{1}a \vee \boxed{1}b)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to \neg\boxed{1}(a \to b)\big)\big) \\
&\quad= \tau_1\big(\Box^*(t_1 \to t_2 \vee t_3 \vee \boxed{1}b)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to \neg\boxed{1}(a \to b)\big)\big) \\
&\qquad\wedge \tau_1\big(\Box^*(t_3 \to \neg\boxed{1}a)\big) \\
&\quad= \tau_1\big(\Box^*(t_1 \to t_2 \vee t_3 \vee t_4)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to \neg\boxed{1}(a \to b)\big)\big) \\
&\qquad\wedge \tau_1\big(\Box^*(t_3 \to \neg\boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_4 \to \boxed{1}b)\big)
\end{aligned}$$

$$
\begin{aligned}
&= \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \tau_1\big(\Box^*\big(t_2 \to \neg\boxed{1}(a \to b)\big)\big) \\
&\quad \wedge \tau_1\big(\Box^*(t_3 \to \neg\boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_4 \to \boxed{1}b)\big) \\
&= \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \tau_1\big(\Box^*(t_2 \to \neg\boxed{1}\neg t_5)\big) \wedge \tau_1\big(\Box^*(t_3 \to \neg\boxed{1}a)\big) \\
&\quad \wedge \tau_1\big(\Box^*(t_4 \to \boxed{1}b)\big) \wedge \tau_1\big(\Box^*\big(t_5 \to \neg(a \to b)\big)\big) \\
&= \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \Box^*(t_2 \to \neg\boxed{1}\neg t_5) \wedge \Box^*(t_3 \to \neg\boxed{1}a) \\
&\quad \wedge \Box^*(t_4 \to \boxed{1}b) \wedge \tau_1\big(\Box^*\big(t_5 \to \neg(a \to b)\big)\big) \\
&= \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \Box^*(t_2 \to \neg\boxed{1}\neg t_5) \wedge \Box^*(t_3 \to \neg\boxed{1}a) \\
&\quad \wedge \Box^*(t_4 \to \boxed{1}b) \wedge \tau_1\big(\Box^*(t_5 \to a)\big) \wedge \tau_1\big(\Box^*(t_5 \to \neg b)\big) \\
&= \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \Box^*(t_2 \to \neg\boxed{1}\neg t_5) \wedge \Box^*(t_3 \to \neg\boxed{1}a) \\
&\quad \wedge \Box^*(t_4 \to \boxed{1}b) \wedge \Box^*(\top \to \neg t_5 \vee a) \wedge \Box^*(\top \to \neg t_5 \vee \neg b)
\end{aligned}
$$

O processo de tradução resulta então em 7 cláusulas — uma inicial, três literais e três modais:

$$
\begin{aligned}
\tau_0(\varphi) = \Box^*(\textbf{início} \to t_1) &\wedge \Box^*(\top \to \neg t_1 \vee t_2 \vee t_3 \vee t_4) \wedge \Box^*(t_2 \to \neg\boxed{1}\neg t_5) \\
&\wedge \Box^*(t_3 \to \neg\boxed{1}a) \wedge \Box^*(t_4 \to \boxed{1}b) \wedge \Box^*(\top \to \neg t_5 \vee a) \wedge \Box^*(\top \to \neg t_5 \vee \neg b)
\end{aligned}
$$

Exemplo 3.2. Façamos agora o processo de tradução para a negação da fórmula φ acima, ou seja, para $\neg\varphi = \neg(\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b))$. Tal resultado será utilizado para exemplificar o método de prova na Seção 3.3.

Iniciamos com τ_0:

$$
\tau_0(\neg\varphi) = \Box^*(\textbf{início} \to t_1) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg\big(\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)\big)\big)\big)
$$

e a tradução prossegue com τ_1:

$$
\begin{aligned}
&\tau_1\big(\Box^*\big(t_1 \to \neg\big(\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)\big)\big)\big) \\
&\quad = \tau_1\big(\Box^*\big(t_1 \to \boxed{1}(a \to b)\big)\big) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg(\boxed{1}a \to \boxed{1}b)\big)\big) \\
&\quad = \tau_1\big(\Box^*(t_1 \to \boxed{1}t_2)\big) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg(\boxed{1}a \to \boxed{1}b)\big)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to (a \to b)\big)\big) \\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg(\boxed{1}a \to \boxed{1}b)\big)\big) \wedge \tau_1\big(\Box^*\big(t_2 \to (a \to b)\big)\big) \\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \tau_1\big(\Box^*(t_1 \to \boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_1 \to \neg\boxed{1}b)\big) \\
&\qquad \wedge \tau_1\big(\Box^*\big(t_2 \to (a \to b)\big)\big) \\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \neg\boxed{1}b) \wedge \tau_1\big(\Box^*\big(t_2 \to (a \to b)\big)\big) \\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \neg\boxed{1}b) \wedge \tau_1\big(\Box^*(t_2 \to \neg a \vee b)\big) \\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \neg\boxed{1}b) \wedge \Box^*(\top \to \neg t_2 \vee \neg a \vee b)
\end{aligned}
$$

Resultando em 5 cláusulas — uma inicial, uma literal e três modais:

$$
\begin{aligned}
\tau_0(\neg\varphi) = \Box^*(\textbf{início} \to t_1) &\wedge \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \neg\boxed{1}b) \\
&\wedge \Box^*(\top \to \neg t_2 \vee \neg a \vee b)
\end{aligned}
$$

3.3 Regras de Inferência para $\mathrm{K}_{(n)}$

A partir desta seção introduzimos o método de prova utilizado neste trabalho, baseado no método de resolução, apresentado originalmente por Nalon and Dixon [2007]. Nesta seção mostramos as regras de inferência para $\mathrm{K}_{(n)}$ e apresentamos alguns exemplos do seu uso. No trabalho de Nalon and Dixon [2007] também é possível consultar as provas de correção, completude e terminação de cada uma das regras de inferência introduzidas nesta seção.

Através destas regras é possível encontrar uma contradição para qualquer conjunto insatisfatível de cláusulas na FNS_K. Para outros sistemas modais, além das regras mostradas nesta seção também é necessário adicionar outras regras de inferência que capturem as restrições adicionais nas relações de acessibilidade, conforme mostrado na Seção 3.4.

Nas regras mostradas a seguir, $l, l', l_i, l'_i \in \mathcal{L}$ $(i \in \mathbb{N})$; δ, δ' são disjunções de literais.

Resolução de literais

Estas regras correspondem à resolução clássica, aplicada à porção proposicional da lógica combinada. Uma cláusula inicial pode ser resolvida tanto com uma cláusula literal quanto com outra cláusula inicial (IRES1 e IRES2).

$$\text{[IRES1]}\quad \frac{\Box^*(\top \to \delta \vee l) \qquad \Box^*(\mathbf{início} \to \delta' \vee \neg l)}{\Box^*(\mathbf{início} \to \delta \vee \delta')} \qquad \text{[IRES2]}\quad \frac{\Box^*(\mathbf{início} \to \delta \vee l) \qquad \Box^*(\mathbf{início} \to \delta' \vee \neg l)}{\Box^*(\mathbf{início} \to \delta \vee \delta')}$$

Podem ser resolvidas também duas cláusulas literais (LRES):

$$\text{[LRES]}\quad \frac{\Box^*(\top \to \delta \vee l) \qquad \Box^*(\top \to \delta' \vee \neg l)}{\Box^*(\top \to \delta \vee \delta')}$$

Resolução modal

Estas regras são aplicadas entre cláusulas que se referem ao mesmo contexto; isto é, elas devem se referir ao mesmo agente. Por exemplo, pode-se resolver duas ou mais cláusulas i-modais (MRES e GEN2); ou várias cláusulas i-modais e uma cláusula literal (GEN1 e GEN3). As regras de inferência modais são:

$$\text{[MRES]}\quad \frac{\Box^*(l_1 \to \boxed{i} l) \qquad \Box^*(l_2 \to \neg\boxed{i} l)}{\Box^*(\top \to \neg l_1 \vee \neg l_2)} \qquad \text{[GEN2]}\quad \frac{\Box^*(l'_1 \to \boxed{i} l_1) \qquad \Box^*(l'_2 \to \boxed{i} \neg l_1) \qquad \Box^*(l'_3 \to \neg\boxed{i} \neg l_2)}{\Box^*(\top \to \neg l'_1 \vee \neg l'_2 \vee \neg l'_3)}$$

$$\text{[GEN1]}\quad \frac{\begin{array}{c}\Box^*(l'_1 \to \boxed{i}\neg l_1)\\ \vdots\\ \Box^*(l'_m \to \boxed{i}\neg l_m)\\ \Box^*(l' \to \neg\boxed{i}\neg l)\\ \Box^*(\top \to l_1 \vee \cdots \vee l_m \vee \neg l)\end{array}}{\Box^*(\top \to \neg l'_1 \vee \cdots \vee \neg l'_m \vee \neg l')} \qquad \text{[GEN3]}\quad \frac{\begin{array}{c}\Box^*(l'_1 \to \boxed{i}\neg l_1)\\ \vdots\\ \Box^*(l'_m \to \boxed{i}\neg l_m)\\ \Box^*(l' \to \neg\boxed{i}\neg l)\\ \Box^*(\top \to l_1 \vee \cdots \vee l_m)\end{array}}{\Box^*(\top \to \neg l'_1 \vee \cdots \vee \neg l'_m \vee \neg l')}$$

MRES pode ser facilmente compreendida, pois ela é equivalente à resolução proposicional clássica — uma fórmula e sua negação não podem ser verdadeiras no mesmo mundo.

A regra GEN1 corresponde à regra da necessidade e várias aplicações da resolução clássica: como as cláusulas são verdadeiras em todos os mundos, a cláusula literal implica em $\boxed{i}(l_1 \vee \cdots \vee l_m \vee \neg l)$. Pela semântica da implicação, podemos reescrever a última fórmula como $\boxed{i}(\neg l_1 \to (l_2 \vee \cdots \vee l_m \vee \neg l))$. Como estamos em $\mathrm{K}_{(n)}$, o axioma $\mathbf{K_i}$ é válido; então aplicamos $\mathbf{K_i}$ para obter $\boxed{i}\neg l_1 \to \boxed{i}(l_2 \vee \cdots \vee l_m \vee \neg l)$. Pela semântica da implicação, obtemos então $\neg\boxed{i}\neg l_1 \vee \boxed{i}(l_2 \vee \cdots \vee l_m \vee \neg l)$. Repetindo o procedimento para cada um dos outros literais l_i, chegamos a $\neg\boxed{i}\neg l_1 \vee \cdots \vee \neg\boxed{i}\neg l_m \vee \boxed{i}\neg l$. A partir daí, os literais modais desta fórmula podem ser resolvidos, um por um, com seus complementos nas outras cláusulas. GEN2 é um caso especial de GEN1, pois podemos incluir a tautologia $\Box^*(\top \to l_1 \vee \neg l_1 \vee \neg l_2)$ como uma quarta cláusula, transformando-a em uma instância de GEN1.

Para GEN3, a cláusula i-negativa implica em $l' \to \neg\boxed{i}\neg l$. Nos mundos em que l' é falso, o resolvente é trivialmente verdadeiro. Para cada mundo s onde l' é verdadeiro, então $\neg\boxed{i}\neg l$ é verdadeiro e, portanto, aquele mundo tem acesso a pelo menos um mundo. Para cada mundo s' acessível por s, a cláusula literal implica em $l_1 \vee \cdots \vee l_m$. Assim, pela semântica de $\boxed{i}$, em cada mundo s temos $\neg\boxed{i}\neg l_1 \vee \cdots \vee \neg\boxed{i}\neg l_m$. A partir daí verifica-se pela definição de disjunção que para algum $k, 1 \leq k \leq m$, $\neg\boxed{i}\neg l_k$ é verdadeiro em s. Assim, $\boxed{i}\neg l_k$ é falso em s; pela semântica da implicação e considerando a k-ésima cláusula de GEN3, temos que l'_k também é falso em s. A partir daí é trivial obter o resolvente em GEN3.

Simplificação

Assumimos no método de resolução as simplificações padrão da lógica proposicional clássica para manter as cláusulas tão simples quanto possível. Por exemplo, $\delta \vee l \vee l$ no lado direito de uma cláusula inicial ou literal seria reescrita como $\delta \vee l$. Além disso, o método também não introduz quaisquer cláusulas repetidas no conjunto de cláusulas e tautologias.

Exemplo 3.3. Provamos que a fórmula $\varphi = \boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b)$, uma aplicação do axioma $\mathbf{K_i}$, é válida em $\mathrm{K}_{(1)}$. Para tal, mostramos que $\neg\varphi$ é uma contradição — portanto, insatisfatível — aplicando o método de prova acima. Primeiramente, traduzimos $\neg\varphi = \neg(\boxed{1}a \to \neg\boxed{1}\neg a)$ para a FNS_K.

A transformação para a FNS_K já foi mostrada no Exemplo 3.2 e resulta em:

$$\begin{aligned}\tau_0(\neg\varphi) = {} & \Box^*(\textbf{início} \to t_1) \wedge \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \neg\boxed{1}b)\\ & \wedge \Box^*(\top \to \neg t_2 \vee \neg a \vee b)\end{aligned}$$

Portanto, assim segue a resolução para $\neg\varphi$:

1.	$\textbf{início} \to t_1$	
2.	$t_1 \to \boxed{1}t_2$	
3.	$t_1 \to \boxed{1}a$	
4.	$t_1 \to \neg\boxed{1}b$	
5.	$\top \to \neg t_2 \vee \neg a \vee b$	
6.	$\top \to \neg t_1$	[2, 3, 4, 5, GEN1]
7.	$\textbf{início} \to \bot$	[6, 1, IRES1]

Exemplo 3.4. Mostramos aqui um exemplo com dois agentes que utiliza a regra GEN3. Seja $\varphi = \boxed{1}\boxed{2}(a \to \neg a) \to (\boxed{1}\boxed{2}a \to \boxed{1}\boxed{2}b)$. Primeiramente transformamos $\neg\varphi$ para a forma normal:

$$
\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \to t_1) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg\big(\boxed{1}\boxed{2}(a \to \neg a) \to (\boxed{1}\boxed{2}a \to \boxed{1}\boxed{2}b)\big)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \to \neg\big(\boxed{1}\boxed{2}(a \to \neg a) \to (\boxed{1}\boxed{2}a \to \boxed{1}\boxed{2}b)\big)\big)\big)\\
&\quad = \tau_1\big(\Box^*\big(t_1 \to \boxed{1}\boxed{2}(a \to \neg a)\big)\big) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg(\boxed{1}\boxed{2}a \to \boxed{1}\boxed{2}b)\big)\big)\\
&\quad = \tau_1\big(\Box^*(t_1 \to \boxed{1}t_2)\big) \wedge \tau_1\big(\Box^*(t_1 \to \boxed{1}\boxed{2}a)\big) \wedge \tau_1\big(\Box^*(t_1 \to \neg\boxed{1}\boxed{2}b)\big)\\
&\qquad \wedge \tau_1\big(\Box^*\big(t_2 \to \boxed{2}(a \to \neg a)\big)\big)\\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \tau_1\big(\Box^*(t_1 \to \boxed{1}t_3)\big) \wedge \tau_1\big(\Box^*(t_1 \to \neg\boxed{1}\neg t_4)\big)\\
&\qquad \wedge \tau_1\big(\Box^*(t_2 \to \boxed{2}t_5)\big) \wedge \tau_1\big(\Box^*(t_3 \to \boxed{2}a)\big) \wedge \tau_1\big(\Box^*(t_4 \to \neg\boxed{2}b)\big)\\
&\qquad \wedge \tau_1\big(\Box^*\big(t_5 \to (a \to \neg a)\big)\big)\\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}t_3) \wedge \Box^*(t_1 \to \neg\boxed{1}\neg t_4) \wedge \Box^*(t_2 \to \boxed{2}t_5)\\
&\qquad \wedge \Box^*(t_3 \to \boxed{2}a) \wedge \Box^*(t_4 \to \neg\boxed{2}b) \wedge \tau_1\big(\Box^*(t_5 \to \neg a)\big)\\
&\quad = \Box^*(t_1 \to \boxed{1}t_2) \wedge \Box^*(t_1 \to \boxed{1}t_3) \wedge \Box^*(t_1 \to \neg\boxed{1}\neg t_4) \wedge \Box^*(t_2 \to \boxed{2}t_5)\\
&\qquad \wedge \Box^*(t_3 \to \boxed{2}a) \wedge \Box^*(t_4 \to \neg\boxed{2}b) \wedge \Box^*(\top \to \neg t_5 \vee \neg a)
\end{aligned}
$$

E então aplicamos as regras de inferência às cláusulas:

1.	$\textbf{início} \to t_1$	
2.	$t_1 \to \boxed{1}t_2$	
3.	$t_1 \to \boxed{1}t_3$	
4.	$t_1 \to \neg\boxed{1}\neg t_4$	
5.	$t_2 \to \boxed{2}t_5$	
6.	$t_3 \to \boxed{2}a$	
7.	$t_4 \to \neg\boxed{2}b$	
8.	$\top \to \neg t_5 \vee \neg a$	
9.	$\top \to \neg t_2 \vee \neg t_3 \vee \neg t_4$	[5, 6, 7, 8, GEN3]
10.	$\top \to \neg t_1$	[2, 3, 4, 9, GEN1]
11.	$\textbf{início} \to \bot$	[10, 1, IRES1]

3.4 Regras de Inferência para outros Sistemas Modais Normais

Como visto na Seção 2.3.3, considerando os axiomas $\mathbf{D_i}$, $\mathbf{T_i}$, $\mathbf{B_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$, existem 15 sistemas modais normais. Cada um deles apresenta propriedades nas suas relações de acessibilidade de acordo com os axiomas que são válidos em seus modelos. Conforme vimos na Seção 2.3.1, a validade dos axiomas $\mathbf{D_i}$, $\mathbf{T_i}$, $\mathbf{B_i}$, $\mathbf{4_i}$ e $\mathbf{5_i}$ confere às relações de acessibilidade as propriedades *serial*, *reflexiva*, *simétrica*, *transitiva* e *euclidiana*, respectivamente, que restringem a classe dos modelos de um sistema apenas àqueles que possuam tais propriedades. Além disso, a presença de algumas combinações das propriedades acima nas relações de acessibilidade de um modelo implica na presença de outras propriedades, como visto na Seção 2.3.2.

A seguir são introduzidas regras de inferência que adicionam cláusulas ao cálculo que capturam essas restrições nas relações de acessibilidade. Estas regras de inferência devem ser aplicadas juntamente com IRES1, IRES2, LRES, MRES, GEN1, GEN2 e GEN3 a um conjunto de cláusulas na FNS_K até que uma contradição seja encontrada ou não seja possível gerar novas cláusulas. Todas as regras de inferência introduzidas nesta seção foram apresentadas originalmente por Nalon and Dixon [2007]. Na mesma referência podem ser vistas as provas de correção, terminação e completude para cada uma das regras a seguir.

Combinando cada uma dessas regras, que individualmente capturam as restrições nas relações de acessibilidade, obtém-se cálculos de resolução corretos e completos para cada um dos 15 sistemas modais normais. Por exemplo, para termos um cálculo de resolução correto e completo para $\mathrm{KD45}_{(n)}$, basta utilizarmos um processo de resolução que contenha as sete regras de inferência da resolução para $\mathrm{K}_{(n)}$ adicionadas das regras para sistemas seriais, para sistemas transitivos e para sistemas euclidianos.

3.4.1 Regra para sistemas seriais

A regra de inferência SER é aplicada a uma cláusula i-positiva, gerando uma cláusula i-negativa.

$$[\mathrm{SER}] \quad \frac{\Box^*(l_1 \to \boxed{i}l)}{\Box^*(l_1 \to \neg\boxed{i}\neg l)}$$

Esta regra aplica o axioma $\mathbf{D_i}$ às cláusulas i-positivas. Em qualquer modelo M *serial*, pelo axioma $\mathbf{D_i}$ temos que $(M,s) \models \boxed{i}l \to \neg\boxed{i}\neg l$ para todos os mundos s em M, onde l é um literal. Se $\Box^*(l_1 \to \boxed{i}l)$, então pela semântica de $\Box^*$, $(M,s) \models l_1 \to \boxed{i}l, \forall s \in \mathcal{S}$. Portanto, utilizando o raciocínio da lógica proposicional clássica, se $(M,s) \models l_1 \to \boxed{i}l$ e $(M,s) \models \boxed{i}l \to \neg\boxed{i}\neg l$, então $(M,s) \models l_1 \to \neg\boxed{i}\neg l$, $\forall s \in \mathcal{S}$. Pela semântica de $\Box^*$, temos então que $\Box^*(l_1 \to \neg\boxed{i}\neg l)$.

A prova formal para a correção, completude e terminação de SER pode ser vista em [Nalon and Dixon, 2007].

Exemplo 3.5. Note que ao incluir SER às regras para $\mathrm{K}_{(n)}$, obtemos o cálculo para o sistema $\mathrm{D}_{(n)}$. Mostramos então que, incluindo a regra de inferência SER, podemos

demonstrar que a fórmula $\varphi = \boxed{1}a \rightarrow \neg\boxed{1}\neg a$, uma aplicação do axioma $\mathbf{D_i}$, é válida em $\mathrm{D}_{(1)}$. Primeiramente, traduzimos $\neg\varphi = \neg(\boxed{1}a \rightarrow \neg\boxed{1}\neg a)$ para a FNS_K.

$$\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \rightarrow t_1) \wedge \tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow \neg\boxed{1}\neg a)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow \neg\boxed{1}\neg a)\big)\big)\\
&\qquad = \tau_1\big(\Box^*(t_1 \rightarrow \boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg\neg\boxed{1}\neg a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \boxed{1}a) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \boxed{1}\neg a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \boxed{1}a) \wedge \Box^*(t_1 \rightarrow \boxed{1}\neg a)
\end{aligned}$$

Aplicamos agora as regras de inferência às cláusulas para derivar a contradição.

1.	$\textbf{início} \rightarrow t_1$	
2.	$t_1 \rightarrow \boxed{1}a$	
3.	$t_1 \rightarrow \boxed{1}\neg a$	
4.	$t_1 \rightarrow \neg\boxed{1}\neg a$	[2, SER]
5.	$\top \rightarrow \neg t_1$	[3, 4, MRES]
6.	$\textbf{início} \rightarrow \bot$	[5, 1, IRES1]

3.4.2 Regra para sistemas reflexivos

A regra de inferência REF é aplicada a uma cláusula i-positiva, gerando uma cláusula literal.

$$[\text{REF}] \quad \frac{\Box^*(l_1 \rightarrow \boxed{i}l)}{\Box^*(\top \rightarrow \neg l_1 \vee l)}$$

Esta regra aplica o axioma $\mathbf{T_i}$ às cláusulas i-positivas. Em qualquer modelo M *reflexivo*, pelo axioma $\mathbf{T_i}$ temos que $(M, s) \models \boxed{i}l \rightarrow l$ para todos os mundos s em M, onde l é um literal. Se $\Box^*(l_1 \rightarrow \boxed{i}l)$, então pela semântica de $\Box^*$, $(M, s) \models l_1 \rightarrow \boxed{i}l, \forall s \in \mathcal{S}$. Portanto, utilizando o raciocínio da lógica proposicional clássica, se $(M, s) \models l_1 \rightarrow \boxed{i}l$ e $(M, s) \models \boxed{i}l \rightarrow l$, então $(M, s) \models l_1 \rightarrow l$, $\forall s \in \mathcal{S}$. Pela semântica de $\Box^*$, temos então que $\Box^*(l_1 \rightarrow l)$. Por fim, pela semântica da implicação, $\Box^*(\top \rightarrow \neg l_1 \vee l)$.

A prova formal para a correção, completude e terminação de REF pode ser vista em [Nalon and Dixon, 2007].

Exemplo 3.6. Mostramos que, ao incluir a regra REF no cálculo de resolução, podemos demonstrar que $\varphi = \boxed{1}a \rightarrow a$ é válido. Primeiramente, traduzimos $\neg\varphi = \neg(\boxed{1}a \rightarrow a)$ para a FNS_K.

$$\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \rightarrow t_1) \wedge \tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow a)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow a)\big)\big)\\
&\qquad = \tau_1\big(\Box^*(t_1 \rightarrow \boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \boxed{1}a) \wedge \Box^*(\top \rightarrow \neg t_1 \vee \neg a)
\end{aligned}$$

Aplicamos agora as regras de inferência às cláusulas para derivar a contradição.

1. **início** $\to t_1$
2. $t_1 \to \boxed{1}a$
3. $\top \to \neg t_1 \vee \neg a$

---

4. $\top \to \neg t_1 \vee a$ [2, REF]
5. $\top \to \neg t_1$ [3, 4, LRES]
6. **início** $\to \bot$ [5, 1, IRES1]

Exemplo 3.7. Mostramos também que, ao incluir a regra REF no cálculo de resolução, podemos demonstrar que $\varphi = \boxed{1}a \to \neg\boxed{i}\neg a$ é válido como esperado, pois todo sistema reflexivo é também serial. A tradução de $\neg\varphi$ para a FNS_K já foi feita no Exemplo 3.5, e resulta em:

$$\tau_0(\neg\varphi) = \Box^*(\textbf{início} \to t_1) \wedge \Box^*(t_1 \to \boxed{1}a) \wedge \Box^*(t_1 \to \boxed{1}\neg a)$$

Aplicamos agora as regras de inferência às cláusulas para derivar a contradição.

1. **início** $\to t_1$
2. $t_1 \to \boxed{1}a$
3. $t_1 \to \boxed{1}\neg a$

---

4. $\top \to \neg t_1 \vee a$ [2, REF]
5. $\top \to \neg t_1 \vee \neg a$ [3, REF]
6. $\top \to \neg t_1$ [4, 5, LRES]
7. **início** $\to \bot$ [6, 1, IRES1]

3.4.3 Regra para sistemas simétricos

A regra de inferência SYM é aplicada a uma cláusula i-positiva, gerando uma cláusula i-positiva.

$$\text{[SYM]} \quad \frac{\Box^*(l_1 \to \boxed{i}\neg l)}{\Box^*(l \to \boxed{i}\neg l_1)}$$

Esta regra aplica a propriedade de simetria a todas as relações de acessibilidade de todos os agentes $i \in \mathcal{A}$. Se $\Box^*(l_1 \to \boxed{i}\neg l)$ está no conjunto de cláusulas, então considere um mundo s em um modelo M tal que $(M, s) \models l$.

i) Se s não tem acesso a qualquer mundo pelo agente i, então trivialmente $(M, s) \models \boxed{i}\neg l_1$.

ii) Se existe um mundo t tal que $(s, t) \in \mathcal{R}_i$, como M e simétrico, $(t, s) \in \mathcal{R}_i$. Como $\Box^*(l_1 \to \boxed{i}\neg l)$, pela semântica de $\Box^*$ temos $(M, t) \models l_1 \to \boxed{i}\neg l$. Suponha que $(M, t) \models l_1$; então, pela semântica de $\boxed{i}$ e como $(t, s) \in \mathcal{R}_i$, $(M, s) \models \neg l$. Mas como já foi definido anteriormente que $(M, s) \models l$, então verifica-se que $(M, t) \not\models l_1$, $\forall t \in \mathcal{S}$ tal que $(s, t) \in \mathcal{R}_i$. Assim, pela semântica de $\boxed{i}$, $(M, s) \models \neg l_1$.

A partir dos casos acima e pela semântica da implicação e de $\Box^*$, fica claro que se $\Box^*(l_1 \to \boxed{i}\neg l)$ em um modelo simétrico, então $\Box^*(l \to \boxed{i}\neg l_1)$, como determinado pela regra SYM.

A prova formal para a correção, completude e terminação de SYM pode ser vista em [Nalon and Dixon, 2007].

Exemplo 3.8. Mostramos que, ao incluir a regra SYM no cálculo de resolução, podemos demonstrar que $\varphi = a \to \boxed{1}\neg\boxed{1}\neg a$ é válido. Primeiramente, traduzimos $\neg\varphi = \neg(a \to \boxed{1}\neg\boxed{1}\neg a)$ para a FNS_K.

$$\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \to t_1) \wedge \tau_1\big(\Box^*\big(t_1 \to \neg(a \to \boxed{1}\neg\boxed{1}\neg a)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \to \neg(a \to \boxed{1}\neg\boxed{1}\neg a)\big)\big)\\
&\qquad = \tau_1\big(\Box^*(t_1 \to a)\big) \wedge \tau_1\big(\Box^*(t_1 \to \neg\boxed{1}\neg\boxed{1}\neg a)\big)\\
&\qquad = \Box^*(\top \to \neg t_1 \vee a) \wedge \tau_1\big(\Box^*(t_1 \to \neg\boxed{1}\neg t_2)\big) \wedge \tau_1\big(\Box^*(t_2 \to \boxed{1}\neg a)\big)\\
&\qquad = \Box^*(\top \to \neg t_1 \vee a) \wedge \Box^*(t_1 \to \neg\boxed{1}\neg t_2) \wedge \Box^*(t_2 \to \boxed{1}\neg a)
\end{aligned}$$

Aplicamos então as regras de inferência às cláusulas para derivar a contradição.

1.	$\textbf{início} \to t_1$	
2.	$\top \to \neg t_1 \vee a$	
3.	$t_1 \to \neg\boxed{1}\neg t_2$	
4.	$t_2 \to \boxed{1}\neg a$	
5.	$a \to \boxed{1}\neg t_2$	[4, SYM]
6.	$\top \to \neg t_1 \vee \neg a$	[3, 5, MRES]
7.	$\top \to \neg t_1$	[2, 6, LRES]
8.	$\textbf{início} \to \bot$	[7, 1, IRES1]

3.4.4 Regra para sistemas transitivos

Há apenas uma regra de inferência para os sistemas transitivos, TRANS, que é aplicada a uma cláusula i-positiva, gerando três cláusulas como conclusão: uma cláusula literal e duas cláusulas i-positivas.

$$\text{[TRANS]}\quad \frac{\Box^*(l_1 \to \boxed{i}l)}{\begin{array}{l}\Box^*(\top \to \neg l_1 \vee nec_{i,l})\\ \Box^*(nec_{i,l} \to \boxed{i}l)\\ \Box^*(nec_{i,l} \to \boxed{i}nec_{i,l})\end{array}}$$

Para sistemas modais transitivos temos que estender o conjunto de símbolos proposicionais. Para cada literal l que apareça no conjunto original de cláusulas e para cada agente $i \in \mathcal{A}$, precisamos de um novo símbolo proposicional $nec_{i,l}$. Note que esta regra não deve ser aplicada a cláusulas i-positivas cujo lado direito da implicação seja $\boxed{i}nec_{i,l}$ ou $\boxed{i}pos_{i,l}$ (no caso do sistema também ser euclidiano; veja a próxima subseção). Aplicar TRANS a $\Box^*(nec_{i,l} \to \boxed{i}l)$ resulta em duas cláusulas que já estão no conjunto de cláusulas (mais especificamente os dois últimos resolventes) e uma tautologia. Aplicar TRANS a $\Box^*(nec_{i,l} \to \boxed{i}nec_{i,l})$ requer a criação de novos símbolos proposicionais (por exemplo, $nec_{i,nec_{i,l}}$), o que é problemático para a terminação e não é necessário para a completude. Portanto, aplicar TRANS às cláusulas mencionadas não é permitido.

Intuitivamente, esse símbolo proposicional é um novo nome para $\boxed{i}l$ — a primeira cláusula gerada efetua a substituição na cláusula original, a segunda ancora $nec_{i,l}$ como um novo nome para $\boxed{i}l$ e a terceira aplica o axioma $\mathbf{4_i}$ para l: $\boxed{i}l \rightarrow \boxed{i}\boxed{i}l$. O processo de renomear $\boxed{i}l$ é feito pois não podemos ter um literal modal à esquerda da implicação nas cláusulas da FNS_K, decisão que simplifica o método de prova como um todo.

A prova formal para a correção, completude e terminação de TRANS pode ser vista em [Nalon and Dixon, 2007].

Exemplo 3.9. Ao incluir a regra TRANS no cálculo de resolução, podemos demonstrar que $\varphi = \boxed{1}a \rightarrow \boxed{1}\boxed{1}a$ é válido. Primeiramente, traduzimos $\neg\varphi = \neg(\boxed{1}a \rightarrow \boxed{1}\boxed{1}a)$ para a FNS_K.

$$
\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \rightarrow t_1) \wedge \tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow \boxed{1}\boxed{1}a)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\boxed{1}a \rightarrow \boxed{1}\boxed{1}a)\big)\big)\\
&\qquad = \tau_1\big(\Box^*(t_1 \rightarrow \boxed{1}a)\big) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg\boxed{1}\boxed{1}a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \boxed{1}a) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg\boxed{1}\neg t_2)\big) \wedge \tau_1\big(\Box^*(t_2 \rightarrow \neg\boxed{1}a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \boxed{1}a) \wedge \Box^*(t_1 \rightarrow \neg\boxed{1}\neg t_2) \wedge \Box^*(t_2 \rightarrow \neg\boxed{1}a)
\end{aligned}
$$

Aplicamos então as regras de inferência às cláusulas para derivar a contradição.

1.	$\textbf{início} \rightarrow t_1$	
2.	$t_1 \rightarrow \boxed{1}a$	
3.	$t_1 \rightarrow \neg\boxed{1}\neg t_2$	
4.	$t_2 \rightarrow \neg\boxed{1}a$	
5.	$\top \rightarrow \neg t_1 \vee nec_{1,a}$	[2, TRANS]
6.	$nec_{1,a} \rightarrow \boxed{1}a$	[2, TRANS]
7.	$nec_{1,a} \rightarrow \boxed{1}nec_{1,a}$	[2, TRANS]
8.	$\top \rightarrow \neg t_2 \vee \neg nec_{1,a}$	[4, 6, MRES]
9.	$\top \rightarrow \neg t_1 \vee \neg nec_{1,a}$	[7, 3, 8, GEN1]
10.	$\top \rightarrow \neg t_1$	[5, 9, LRES]
11.	$\textbf{início} \rightarrow \bot$	[10, 1, IRES1]

3.4.5 Regras para sistemas euclidianos

Temos duas regras de inferência para sistemas euclidianos, EUC1 e EUC2. A primeira, EUC1, é aplicada a uma cláusula i-negativa e gera quatro cláusulas: uma cláusula literal, uma cláusula i-negativa e duas cláusulas i-positivas. A segunda, EUC2, é aplicada a uma cláusula i-positiva e gera também quatro cláusulas: uma cláusula i-negativa e três cláusulas i-positivas.

$$
\text{[EUC1]}\quad \frac{\Box^*(l_1 \rightarrow \neg\boxed{i}\neg l)}{\begin{array}{r}\Box^*(\top \rightarrow \neg l_1 \vee pos_{i,l})\\ \Box^*(pos_{i,l} \rightarrow \neg\boxed{i}\neg l)\\ \Box^*(\neg pos_{i,l} \rightarrow \boxed{i}\neg l)\\ \Box^*(pos_{i,l} \rightarrow \boxed{i}pos_{i,l})\end{array}}
\qquad
\text{[EUC2]}\quad \frac{\Box^*(l_1 \rightarrow \boxed{i}l)}{\begin{array}{r}\Box^*(pos_{i,l_1} \rightarrow \boxed{i}l)\\ \Box^*(pos_{i,l_1} \rightarrow \neg\boxed{i}\neg l_1)\\ \Box^*(\neg pos_{i,l_1} \rightarrow \boxed{i}\neg l_1)\\ \Box^*(pos_{i,l_1} \rightarrow \boxed{i}pos_{i,l_1})\end{array}}
$$

Para sistemas euclidianos também estendemos o conjunto de proposições. Para cada literal l que apareça no conjunto original de cláusulas e para cada agente $i \in \mathcal{A}$, precisamos de um novo símbolo proposicional $pos_{i,l}$. A regra EUC1 não deve ser aplicada a cláusulas cujo lado direito da implicação seja $\neg\boxed{i}\neg pos_{i,l}$ ou $\neg\boxed{i}\neg nec_{i,l}$. Além disso, a regra EUC2 não deve ser aplicada a cláusulas cujo lado esquerdo da implicação seja $pos_{i,l}$ ou $nec_{i,l}$ Aplicar a regra a cláusulas destes formatos não é incorreto, porém resulta em tautologias ou cláusulas que não são necessárias para a completude do cálculo, além de causar problemas para a terminação.

Intuitivamente, esse símbolo proposicional é um novo nome para $\neg\boxed{i}\neg l$, isto é, $\diamondsuit l$. Na primeira regra, EUC1, a primeira cláusula gerada efetua a substituição na cláusula original, a segunda e a terceira cláusulas ancoram $pos_{i,l}$ como um novo nome para $\neg\boxed{i}\neg l$, enquanto a última aplica o axioma $\mathbf{5_i}$ para l: $\diamondsuit l \rightarrow \boxed{i}\diamondsuit l$.

Em EUC2, é necessário manipular a cláusula $\Box^*(l_1 \rightarrow \boxed{i}l)$ para chegar ao resultado. Considere que $\Box^*(l_1 \rightarrow \boxed{i}l)$ está no conjunto de cláusulas. Pelo raciocínio proposicional, temos $\Box^*(\neg\boxed{i}l \rightarrow \neg l_1)$. Pela semântica de $\Box^*$, $\Box^*(\boxed{i}(\neg\boxed{i}l \rightarrow \neg l_1))$. Através do axioma $\mathbf{K_i}$, obtemos $\Box^*(\boxed{i}\neg\boxed{i}l \rightarrow \boxed{i}\neg l_1)$. Então, pelo raciocínio proposicional, $\Box^*(\neg\boxed{i}\neg l_1 \rightarrow \neg\boxed{i}\neg\boxed{i}l)$. É possível reescrever o axioma $\mathbf{5_i}$ como $\diamondsuit\boxed{i}\varphi \rightarrow \boxed{i}\varphi$, através do raciocínio proposicional e da definição de $\diamondsuit$. Então, pelo axioma $\mathbf{5_i}$ verificamos que o lado direito da implicação pode ser simplificado para $\boxed{i}l$. Logo, $\Box^*(\neg\boxed{i}\neg l_1 \rightarrow \boxed{i}l)$. Desta forma, utilizamos pos_{i,l_1} para substituir $\neg\boxed{i}\neg l_1$. Assim, a primeira cláusula gerada por EUC2 efetua a substituição na cláusula original, enquanto as outras três cláusulas efetuam o mesmo papel que em EUC1.

A prova formal para a correção, completude e terminação de EUC1 e EUC2 podem ser vistas em [Nalon and Dixon, 2007].

Exemplo 3.10. Incluindo as regras EUC1 e EUC2 no cálculo de resolução, podemos demonstrar que $\varphi = \neg\boxed{1}\neg a \rightarrow \boxed{1}\neg\boxed{1}\neg a$ é válido. Primeiramente, traduzimos $\neg\varphi = \neg(\neg\boxed{1}\neg a \rightarrow \boxed{1}\neg\boxed{1}\neg a)$ para a FNS_K.

$$
\begin{aligned}
&\tau_0(\neg\varphi) = \Box^*(\textbf{início} \rightarrow t_1) \wedge \tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\neg\boxed{1}\neg a \rightarrow \boxed{1}\neg\boxed{1}\neg a)\big)\big)\\
&\tau_1\big(\Box^*\big(t_1 \rightarrow \neg(\neg\boxed{1}\neg a \rightarrow \boxed{1}\neg\boxed{1}\neg a)\big)\big)\\
&\qquad = \tau_1\big(\Box^*(t_1 \rightarrow \neg\boxed{1}\neg a)\big) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg\boxed{1}\neg\boxed{1}\neg a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \neg\boxed{1}\neg a) \wedge \tau_1\big(\Box^*(t_1 \rightarrow \neg\boxed{1}\neg t_2)\big) \wedge \tau_1\big(\Box^*(t_2 \rightarrow \boxed{1}\neg a)\big)\\
&\qquad = \Box^*(t_1 \rightarrow \neg\boxed{1}\neg a) \wedge \Box^*(t_1 \rightarrow \neg\boxed{1}\neg t_2) \wedge \Box^*(t_2 \rightarrow \boxed{1}\neg a)
\end{aligned}
$$

Aplicamos então as regras de inferência às cláusulas para derivar a contradição.

1.	$\textbf{início} \rightarrow t_1$	
2.	$t_1 \rightarrow \neg\boxed{1}\neg a$	
3.	$t_1 \rightarrow \neg\boxed{1}\neg t_2$	
4.	$t_2 \rightarrow \boxed{1}\neg a$	
5.	$\top \rightarrow \neg t_1 \vee pos_{i,t_2}$	[3, EUC1]
6.	$pos_{i,t_2} \rightarrow \neg\boxed{1}\neg t_2$	[3, EUC1]
7.	$\neg pos_{i,t_2} \rightarrow \boxed{1}\neg t_2$	[3, EUC1]
8.	$pos_{i,t_2} \rightarrow \boxed{1}pos_{i,t_2}$	[3, EUC1]
9.	$pos_{i,t_2} \rightarrow \boxed{1}\neg a$	[4, EUC2]

10. $\top \rightarrow \neg t_1 \vee \neg pos_{i,t_2}$ [2, 9, MRES]
11. $\top \rightarrow \neg t_1$ [5, 10, LRES]
12. **início** $\rightarrow \bot$ [11, 1, IRES1]

Note que, ao aplicar EUC2 na cláusula 4, os últimos três resolventes já haviam sido gerados anteriormente pela aplicação de EUC1 na cláusula 3.

3.5 Subsunção no Método de Resolução Modal

Pretendemos mostrar que a aplicação da subsunção de cláusulas literais no método de resolução modal mostrado neste capítulo preserva a completude do método, mantendo a satisfatibilidade do conjunto de cláusulas. Neste trabalho não utlizamos a subsunção na parte modal da linguagem.

Para o método de resolução apresentado por Nalon and Dixon [2007] mostramos inicialmente que, dado um conjunto de cláusulas Σ, se existe uma cláusula $C_1 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n)$ em Σ, então podemos excluir de Σ qualquer cláusula $C_2 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n \vee l)$ sem modificar a completude do método, onde $l_1, \ldots, l_n, l$ são literais.

Lema 3.1. *Seja Σ um conjunto de cláusulas. Se Σ é insatisfatível e $C_1 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n)$, $C_2 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n \vee l_0)$ são cláusulas em Σ, então $\Sigma - \{C_2\}$ é insatisfatível.*

Demonstração. Seja Σ um conjunto de cláusulas insatisfatível e $C_1 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n)$, $C_2 = \Box^*(\top \rightarrow l_1 \vee \cdots \vee l_n \vee l_0)$ cláusulas em Σ, onde $l_0, l_1, \ldots, l_n$ são literais.

Para as regras IRES1 e LRES, a completude após a exclusão de C_2 do conjunto de cláusulas é assegurada pela completude da subsunção aplicada à lógica proposicional [Kowalski, 1970]. As regras IRES2, MRES, GEN2, REF, SER, TRANS, EUC1, EUC2 e SYM não utilizam cláusulas literais, portanto a completude delas não é afetada com a exclusão de C_2 do conjunto de cláusulas. Portanto, nos resta provar que a exclusão de C_2 mantém a completude de GEN1 e GEN3.

1. Se C_2 pode ser utilizada em GEN1, então Σ tem n cláusulas i-positivas $\Box^*(l''_1 \rightarrow \boxed{i}\neg l'_1)$, $\ldots$, $\Box^*(l''_n \rightarrow \boxed{i}\neg l'_n)$, $n \geq 0$, e uma cláusula i-negativa $\Box^*(l''_0 \rightarrow \neg\boxed{i} l'_0)$, onde todo l'_j é igual a algum l_i, $0 \leq i, j \leq n$. A aplicação de GEN1 em C_2 juntamente com estas cláusulas produz o resolvente $\Box^*(\top \rightarrow \neg l''_0 \vee \cdots \vee \neg l''_n)$. Consideramos então dois casos.

   (a) Se $l_0 = l'_0$, então podemos aplicar GEN3 a C_1 juntamente com as n cláusulas i-positivas e a cláusula i-negativa mencionadas acima e obter o mesmo resolvente: $\Box^*(\top \rightarrow \neg l''_0 \vee \cdots \vee \neg l''_n)$. Assim, podemos retirar C_2 de Σ sem alterar a satisfatibilidade do método, pois conseguimos gerar as mesmas cláusulas sem ele.

   (b) Se $l_0 = l'_k$, $1 \leq k \leq n$, então podemos aplicar GEN1 a C_1 juntamente com as cláusulas i-positivas, exceto por $\Box^*(l''_k \rightarrow \boxed{i}\neg l'_k)$, e a cláusula i-negativa e obter o resolvente $\Box^*(\top \rightarrow \neg l''_0 \vee \cdots \vee \neg l''_{k-1} \vee \neg l''_{k+1} \vee \cdots \vee \neg l''_n)$. Como o resolvente gerado por C_2 tem todos os literais do resolvente gerado por C_1 e mais um literal (no caso, $\neg l''_k$), pela hipótese feita podemos retirar o resolvente de C_2 de Σ sem alterar a satisfatibilidade do método.

2. Se C_2 pode ser utilizada em GEN3, então Σ tem $n+1$ cláusulas i-positivas $\Box^*(l''_0 \rightarrow \boxed{i}\neg l'_0)$, $\ldots$, $\Box^*(l''_n \rightarrow \boxed{i}\neg l'_n)$, $n \geq 0$, e uma cláusula i-negativa $\Box^*(l'' \rightarrow \neg\boxed{i}\neg l')$, onde todo l'_j é igual a algum l_i, $0 \leq i, j \leq n$. Assim, $l_0 = l'_k$ para algum k tal que $0 \leq k \leq n$. A aplicação de GEN3 em C_2 juntamente com estas cláusulas produz o resolvente $\Box^*(\top \rightarrow \neg l''_0 \vee \cdots \vee \neg l''_n \vee \neg l'')$.

   Portanto, podemos aplicar GEN3 a C_1 juntamente com as cláusulas i-positivas, exceto por $\Box^*(l''_k \rightarrow \boxed{i}\neg l'_k)$, e a cláusula i-negativa e obter o resolvente $\Box^*(\top \rightarrow \neg l''_0 \vee \cdots \vee \neg l''_{k-1} \vee \neg l''_{k+1} \vee \cdots \vee \neg l''_n \vee \neg l'')$. Como o resolvente gerado por C_2 tem todos os literais do resolvente gerado por C_1 e mais um literal (no caso, $\neg l''_k$), pela hipótese feita podemos retirar o resolvente de C_2 de Σ sem alterar a satisfatibilidade do método.

Assim, mostramos que retirar C_2 de Σ não modifica a completude de qualquer uma das 13 regras de inferência do método. Portanto, como as 13 regras são corretas e completas, conclui-se que se Σ é insatisfatível, então $\Sigma - \{C_2\}$ é insatisfatível. □

Por fim, podemos provar que a estratégia de subsunção preserva a completude do método.

Teorema 3.2. *Seja Σ um conjunto de cláusulas. Se Σ é insatisfatível, C_1 e C_2 são cláusulas em Σ e $C_1 \subseteq C_2$, então $\Sigma - \{C_2\}$ é insatisfatível.*

Demonstração. A partir do resultado do lema anterior, a prova é dada trivialmente por indução sobre a quantidade de literais que C_1 tem a menos que C_2. □

Capítulo 4

Implementação

Neste capítulo apresentamos os detalhes da implementação baseada no método descrito no Capítulo 3. Primeiramente mostramos uma visão geral do programa, para depois mostrar as estruturas de dados utilizadas e, por fim, detalhar cada módulo que compõe o programa.

O provador de teoremas foi implementado na linguagem C++. Para a tradução das fórmulas e para o método de resolução modal foram utilizadas apenas as bibliotecas padrão da linguagem. Para implementar o *analisador léxico* (ou *tokenizer*) e o *analisador sintático* (ou *parser*), que em conjunto leem uma sequência de caracteres com uma fórmula na linguagem de $K_{(n)}$ e a transformam em uma árvore sintática de operadores e operandos, foram utilizados dois *softwares* livres frequentemente empregados para esse fim: o *Flex* [Paxson, 2012] e o *GNU Bison* [Demaille et al., 2013].

Na versão do programa apresentada neste trabalho foram implementadas as regras de inferência para $K_{(n)}$ IRES1, IRES2, LRES, MRES, GEN1, GEN2 e GEN3, além das regras SER, REF, SYM, TRANS, EUC1 e EUC2, para sistemas seriais, reflexivos, simétricos, transitivos e euclidianos, respectivamente. Desta forma, a atual versão efetua o cálculo de resolução modal para os 15 sistemas descritos na Seção 2.3.3.

4.1 Visão Geral

O funcionamento do programa inicia a partir da leitura de um arquivo com fórmulas para serem provadas e da leitura de argumentos utilizados ao se executar o programa. A forma geral para se executar o programa é:

```
<nome_do_programa> [opções] <nome_do_arquivo>
```

No comando acima, `<nome_do_arquivo>` é o nome de um arquivo-texto que contenha fórmulas na linguagem de $K_{(n)}$, com a denominação para operadores e operandos especificada na Seção 4.3.1. As opções permitidas são as seguintes:

- `-h` Mostra a ajuda
- `-d` Adiciona o axioma $\mathbf{D_i}$ ao sistema lógico
- `-t` Adiciona o axioma $\mathbf{T_i}$ ao sistema lógico
- `-b` Adiciona o axioma $\mathbf{B_i}$ ao sistema lógico
- `-4` Adiciona o axioma $\mathbf{4_i}$ ao sistema lógico

`-5` Adiciona o axioma $\mathbf{5_i}$ ao sistema lógico

`-v0` A saída mostra apenas se a fórmula é válida, satisfatível ou insatisfatível

`-v1` (*default*) A saída mostra as cláusulas do método de resolução para cada fórmula e seu resultado

`-v2` A saída mostra as árvores sintáticas obtidas antes e após a tradução para a FNS_K e o processo de resolução para cada fórmula, com seu resultado

Note que o axioma $\mathbf{K_i}$ é sempre válido, embora o programa também aceite a opção '`-k`' (que não tem qualquer efeito). É possível juntar várias opções em uma só: por exemplo, a opção '`-tb`' define o uso do sistema $\text{KTB}_{(n)}$; o mesmo resultado seria obtido utilizando as opções '`-t -b`', ou mesmo '`-ktb`'.

Após abrir o arquivo-texto, o programa considera que cada linha no arquivo possui uma fórmula da lógica modal, aplicando o processo de resolução separadamente para cada uma das fórmulas e pulando linhas em branco. Caso alguma linha não possua uma fórmula bem-formada, o programa reporta o erro e prossegue com as linhas seguintes. Cada linha de texto é enviada ao *analisador sintático* (*parser*) para que este construa uma árvore sintática com a hierarquia de operandos e operadores da fórmula lida, de acordo com a gramática BNF que representa a gramática definida pela linguagem de $\text{K}_{(n)}$. Para isso ele utiliza o *analisador léxico* (*tokenizer*), que recebe essa sequência de caracteres e retorna uma sequência de símbolos léxicos (ou *tokens*), que podem ser manipulados mais facilmente pelo analisador sintático. Os símbolos léxicos definidos pelo analisador léxico podem ser vistos na Seção 4.3.1, enquanto a gramática do analisador sintático e o processo de construção da árvore sintática são detalhados na Seção 4.3.2. A estrutura da árvore sintática em si pode ser vista na Seção 4.2.1.

A árvore sintática é então enviada ao *tradutor*, que aplica as regras mostradas na Seção 3.2 à árvore para transformá-la para a FNS_K. O analisador sintático não pode construir a árvore já na FNS_K porque o GNU Bison trabalha com a estratégia *bottom-up*, enquanto o algoritmo de tradução para a FNS_K utiliza a estratégia *top-down*. Após o final da tradução, a árvore sintática estará no formato de uma conjunção de cláusulas, onde cada cláusula pode ser inicial, literal ou modal, conforme as definições da Seção 3.2. O funcionamento do módulo de tradução é detalhado na Seção 4.3.3. A árvore sintática na forma normal é então repassada a um *módulo de montagem de cláusulas*, que lê a árvore e prepara, a partir dela, as estruturas de dados que serão utilizadas no método de resolução modal apresentado neste trabalho. O funcionamento do módulo supracitado pode ser visto na Seção 4.3.4, enquanto as estruturas de dados utilizadas no método de resolução são mostradas na Seção 4.2.2.

Por fim, a estrutura pronta é entregue ao *módulo de resolução modal*. Através das opções passadas ao executar o programa, ele determina se utilizará as regras de inferência mostradas na Seção 3.4 no cálculo de resolução e quais serão usadas, aplicando as regras repetidamente ao conjunto de cláusulas até que se encontre uma contradição ou não seja possível gerar novas cláusulas. O módulo de resolução modal utiliza *hashes* presentes na estrutura de dados para encontrar mais rapidamente cláusulas que possam ser utilizadas em regras de inferência, além de aplicar estratégias que buscam encontrar a contradição em um menor número de passos, determinando quais as primeiras cláusulas e regras de inferência a serem escolhidas. Tais estratégias são descritas na Seção 4.3.5.

Após a obtenção do resultado do cálculo de resolução, o programa verifica se alguma das opções `-v0`, `-v1` ou `-v2` foi dada e mostra a saída de acordo com a opção escolhida (ou com a opção *default*, se o usuário não escolheu qualquer uma dessas opções), seguindo então para a próxima fórmula. O processo descrito acima é mostrado na Figura 4.1.

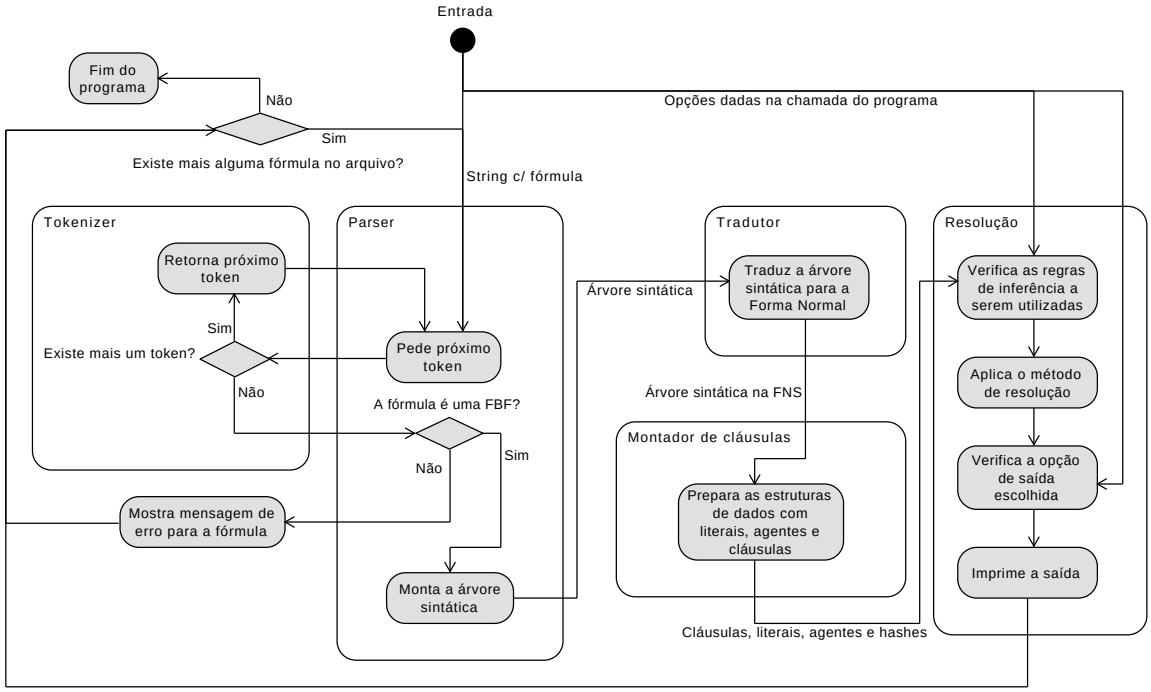

Figura 4.1: Visão geral do processo de execução do programa.

A implementação de um formato de entrada de dados que permita fornecer as cláusulas prontas para serem enviadas diretamente à resolução, sem necessidade da tradução, está planejada para versões futuras do programa.

4.2 Estruturas de Dados

A implementação utiliza certas estruturas de dados para representar a árvore sintática, assim como para reproduzir um conjunto de cláusulas, cada uma contendo disjunções de literais ou literais modais. Além disso, também são utilizados *hashes* para agilizar o processo de busca de literais específicos nas cláusulas.

Detalhamos nas seções seguintes as estruturas de dados utilizadas neste trabalho.

4.2.1 Árvore sintática

A árvore sintática utilizada é baseada em uma estrutura comum de árvore, onde cada nó pode ter um número qualquer de filhos. Na implementação utilizada, cada nó tem, além dos ponteiros para os nós filhos, um ponteiro para o nó pai, de forma a facilitar certos procedimentos executados na tradução para a FNS_K. O analisador sintático reserva o uso de mais de dois filhos em um nó apenas para os nós que representam conjunções ou disjunções. Cada nó possui os seguintes atributos:

i) um tipo de nó (que pode ser qualquer um dentre os tipos 'IFF', 'IMPLY', 'AND', 'OR', 'NECESSARY', 'POSSIBLE' ou 'LITERAL');

ii) um nome, que é uma *string* utilizada pelo analisador sintático para armazenar o nome do símbolo proposicional, no caso de um nó do tipo LITERAL, ou o nome do agente, no caso de um nó do tipo NECESSARY ou POSSIBLE;

iii) um escopo, que é uma *string* com o nome do literal à esquerda da implicação, usada durante o processo de tradução; e

iv) uma variável booleana que indica se o nó está negado ou não.

Os atributos do nó sintático revelam algumas decisões de projeto. O uso de atributos para o tipo de nó e para o nome de um símbolo proposicional ou agente são facilmente compreendidos, então explicaremos os outros dois atributos.

Na tradução para a FNS_K, inicialmente a fórmula φ a ser traduzida é colocada no formato $\Box^*(\textbf{início} \rightarrow x) \wedge \tau_1(\Box^*(x \rightarrow \varphi))$, onde x é um novo símbolo proposicional. Além disso, todas as subfórmulas geradas por τ_1 estão na forma $\Box^*(x \rightarrow \omega)$, onde x é um símbolo proposicional e ω é uma fórmula. Assim, para evitar a criação de novos nós IMPLY durante o processo de tradução, cada nó possui um campo *string* para armazenar o nome do símbolo proposicional à esquerda da implicação (o x dos casos acima). Chamamos este atributo de *escopo* pois, semanticamente, ele define em quais mundos a fórmula à direita da implicação é satisfeita. Consideramos que o escopo é $\top$ quando o campo *escopo* está vazio.

Cada nó possui também uma variável booleana definindo se o operador ou operando está negado ou não. Tal abordagem facilita o processo de tradução, pois podemos diferenciar, por exemplo, as regras $\tau_1(\Box^*(x \rightarrow \varphi \wedge \psi))$ e $\tau_1(\Box^*(x \rightarrow \neg(\varphi \wedge \psi)))$ observando um único nó, sem precisar verificar um nível de profundidade abaixo. Com isto também simplificamos a aplicação da regra $\tau_1(\Box^*(x \rightarrow \neg\neg\varphi)) = \tau_1(\Box^*(x \rightarrow \varphi))$, pois o caso de uma dupla negação apenas inverte a variável booleana de negação duas vezes.

Cada nó possui também ponteiros para os nós filhos e para o nó pai, além de métodos comuns em uma árvore, como inclusão e remoção de filhos, e um método para gerar uma cópia de um ramo completo (um filho e toda a estrutura de nós abaixo dele), utilizado para as regras de tradução que lidam com o operador de bi-implicação, '$\leftrightarrow$'.

4.2.2 Estruturas para a resolução

Durante o processo de montagem das cláusulas, os dados da árvore sintática são repassados para estruturas específicas que armazenam as cláusulas a serem utilizadas na resolução. Tais estruturas são modificadas durante o processo de resolução, através da inclusão de novas cláusulas — e de novos símbolos proposicionais, no caso das regras TRANS, EUC1 e EUC2. Através dessas estruturas é necessário ter conhecimento de quais literais referem-se ao mesmo símbolo proposicional, portanto define-se um *id* — um número identificador único — para cada símbolo proposicional. Durante a resolução utilizamos apenas o *id* dos símbolos proposicionais, para que não seja necessário ficar comparando *strings*. Os agentes também são tratados por um *id* definido internamente.

Além disso, é desejável ter *hashes* que agilizem buscas realizadas frequentemente nas regras de resolução, como descobrir quais cláusulas têm um literal específico (negado

ou não), ou quais cláusulas são i-negativas. As seguintes estruturas são usadas para a resolução:

i) um vetor de *strings* com o nome de cada símbolo proposicional (onde o *id* de cada símbolo proposicional é a sua posição no vetor);

ii) um vetor de *strings* com o nome de cada agente (onde o *id* de cada agente é a sua posição no vetor);

iii) uma tabela *hash* que retorna o *id* de um símbolo proposicional a partir do seu nome, ou se ainda não há uma entrada com aquele nome no vetor de símbolos;

iv) uma tabela *hash* que retorna o *id* de um agente a partir do seu nome, ou se ainda não há uma entrada com aquele nome no vetor de agentes;

v) um vetor de cláusulas (onde o *id* de cada cláusula é a sua posição no vetor);

vi) dois vetores de *ints* que armazenam o *id* de todas as cláusulas i-positivas e i-negativas, respectivamente, sem distinção de agente; e

vii) uma tabela *hash* que recebe um literal e retorna o *id* das cláusulas que possuem aquele literal.

Os vetores dos itens i) e ii) são usados para definir o *id* dos símbolos e agentes e para imprimir a saída na tela, pois todos os símbolos e agentes são tratados internamente como *ids*. As tabelas *hash* nos itens iii) e iv) são utilizadas na montagem das cláusulas, ao ler a árvore sintática, para saber se cada símbolo proposicional ou agente já recebeu um *id*, e qual é esse *id*. O item v) é um vetor da estrutura Cláusula, detalhada adiante. Os vetores mencionados no item vi) são utilizados para encontrar rapidamente todas as cláusulas i-positivas ou i-negativas para aplicação extensiva das regras modais GEN1, GEN2, GEN3, SER, REF, SYM, TRANS, EUC1 e EUC2. A última tabela de *hash* do item vii) é útil para agilizar todo o processo de resolução, permitindo encontrar rapidamente todas as cláusulas que possuam um certo literal, seja ele modal ou proposicional; este *hash* é amplamente utilizado nas regras LRES, MRES, GEN1, GEN2 e GEN3.

Para as tabelas *hash* o programa usa o `unordered_map`, da biblioteca padrão do C++11; caso o compilador utilizado não implemente as especificações do C++11, é possível substituí-las por `map`, da biblioteca padrão do C++, sem necessidade de alteração de qualquer outro código, ao custo de uma pequena perda de desempenho. O programa faz uso de duas estruturas adicionais, criadas especificamente para esta implementação: uma para literais (tanto proposicionais quanto modais) e uma para cláusulas.

Literal

Nas Seções 3.1.1 e 3.2 apresentamos os conceitos de literal e literal modal: um literal é um símbolo proposicional ou sua negação, enquanto um literal modal é uma fórmula nos formatos $\Box_i l$ ou $\neg\Box_i l$, onde l é um literal e i é um agente. Considerando que uma cláusula na FNS_K pode ter literais proposicionais ou modais, decidimos criar uma estrutura que engloba os dois tipos de literais. Assim, a estrutura Literal contém os seguintes atributos:

i) o *id* do símbolo proposicional;

ii) uma variável booleana indicando se o símbolo proposicional está negado;

iii) uma variável booleana indicando se este é um literal modal;

iv) uma variável booleana indicando se o operador $\boxed{i}$ está negado (sem efeito se não for um literal modal); e

v) o *id* de um agente, caso seja um literal modal (sem efeito se não for um literal modal).

Desta forma, um elemento do tipo Literal pode ser da forma 'p', '$\neg p$', '$\boxed{i}p$', '$\boxed{i}\neg p$', '$\neg\boxed{i}p$' ou '$\neg\boxed{i}\neg p$', onde p é um símbolo proposicional e i é um agente.

Cláusula

Nesta implementação, uma cláusula é uma disjunção de elementos do tipo Literal, representada internamente como um vetor de Literal. Além disso, ele contém um elemento do tipo Literal como escopo, onde o termo escopo é utilizado da mesma forma que na árvore sintática — o literal à esquerda da implicação. Embora teoricamente a implementação permita um literal modal como escopo, os módulos de montagem de cláusulas e de resolução restringem o escopo apenas aos literais proposicionais. A estrutura Cláusula possui os atributos a seguir:

i) um vetor de Literal que representa a disjunção de literais ou um literal modal;

ii) um vetor de Literal com os literais utilizados como escopo; e

iii) uma lista de *ints* e uma *string* que contêm, respectivamente, o *id* das cláusulas utilizadas para criação desta cláusula e qual o nome da regra que criou a cláusula, no caso desta cláusula ter sido resolvente de uma regra de inferência.

No método apresentado utilizamos apenas um literal como escopo, portanto os módulos de montagem de cláusulas e de resolução restringem o uso do vetor de literais de escopo para apenas um literal. No caso do escopo ser $\top$, este vetor é deixado vazio.

Caso a cláusula tenha sido gerada no processo de tradução para a FNS_K e não como resolvente de uma regra de inferência, a lista de *ints* e a *string* são mantidas vazias. Esses dois campos são usados ao imprimir a saída da resolução. Na cláusula também são implementados métodos para comparar cláusulas, para adicionar um literal e para ordenar os literais de acordo com o *id* do símbolo proposicional.

O programa sempre faz a ordenação do vetor que armazena a disjunção de literais antes da cláusula ser adicionada ao conjunto de cláusulas. Assim, diminuímos o tempo de comparação entre cláusulas ao verificarmos se uma cláusula gerada já está presente no conjunto de cláusulas, pois evitamos a necessidade de comparação literal por literal, comparando apenas blocos de memória. Essa ordenação também torna mais rápido para o programa verificar se uma cláusula gerada é subsumida por outra no conjunto de cláusulas.

4.3 Descrição dos Módulos

Como mencionado na Seção 4.1, o programa é dividido em alguns módulos: um *analisador léxico*, que separa a *string* com a fórmula modal em uma sequência de símbolos

léxicos para o analisador sintático; um *analisador sintático*, que através de uma gramática pré-definida e dos símbolos léxicos providos pelo analisador léxico constrói uma árvore sintática para a fórmula, ou retorna um erro caso a fórmula não seja bem-formada; um *tradutor*, que transforma a árvore sintática para colocá-la na FNS_K; um *módulo de montagem de cláusulas*, que prepara, a partir da árvore normalizada, as estruturas de dados a serem utilizados na resolução; e um *módulo de resolução modal*, que aplica as regras de inferência apresentadas no Capítulo 3 às cláusulas, determinando se a fórmula provida é uma contradição ou não. Detalhamos a seguir o funcionamento de cada um desses módulos.

4.3.1 Analisador léxico

O *analisador léxico*, implementado no Flex, recebe uma *string* que supostamente possui uma fórmula da lógica modal e a separa em símbolos léxicos, também chamados de *tokens*, enviando cada símbolo lido ao analisador sintático, que por sua vez pode usar esses símbolos juntamente com sua gramática para determinar a hierarquia dos operadores e operandos. A lista dos símbolos léxicos reconhecidos pelo analisador léxico, juntamente com os caracteres ou conjuntos de caracteres aos quais o analisador associa cada símbolo, pode ser vista na Tabela 4.1.

Símbolo léxico	**Sequências de caracteres**
AND	`^`, `&`, `and`
OR	`+`, `|`, `or`, `v`
NOT	¬, `!`, `-`, ∼, `not`, `neg`
IMPLY	`->`, `=>`, `then`, `imp`, `imply`, `implies`
IFF	`<->`, `<=>`, `iff`
NECESSARY	`L`, `box`, `nec`, `necessary`
POSSIBLE	`M`, `dia`, `pos`, `diamond`, `possible`
NAME	<alpha>`[`<alnum>`_]`*
NUMBER	<digit>+

Tabela 4.1: Símbolos definidos pelo analisador léxico.

Na lista acima, <alpha> representa um caractere alfabético, `[`<alnum>`_]`* representa uma sequência de zero ou mais caracteres alfanuméricos ou '`_`' (*underscore*) e <digit>+ representa uma sequência de um ou mais dígitos. Todas as palavras reservadas são *case-insensitive*, exceto '`L`' e '`M`', que quando minúsculas são reconhecidas como um NAME. O uso de '`L`' e '`M`' para necessidade e possibilidade, respectivamente, visam dar compatibilidade à notação usada por Hughes and Cresswell [1996].

A regra NAME serve para capturar nomes de símbolos proposicionais e de agentes. Agentes também podem ser definidos simplesmente por números, o que justifica o símbolo NUMBER. O analisador léxico ignora qualquer caractere que não seja um dos definidos acima, um espaço em branco ou parênteses.

As regras TRUE e FALSE não chegaram a ser implementadas por falta de tempo, pois o tradutor para FNS_K ainda não contém regras que lidem com tais operadores nulários. A sua implantação está prevista para versões futuras.

4.3.2 Analisador sintático

Cada símbolo léxico é enviado ao analisador sintático, que os utiliza para construir a árvore sintática que representa a hierarquia dos operadores e operandos na fórmula.

O analisador sintático, implementado no GNU Bison, possui uma definição de gramática BNF que representa a gramática da linguagem modal de $K_{(n)}$:

```
<fórmula>          ::= <equivalência>
<equivalência>     ::= <equivalência> IFF <implicação>
                     | <implicação>
<implicação>       ::= <implicação> IMPLY <conjunção>
                     | <conjunção>
<conjunção>        ::= <conjunção> AND <disjunção>
                     | <disjunção>
<disjunção>        ::= <disjunção> OR <negação>
                     | <negação>
<negação>          ::= NOT <negação>
                     | <modal>
<modal>            ::= NECESSARY <arg_modal>
                     | POSSIBLE <arg_modal>
                     | <agrupamento>
<arg_modal>        ::= NAME <negação>
                     | NUMBER <negação>
                     | <negação>
<agrupamento>      ::= ( <fórmula> )
                     | <fórmula_atômica>
<fórmula_atômica>  ::= NAME
```

A regra *<arg_modal>* permite o uso de construções como '`box 1 p`' ou '`box alice p`' para determinar um agente específico para um operador modal (nos casos mencionados o agente é representado como '1' ou como 'alice', respectivamente). Se o usuário não especificar um agente para um operador modal, o programa considera um agente *default*, sem nome. O uso de *<negação>* dentro de *<arg_modal>* permite construções como '`not box 1 not box 2 p`', que se traduz em $\neg\boxed{1}\neg\boxed{2}p$.

Perceba que a BNF utilizada considera que a disjunção tem precedência maior que a conjunção. Além disso, todos os operadores binários são associativos à esquerda. Assim, a partir da BNF e da definição dos símbolos feita no analisador léxico obtemos a Tabela 4.2, que mostra a precedência para os operadores. Na tabela, os operadores são listados de cima para baixo em ordem decrescente de precedência, onde todos os operadores são *case-insensitive*, exceto '`L`' e '`M`'.

Precedência	Operador	Descrição
1	`()`	Delimitador para agrupamento de subfórmulas

2	¬, `!`, `-`, ∼, `not`, `neg`
	`L`, `box`, `nec`, `necessary`
	`M`, `dia`, `diamond`, `pos`, `possible`
		Negação
		Operador □
		Operador ◇

Precedência	Operador	Descrição
3	`+`, `|`, `or`, `v`	Disjunção
4	`^`, `&`, `and`	Conjunção
5	`->`, `=>`, `then`, `imp`, `imply`, `implies`	Implicação
6	`<->`, `<=>`, `iff`	Equivalência (bi-implicação)

Tabela 4.2: Precedência dos operadores na gramática BNF do programa.

Após construir a hierarquia de operadores e operandos através da BNF mostrada acima, o *parser* constrói uma árvore sintática a partir da fórmula lida. Nessa árvore, cada nó representa um operador ou um operando, exceto pelo operador de negação; conforme visto na Seção 4.2.1, cada nó possui uma *flag* definindo se o operador ou operando está negado ou não. Por exemplo, a fórmula $\neg(\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b))$ seria representada como mostramos na Figura 4.2, onde a *string* abaixo do tipo do nó representa o nome do símbolo proposicional ou do agente.

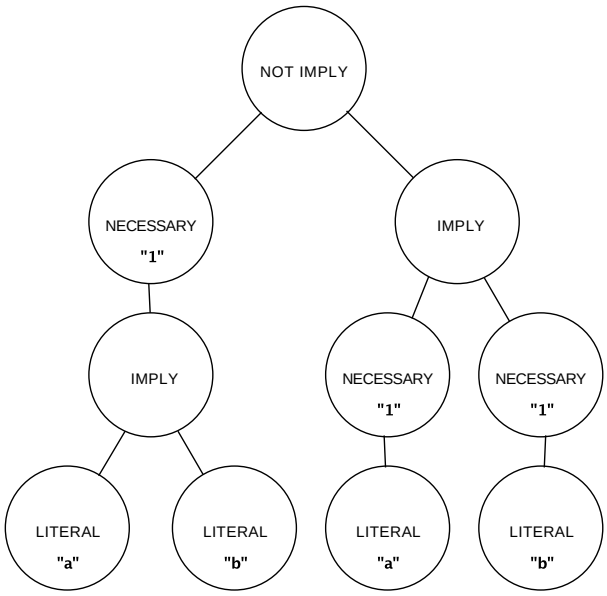

Figura 4.2: Árvore sintática para $\neg(\boxed{1}(a \to b) \to (\boxed{1}a \to \boxed{1}b))$.

4.3.3 Tradutor

A árvore sintática é então passada ao *tradutor*, onde esta é traduzida para a FNS_K. São aplicadas as regras descritas na Seção 3.2 de forma recursiva a cada ramo, até que a fórmula esteja na FNS_K, estando separada como uma conjunção de cláusulas iniciais, literais e modais.

Conforme visto na Seção 4.2.1, cada nó possui um campo *string* denominado como escopo. No processo de tradução para a FNS_K, esse campo é utilizado para armazenar o nome do símbolo proposicional à esquerda da implicação ao lidarmos com fórmulas do tipo $\Box^*(x \to \varphi)$. Caso esse campo esteja vazio, considera-se que o lado esquerdo da implicação é '$\top$'.

Consideramos não ser necessário descrever neste texto as regras aplicadas pelo tradutor na árvore sintática para transformá-la para a forma normal, pois elas reproduzem fielmente as regras mostradas na Seção 3.2, exceto pela regra de eliminação de dupla negação que não é necessária para a estrutura de dados utilizada. Nas regras mostradas naquela seção, considera-se que a função τ_1 representa que um nó e todos os nós abaixo dele ainda precisam sofrer regras de transformação.

Como o operador $\Box^*$ é aplicado a todas as cláusulas (com o sentido de indicar que aquela cláusula é válida em todos os mundos), não há necessidade de representar esse operador na árvore sintática. Cada novo símbolo proposicional produzido pelo método é nomeado como `_t1`, `_t2`, ..., para quantos novos símbolos proposicionais forem necessários. O caractere *underscore* no início do nome de cada símbolo proposicional gerado existe para diferenciá-los dos símbolos proposicionais definidos pelo usuário pois, como vimos na Seção 4.3.1, uma *string* iniciada com *underscore* não é considerada um *token* válido.

Por exemplo, ao final do processo de tradução da árvore sintática mostrada na Figura 4.2, a árvore teria a configuração mostrada na Figura 4.3. Na figura, a *string* na parte de cima de um nó representa o escopo, enquanto a parte de baixo representa o nome do símbolo proposicional ou do agente.

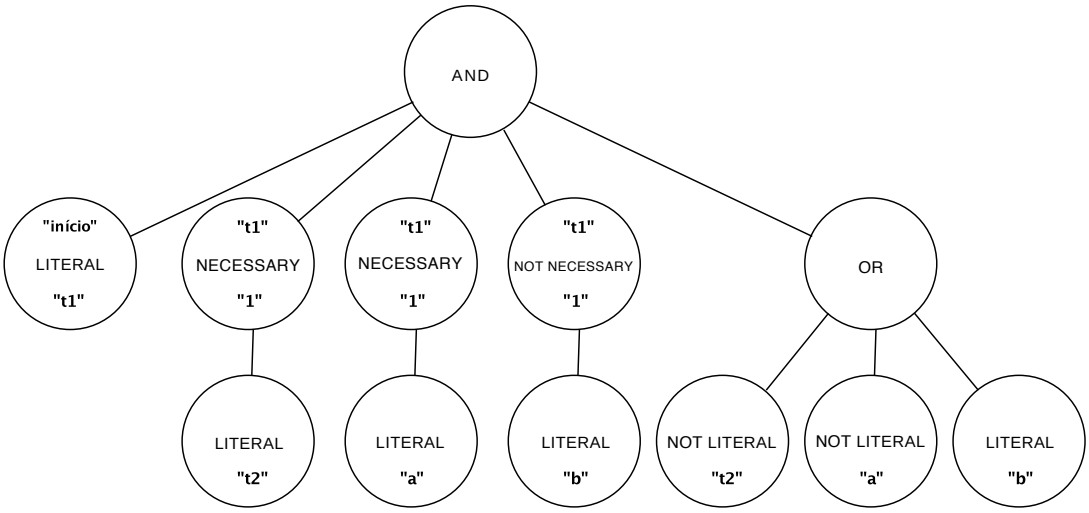

Figura 4.3: Árvore da Figura 4.2 após a tradução para a FNS_K.

4.3.4 Montagem de cláusulas

O *módulo de montagem de cláusulas* lê a árvore sintática resultante do processo de tradução e prepara as estruturas descritas na Seção 4.2.2, que serão utilizadas durante a resolução.

Como a árvore sintática está no formato de uma conjunção de cláusulas, cada filho do nó AND que está na raiz da árvore sintática é a raiz de uma cláusula. Então, para cada cláusula, este módulo executa o seguinte algoritmo:

1. Crie um elemento do tipo Cláusula, com escopo e lista de literais vazios.

2. Verifique a *string* no escopo do nó raiz da cláusula, na árvore sintática. Se ela não for vazia, adicione um escopo à cláusula criada, da forma descrita a seguir.

   (a) Verifique na tabela *hash* de símbolos se já existe um *id* para aquela *string*. Se o símbolo proposicional não possui um *id* associado a ele, adicione uma entrada para a *string* lida no vetor de símbolos, onde o *id* será a posição dessa entrada no vetor, e atualize a tabela *hash* de símbolos de acordo.

   (b) Adicione ao escopo da cláusula um literal proposicional não negado com o *id* do símbolo proposicional.

3. Verifique o tipo do nó raiz da cláusula, na árvore sintática.

   (a) Se o tipo for NECESSARY, verifique o agente neste nó e o nome do símbolo proposicional que está no nó filho.

      i. Se o símbolo lido não estiver presente no vetor de símbolos, adicione-o ao vetor e atualize a tabela *hash* de símbolos.

      ii. Se o agente lido não estiver presente no vetor de agentes, adicione-o ao vetor e atualize a tabela *hash* de agentes.

      iii. Adicione um literal modal à lista de literais da cláusula, com os *ids* do símbolo proposicional e do agente, o valor de negação de $\boxed{i}$ presente no nó raiz da cláusula na árvore e o valor de negação do símbolo proposicional presente no nó filho.

   (b) Se o tipo for OR, para cada um dos nós filhos:

      i. Se o nome do símbolo proposicional não estiver presente no vetor de símbolos, adicione-o ao vetor e atualize a tabela *hash* de símbolos.

      ii. Crie um literal proposicional l com o *id* do símbolo proposicional e o valor de negação determinado neste nó.

      iii. Procure se já existe algum literal do formato l na cláusula. Se houver, não há necessidade de adicionar este literal à cláusula; portanto, pule para o próximo nó.

      iv. Procure se já existe algum literal do formato $\neg l$ na cláusula (valor de negação invertido). Se houver, esta cláusula é uma tautologia; portanto, delete a cláusula criada e pule para o nó raiz da próxima cláusula, na árvore sintática.

      v. Adicione o literal criado à cláusula.

   (c) Se o tipo for LITERAL, realize o mesmo procedimento do item anterior, considerando como se este nó fosse um dos nós filhos de OR.

4. Procure se há no vetor de cláusulas alguma cláusula igual à que geramos. Se houver, delete a cláusula criada e pule para o nó raiz da próxima cláusula, na árvore sintática.

5. Adicione a cláusula criada ao vetor de cláusulas. A posição da cláusula no vetor será o seu *id*.

6. Para cada literal da cláusula, atualize a tabela *hash* de literais, incluindo o *id* desta cláusula, pois ela possui aquele literal.

7. Se a cláusula gerada for i-positiva, inclua o id desta cláusula no índice de cláusulas i-positivas.

8. Se a cláusula gerada for i-negativa, inclua o id desta cláusula no índice de cláusulas i-negativas.

Após efetuar o algoritmo acima para todas as cláusulas na árvore sintática, a estrutura de cláusulas estará pronta para ser utilizada pelo módulo de resolução.

4.3.5 Resolução

Com a estrutura de cláusulas pronta, podemos efetuar o processo de resolução. O *módulo de resolução* recebe como entrada a estrutura citada na Seção 4.2.2 e a lista de opções escolhida pelo usuário ao executar o programa.

Primeiramente, o módulo de resolução verifica as opções dadas pelo usuário para saber quais regras de inferência poderá utilizar no cálculo. As regras IRES1, IRES2, LRES, MRES, GEN1, GEN2 e GEN3 podem sempre ser utilizadas, pois estas são aplicáveis em toda a classe de sistemas modais normais. A regra SER é usada quando a opção `-d` é escolhida. De forma similar, `-t` ativa a regra REF, `-b` ativa a regra SYM, `-4` ativa a regra TRANS e `-5` ativa as regras EUC1 e EUC2. Podemos combinar essas opções para incluir mais de uma regra.

Após isso, o módulo inicia o processo de resolução. Utilizamos uma estratégia de resolução linear, descrita na Seção 3.1.2, para a resolução modal. A parte do cálculo que trata das cláusulas proposicionais, com o uso das regras LRES, IRES1 e IRES2, é feita através do mesmo procedimento da resolução linear descrita naquela seção; porém, para garantir a completude do método, aplicamos todas as regras de inferência modais possíveis em cada ramo da árvore de prova.

Durante a execução do método também utilizamos a estratégia de *forward subsumption* descrita na Seção 3.1.3, cuja completude foi provada na Seção 3.5. Assim, toda vez que produzimos uma nova cláusula por qualquer regra, esta cláusula só é adicionada ao vetor de cláusulas se ela não for subsumida por outra cláusula que já esteja no vetor.

Para reduzir a aplicação de regras modais em cada árvore de prova, iniciamos com uma fase de resolução puramente modal, onde são aplicadas todas as regras de inferência nas quais apenas cláusulas modais são resolvidas (SER, SYM, TRANS, EUC1, EUC2, REF, MRES e GEN2), até que não seja mais possível gerar cláusulas únicas a partir destas regras. Desta forma, a única resolução modal necessária dentro de cada ramo da árvore de prova será aquela realizada pelas regras GEN1 e GEN3, pois elas duas geram apenas cláusulas literais.

1. Dentre as regras SER, SYM, TRANS, EUC1 e EUC2, aplicamos repetidamente aquelas que estiverem na lista de regras do cálculo, até que não seja possível gerar novas cláusulas com as regras citadas.

2. Para cada cláusula i-positiva, aplicamos REF se ela estiver na lista de regras do cálculo. Também verificamos se existem outras cláusulas que, em conjunto com esta, permitam a aplicação de MRES ou GEN2, aplicando estas duas regras quando possível.

As regras SER, SYM, TRANS, EUC1 e EUC2 são executadas antes das outras pois todas elas utilizam cláusulas modais e geram cláusulas modais. Assim, cada uma destas cinco regras pode gerar cláusulas utilizadas pelas outras quatro e também por outras regras de resolução. Como todas as outras regras de inferência geram apenas cláusulas proposicionais, podemos aplicar estas cinco extensivamente antes da aplicação das demais. Deixamos as regras REF, MRES e GEN2 para o final da fase de resolução modal, pois estas últimas utilizam apenas cláusulas modais e geram cláusulas literais.

Depois do término da fase de resolução modal pelas regras acima, passamos para uma fase de resolução modal-proposicional através das regras restantes. Nessa fase, utilizamos uma adaptação da estratégia de *resolução linear* apresentada na Seção 3.1.2, porém fazendo o *backtracking* somente após todas as possíveis aplicações de resolução modal terem sido concluídas.

3. Faça a resolução linear pela regra LRES (escolhendo um par de cláusulas e, após isso, resolvendo sempre com a última cláusula gerada) até que se encontre uma contradição, a cláusula $\top \rightarrow \neg$`_t1` ou não seja possível gerar novas cláusulas.

   (a) Se conseguimos derivar uma contradição, a incluímos no conjunto de cláusulas e terminamos o processo de resolução.

   (b) Se produzimos a cláusula $\top \rightarrow \neg$`_t1`, note que, pela função de transformação τ_0 e pela denominação utilizada para os símbolos proposicionais gerados pelo programa, a primeira cláusula do conjunto sempre será **início** $\rightarrow$ `_t1`. Portanto, nesse caso podemos aplicar IRES1 entre as duas cláusulas e derivar uma contradição.

   (c) Se não é possível produzir novas cláusulas por LRES e nenhuma contradição foi encontrada, vá para o passo 4.

4. Verifique se é possível gerar uma nova cláusula através de GEN1 ou GEN3.

   (a) Se for possível, aplique as regras GEN1 e GEN3 repetidamente até não seja possível gerar novas cláusulas por estas duas regras. Vá para o passo 3.

   (b) Se não for possível, faça *backtrack* da última cláusula gerada e prossiga produzindo outra cláusula que não tenha sido produzida naquele passo.

5. Repita até que não seja possível produzir mais cláusulas.

Dessa forma, seguimos alternando entre uma fase de resolução linear proposicional (LRES) e uma fase de resolução modal-proposicional (GEN1 e GEN3) até que a contradição seja encontrada ou não seja possível produzir uma nova cláusula, onde então fazemos *backtrack* e continuamos explorando todos os ramos possíveis da árvore de prova, até que encontremos a contradição ou não existam mais ramos a serem explorados.

LRES produz cláusulas potencialmente utilizáveis por GEN1 e GEN3; além disso, GEN1 e GEN3 produzem cláusulas potencialmente utilizáveis por LRES. Por isso alternamos a utilização destas regras até que não seja possível gerar novas cláusulas por LRES, GEN1 ou GEN3. A aplicação das regras puramente modais no início da resolução e a aplicação extensiva de GEN1 e GEN3 em todos os ramos garante que toda a resolução modal é feita para cada ramo.

Neste trabalho não implementamos estratégias específicas para a escolha do primeiro par de cláusulas da resolução linear por LRES; a escolha é feita para o primeiro par de cláusulas encontrado que permita a aplicação da regra. Pretendemos implementar, como trabalho futuro, estratégias específicas para aumentar a probabilidade de se derivar uma contradição mais rapidamente no caso médio.

Como a única cláusula com escopo **início** gerada pela processo de tradução para a FNS_K é **início** $\rightarrow$ `_t1`, então essa fórmula só poderá ser útil para encontrar uma contradição se houver uma cláusula $\top \rightarrow \neg$`_t1`. Assim, IRES1 é utilizada no máximo uma vez durante todo o processo de resolução. Além disso, como não outras cláusulas com escopo **início**, a regra IRES2 não precisa ser utilizada. Tal observação se deve ao fato da forma que é feita a tradução para a FNS_K; portanto, será necessário o uso de IRES2 e múltiplos usos de IRES1 quando introduzirmos a funcionalidade de fornecer cláusulas prontas na entrada ao invés de apenas fórmulas. A regra IRES2 está implementada no código e será necessária sua utilização quando implementarmos a entrada direta no programa de cláusulas na forma normal, prevista como implementação futura, no caso do usuário fornecer mais de uma cláusula com escopo **início**.

4.4 Exemplo de Uso

Nesta seção mostramos os passos necessários para instalar e executar o programa, bem como mostramos um pequeno exemplo de entrada e as saídas correspondentes utilizando dois sistemas modais diferentes.

O código-fonte com a versão do programa produzida para este Trabalho de Graduação, bem como a última versão do programa, podem ser obtidos na página da Prof.ª Dr.ª Cláudia Nalon no *site* do Departamento de Ciência da Computação (CIC) desta universidade [Nalon, 2013].

4.4.1 Instalação

Os passos de instalação a seguir foram testados em ambientes Linux e Mac OS X. Para efetuar a instalação, são necessários:

1. o compilador *g++*, presente no GCC [Stallman et al., 2013];
2. o gerador de analisadores léxicos *Flex* [Paxson, 2012];
3. o gerador de analisadores sintáticos *Bison* [Demaille et al., 2013]; e
4. a ferramenta *GNU Make* [Smith, 2010].

A instalação foi testada com o *g++* versão 4.7.3, *flex* versão 2.5.37, *bison* versão 3.0 e *make* versão 3.82. Após obter o arquivo com o código-fonte do sistema e descomprimi-lo em algum diretório, o usuário deve ir ao diretório com o código-fonte pela linha de comando e utilizar o comando:

```
make
```

Após isso será gerado um executável com o nome `modal`. O usuário poderá também utilizar os comandos '`make clean`' para deletar todos os produtos da compilação (arquivos

com extensão '.o' e aqueles gerados pelo Flex e pelo Bison). O comando '`make purge`' também está disponível, que remove o executável e os arquivos que seriam excluídos por '`make clean`'.

4.4.2 Arquivo de entrada

O arquivo de entrada de fórmulas para o provador deve ser um arquivo-texto simples, com uma fórmula por linha. Linhas em branco ou somente com caracteres de espaço em branco são ignoradas. O provador tenta aplicar o método de resolução separadamente para cada fórmula, retornando o resultado da resolução de uma fórmula antes de iniciar o método para a próxima. A entrada de fórmulas da forma "{premissas} $\vdash$ conclusão" ainda não está implementada e é prevista como trabalho futuro.

A lista de símbolos possíveis para representar cada operando é mostrada na Seção 4.3.1. Uma das formas possíveis de representação dos operadores, que utilizamos no exemplo a seguir, é:

- '`not`' para representar '$\neg$';
- '`and`' para '$\wedge$';
- '`or`' para '$\vee$';
- '`->`' para '$\rightarrow$';
- '`<->`' para '$\leftrightarrow$;
- '`box`' para '$\Box$'; e
- '`diamond`' para '$\Diamond$'.

Podemos representar $\boxed{i}$, $i \in \mathcal{A}$ colocando o nome do agente entre o operador modal e o operando, separados por um caractere de espaço em branco. A gramática verifica pelo contexto que aquele é o nome do agente. Por exemplo, podemos usar '`box Alice p`' para dizer que '$\Box p$' para o agente Alice.

Exemplo 4.1. Mostramos um exemplo simples de arquivo de entrada com duas fórmulas: a primeira é uma aplicação direta do axioma $\mathbf{4_i}$ para um agente, enquanto a segunda mostra um encadeamento de operadores modais válido em $S5_{(2)}$.

```
diamond diamond p -> diamond p

box a diamond b box a p -> diamond b diamond a box a p
```

4.4.3 Execução

Supondo que o arquivo executável continua com o nome padrão, `modal`, utilizamos o seguinte comando no diretório com o executável para chamar o programa:

```
./modal [opções] <arquivo_de_entrada>
```

Nas opções podemos escolher o sistema modal a ser utilizado; por exemplo, a opção '`-t`' (ou '`-kt`') utiliza o sistema modal $KT_{(n)}$, '`-d45`' (ou '`-kd45`') escolhe o sistema $KD45_{(n)}$ e '`-t5`' (ou '`-kt5`') usa o sistema $S5_{(n)}$. Caso não seja escolhido nenhum sistema modal, o sistema padrão $K_{(n)}$ é utilizado. Existem também opções para a verbosidade do programa — a lista completa de opções pode ser vista na Seção 4.1.

Após executar o comando, o programa lê cada linha do arquivo-texto como uma fórmula separada e aplica a resolução modal para aquela fórmula. Primeiramente o programa aplica o método de resolução modal à negação da fórmula, para tentar provar que ela é válida. Caso não seja possível provar a validade, então aplica-se o método de resolução modal à fórmula diretamente para tentar provar que ela é uma contradição.

Exemplo 4.2. Mostramos o comando para executar o programa e a saída quanto tentamos provar as fórmulas do Exemplo 4.1 em $K_{(n)}$. Supomos aqui que o arquivo de entrada chama-se `input.txt`. Note que as duas fórmulas não são válidas em $K_{(n)}$.

```
./modal input.txt
```

```
Using modal system K

Formula f1: diamond diamond p -> diamond p

Trying resolution on ¬(f1):
1. _start => _t1
2. _t1 => ¬□¬_t2
3. _t2 => ¬□¬p
4. _t1 => □¬p
------------------------------------
5. true => ¬_t1 v ¬_t2                                    [MRES] 4,3

4 initial clauses + 1 clause(s) generated during resolution.

---

Trying resolution on f1:
1. _start => _t1
2. true => ¬_t1 v _t2 v _t3
3. _t2 => □_t4
4. _t4 => □¬p
5. _t3 => ¬□¬p
------------------------------------
6. true => ¬_t3 v ¬_t4                                    [MRES] 4,5

5 initial clauses + 2 clause(s) generated during resolution (1 backtracked).

---

f1: diamond diamond p -> diamond p cannot be proven.

================================================================================

Formula f2: box a diamond b box a p -> diamond b box a box a p

Trying resolution on ¬(f2):
1. _start => _t1
2. _t1 => □[a]_t2
3. _t2 => ¬□[b]¬_t3
4. _t3 => □[a]p
5. _t1 => □[b]_t4
6. _t4 => ¬□[a]¬_t5
7. _t5 => ¬□[a]p
------------------------------------
8. true => ¬_t3 v ¬_t5                                    [MRES] 4,7
```

```
7 initial clauses + 1 clause(s) generated during resolution.

---

Trying resolution on f2:
1. _start => _t1
2. true => ¬_t1 v _t2 v _t3
3. _t2 => ¬□[a]¬_t4
4. _t4 => □[b]_t5
5. _t5 => ¬□[a]p
6. _t3 => ¬□[b]¬_t6
7. _t6 => □[a]_t7
8. _t7 => □[a]p
-----------------------------------
9. true => ¬_t5 v ¬_t7                                    [MRES] 8,5

8 initial clauses + 1 clause(s) generated during resolution.

---

f2: box a diamond b box a p -> diamond b box a box a p cannot be proven.
```

Exemplo 4.3. Apresentamos a saída para o mesmo arquivo de entrada do exemplo anterior, porém para o sistema $\mathrm{KT5}_{(n)} = \mathrm{S5}_{(n)}$.

```
./modal -kt5 input.txt
```

```
Using modal system KT5

Formula f1: diamond diamond p -> diamond p

Trying resolution on ¬(f1):
1. _start => _t1
2. _t1 => ¬□¬_t2
3. _t2 => ¬□¬p
4. _t1 => □¬p
-----------------------------------
5. true => ¬_t1 v _pos(_t2)                               [EUC1] 2
6. _pos(_t2) => ¬□¬_t2                                    [EUC1] 2
7. ¬_pos(_t2) => □¬_t2                                    [EUC1] 2
8. _pos(_t2) => □_pos(_t2)                                [EUC1] 2
9. true => ¬_t2 v _pos(p)                                 [EUC1] 3
10. _pos(p) => ¬□¬p                                       [EUC1] 3
11. ¬_pos(p) => □¬p                                       [EUC1] 3
12. _pos(p) => □_pos(p)                                   [EUC1] 3
13. _pos(_t1) => □¬p                                      [EUC2] 4
14. _pos(_t1) => ¬□¬_t1                                   [EUC2] 4
15. ¬_pos(_t1) => □¬_t1                                   [EUC2] 4
16. _pos(_t1) => □_pos(_t1)                               [EUC2] 4
17. true => ¬_t1 v ¬p                                     [REF] 4
18. true => ¬_t1 v ¬_pos(p)                               [MRES] 4,10
19. true => ¬_t1 v ¬_t2                                   [MRES] 4,3
20. true => ¬_t2 v _pos(_t2)                              [REF] 7
21. true => ¬p v _pos(p)                                  [REF] 11
22. true => ¬p v ¬_pos(_t1)                               [REF] 13
23. true => ¬_pos(p) v ¬_pos(_t1)                         [MRES] 13,10
24. true => ¬_t2 v ¬_pos(_t1)                             [MRES] 13,3
25. true => ¬_t1 v _pos(_t1)                              [REF] 15
26. true => ¬_t1 v ¬_pos(_t1)                             [GEN1] 16,2,24
27. true => ¬_t1                                          [LRES] 26,25
28. _start => FALSE                                       [IRES1] 27,1

4 initial clauses + 24 clause(s) generated during resolution.

---

f1: diamond diamond p -> diamond p is valid.
```

```
Formula f2: box a diamond b box a p -> diamond b diamond a box a p

Trying resolution on ¬(f2):
1. _start => _t1
2. _t1 => □[a]_t2
3. _t2 => ¬□[b]¬_t3
4. _t3 => □[a]p
5. _t1 => □[b]_t4
6. _t4 => □[a]_t5
7. _t5 => ¬□[a]p
```

8. _pos(a,_t1) => □[a]_t2	[EUC2]	2
9. _pos(a,_t1) => ¬□[a]¬_t1	[EUC2]	2
10. ¬_pos(a,_t1) => □[a]¬_t1	[EUC2]	2
11. _pos(a,_t1) => □[a]_pos(a,_t1)	[EUC2]	2
12. true => ¬_t2 v _pos(b,_t3)	[EUC1]	3
13. _pos(b,_t3) => ¬□[b]¬_t3	[EUC1]	3
14. ¬_pos(b,_t3) => □[b]¬_t3	[EUC1]	3
15. _pos(b,_t3) => □[b]_pos(b,_t3)	[EUC1]	3
16. _pos(a,_t3) => □[a]p	[EUC2]	4
17. _pos(a,_t3) => ¬□[a]¬_t3	[EUC2]	4
18. ¬_pos(a,_t3) => □[a]¬_t3	[EUC2]	4
19. _pos(a,_t3) => □[a]_pos(a,_t3)	[EUC2]	4
20. _pos(b,_t1) => □[b]_t4	[EUC2]	5
21. _pos(b,_t1) => ¬□[b]¬_t1	[EUC2]	5
22. ¬_pos(b,_t1) => □[b]¬_t1	[EUC2]	5
23. _pos(b,_t1) => □[b]_pos(b,_t1)	[EUC2]	5
24. _pos(a,_t4) => □[a]_t5	[EUC2]	6
25. _pos(a,_t4) => ¬□[a]¬_t4	[EUC2]	6
26. ¬_pos(a,_t4) => □[a]¬_t4	[EUC2]	6
27. _pos(a,_t4) => □[a]_pos(a,_t4)	[EUC2]	6
28. true => ¬_t5 v _pos(a,¬p)	[EUC1]	7
29. _pos(a,¬p) => ¬□[a]p	[EUC1]	7
30. ¬_pos(a,¬p) => □[a]p	[EUC1]	7
31. _pos(a,¬p) => □[a]_pos(a,¬p)	[EUC1]	7
32. true => ¬_t1 v _t2	[REF]	2
33. true => ¬_t3 v p	[REF]	4
34. true => ¬_t3 v ¬_pos(a,¬p)	[MRES]	4,29
35. true => ¬_t3 v ¬_t5	[MRES]	4,7
36. true => ¬_t1 v _t4	[REF]	5
37. true => ¬_t4 v _t5	[REF]	6
38. true => _t2 v ¬_pos(a,_t1)	[REF]	8
39. true => ¬_t1 v _pos(a,_t1)	[REF]	10
40. true => ¬_t3 v _pos(b,_t3)	[REF]	14
41. true => p v ¬_pos(a,_t3)	[REF]	16
42. true => ¬_pos(a,_t3) v ¬_pos(a,¬p)	[MRES]	16,29
43. true => ¬_t5 v ¬_pos(a,_t3)	[MRES]	16,7
44. true => ¬_t3 v _pos(a,_t3)	[REF]	18
45. true => _t4 v ¬_pos(b,_t1)	[REF]	20
46. true => ¬_t1 v _pos(b,_t1)	[REF]	22
47. true => _t5 v ¬_pos(a,_t4)	[REF]	24
48. true => ¬_t4 v _pos(a,_t4)	[REF]	26
49. true => p v _pos(a,¬p)	[REF]	30
50. true => ¬_t1 v _pos(b,_t3)	[LRES]	32,12
51. true => ¬_pos(a,_t1) v _pos(a,_t4)	[GEN1]	26,9,36
52. true => ¬_t1 v _t5	[LRES]	37,36
53. true => ¬_t1 v ¬_pos(a,_t3)	[LRES]	52,43
54. true => ¬_t1 v ¬_t3	[LRES]	53,44
55. true => ¬_pos(a,_t1) v ¬_pos(a,_t3)	[GEN1]	19,9,53
56. true => ¬_t3 v ¬_t4	[LRES]	37,35
57. true => ¬_t3 v ¬_pos(b,_t1)	[LRES]	56,45
58. true => ¬_t2 v ¬_pos(b,_t1)	[GEN1]	23,3,57
59. true => ¬_t4 v _pos(a,¬p)	[LRES]	37,28
60. true => ¬_pos(b,_t1) v _pos(a,¬p)	[LRES]	59,45
61. true => ¬_t1 v _pos(a,¬p)	[LRES]	60,46
62. true => ¬_pos(b,_t3) v ¬_pos(b,_t1)	[GEN1]	23,13,57
63. true => ¬_pos(a,_t1) v _pos(b,_t3)	[LRES]	38,12

```
64. true => ¬_pos(a,_t1) v ¬_pos(b,_t1)                    [LRES] 63,62
65. true => ¬_t1 v ¬_pos(b,_t1)                            [LRES] 64,39
66. true => ¬_t1                                           [LRES] 65,46
67. _start => FALSE                                        [IRES1] 66,1

7 initial clauses + 60 clause(s) generated during resolution.

---

f2: box a diamond b box a p -> diamond b diamond a box a p is valid.
```

Capítulo 5

Conclusão

Neste trabalho apresentamos a implementação de um provador de teoremas para lógicas modais normais com múltiplos agentes, utilizando o método de resolução modal proposto por Nalon and Dixon [2007]. Com ele é possível provar teoremas para o sistema $K_{(n)}$, que não possui restrições nas relações de acessibilidade entre os mundos para cada agente, e também para sistemas modais normais cujas relações de acessibilidade sejam seriais, reflexivas, simétricas, transistivas ou euclidianas, ou qualquer combinação destas cinco. Assim, totalizamos quinze sistemas modais normais distintos para os quais o provador apresentado consegue apresentar provas de validade e satisfatibilidade, conforme descrito na Seção 2.3.3.

O objetivo do trabalho era implementar o cálculo de resolução proposto por Nalon and Dixon [2007] em sua totalidade. Consideramos que tal objetivo foi alcançado, embora ainda haja trabalho para ser feito na implementação. As constantes TRUE e FALSE não foram implementadas; além disso, ainda há necessidade de alterar a interface do programa, tornando a entrada e a saída mais intuitivas — uma das opções é uma integração com a interface para assistentes de prova *Proof General* [Aspinall, 2012] — e oferecendo mais opções ao usuário para entrada e saída. Para que o programa gerado possa ser utilizado pelo público, ainda é necessário efetuar testes de *benchmark* comparativos entre esta implementação e os outros sistemas que também efetuam provas automáticas de teoremas para lógicas modais normais, além de testar outras estratégias para ganho de eficiência.

Uma das maiores dificuldades encontradas na realização deste trabalho foi o tempo disponível para completá-lo, pois o projeto inteiro foi feito em um semestre, devido à mudança de orientadora e de projeto ao início da disciplina *Trabalho de Graduação 2*. Isso, juntamente com o fato de que o aluno não possuía qualquer conhecimento prévio sobre lógica modal, fez com que fosse necessário acomodar o estudo da teoria envolvida, implementação, testes e escrita deste texto no período de quatro meses. Outra dificuldade encontrada foi a falta de experiência no desenvolvimento de provadores de teoremas, que fez com que fossem tomadas algumas decisões de projeto ingênuas no início do desenvolvimento, das quais uma parte permaneceu no código final. Acredita-se que o produto gerado teria maior qualidade se o projeto fosse feito nos dois semestres usuais, uma vez que faltaram apenas as regras de transformação para fórmulas com os operadores TRUE e FALSE na implementação e a realização de testes mais extensivos para o sistema.

O código-fonte com a versão do programa produzida para este Trabalho de Graduação, bem como a última versão do programa, podem ser obtidos na página da Prof.ª Dr.ª Cláu-

dia Nalon no *site* do Departamento de Ciência da Computação (CIC) desta universidade [Nalon, 2013]. Dessa forma, quem quiser avaliá-lo poderá compilar e testar o programa.

Pretende-se continuar o desenvolvimento deste projeto após o término deste Trabalho de Gradução. Quanto aos aprimoramentos futuros pretendemos, no curto prazo: (i) implementar os operadores TRUE e FALSE; (ii) refatorar o código para aumentar a sua legibilidade e melhorar a performance do programa; (iii) prover novas opções para entrada, permitindo a entrada de cláusulas prontas na forma normal e um esquema que permita tomar um número qualquer de fórmulas como premissa e uma fórmula como conclusão; (iv) implementar saída em formato LaTeX; e (v) realizar testes de *benchmark*, procurando melhorar a eficiência do código. A longo prazo pretende-se expandir esta aplicação para abranger outras lógicas modais normais, através da introdução de novas regras de inferência, e combinar a resolução implementada neste trabalho com o cálculo de resolução para lógicas temporais, a fim de implementar um provador de teoremas para fusões das lógicas modais normais com lógicas temporais.

Referências

D. Aspinall. Proof General. http://proofgeneral.inf.ed.ac.uk/, oct 2012. Version 4.2. 61

W. Bibel. *Automated Theorem Proving*. Vieweg, 2^{nd} edition, 1987. 22

P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 2002. 8

H. K. Buning and T. Letterman. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0521630177. 25

B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, Cambridge, UK, 1980. 1, 2, 7, 8, 18, 19

A. Demaille, J. E. Denny, and P. Eggert. Bison: GNU parser generator. http://www.gnu.org/software/bison/, jul 2013. Version 3.0. 41, 54

M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, 1990. 21, 22

D. M. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. Many-dimensional modal logics: theory and applications. In *Studies in Logic and the Foundations of Mathematics*, volume 148. Elsevier, Amsterdam, 2003. 5, 8

J. Garson. Modal Logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2013 edition, 2013. 19

G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996. 47

R. Kowalski. The case for using equality axioms in automatic demonstration. In *Proc. of the Symposium on Automatic Demonstration*, number 125 in Lecture Notes in Mathematics, pages 112–127. Springer-Verlag, 1970. 25, 39

L. Lambert, E. Hemaspaandra, C. Homan, and M. Van Wie. Modal Logic in Computer Science. Master's thesis, B. Thomas Golisano College of Computing and Information Sciences, 2004. 1

R. C. T. Lee. *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms*. PhD thesis, University of California, Berkeley, 1967. 25

D. W. Loveland. A Linear Format for Resolution. In *Proceedings of the IRIA Symposium on Automatic Demonstration*, pages 147–162, New York, 1970. Springer-Verlag. 25

C. Nalon. Implementação do provador de teoremas para lógicas modais normais. http://www.cic.unb.br/~nalon, dec 2013. 54, 62

C. Nalon and C. Dixon. Clausal resolution for normal modal logics. *Journal of Algorithms*, 62:117–134, 2007. vii, ix, 1, 2, 6, 8, 11, 21, 26, 28, 30, 33, 34, 36, 37, 38, 39, 61

V. Paxson. Flex: fast lexical analyzer generator. http://flex.sourceforge.net/, aug 2012. Version 2.5.37. 41, 54

J. A. Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *JACM*, 12(1):23–41, jan 1965. 2, 21, 24

P. Smith. GNU Make. http://www.gnu.org/software/make/, jul 2010. Version 3.82. 54

R. Stallman, D. Edelsohn, K. R. Ghazi, J. A. Law, M. Lehmann, J. Merrill, D. Miller, T. Moene, J. Myers, G. Pfeifer, J. Sherrill, and J. Wilson. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, apr 2013. Version 4.7.3. 54

E. N. Zalta. Basic Concepts in Modal Logic. http://mally.stanford.edu/notes.pdf, Center for the Study of Language and Information, Stanford University, 1995. 5

Apêndice A

Código-fonte

A.1 makefile

```
CC=g++ -Wall -Wextra -O3
LEX=flex
YACC=bison -d

all: modal

modal: lex.yy.o parser.tab.o syntax_node.o translator.o propositional_symbol.o \
    literal.o clause.o resolution.o main.o
        ${CC} main.o resolution.o clause.o literal.o propositional_symbol.o translator.o \
            syntax_node.o parser.tab.o lex.yy.o -lfl -o modal

main.o: main.cpp parser_utils.h syntax_node.hpp errorinfo.h translator.hpp resolution.hpp
        ${CC} -c main.cpp

resolution.o: resolution.cpp resolution.hpp propositional_symbol.hpp clause.hpp \
    syntax_node.hpp translator.hpp
        ${CC} -c resolution.cpp

clause.o: clause.cpp clause.hpp literal.hpp propositional_symbol.hpp
        ${CC} -c clause.cpp

literal.o: literal.cpp literal.hpp
        ${CC} -c literal.cpp

propositional_symbol.o: propositional_symbol.cpp propositional_symbol.hpp
        ${CC} -c propositional_symbol.cpp

translator.o: translator.cpp translator.hpp syntax_node.hpp
        ${CC} -c translator.cpp

syntax_node.o: syntax_node.cpp syntax_node.hpp
        ${CC} -c syntax_node.cpp

parser.tab.o: parser.tab.cpp parser_utils.h syntax_node.hpp errorinfo.h
        ${CC} -c parser.tab.cpp

lex.yy.o: lex.yy.c parser.tab.hpp parser_utils.h
        ${CC} -c lex.yy.c

# Runs bison
parser.tab.cpp parser.tab.hpp: parser.ypp
        ${YACC} -v parser.ypp

# Runs flex
lex.yy.c: tokenizer.l
        ${LEX} tokenizer.l

clean:
        -rm -f *.o lex.yy.c *.tab.* *.output

purge:
        -rm -f *.o lex.yy.c *.tab.* *.output modal
```

A.2 syntax_node.hpp

```
#ifndef SYNTAX_NODE_HPP
#define SYNTAX_NODE_HPP
```

```
#include <vector>
#include <string>

// A node for the syntax tree of a modal logic formula.
struct SyntaxNode {
    public:
    // An enumeration for the possible node types.
    enum NodeType {
        ntAnd,       // AND (node can have several children)
        ntOr,        // OR (node can have several children)
        ntImply,     // IMPLY ( => )
        ntIff,       // IFF ( <=> )
        ntNecessary,// NECESSARY, or BOX
        ntPossible, // POSSIBLE, or DIAMOND
        ntTrue,      // TRUE ( \top )
        ntFalse,     // FALSE ( \bot )
        ntLiteral,   // a LITERAL, can be negated like every other node
        ntRoot       // only used for the ROOT of the tree before translation
    };

    /* PUBLIC MEMBERS */
    NodeType type;
    std::string name;
    std::string scope;
    bool negated;

    /* CONSTRUCTORS */
    // Constructor that sets the node type and leaves everything else blank
    SyntaxNode(SyntaxNode::NodeType type);
    // "Copy" constructor, does not copy parent or children
    SyntaxNode(SyntaxNode* const node);

    /* GETTERS */
    // Returns a pointer to the parent node of this node.
    SyntaxNode* parent() const;
    // Returns the n-th child of this node (n is zero-based).
    SyntaxNode* child(size_t pos) const;
    // Returns how many children this node has.
    size_t num_children() const;
    // Returns if this node is ready.
    bool ready() const;
    // Returns which child (first, second etc.) of its parent this node is.
    int which_child() const;

    /* OTHER PUBLIC METHODS */
    // Add a child to the right of this node
    void add_child(SyntaxNode& child);
    /* Add a child in the n-th position of this node (n is zero-based)
     *
     * If pos is greater or equal to the current number of children of this
     * node, it just adds to the right
     */
    void add_child_at(size_t pos, SyntaxNode& child);
    /* Remove the n-th child of this node.
     *
     * It does not delete the child node, just removes it from the children
     * list of the current node.
     */
    void remove_child(size_t pos);
    // Toggle "negated" status.
    void toggle_negate();
    // Set this node as ready.
    void set_ready();
    // Reverse the order of this node's children, useful to transform IFF
    void reverse_children();
    // Print the hierarchy of the tree, taking this node as the root of it.
    void print_tree(unsigned level = 0);

    private:
    /** PRIVATE MEMBERS **/
    std::vector<SyntaxNode*> _children; ///< Vector of children nodes.
    SyntaxNode* _parent;    ///< ID of parent node, -1 for no parent
    // If this node is ready. Used in the translation to SNF.
    bool _ready;
};

#endif //SYNTAX_NODE_HPP
```

A.3 syntax_node.cpp

```
#include <iostream>  // use in print_tree
#include <algorithm> // std::reverse()
#include "syntax_node.hpp"

SyntaxNode::SyntaxNode(NodeType ptype): type(ptype), name(""), scope(""),
    negated(false), _parent(NULL), _ready(false)
{
```

```
}

SyntaxNode::SyntaxNode(SyntaxNode* const node): type(node->type),
    name(node->name), scope(node->scope), negated(node->negated),
    _parent(NULL), _ready(node->_ready)
{
}

// Getters
SyntaxNode* SyntaxNode::parent() const
{
    return this->_parent;
}

SyntaxNode* SyntaxNode::child(size_t pos) const
{
    if (pos >= this->_children.size())
    {
        throw "Error in SyntaxNode::child() - index out of bounds";
    }

    return this->_children[pos];
}

size_t SyntaxNode::num_children() const
{
    return this->_children.size();
}

bool SyntaxNode::ready() const
{
    return this->_ready;
}

int SyntaxNode::which_child() const
{
    if (this->_parent == NULL)
    {
        return -1;
    }

    SyntaxNode* par = this->_parent;
    for (size_t i = 0; i < par->_children.size(); i++)
    {
        if (par->_children[i] == this)
        {
            return i;
        }
    }

    // TODO: throw an exception here
    return -1;
}

// Other methods
void SyntaxNode::add_child(SyntaxNode& child)
{
    this->_children.push_back(&child);
    child._parent = this;
}

void SyntaxNode::add_child_at(size_t pos, SyntaxNode& child)
{
    if (pos < this->_children.size())
    {
        std::vector<SyntaxNode*>::iterator it = this->_children.begin();
        this->_children.insert(it+pos, &child);
    }
    else
    {
        this->_children.push_back(&child);
    }

    child._parent = this;
}

void SyntaxNode::remove_child(size_t pos)
{
    if (pos >= this->_children.size())
    {
        throw "Error in SyntaxNode::remove_child() - index out of bounds";
    }

    std::vector<SyntaxNode*>::iterator it = this->_children.begin();
    SyntaxNode* child = *(it+pos);
    child->_parent = NULL;
    this->_children.erase(it+pos);
}

void SyntaxNode::toggle_negate()
{
    this->negated = !this->negated;
}

void SyntaxNode::set_ready()
```

```
{
    this->_ready = true;
}

void SyntaxNode::reverse_children()
{
    std::reverse(this->_children.begin(), this->_children.end());
}

void SyntaxNode::print_tree(unsigned level)
{
    using std::cout;
    using std::endl;

    std::string nodetype_strings[] =
    {
        "AND", "OR", "IMPLY", "IFF", "NECESSARY", "POSSIBLE", "TRUE",
        "FALSE", "LITERAL", "ROOT"
    };

    for (unsigned i = 0; i < level; i++)
    {
        cout << "  ";
    }

    if (this->scope == "")
    {
        //cout << "[true] ";
    }
    else if (this->scope == "_start")
    {
        cout << "[_start] ";
    }
    else
    {
        cout << "[" << this->scope << "] ";
    }

    if (this->negated)
    {
        cout << "NOT ";
    }

    cout << nodetype_strings[this->type];

    switch (this->type)
    {
        case SyntaxNode::ntLiteral:
            cout << " " << this->name;
            break;
        case SyntaxNode::ntNecessary:
        case SyntaxNode::ntPossible:
            cout << " " << this->name;
            break;
        default:
            break;
    }

    cout << endl;

    for (std::vector<SyntaxNode*>::iterator i = this->_children.begin();
        i != this->_children.end(); i++)
    {
        (*i)->print_tree(level + 1);
    }
}
```

A.4 parser_utils.h

```
#ifndef __PARSER_UTILS_H
#define __PARSER_UTILS_H

#include <vector>
#include <string>
#include <stack>
#include <cstring>
#include "syntax_node.hpp"
#include "errorinfo.h"


/* This parses the specified buffer. It appends all the generated SyntaxNodes
 * into "nodes" vector, and returns the index of the root for the generated
 * Syntax Tree.
 *
 * If there is a parse error, returns "-1" and fills out the ErrorInfo.
 */
SyntaxNode* parse_x(const char *buf, ErrorInfo *einfo);
```

```
// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

/* The following is seen only by Flex and Bison */

/* TParseParams structure -- this stores the current state of the parser and the
 * lexer, for reentrancy (ie. no global variables).
 *
 * The parser likes to have a pointer to its "TParseParams" state, while the
 * lexer likes to have a pointer to its "yyscanner" state, and both must be
 * retrievable from each other. So yyscanner will actually contain a pointer
 * back to its TParseParams in its yyget_extra() field.
 *
 * Note that yyscanner is maintained by yylex (and calls to yylex_init and
 * yylex_destroy), while TParseParams is maintained by us.
 */
struct TParseParams {
    // Constructor
    TParseParams(const char *buf_)
        : buf(buf_), pos(0), length(strlen(buf)) {}
    // A dummy assignment operator to stop the compiler complaining
    //operator=(const TParseParams &) {exit(0);}
    ErrorInfo *einfo;       // in case there is an error

    void *yyscanner;        // state of the lexer, used by Flex
    //std::vector<SyntaxNode> &nodes; // all syntax nodes are stored in here
    const char *buf;        // buffer for the source expression
    unsigned int pos;       // position of the lexer in the buffer
    unsigned int length;    // length of the buffer
    SyntaxNode* res;        // the most recent SyntaxNode produced, for Bison
    std::stack<std::string> sval;
};

// These are just prototypes for functions buried inside the lexer/parser.
#define YYSTYPE SyntaxNode*

extern "C" int yylex (SyntaxNode**, void *yyscanner);
//int yylex (SyntaxNode**, void *yyscanner);
int yylex_init (void** yyscanner);
int yylex_destroy (void* yyscanner);
void yyset_extra (void *dat,void *yyscanner);
void *yyget_extra (void *yyscanner);
int yyget_lineno( void* yscanner);
int yyget_column( void* yscanner);
int yyparse(void*);

#endif //__PARSER_UTILS_H
```

A.5 errorinfo.h

```
#ifndef __ERRORINFO_H
#define __ERRORINFO_H

#include <string>

/* Struct for returning error messages */
struct ErrorInfo {
    ErrorInfo() : msg(""), line(-1) {}
    std::string msg;
    int line;
};

#endif //__ERRORINFO_H
```

A.6 tokenizer.l

```
/*
 * Scanner for modal formulae
 */

%{
    #include <vector>
    #include <string>
    #include <iostream>
    using namespace std;
    #include "parser_utils.h"
    #include "parser.tab.hpp"

    #define YY_NEVER_INTERACTIVE 1
    #define YY_EXIT_FAILURE ((void)yyscanner, 2)
    //#define YY_DECL int yylex(int *yylval_param, void *yyscanner)
```

```
    /* We wish to read input from a block of memory. So, a pointer to the block
     * of memory was part of our state. This input proc simply reads from it.
     */
    #define YY_INPUT(buf,result,max_size) \
        result=yy_input_proc(buf,max_size,yyscanner)
    int yy_input_proc(char *buf, int size, yyscan_t yyscanner) {
        TParseParams *param = (TParseParams*)yyget_extra(yyscanner);
        if (param->pos >= param->length)
            return 0;
        int num = param->length - param->pos;
        if (num > size)
            num = size;
        memcpy(buf, param->buf + param->pos, num);
        param->pos += num;
        return num;
    }
%}

%option bison-bridge
%option reentrant
%option noyywrap
%option noinput
%option nounput
%option noyyalloc
%option noyyrealloc
%option noyyfree
%option yylineno

AND           "^"|"&"|(?i:and)
OR            "+"|"|"|(?i:or)|(?i:v)
NOT           "$\neg$"|"!"|"-"|"~"|(?i:not)|(?i:neg)
IMPLY         "->"|"=>"|(?i:imp)|(?i:imply)|(?i:implies)
IFF           "<->"|"<=>"|(?i:iff)
NECESSARY     "L"|"N"|(?i:box)|(?i:nec)|(?i:necessary)
POSSIBLE      "M"|"P"|(?i:dia)|(?i:pos)|(?i:diamond)|(?i:possible)
/*TRUE         "T"|(?i:true)*/
/*FALSE        "F"|(?i:false)*/
NAME          [[:alpha:]][[:alnum:]_]*
NUMBER        [[:digit:]]+

%%

[ \r\t\n]    { /* ignore blank spaces -- also ignore \n by now */ }
{AND}        { return TAND; }
{OR}         { return TOR; }
{NOT}        { return TNOT; }
{IMPLY}      { return TIMPLY; }
{IFF}        { return TIFF; }
{NECESSARY}  { return TNECESSARY; }
{POSSIBLE}   { return TPOSSIBLE; }
{TRUE}       { return TTRUE; }
{FALSE}      { return TFALSE; }
"("          { return TLPAREN; }
")"          { return TRPAREN; }
{NAME}       {
    TParseParams *pp = (TParseParams*)yyget_extra(yyscanner);
    pp->sval.push(strdup(yytext));
    return TNAME;
}
{NUMBER}     {
    TParseParams *pp = (TParseParams*)yyget_extra(yyscanner);
    pp->sval.push(strdup(yytext));
    return TNUMBER;
}
.            { }

%%

void yyfree (void * ptr, yyscan_t yyscanner)
{
    (void)yyscanner;
    free(ptr);
}

void *yyalloc (yy_size_t size, yyscan_t yyscanner)
{
    (void)yyscanner;
    return malloc(size);
}

void *yyrealloc (void * ptr, yy_size_t size, yyscan_t yyscanner)
{
    (void)yyscanner;
    return realloc(ptr, size );
}
```

A.7 parser.ypp

```
/*
 * Parser for modal formulae
 */

%{
    #include <vector>
    #include <string>
    #include <iostream>
    #include <stdexcept>
    using namespace std;
    #include "parser_utils.h"
    #include "syntax_node.hpp"
    #include "errorinfo.h"

    #define OUT_OF_BOUNDS out_of_range("Error in parser.ypp - index out of \
 bounds")
    #define YYERROR_VERBOSE 1
    //#define YYSTYPE SyntaxNode*

    /* This first "param" is the name of the formal argument that gets passed to
     * the parser, whose job is to keep the state.
     * The second "param" is the name of the actual argument to parse to the
     * lexer, which will be its state.
     */
    #define YYPARSE_PARAM param
    #define YYLEX_PARAM ((TParseParams*)param)->yyscanner

    /* When an error message is generated, the parser calls this function and we
     * record the error message. Then it backtracks its call-stack until it
     * meets an appropriate 'error' rule. That's where we fill in the
     * einfo->line number, and abort.
     */
    #define yyerror(msg) my_yyerror(msg,param)
    void my_yyerror(const char *c,void *param)
    {
        TParseParams *pp=(TParseParams*)param;
        pp->einfo->msg=c;
    }

    /* Following function parse_x just initializes the state (for both
     * parser and lexer), then invokes the parser.
     */
    SyntaxNode* parse_x(const char *buf, ErrorInfo *einfo)
    {
        TParseParams pp(buf);
        ErrorInfo dummy_info;
        pp.einfo=einfo; if (pp.einfo==0) pp.einfo=&dummy_info;
        yylex_init(&pp.yyscanner);
        yyset_extra(&pp, pp.yyscanner);
        int res=yyparse(&pp);
        yylex_destroy(pp.yyscanner);
        if (res==1) return NULL; else return pp.res;
    }
%}
%pure_parser
%start file

%left TIFF
%left TIMPLY
%left TAND
%left TOR
%right TNOT
%right TPOSSIBLE TNECESSARY
%token TNUMBER
%token TLPAREN TRPAREN
%token TTRUE TFALSE TNAME

%%

file:
    formula
    {
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntRoot);
        sn->add_child(*($1));
        TParseParams *pp = (TParseParams*)param;
        $$ = pp->res = sn;
    }
    | error
    {
        TParseParams *pp = (TParseParams*)param;
        pp->einfo->line = yyget_lineno(pp->yyscanner);
        pp->einfo->msg = "Formula is not well-formed.";
        YYABORT;
    };
formula:
    equivalence { $$ = $1; };
equivalence:
    equivalence TIFF implication
    {
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntIff);
        sn->add_child(*($1));
        sn->add_child(*($3));
        $$ = sn;
    }
    | implication { $$  = $1; };
```

```
implication:
    implication TIMPLY conjunctive
    {
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntImply);
        sn->add_child(*($1));
        sn->add_child(*($3));
        $$ = sn;
    }
    | conjunctive { $$ = $1; };
conjunctive:
    conjunctive TAND disjunctive
    {
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntAnd);
        sn->add_child(*($1));
        sn->add_child(*($3));
        $$ = sn;
    }
    | disjunctive { $$ = $1; };
disjunctive:
    disjunctive TOR negate
    {
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntOr);
        sn->add_child(*($1));
        sn->add_child(*($3));
        $$ = sn;
    }
    | negate { $$ = $1; };
negate:
    TNOT negate
    {
        SyntaxNode *snneg = $2;
        snneg->toggle_negate();
        $$ = $2;
    }
    | modal { $$ = $1; };
modal:
    TNECESSARY modalarg
    {
        TParseParams *pp = (TParseParams*)param;
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntNecessary);
        sn->name = pp->sval.top();
        pp->sval.pop();
        sn->add_child(*($2));
        $$ = sn;
    }
    | TPOSSIBLE modalarg
    {
        TParseParams *pp = (TParseParams*)param;
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntPossible);
        sn->name = pp->sval.top();
        pp->sval.pop();
        sn->add_child(*($2));
        $$ = sn;
    }
    | grouping { $$ = $1; };
modalarg:
    TNAME negate { $$ = $2; }
    | TNUMBER negate { $$ = $2; }
    | negate
    {
        TParseParams *pp = (TParseParams*)param;
        pp->sval.push("");
        $$ = $1;
    };
grouping:
    TLPAREN formula TRPAREN { $$ = $2; }
    | atomic { $$ = $1; };
atomic:
    TNAME
    {
        TParseParams *pp = (TParseParams*)param;
        SyntaxNode *sn = new SyntaxNode(SyntaxNode::ntLiteral);
        sn->name = pp->sval.top();
        pp->sval.pop();
        $$ = sn;
    };

%%
```

A.8 translator.hpp

```
#ifndef __TRANSLATOR_HPP
#define __TRANSLATOR_HPP

#include "syntax_node.hpp"

class Translator {
    private:
```

```
    // attributes
    static unsigned _last_tmp;

    // private methods
    Translator();
    static SyntaxNode* _new_literal_node(bool ready);
    static bool _join_with_parent(SyntaxNode* node);
    static void _duplicate_children(SyntaxNode *source, SyntaxNode *target);

    public:
    // public methods
    static void translate_to_snf(SyntaxNode* root);
};

#endif // __TRANSLATOR_HPP
```

A.9 translator.cpp

```
#include <vector>
#include <sstream>
#include <iostream>
using namespace std;

#include "translator.hpp"

/*--- Static variables initialization ---*/
unsigned Translator::_last_tmp = 0;

SyntaxNode* Translator::_new_literal_node(bool ready)
{
    _last_tmp++;
    stringstream sstm;
    sstm << "_t" << _last_tmp;

    SyntaxNode* newNode = new SyntaxNode(SyntaxNode::ntLiteral);

    if (ready)
    {
        newNode->set_ready();
    }
    newNode->name = sstm.str();

    return newNode;
}

bool Translator::_join_with_parent(SyntaxNode *node)
{
    SyntaxNode* parent = node->parent();
    if (!node->negated && !parent->negated && (node->type == parent->type))
    {
        int pos = node->which_child();

        parent->remove_child(pos);
        for (size_t i = 0; i < node->num_children(); i++)
        {
            parent->add_child_at(pos, *node->child(i));
            pos++;
        }

        return true;
    }

    return false;
}

void Translator::_duplicate_children(SyntaxNode *source, SyntaxNode *target)
{
    for (size_t i = 0; i < source->num_children(); i++)
    {
        SyntaxNode *child = source->child(i);

        // Create a new node with copied data, but not in the tree yet
        SyntaxNode *new_child = new SyntaxNode(child);

        target->add_child(*new_child);

        _duplicate_children(child, new_child);
    }
}

void Translator::translate_to_snf(SyntaxNode *node)
{
    if (!node->ready())
    {
        switch (node->type)
        {
            // Root, creates start and first generated literal
            case SyntaxNode::ntRoot:
```

```
        {
            // Reset index of temporary variables
            _last_tmp = 0;

            // Change to AND, no need to see parent since there's no parent
            node->type = SyntaxNode::ntAnd;

            // Create new literal and insert into nodes vector
            SyntaxNode *tmp_literal = _new_literal_node(true);
            tmp_literal->scope = "_start";

            // Add new node as root's child at left
            node->add_child_at(0, *tmp_literal);

            // Change other child's scope
            node->child(1)->scope = tmp_literal->name;

            // Remove transform flag and scope
            node->set_ready();
            node->scope = "";
        }
        break;
    case SyntaxNode::ntIff:
        {
            // Change this to be an AND node
            node->type = SyntaxNode::ntAnd;

            // Create an Imply child, change all of this children to
            // be children of Imply
            SyntaxNode *newNode1 = new SyntaxNode(SyntaxNode::ntImply);
            while (node->num_children() > 0)
            {
                SyntaxNode *child = node->child(0);
                node->remove_child(0);
                newNode1->add_child(*child);
            }
            node->add_child(*newNode1);

            // Create a second Imply child
            SyntaxNode *newNode2 = new SyntaxNode(SyntaxNode::ntImply);
            node->add_child(*newNode2);

            // Duplicate children of Imply1 to Imply2 in reverse order,
            // so we have (A <=> B) = (A => B) ^ (B => A)
            _duplicate_children(node->child(0), node->child(1));
            newNode2->reverse_children();
        }
        break;
    case SyntaxNode::ntImply:
        {
            if (!node->negated)
            {
                // (A -> B) => (not A v B)
                node->type = SyntaxNode::ntOr;
                node->child(0)->toggle_negate();
            }
            else
            {
                // not (A -> B) => (A ^ not B)
                node->type = SyntaxNode::ntAnd;
                node->negated = false;
                node->child(1)->toggle_negate();
            }
        }
        break;
    case SyntaxNode::ntAnd:
        {
            if (!node->negated)
            {
                // Change child scope if there is no scope defined
                for (size_t i = 0; i < node->num_children(); i++)
                {
                    if (node->child(i)->scope == "")
                    {
                        node->child(i)->scope = node->scope;
                    }
                }
                // Remove transform flag and scope
                node->set_ready();
                node->scope = "";
                // Check if possible to join with parent
                if (_join_with_parent(node))
                {
                    translate_to_snf(node->child(0));
                }
            }
            else
            {
                // Apply DeMorgan
                node->type = SyntaxNode::ntOr;
                node->toggle_negate();
                for (size_t i = 0; i < node->num_children(); i++)
                {
                    node->child(i)->toggle_negate();
                }
```

```
            }
        }
        break;
    case SyntaxNode::ntPossible:
        {
            // Possible(A) => not Necessary(not A)
            node->type = SyntaxNode::ntNecessary;
            node->toggle_negate();
            node->child(0)->toggle_negate();
        }
        break;
    case SyntaxNode::ntNecessary:
        {
            // Check if modal clause
            if (node->child(0)->type == SyntaxNode::ntLiteral)
            {
                // If it is, just remove transform flag and keep scope
                node->set_ready();
                node->child(0)->set_ready();
            }
            else
            {
                // Change to AND
                node->type = SyntaxNode::ntAnd;

                // Create modal child at left
                SyntaxNode* newNode = new SyntaxNode(
                    SyntaxNode::ntNecessary);
                newNode->scope = node->scope;
                newNode->name = node->name;
                node->name = "";
                node->add_child_at(0, *newNode);

                // Create new literal and insert as a child of the
                // modal we just created
                SyntaxNode *new_literal = _new_literal_node(false);
                SyntaxNode* modalNode = node->child(0);
                modalNode->add_child(*new_literal);
                // Change the other child's scope
                node->child(1)->scope = new_literal->name;
                // Conformity to rule not Necessary(A)
                if (node->negated)
                {
                    node->negated = false;
                    modalNode->negated = true;
                    modalNode->child(0)->negated = true;
                    node->child(1)->toggle_negate();
                }
            }
        }
        break;
    case SyntaxNode::ntOr:
        {
            if (node->negated)
            {
                // Apply DeMorgan
                node->type = SyntaxNode::ntAnd;
                node->toggle_negate();
                for (size_t i = 0; i < node->num_children(); i++)
                {
                    node->child(i)->toggle_negate();
                }
            }
            else
            {
                // Check each child for special cases
                size_t i = 0;
                while (i < node->num_children())
                {
                    SyntaxNode* child = node->child(i);

                    // If this child is "A => B"
                    if ((child->type == SyntaxNode::ntImply)
                        && !child->negated)
                    {
                        child->type = SyntaxNode::ntOr;
                        child->child(0)->toggle_negate();
                        _join_with_parent(child);
                    }
                    // If this child is "not (A ^ B)"
                    else if ((child->type == SyntaxNode::ntAnd)
                        && child->negated)
                    {
                        child->type = SyntaxNode::ntOr;
                        child->toggle_negate();
                        for (size_t j = 0; j < child->num_children();
                            j++)
                        {
                            child->child(j)->toggle_negate();
                        }
                        _join_with_parent(child);
                    }
                    // If this child is anything more complex than
                    // a literal
                    else if (child->type != SyntaxNode::ntLiteral)
```

```
                    {
                        SyntaxNode* parent = node->parent();
                        SyntaxNode *new_node = _new_literal_node(false);

                        node->remove_child(i);
                        node->add_child_at(i, *new_node);

                        if ((parent->type == SyntaxNode::ntAnd)
                            && !parent->negated)
                        {
                            parent->add_child(*child);
                            child->scope  = new_node->name;
                        }
                        else
                        {
                            // ERROR METHOD, shouldn't happen
                        }

                        // Now we got a literal, advance to next child
                        i++;
                    }
                    else
                    {
                        // Literal found, advance to next child
                        i++;
                    }
                }

                // This is already a disjunction of literals, do final rule
                // Create a new literal with negated scope
                SyntaxNode* scope_copy =  new SyntaxNode(
                    SyntaxNode::ntLiteral);
                scope_copy->name = node->scope;
                scope_copy->negated = true;
                scope_copy->set_ready();

                node->add_child_at(0, *scope_copy);

                // Remove transform flag and adjust scope
                node->set_ready();
                node->scope = "";

                // Finishes all other literals
                for (i = 0; i < node->num_children(); i++)
                {
                    node->child(i)->set_ready();
                }
            }
        }
        break;
    case SyntaxNode::ntLiteral:
        {
            // Make a new literal for the scope
            SyntaxNode* scope_copy =  new SyntaxNode(
                SyntaxNode::ntLiteral);
            scope_copy->name = node->scope;
            scope_copy->negated = true;
            scope_copy->set_ready();

            node->add_child(*scope_copy);

            // Make a copy of this literal as well
            SyntaxNode *new_literal =  new SyntaxNode(
                SyntaxNode::ntLiteral);
            new_literal->name = node->name;
            new_literal->negated = node->negated;
            new_literal->set_ready();

            node->add_child(*new_literal);

            // Transform this into an OR
            node->type = SyntaxNode::ntOr;
            node->negated = false;
            node->set_ready();
            node->scope = "";

            // Parent is not atOr, or else we would be in other branch,
            // but ok...
            _join_with_parent(node);
        }
        break;
    case SyntaxNode::ntTrue:
        {
            // TODO
        }
        break;
    case SyntaxNode::ntFalse:
        {
            // TODO
        }
        break;
    default:
        // TODO: ERROR METHOD
        break;
}
```

```
    }

    if (node->ready())
    {
        for (size_t i = 0; i < node->num_children(); i++)
        {
            translate_to_snf(node->child(i));
        }
    }
    else
    {
        translate_to_snf(node);
    }
}
```

A.10 propositional_symbol.hpp

```
#ifndef __PROPOSITIONAL_SYMBOL_HPP
#define __PROPOSITIONAL_SYMBOL_HPP

#include <string>
#include <vector>

class PropositionalSymbol {
    // attributes
    int id; // do we need this?
    std::string name;
    // indexes for which clauses have literals with this propositional symbol,
    // each index for one of the possible literal types (proposition or modal,
    // negated or not negated
    std::vector<int> cl_prop[2];
    std::vector<int> cl_pmod[2];
    std::vector<int> cl_nmod[2];

    public:
    // constructor
    PropositionalSymbol(int, std::string);

    // methods
    std::string get_name();
    void add_clause(int, bool, bool, bool);
    void delete_clause(int, bool, bool=false, bool=false);
    int num_clauses(bool, bool=false, bool=false);
    int get_clause_id(int, bool, bool=false, bool=false);
    int get_id();
};

#endif //__PROPOSITIONAL_SYMBOL_HPP
```

A.11 propositional_symbol.cpp

```
#include "propositional_symbol.hpp"

using namespace std;

PropositionalSymbol::PropositionalSymbol(int pid, string pname)
{
    this->id = pid;
    this->name = pname;
    this->cl_prop[0] = vector<int>();
    this->cl_prop[1] = vector<int>();
    this->cl_pmod[0] = vector<int>();
    this->cl_pmod[1] = vector<int>();
    this->cl_nmod[0] = vector<int>();
    this->cl_nmod[1] = vector<int>();
}

string PropositionalSymbol::get_name()
{
    return this->name;
}

void PropositionalSymbol::add_clause(int cid, bool neg, bool modal,
    bool modal_neg)
{
    if (!modal) {
        this->cl_prop[(int)neg].insert(this->cl_prop[(int)neg].begin(),cid);
    }
    else if (!modal_neg) {
        this->cl_pmod[(int)neg].insert(this->cl_pmod[(int)neg].begin(),cid);
    }
```

```
    else {
        this->cl_nmod[(int)neg].insert(this->cl_nmod[(int)neg].begin(),cid);
    }
}

void PropositionalSymbol::delete_clause(int cid, bool neg, bool modal,
    bool modal_neg)
{
    if (!modal) {
        for (vector<int>::iterator it = cl_prop[(int)neg].begin();
            it != cl_prop[(int)neg].end(); it++)
        {
            if ((*it) == cid)
            {
                cl_prop[(int)neg].erase(it);
                return;
            }
        }
    }
    else if (!modal_neg) {
        for (vector<int>::iterator it = cl_pmod[(int)neg].begin();
             it != cl_pmod[(int)neg].end(); it++)
        {
            if ((*it) == cid)
            {
                cl_pmod[(int)neg].erase(it);
                return;
            }
        }
    }
    else {
        for (vector<int>::iterator it = cl_nmod[(int)neg].begin();
             it != cl_nmod[(int)neg].end(); it++)
        {
            if ((*it) == cid)
            {
                cl_nmod[(int)neg].erase(it);
                return;
            }
        }
    }
}

int PropositionalSymbol::num_clauses(bool neg, bool modal, bool modal_neg)
{
    if (!modal) {
        return cl_prop[(int)neg].size();
    }
    else if (!modal_neg) {
        return cl_pmod[(int)neg].size();
    }
    else {
        return cl_nmod[(int)neg].size();
    }
}

int PropositionalSymbol::get_clause_id(int pos, bool neg, bool modal,
    bool modal_neg)
{
    if (!modal) {
        return cl_prop[(int)neg][pos];
    }
    else if (!modal_neg) {
        return cl_pmod[(int)neg][pos];
    }
    else {
        return cl_nmod[(int)neg][pos];
    }
}

int PropositionalSymbol::get_id()
{
    return this->id;
}
```

A.12 literal.hpp

```
#ifndef __LITERAL_HPP
#define __LITERAL_HPP

struct Literal {
    // attributes
    int id;
    bool negated;
    bool modal;
    bool modal_negated;
    int agent;
```

```
    // constructor
    Literal(int, bool, bool, bool, int = 0);

    // operators
    bool operator<(const Literal& other) const;
    bool operator==(const Literal& other) const;
};

#endif //__LITERAL_HPP
```

A.13 literal.cpp

```
#include <iostream>
#include "literal.hpp"

using namespace std;

Literal::Literal(int pid, bool pnegated, bool pmodal, bool pmodal_negated,
    int pagent): id(pid), negated(pnegated), modal(pmodal),
    modal_negated(pmodal_negated), agent(pagent)
{
}

bool Literal::operator<(const Literal& other) const {
    return (this->id < other.id);
}

bool Literal::operator==(const Literal& other) const {
    if ((this->id == other.id) && (this->negated == other.negated)
        && (this->modal == other.modal)
        && (this->modal_negated == other.modal_negated)
        && (this->agent == other.agent))
    {
        return true;
    }

    return false;
}
```

A.14 clause.hpp

```
#ifndef __CLAUSE_HPP
#define __CLAUSE_HPP

#include <string>
#include <vector>
#include "literal.hpp"
#include "propositional_symbol.hpp"

class Clause {
    // attributes
    std::vector<Literal> scope;
    std::vector<Literal> literals;
    std::string creation_method;
    std::vector<PropositionalSymbol>* symbols;

    bool _compare_literals(Literal, Literal);

    public:
    // "Defines" -- is there a better practice for this?
    static const int SCOPE_TRUE = -1;

    // constructor
    Clause(int = Clause::SCOPE_TRUE, bool = false);

    // getters
    int num_literals();
    int get_literal_id(int);
    bool is_negated(int);
    bool is_modal(int);
    bool is_modal_negated(int);
    int get_agent(int);
    int get_scope();
    bool is_scope_negated();
    std::string get_creation_method();

    // setters
    void set_creation_method(std::string);

    // operators
    bool operator<(const Clause& other) const;
    bool operator==(const Clause&) const;
```

```
    // other methods
    void add_literal(Literal);
    int has_literal(int);
    void sort_literals();
};

#endif //__CLAUSE_HPP
```

A.15 clause.cpp

```
#include "clause.hpp"
#include <algorithm>
#include <functional>

using namespace std;

Clause::Clause(int pscope, bool negated) {
    if (pscope != Clause::SCOPE_TRUE)
    {
        this->scope.push_back(Literal(pscope, negated, false, false));
    }
}

bool Clause::_compare_literals(Literal l1, Literal l2)
{
    int count1 = (*this->symbols)[l1.id].num_clauses(!l1.negated);
    int count2 = (*this->symbols)[l2.id].num_clauses(!l2.negated);
    return (count1 < count2);
}

int Clause::num_literals() {
    return this->literals.size();
}

int Clause::get_literal_id(int pos) {
    return this->literals[pos].id;
}

bool Clause::is_negated(int pos) {
    return this->literals[pos].negated;
}

bool Clause::is_modal(int pos) {
    return this->literals[pos].modal;
}

bool Clause::is_modal_negated(int pos) {
    return this->literals[pos].modal_negated;
}

int Clause::get_agent(int pos) {
    return this->literals[pos].agent;
}

int Clause::get_scope() {
    if (!this->scope.empty())
    {
        return this->scope[0].id;
    }

    return Clause::SCOPE_TRUE;
}

bool Clause::is_scope_negated() {
    if (this->scope.empty())
    {
        return false;
    }

    return this->scope[0].negated;
}

string Clause::get_creation_method() {
    return this->creation_method;
}

void Clause::set_creation_method(string method) {
    this->creation_method = method;
}

bool Clause::operator<(const Clause& other) const {
    return (this->literals.size() < other.literals.size());
}

bool Clause::operator==(const Clause& other) const {
    if ((this->scope == other.scope) && (this->literals == other.literals))
    {
```

```
        return true;
    }

    return false;
}

void Clause::add_literal(Literal lit) {
    literals.push_back(lit);
}

int Clause::has_literal(int lit_id) {
    for (unsigned int i = 0; i < this->literals.size(); i++) {
        if (literals[i].id == lit_id)
            return i;
    }
    return -1;
}

void Clause::sort_literals() {
    sort(literals.begin(), literals.end());
}
```

A.16 resolution.hpp

```
#ifndef __RESOLUTION_HPP
#define __RESOLUTION_HPP

#include <string>
#include <vector>
#include <map>
//#include <unordered_map>
#include <set>
#include <stack>
#include <utility>
#include "propositional_symbol.hpp"
#include "clause.hpp"
#include "syntax_node.hpp"

class Resolution {
    // uses default constructor

    /*** ATTRIBUTES ***/
    // a clause vector with literals
    std::vector<Clause> clauses;
    // A vector with info from propositional symbols (name and which clauses
    // have literals with them)
    std::vector<PropositionalSymbol> symbols;
    // a name => id hashmap for propositional symbols (to import the clauses
    // from the syntax tree)
    std::map<std::string, int> symbol_hash;
    // an id => name vector for printing agents
    std::vector<std::string> agent_names;
    // a name => id hashmap for agents (to import the clauses from the
    // syntax tree)
    std::map<std::string, int> agent_hash;
    // An index for negated modal clauses, frequently used in GEN1, GEN2 and
    // GEN3
    std::vector<int> neg_modal_clauses;
    // Number of clauses generated from the SNF conversion
    int num_initial_clauses;
    // Total number of valid clauses generated, including the ones erased in
    // backtracking
    int num_total_clauses;
    // Number of clauses generated after the modal phase
    int num_init_and_modal;
    // Backup of last backtracked clause, just to avoid making the same clause
    // over and over
    Clause last_backtracked;

    // private methods
    bool _mount_clause_from_node(SyntaxNode&, int);
    bool _add_literal_node_to_clause(SyntaxNode*, int, bool = false,
        bool = false, std::string = "");
    bool _add_literal_to_clause(int, int, bool, bool=false, bool=false, int=0);
    void _add_clause_to_indexes(int);
    int _get_scope_id(SyntaxNode&);
    bool _last_clause_is_unique();
    void _delete_last_clause();
    bool _modal_phase();
    bool _linear_resolution();
    bool _propositional_step(int);

    // move to InferenceRule class in future
    bool _do_mres_with_compatible_clauses(int);
    bool _do_gen2_with_compatible_clauses(int);
    bool _do_ref_with_compatible_clauses(int);
    bool _do_ser_with_compatible_clauses(int);
    bool _do_sym_with_compatible_clauses(int);
```

```
    bool _try_gen1();
    bool _try_gen3();
    bool _pres(int, int, int);
    bool _mres(int, int);
    bool _gen1(int, int, std::set<int>);
    bool _gen2(int, int, int);
    bool _gen3(std::set<int>, int, int);
    bool _ref(int);
    bool _ser(int);
    bool _sym(int);

    public:
    // methods
    void import_syntax_tree(SyntaxNode&);
    bool print_clauses();
    bool resolve();
    // getters
    unsigned num_clauses();
    unsigned initial_clauses();
    unsigned total_clauses();
};

#endif //__RESOLUTION_HPP
```

A.17 resolution.cpp

```
#include "resolution.hpp"
#include "literal.hpp"

#include <iostream>
#include <algorithm>
#include <iterator>
#include <sstream>
#include <iomanip>
#include <queue>

using namespace std;

extern bool axiom_d;
extern bool axiom_t;
extern bool axiom_b;

bool Resolution::_mount_clause_from_node(SyntaxNode& clause_root, int clause_id)
{
    // If clause is just a literal, add it and return
    if (clause_root.type == SyntaxNode::ntLiteral)
    {
        this->_add_literal_node_to_clause(&clause_root, clause_id);
        return true;
    }

    for (size_t i = 0; i < clause_root.num_children(); i++)
    {
        SyntaxNode *clause_child_node = clause_root.child(i);
        bool clause_is_relevant;

        // Add child to clause
        if (clause_root.type == SyntaxNode::ntNecessary)
        {
            clause_is_relevant = this->_add_literal_node_to_clause(
                clause_child_node, clause_id, true, clause_root.negated,
                clause_root.name);
        }
        else
        {
            clause_is_relevant = this->_add_literal_node_to_clause(
                clause_child_node, clause_id);
        }

        // Delete clause if it is not important for us (tautology or not unique)
        if (!clause_is_relevant)
        {
            this->clauses.pop_back();
            return false;
        }
    }

    return true;
}

// TODO: change to _add_child_node_to_clause and treat modal literals too
bool Resolution::_add_literal_node_to_clause(SyntaxNode* node, int clause_id,
    bool modal, bool modal_negated, string agent)
{
    string name = node->name;
    int symbol_id;

    if (this->symbol_hash.count(name) == 0)
```

```
    {
        // If literal name is not in hash, create new propositional symbol and
        // include in hash
        symbol_id = this->symbols.size();
        this->symbols.push_back(PropositionalSymbol(symbol_id, name));
        this->symbol_hash[name] = symbol_id;
    }
    else
    {
        symbol_id = this->symbol_hash.at(name);
    }

    int agent_id;

    // If agent has name and is not in hash, create new agent in hash
    if (agent == "")
    {
        agent_id = 0;
    }
    else if (this->agent_hash.count(agent) == 0)
    {
        agent_id = this->agent_names.size();
        this->agent_names.push_back(agent);
        this->agent_hash[agent] = agent_id;
    }
    else
    {
        agent_id = this->agent_hash.at(agent);
    }

    return this->_add_literal_to_clause(symbol_id, clause_id, node->negated,
        modal, modal_negated, agent_id);
}

bool Resolution::_add_literal_to_clause(int symbol_id, int clause_id,
    bool negated, bool modal, bool modal_negated, int agent_id)
{
    // Search if the literal is already in this clause
    int repeated_id = this->clauses[clause_id].has_literal(symbol_id);
    if (repeated_id > -1)
    {
        if (negated == clauses[clause_id].is_negated(repeated_id))
        {
            return true; // finishes and does nothing
        }
        else
        {
            return false; // indicates that this clause should be erased
        }
    }

    // Include in clause
    this->clauses[clause_id].add_literal(Literal(symbol_id, negated, modal,
        modal_negated, agent_id));

    return true;
}

void Resolution::_add_clause_to_indexes(int clause_id)
{
    // Include this clause in the relevant indexes of its literals
    for (int i = 0; i < this->clauses[clause_id].num_literals(); i++)
    {
        int literal_id = this->clauses[clause_id].get_literal_id(i);
        bool negated = this->clauses[clause_id].is_negated(i);
        bool modal = this->clauses[clause_id].is_modal(i);
        bool modal_negated = this->clauses[clause_id].is_modal_negated(i);
        //int agent = this->clauses[clause_id].get_agent(i);

        this->symbols[literal_id].add_clause(clause_id, negated, modal,
            modal_negated);

        if (modal && modal_negated)
        {
            this->neg_modal_clauses.push_back(clause_id);
        }
    }
}

int Resolution::_get_scope_id(SyntaxNode& clause_root)
{
    if (clause_root.scope == "")
    {
        return Clause::SCOPE_TRUE;
    }

    // Search for the id of this literal in the literal table
    string name = clause_root.scope;
    if (this->symbol_hash.count(name) == 0)
    {
        // If literal name is not in hash, create new propositional symbol and
        // include in hash
        size_t symbol_id = this->symbols.size();
        this->symbols.push_back(PropositionalSymbol(symbol_id, name));
        this->symbol_hash[name] = symbol_id;
```

```
    }

    int scope = this->symbol_hash.at(name);

    return scope;
}

bool Resolution::_last_clause_is_unique()
{
    for (vector<Clause>::iterator it = clauses.begin();
         (it != clauses.end()) && (distance(it,clauses.end()) != 1);
         it++)
    {
        if ((*it) == this->clauses.back())
            return false;
    }

    return true;
}

void Resolution::_delete_last_clause()
{
    Clause* cl = &(this->clauses.back());

    // Update symbol indexes to eliminate this clause from the hashes
    for (int i = 0; i < cl->num_literals(); i++)
    {
        int symbol_id = cl->get_literal_id(i);
        bool negated = cl->is_negated(i);
        this->symbols[symbol_id].delete_clause(this->clauses.size()-1, negated);
    }

    // Copy clause
    this->last_backtracked = this->clauses.back();

    // Delete clause
    this->clauses.pop_back();
}

bool Resolution::_modal_phase()
{
    bool contradiction_found;

    // Rules that create other modal clauses
    for (size_t c1 = 1; c1 < this->clauses.size(); c1++)
    {
        if (axiom_b)
        {
            // SER
            contradiction_found = this->_do_sym_with_compatible_clauses(c1);
            if (contradiction_found)
            {
                return true;
            }
        }
    }

    for (size_t c1 = 1; c1 < this->clauses.size(); c1++)
    {
        if (axiom_d)
        {
            // SER
            contradiction_found = this->_do_ser_with_compatible_clauses(c1);
            if (contradiction_found)
            {
                return true;
            }
        }
    }

    this->num_init_and_modal = this->clauses.size();

    // Rules that only create propositional clauses
    for (int c1 = 1; c1 < this->num_init_and_modal; c1++)
    {
        // MRES
        contradiction_found = this->_do_mres_with_compatible_clauses(c1);
        if (contradiction_found)
        {
            return true;
        }

        // GEN2
        contradiction_found = this->_do_gen2_with_compatible_clauses(c1);
        if (contradiction_found)
        {
            return true;
        }

        // REF
        if (axiom_t)
        {
            contradiction_found = this->_do_ref_with_compatible_clauses(c1);
            if (contradiction_found)
            {
```

```
                return true;
            }
        }
    }

    return false;
}

bool Resolution::_do_mres_with_compatible_clauses(int c1)
{
    Clause* c = &(this->clauses[c1]);

    // Skip if we have anything other than one modal literal in the form
    // "Box(l)" (l can be negated)
    if ((c->num_literals()!=1) || !c->is_modal(0) || c->is_modal_negated(0))
        return false;

    // See if any other clause has a modal literal in the form "not Box(l)"
    int symbol_id = c->get_literal_id(0);
    bool negated_symbol = c->is_negated(0);

    // For each clause found, check agent and do MRES if agents are equal
    for (int i = 0; i < this->symbols[symbol_id].num_clauses(negated_symbol,
        true, true); i++)
    {
        int c2 = this->symbols[symbol_id].get_clause_id(i, negated_symbol, true,
            true);

        // Skip if they don't have the same agent
        // TODO: throw this to PropositionalSymbol class
        if (this->clauses[c1].get_agent(0) != this->clauses[c2].get_agent(0))
            continue;

        this->_mres(c1, c2);

        // Check if we reached a contradiction
        Clause *gen = &clauses.back();
        if (gen->num_literals() == 0)
            return true;

        // Check if we just generated "TRUE => not _t1", id of _t1 is always 1
        if ((gen->num_literals() == 1)
            && (gen->get_scope() == Clause::SCOPE_TRUE)
            && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
        {
            this->_pres(this->clauses.size()-1, 0, 1);
            return true;
        }
    }

    return false;
}

bool Resolution::_do_gen2_with_compatible_clauses(int c1)
{
    Clause* c = &(this->clauses[c1]);

    // Skip if we have anything other than one modal literal in the form
    // "Box(l)" (l can be negated)
    if ((c->num_literals()!=1) || !c->is_modal(0) || c->is_modal_negated(0))
        return false;

    // See if any other clause has a modal literal in the form "Box(not l)"
    int symbol_id = c->get_literal_id(0);
    bool negated_symbol = c->is_negated(0);
    int agent = this->clauses[c1].get_agent(0);

    for (int i = 0; i < this->symbols[symbol_id].num_clauses(!negated_symbol,
        true, false); i++)
    {
        int c2 = this->symbols[symbol_id].get_clause_id(i, !negated_symbol,
            true, false);

        // Skip if they don't have the same agent
        // TODO: throw this to PropositionalSymbol class
        if (this->clauses[c2].get_agent(0) != agent)
            continue;

        // Find all clauses in the form "not Box(l_2)"
        for (size_t j = 0; j < this->neg_modal_clauses.size(); j++)
        {
            int c3 = this->neg_modal_clauses[j];

            // Skip if agent is different
            if (this->clauses[c3].get_agent(0) != agent)
            {
                continue;
            }

            this->_gen2(c1, c2, c3);

            // Check if we reached a contradiction
            Clause *gen = &clauses.back();
            if (gen->num_literals() == 0)
                return true;
```

```
            // Check if we just generated "TRUE => not _t1", id of _t1 is always 1
            if ((gen->num_literals() == 1)
                && (gen->get_scope() == Clause::SCOPE_TRUE)
                && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
            {
                this->_pres(this->clauses.size()-1, 0, 1);
                return true;
            }
        }
    }

    return false;
}

bool Resolution::_do_ref_with_compatible_clauses(int c1)
{
    Clause* c = &(this->clauses[c1]);

    // Skip if we have anything other than one positive clause
    if ((c->num_literals()!=1) || !c->is_modal(0) || c->is_modal_negated(0))
        return false;

    bool created_clause = this->_ref(c1);

    if (created_clause)
    {
        // Check if we reached a contradiction
        Clause *gen = &clauses.back();
        if (gen->num_literals() == 0)
            return true;

        // Check if we just generated "TRUE => not t1", id of _t1 is always 1
        if ((gen->num_literals() == 1)
            && (gen->get_scope() == Clause::SCOPE_TRUE)
            && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
        {
            this->_pres(this->clauses.size()-1, 0, 1);
            return true;
        }
    }

    return false;
}

bool Resolution::_do_ser_with_compatible_clauses(int c1)
{
    Clause* c = &(this->clauses[c1]);

    // Skip if we have anything other than one positive clause
    if ((c->num_literals()!=1) || !c->is_modal(0) || c->is_modal_negated(0))
        return false;

    bool created_clause = this->_ser(c1);

    if (created_clause)
    {
        // Check if we reached a contradiction
        Clause *gen = &clauses.back();
        if (gen->num_literals() == 0)
            return true;

        // Check if we just generated "TRUE => not t1", id of _t1 is always 1
        if ((gen->num_literals() == 1)
            && (gen->get_scope() == Clause::SCOPE_TRUE)
            && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
        {
            this->_pres(this->clauses.size()-1, 0, 1);
            return true;
        }
    }

    return false;
}

bool Resolution::_do_sym_with_compatible_clauses(int c1)
{
    Clause* c = &(this->clauses[c1]);

    // Skip if we have anything other than one positive clause
    if ((c->num_literals()!=1) || !c->is_modal(0) || c->is_modal_negated(0))
        return false;

    bool created_clause = this->_sym(c1);

    if (created_clause)
    {
        // Check if we reached a contradiction
        Clause *gen = &clauses.back();
        if (gen->num_literals() == 0)
            return true;

        // Check if we just generated "TRUE => not t1", id of _t1 is always 1
        if ((gen->num_literals() == 1)
            && (gen->get_scope() == Clause::SCOPE_TRUE)
```

```
            && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
        {
            this->_pres(this->clauses.size()-1, 0, 1);
            return true;
        }
    }

    return false;
}

bool Resolution::_linear_resolution()
{
    // Try to find two clauses that allow propositional resolution,
    // skip clause 1 (start => t1)
    for (size_t c1 = 1; c1 < this->clauses.size(); c1++)
    {
        // Skip this clause if it is a modal clause
        if (this->clauses[c1].is_modal(0))
        {
            continue;
        }

        bool contradiction_found = this->_propositional_step(c1);
        if (contradiction_found)
        {
            return true;
        }
    }

    // No propositional branch found a contradiction, try GEN1 or GEN3
    bool contradiction_found = this->_try_gen1();
    if (contradiction_found)
    {
        return true;
    }

    contradiction_found = this->_try_gen3();
    if (contradiction_found)
    {
        return true;
    }

    return false;
}

bool Resolution::_propositional_step(int c1)
{
    // Make a vector defining the order to solve this clause's literals
    vector<pair<int,int> > lit_order;
    for (int i = 0; i < this->clauses[c1].num_literals(); i++)
    {
        // Skip literal t1
        if (this->clauses[c1].get_literal_id(i) == 1)
        {
            continue;
        }

        int symbol_id = this->clauses[c1].get_literal_id(i);
        bool negated = this->clauses[c1].is_negated(i);
        int comb_count = this->symbols[symbol_id].num_clauses(!negated);

        lit_order.push_back(make_pair(comb_count, i));
    }
    // Sort by number of resolution possibilities, ASC (start on the literal
    // with the least possibilities for resolution)
    sort(lit_order.begin(), lit_order.end());

    // Try to resolve each literal in the order determined before
    for (vector<pair<int,int> >::iterator it = lit_order.begin();
        it != lit_order.end(); it++)
    {
        int symbol_id = this->clauses[c1].get_literal_id(it->second);
        bool negated = this->clauses[c1].is_negated(it->second);

        // Try with each clause that has a resolution with this literal
        for (int idx = 0;
             idx < this->symbols[symbol_id].num_clauses(!negated); idx++)
        {
            int c2 = this->symbols[symbol_id].get_clause_id(idx, !negated);

            // Check to avoid duplicates (like doing 2,10 and then 10,2)
            if (c1 < c2)
                continue;

            // Apply LRES with the next possible clause
            bool clause_created = this->_pres(c1, c2, symbol_id);

            // If it didn't create a clause (it would be a tautology or repeated
            // clause), skip to the next clause
            if (!clause_created)
                continue;

            // Ok, it did create a clause, let's inspect it
            this->num_total_clauses++;
```

```
            // Check if we reached a contradiction
            Clause *gen = &clauses.back();
            if (gen->num_literals() == 0)
                return true;

            // Check if we just generated "TRUE => not t1", id of _t1 is always 1
            if ((gen->num_literals() == 1)
                && (gen->get_scope() == Clause::SCOPE_TRUE)
                && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
            {
                this->num_total_clauses++;
                this->_pres(this->clauses.size()-1, 0, 1);
                return true;
            }

            // Check if we just repeated the last backtracked clause
            if ((this->num_total_clauses > (int)this->clauses.size())
                && (this->clauses.back() == this->last_backtracked))
            {
                _delete_last_clause();
                continue;
            }

            // Call the propositional step for the newest clause created
            bool contradiction_found = this->_propositional_step(
                this->clauses.size() - 1);

            // Check if the branch found a contradiction
            if (contradiction_found)
                return true;

            // Remove clause generated, since it gave no solution
            _delete_last_clause();
        }
    }

    // No propositional branch found a contradiction, try GEN1 or GEN3
    if (c1 >= this->num_init_and_modal)
    {
        bool contradiction_found = this->_try_gen1();
        if (contradiction_found)
        {
            return true;
        }

        contradiction_found = this->_try_gen3();
        if (contradiction_found)
        {
            return true;
        }
    }

    return false;
}

bool Resolution::_try_gen1()
{
    // For each clause c1 in the form "not Box(not l)", try to find a disjunctive
    // clause with the literal "not l" (l can be negated or not)
    for (vector<int>::iterator it = this->neg_modal_clauses.begin();
        it != this->neg_modal_clauses.end(); it++)
    {
        int c1 = *it;
        int l_symbol_id = this->clauses[c1].get_literal_id(0);
        bool l_negated = this->clauses[c1].is_negated(0);
        int l_agent = this->clauses[c1].get_agent(0);

        // For each disjunctive clause c2 with the literal "not l", see if it is
        // possible to do GEN1
        for (int i=0; i<this->symbols[l_symbol_id].num_clauses(l_negated); i++)
        {
            int c2 = this->symbols[l_symbol_id].get_clause_id(i, l_negated);

            // Try to find clauses in the form "Box(not l_j)" for every other
            // literal "l_j" in c2
            bool gen1_impossible = false;
            set<int> pos_mod_clauses;
            for (int j = 0; j < this->clauses[c2].num_literals(); j++)
            {
                int j_symbol_id = this->clauses[c2].get_literal_id(j);
                bool j_negated = this->clauses[c2].is_negated(j);

                PropositionalSymbol *ps = &(this->symbols[j_symbol_id]);

                // Skip literal "l", it was already matched with c1
                if (j_symbol_id == l_symbol_id)
                {
                    continue;
                }
                // If we found, mark all clauses of that form to be included
                // in GEN1
                else if (ps->num_clauses(!j_negated, true, false) > 0)
                {
                    // TODO: throw agent checking for PropositionalSymbol
                    bool inserted = false;
```

```
                    for (int k = 0; k < ps->num_clauses(
                        !j_negated, true, false); k++)
                    {
                        int k_clause_id = ps->get_clause_id(k, !j_negated, true,
                            false);
                        if (this->clauses[k_clause_id].get_agent(0) == l_agent)
                        {
                            pos_mod_clauses.insert(k_clause_id);
                            inserted = true;
                        }
                    }

                    if (!inserted)
                    {
                        gen1_impossible = true;
                        break;
                    }
                }
                // If there's no modal clause matching, it's impossible to
                // apply GEN1 here, stop inspecting c2 literals
                else
                {
                    gen1_impossible = true;
                    break;
                }
            }

            // If GEN1 is impossible, skip this c2 clause and try for next one
            if (gen1_impossible)
            {
                continue;
            }

            // If GEN1 is possible, apply it and inspect the result
            bool clause_created = this->_gen1(c1, c2, pos_mod_clauses);

            // If no clause was created (tautology or repeated), skip to next c2
            if (!clause_created)
            {
                continue;
            }

            // Check if we reached a contradiction
            this->num_total_clauses++;
            Clause *gen = &clauses.back();
            if (gen->num_literals() == 0)
            {
                return true;
            }

            // Check if we just generated "TRUE => not t1", id of _t1 is always 1
            if ((gen->num_literals() == 1)
                && (gen->get_scope() == Clause::SCOPE_TRUE)
                && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
            {
                this->num_total_clauses++;
                this->_pres(this->clauses.size()-1, 0, 1);
                return true;
            }

            // Check if we just repeated the last backtracked clause
            if ((this->num_total_clauses > (int)this->clauses.size())
                && (this->clauses.back() == this->last_backtracked))
            {
                _delete_last_clause();
                continue;
            }

            // New propositional clause was created, call linear propositional
            // resolution again on the new clause
            bool contradiction_found = this->_propositional_step(
                this->clauses.size() - 1);

            // Check if the branch found a contradiction
            if (contradiction_found)
                return true;

            // Remove clause generated, since it gave no solution
            _delete_last_clause();
        }
    }

    return false;
}

bool Resolution::_try_gen3()
{
    // Give up if there's no negative modal clause
    if (this->neg_modal_clauses.empty())
    {
        return false;
    }

    // A hash to store which propositional clauses were already tested to not
```

```
// being able to use rule GEN3
map<int,bool> discarded_clauses;

// Lookup each positive modal clause in the form "Box(not l1)", skip clause 1
// (start => t1). "l1" can be negated or not.
for (unsigned pos_c1 = 1; pos_c1 < this->clauses.size(); pos_c1++)
{
    // If this is not a positive modal clause, skip it
    if (!this->clauses[pos_c1].is_modal(0)
        || this->clauses[pos_c1].is_modal_negated(0))
    {
        continue;
    }

    // Inspect each propositional clause that has literal "l1"
    unsigned symbol_id = this->clauses[pos_c1].get_literal_id(0);
    bool negated = this->clauses[pos_c1].is_negated(0);
    int agent = this->clauses[pos_c1].get_agent(0);
    for (int i = 0; i < this->symbols[symbol_id].num_clauses(!negated);
        i++)
    {
        unsigned prop_id = this->symbols[symbol_id].get_clause_id(i,
            !negated);

        // If we already tested this propositional clause to not allow GEN3,
        // skip it
        if (discarded_clauses.count(prop_id) > 0)
        {
            continue;
        }

        Clause* prop = &(this->clauses[prop_id]);
        set<int> pos_mod_clauses;
        pos_mod_clauses.insert(pos_c1);
        bool gen3_impossible = false;

        // For each literal in the propositional clause, check if there is
        // at least one positive modal clause that allows GEN3 with the same
        // agent of pos_c1 (the first positive modal clause chosen)
        for (int lit = 0; lit < prop->num_literals(); lit++)
        {
            bool inserted = false;

            unsigned lit_symbol_id = prop->get_literal_id(lit);
            bool lit_negated = prop->is_negated(lit);

            for (int j = 0; j < this->symbols[lit_symbol_id].num_clauses(
                !lit_negated, true, false); j++)
            {
                unsigned pos_ci =
                    this->symbols[lit_symbol_id].get_clause_id(j,
                    !lit_negated, true, false);

                if (pos_ci == pos_c1)
                {
                    inserted = true;
                    continue;
                }

                if (this->clauses[pos_ci].get_agent(0) == agent)
                {
                    pos_mod_clauses.insert(pos_ci);
                    inserted = true;
                }
            }

            if (!inserted)
            {
                gen3_impossible = true;
                break;
            }
        }

        if (gen3_impossible)
        {
            discarded_clauses.insert(make_pair(prop_id,true));
            continue;
        }

        // If GEN3 is possible, choose a negative modal clause, apply GEN3
        // and inspect the results
        for (size_t neg_idx = 0; neg_idx < this->neg_modal_clauses.size();
            neg_idx++)
        {
            unsigned neg_c1 = this->neg_modal_clauses[neg_idx];
            bool clause_created = this->_gen3(pos_mod_clauses, neg_c1,
                prop_id);

            // If no clause was created (tautology or repeated), skip to
            // next negative modal clause
            if (!clause_created)
            {
                continue;
            }
```

```
                // Check if we reached a contradiction
                this->num_total_clauses++;
                Clause *gen = &clauses.back();
                if (gen->num_literals() == 0)
                {
                    return true;
                }

                // Check if we just generated "TRUE => not t1", id of _t1 is 1
                if ((gen->num_literals() == 1)
                    && (gen->get_scope() == Clause::SCOPE_TRUE)
                    && (gen->get_literal_id(0) == 1) && gen->is_negated(0))
                {
                    this->num_total_clauses++;
                    this->_pres(this->clauses.size()-1, 0, 1);
                    return true;
                }

                // Check if we just repeated the last backtracked clause
                if ((this->num_total_clauses > (int)this->clauses.size())
                    && (this->clauses.back() == this->last_backtracked))
                {
                    _delete_last_clause();
                    continue;
                }

                // New propositional clause was created, call linear
                // propositional resolution again on the new clause
                bool contradiction_found = this->_propositional_step(
                    this->clauses.size() - 1);

                // Check if the branch found a contradiction
                if (contradiction_found)
                {
                    return true;
                }

                // Remove clause generated, since it gave no solution
                _delete_last_clause();
            }
        }
    }

    return false;
}

// Propositional Resolution (IRES1, IRES2 and LRES)
bool Resolution::_pres(int c1, int c2, int literal)
{
    int scope = Clause::SCOPE_TRUE;
    if ((this->clauses[c1].get_scope() == 0)
        || (this->clauses[c2].get_scope() == 0))
    {
        scope = 0;
    }

    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(scope));

    ostringstream oss;
    string method;
    if (scope == Clause::SCOPE_TRUE)
        method = "[LRES] ";
    else if (this->clauses[c1].get_scope() != this->clauses[c2].get_scope())
        method = "[IRES1] ";
    else
        method = "[IRES2] ";
    oss << (c1+1) << "," << (c2+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Think about an internal function for these
    for (int i = 0; i < clauses[c1].num_literals(); i++)
    {
        if (clauses[c1].get_literal_id(i) != literal)
        {
            this->_add_literal_to_clause(clauses[c1].get_literal_id(i),
                clause_id, clauses[c1].is_negated(i));
        }
    }
    for (int i = 0; i < clauses[c2].num_literals(); i++)
    {
        if (clauses[c2].get_literal_id(i) != literal)
        {
            bool res = this->_add_literal_to_clause(
                clauses[c2].get_literal_id(i), clause_id,
                clauses[c2].is_negated(i));
            if (!res)
            {
                this->clauses.pop_back();
                return false;
            }
        }
    }
```

```
    // Check if the clause is unique
    this->clauses.back().sort_literals();
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// Basic Modal Resolution (MRES)
bool Resolution::_mres(int c1, int c2)
{
    // Create clause with "TRUE" scope
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(Clause::SCOPE_TRUE));

    // Set creation method for print
    ostringstream oss;
    string method = "[MRES] ";
    oss << (c1+1) << "," << (c2+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add scope variables to the new clause, negated
    this->_add_literal_to_clause(this->clauses[c1].get_scope(), clause_id,
        !this->clauses[c1].is_scope_negated());
    this->_add_literal_to_clause(this->clauses[c2].get_scope(), clause_id,
        !this->clauses[c2].is_scope_negated());

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// GEN1
bool Resolution::_gen1(int c1, int c2, set<int> pos_mod)
{
    // Create clause with "TRUE" scope
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(Clause::SCOPE_TRUE));

    // Set creation method for print
    ostringstream oss;
    string method = "[GEN1] ";
    for (set<int>::iterator it = pos_mod.begin(); it != pos_mod.end(); it++)
    {
        oss << (*it) + 1 << ",";
    }
    oss << (c1+1) << "," << (c2+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add scope variables to the new clause, negated
    for (set<int>::iterator it = pos_mod.begin(); it != pos_mod.end(); it++)
    {
        this->_add_literal_to_clause(this->clauses[*it].get_scope(), clause_id,
            !this->clauses[*it].is_scope_negated());
    }
    this->_add_literal_to_clause(this->clauses[c1].get_scope(), clause_id,
        !this->clauses[c1].is_scope_negated());

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// GEN2
bool Resolution::_gen2(int c1, int c2, int c3)
{
    // Create clause with "TRUE" scope
    int clause_id = this->clauses.size();
```

```
    this->clauses.push_back(Clause(Clause::SCOPE_TRUE));

    // Set creation method for print
    ostringstream oss;
    string method = "[GEN2] ";
    oss << (c1+1) << "," << (c2+1) << "," << (c3+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add scope variables to the new clause, negated
    this->_add_literal_to_clause(this->clauses[c1].get_scope(), clause_id,
        !this->clauses[c1].is_scope_negated());
    this->_add_literal_to_clause(this->clauses[c2].get_scope(), clause_id,
        !this->clauses[c2].is_scope_negated());
    this->_add_literal_to_clause(this->clauses[c3].get_scope(), clause_id,
        !this->clauses[c3].is_scope_negated());

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// GEN3
bool Resolution::_gen3(set<int> pos_mod, int neg_c1, int prop_id)
{
    // Create clause with "TRUE" scope
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(Clause::SCOPE_TRUE));

    // Set creation method for print
    ostringstream oss;
    string method = "[GEN3] ";
    for (set<int>::iterator it = pos_mod.begin(); it != pos_mod.end(); it++)
    {
        oss << (*it) + 1 << ",";
    }
    oss << (neg_c1+1) << "," << (prop_id+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add scope variables to the new clause, negated
    for (set<int>::iterator it = pos_mod.begin(); it != pos_mod.end(); it++)
    {
        this->_add_literal_to_clause(this->clauses[*it].get_scope(), clause_id,
            !this->clauses[*it].is_scope_negated());
    }
    this->_add_literal_to_clause(this->clauses[neg_c1].get_scope(), clause_id,
        !this->clauses[neg_c1].is_scope_negated());

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// REF
bool Resolution::_ref(int c1)
{
    // Create clause with "TRUE" scope
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(Clause::SCOPE_TRUE));

    // Set creation method for print
    ostringstream oss;
    string method = "[REF] ";
    oss << (c1+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add scope variable to the new clause, negated
    this->_add_literal_to_clause(this->clauses[c1].get_scope(), clause_id,
        !this->clauses[c1].is_scope_negated());
    // Add c1 literal to the new clause
    this->_add_literal_to_clause(this->clauses[c1].get_literal_id(0), clause_id,
        this->clauses[c1].is_negated(0));
```

```
    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// SER
bool Resolution::_ser(int c1)
{
    // Create clause with the same scope as c1
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(this->clauses[c1].get_scope(),
        this->clauses[c1].is_scope_negated()));

    // Set creation method for print
    ostringstream oss;
    string method = "[SER] ";
    oss << (c1+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add "not Box(not 1)" to the clause
    this->_add_literal_to_clause(this->clauses[c1].get_literal_id(0), clause_id,
        !this->clauses[c1].is_negated(0), true, true,
        this->clauses[c1].get_agent(0));

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

// SYM
bool Resolution::_sym(int c1)
{
    // Create clause with scope being the literal in c1, negated
    int clause_id = this->clauses.size();
    this->clauses.push_back(Clause(this->clauses[c1].get_literal_id(0),
        !this->clauses[c1].is_negated(0)));

    // Set creation method for print
    ostringstream oss;
    string method = "[SYM] ";
    oss << (c1+1);
    method += oss.str();
    this->clauses.back().set_creation_method(method);

    // Add "Box(not l1)" to the clause
    this->_add_literal_to_clause(this->clauses[c1].get_scope(), clause_id,
                                 !this->clauses[c1].is_scope_negated(), true,
                                 false, this->clauses[c1].get_agent(0));

    // Sort literals
    this->clauses.back().sort_literals();

    // Test for unique clause
    if (this->_last_clause_is_unique())
        this->_add_clause_to_indexes(clause_id);
    else
    {
        this->clauses.pop_back();
        return false;
    }

    return true;
}

void Resolution::import_syntax_tree(SyntaxNode& root)
{
    // Reserve position 0 in agent hash for empty agent
    this->agent_names.push_back("");

    // Look up all clauses from the Syntax Tree
    for (size_t i = 0; i < root.num_children(); i++)
    {
        // We are in the root of a clause
        SyntaxNode *clause_root = root.child(i);
```

```
        // Get scope for the new clause to be inserted in formula
        int scope = this->_get_scope_id(*clause_root);

        // Create clause with the right scope
        int clause_id = clauses.size();
        this->clauses.push_back(Clause(scope));

        // Read literals from clause
        bool clause_added = this->_mount_clause_from_node(*clause_root,
            clause_id);

        // Check if the clause is unique
        if (clause_added)
        {
            this->clauses.back().sort_literals();

            if (!this->_last_clause_is_unique())
                this->clauses.pop_back();
        }
    }

    // Sort initial clauses by number of literals, ASC
    //sort(this->clauses.begin()+1, this->clauses.end());

    // Add generated clauses to symbols' indexes
    for (size_t i = 0; i < this->clauses.size(); i++)
        this->_add_clause_to_indexes(i);

    // Set number of initial clauses generated for better printing of the
    // resolution
    this->num_initial_clauses = this->clauses.size();
}

bool Resolution::print_clauses()
{
    bool ret = true;
    int clause_id = 1;

    for (vector<Clause>::iterator it = this->clauses.begin();
        it != this->clauses.end(); it++)
    {
        int unicode_count = 0;

        if (clause_id == (this->num_initial_clauses + 1))
            cout << "————————————————————————————————" << endl;

        ostringstream oss;
        oss << clause_id << ".␣";

        // Print scope
        if (it->is_scope_negated())
        {
            oss << "\u00AC";
            unicode_count++;
        }
        int scope = it->get_scope();
        switch (scope)
        {
            case 0:
                oss << "_start";
                break;
            case Clause::SCOPE_TRUE:
                oss << "true";
                break;
            default:
                string lit_name = this->symbols[scope].get_name();
                oss << lit_name;
                break;
        }
        oss << "␣=>␣";

        // Prints FALSE in case of contradiction
        if (it->num_literals() == 0)
        {
            oss << "FALSE";
            ret = false;
        }

        for (int i = 0; i < it->num_literals(); i++)
        {
            if (i != 0)
            {
                oss << "␣v␣";
            }

            // Print modal part
            if (it->is_modal(i))
            {
                if (it->is_modal_negated(i))
                {
                    oss << "\u00AC";
                    unicode_count++;
                }
                oss << "\u25A1";
```

```
                unicode_count += 2;
                if (it->get_agent(i) > 0)
                {
                    oss << "[" << this->agent_names[it->get_agent(i)] << "]";
                }
            }

            // Print literal part
            if (it->is_negated(i))
            {
                oss << "not ";
                unicode_count++;
            }
            string lit_name = this->symbols[it->get_literal_id(i)].get_name();
            oss << lit_name;
        }

        cout << setw(60 + unicode_count) << left << oss.str();

        // Print creation_method
        cout << it->get_creation_method();

        cout << endl;
        clause_id++;
    }

    return ret;
}

bool Resolution::resolve()
{
    // Apply inference rules in modal phase ([MRES], [GEN2], and if appliable
    // [REF], [SER], [TRANS], [EUC1], [EUC2] and/or [SYM]
    bool contradiction_found = this->_modal_phase();

    this->num_total_clauses = this->num_init_and_modal = this->clauses.size();

    /** PROPOSITIONAL PHASE **/
    // TODO: change to inspect generated clauses here
    if (!contradiction_found)
    {
        contradiction_found = this->_linear_resolution();
    }

    return contradiction_found;
}

unsigned Resolution::num_clauses()
{
    return this->clauses.size();
}

unsigned Resolution::initial_clauses()
{
    return this->num_initial_clauses;
}

unsigned Resolution::total_clauses()
{
    return this->num_total_clauses;
}
```

A.18 main.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cstring>
 //#include <chrono>
#include <sstream>
using namespace std;

#include "parser_utils.h"
#include "syntax_node.hpp"
#include "errorinfo.h"
#include "translator.hpp"
#include "resolution.hpp"

#define NO_INPUT_FILE 0
#define OPTION_INVALID 1

// Global variables (TODO: put that and everything else inside a namespace)
int verbosity = 1;
bool axiom_t = false;
bool axiom_d = false;
bool axiom_4 = false;
bool axiom_5 = false;
```

```
bool axiom_b = false;
ErrorInfo einfo;

bool try_resolution(string formula)
{
    // Parse expression
    einfo.line = -1;
    SyntaxNode* root = parse_x(formula.c_str(), &einfo);

    if (einfo.line > 0)
    {
        cerr << einfo.msg << endl << endl;
        return false;
    }

    // Debug
    if (verbosity > 1)
    {
        cout << "Syntax Tree before translation to SNF:" << endl;
        cout.flush();
        root->print_tree();
        cout.flush();
        cout << endl;
    }

    // Translate nodes to SNF
    Translator::translate_to_snf(root);

    if (verbosity > 1)
    {
        cout << "Syntax Tree after translation to SNF:" << endl;
        root->print_tree();
        cout << endl;
    }

    // Construct clauses and literals
    Resolution resolution;
    resolution.import_syntax_tree(*root);

    // Resolution
    bool contradiction_found = resolution.resolve();

    if (verbosity > 0)
    {
        resolution.print_clauses();
        cout << endl;

        // Detailed info
        unsigned initial_clauses = resolution.initial_clauses();
        unsigned num_clauses = resolution.num_clauses();
        unsigned total_clauses = resolution.total_clauses();
        cout << initial_clauses << " initial clauses +"
        << total_clauses - initial_clauses
        << " clause(s) generated during resolution";
        if (total_clauses > num_clauses)
        {
            cout << " (" << total_clauses - num_clauses << " backtracked)";
        }
        cout << "." << endl << endl ;
    }

    return contradiction_found;
}

void prove_formula(string formula, int formula_number)
{
    // Gets negated formula to pass to our parser
    string negated_formula = "\u00AC(" + formula + ")";

    if (verbosity > 0)
    {
        cout << "Trying resolution on \u00AC(f" << formula_number << "):"
        << endl;
    }
    bool contradiction_found = try_resolution(negated_formula);

    if (einfo.line > 0)
    {
        return;
    }

    if (verbosity > 0)
    {
        cout << "---" << endl << endl;
    }

    if (contradiction_found)
    {
        cout << "f" << formula_number << ": " << formula << " is valid."
            << endl;
        if (verbosity > 0) cout << endl;
    }
    else
    {
        if (verbosity > 0)
```

```
        {
            cout << "Trying␣resolution␣on␣f" << formula_number << ":"
            << endl;
        }

        bool contradiction_found = try_resolution(formula);

        if (verbosity > 0)
        {
            cout << "---" << endl << endl;
        }

        if (contradiction_found)
        {
            cout << "f" << formula_number << ":␣" << formula
                 << "␣is␣unsatisfiable." << endl;
            if (verbosity > 0) cout << endl;
        }
        else
        {
            cout << "f" << formula_number << ":␣" << formula
                 << "␣cannot␣be␣proven." << endl;
            if (verbosity > 0) cout << endl;
        }
    }
}

/*void show_execution_time(chrono::steady_clock::time_point t1)
{
    if (verbosity == 0) cout << endl;
    chrono::steady_clock::time_point t2 = chrono::steady_clock::now();
    chrono::duration<double> time_span =
    chrono::duration_cast<chrono::duration<double>>(t2 - t1);
    cout << "Program executed in " << time_span.count() << " seconds."
    << endl;
}*/

void parse_error(int error_code, char *argv[], int i = 1)
{
    if (error_code == NO_INPUT_FILE)
    {
        cerr << argv[0] << ":␣no␣input␣file" << endl;
        cerr << "Usage:␣" << argv[0] << "␣[options]␣filename" << endl;
        cerr << "Type␣" << argv[0] << "␣-h␣for␣help" << endl;
        exit(-1);
    }

    if (error_code == OPTION_INVALID)
    {
        cerr << argv[i] << "␣is␣not␣a␣valid␣option" << endl;
        cerr << "Usage:␣" << argv[0] << "␣[options]␣filename" << endl;
        cerr << "Type␣" << argv[0]
             << "␣-h␣for␣help␣on␣the␣options␣available" << endl;
        exit(-1);
    }
}

std::string parse_arguments(int argc, char *argv[])
{
    if (argc < 2) parse_error(NO_INPUT_FILE, argv);

    if (strcmp(argv[1], "-h") == 0)
    {
        cout << "Usage:␣" << argv[0] << "␣[options]␣filename" << endl << endl;

        cout << "Options:" << endl;
        cout << "␣␣-h␣␣␣␣␣Display␣this␣information" << endl;
        cout << "␣␣-t␣␣␣␣␣Add␣axiom␣T␣(\u25A1(p)␣->␣p)␣to␣the␣logic␣system"
             << endl;
        cout << "␣␣-d␣␣␣␣␣Add␣axiom␣D␣(\u25A1(p)␣->␣\u2662(p))␣to␣the␣logic␣system"
             << endl;
        cout << "␣␣-b␣␣␣␣␣Add␣axiom␣B␣(p␣->␣\u25A1(p)␣\u2662(p))␣to␣the␣logic␣system"
        << endl;
        cout << "␣␣-k␣␣␣␣␣Redundant,␣as␣axiom␣K␣is␣always␣included" << endl;
        cout << "␣␣-v0␣␣␣␣Show␣only␣validity␣/␣satisfiability␣of␣each␣formula"
             << endl;
        cout << "␣␣-v1␣␣␣␣(default)␣Show␣resolution␣process␣for␣each␣formula"
             << endl;
        cout << "␣␣-v2␣␣␣␣Show␣translation␣to␣SNF␣and␣resolution␣process␣for␣"
             << "each␣formula" << endl;
        cout << endl << endl;

        cout << "Options␣can␣be␣combined,␣so␣-kdt␣can␣be␣used␣to␣get␣the␣KDT␣"
             << "system␣(-dt␣would␣do␣the␣same)." << endl << endl;

        cout << "Axioms␣4,␣5␣and␣B␣will␣be␣implemented␣soon." << endl
             << endl;

        exit(-1);
    }

    for (int i = 1; i < argc - 1; i++)
    {
        if (argv[i][0] != '-') parse_error(OPTION_INVALID, argv, i);
```

```
        char *ptr = argv[i] + 1;
        while (*ptr != 0)
        {
            if (*ptr == 'k') { /* do nothing */ }
            else if (*ptr == 'd') axiom_d = true;
            else if (*ptr == 't') axiom_t = true;
            else if (*ptr == 'b') axiom_b = true;
            else if (*ptr == '4')
            {
                cerr << "Axiom 4 is not implemented." << endl;
                exit(-1);
            }
            else if (*ptr == '5')
            {
                cerr << "Axiom 5 is not implemented." << endl;
                exit(-1);
            }
            else if (*ptr == 'v')
            {
                ptr++;
                if (*ptr == '0') verbosity = 0;
                else if (*ptr == '1') verbosity = 1;
                else if (*ptr == '2') verbosity = 2;
                else parse_error(OPTION_INVALID, argv, i);
            }
            else parse_error(OPTION_INVALID, argv, i);

            ptr++;
        }
    }

    return string( argv[argc-1] );
}

std::string modal_system()
{
    string system = "K";
    if (axiom_d) system += "D";
    if (axiom_t) system += "T";
    if (axiom_b) system += "B";
    if (axiom_4) system += "4";
    if (axiom_5) system += "5";

    return system;
}

int main(int argc, char *argv[])
{
    try {
        //chrono::steady_clock::time_point t1 = chrono::steady_clock::now();

        string filename = parse_arguments(argc, argv);

        ifstream myfile(filename.c_str());
        if (!myfile.is_open())
        {
            cerr << "Could not open file " << filename << endl;
            cerr << "Usage: " << argv[0] << " [options] filename" << endl;
            cerr << "Type " << argv[0] << " -h for help" << endl;
            exit(-1);
        }

        unsigned formula_number = 0;
        cout << "Using modal system " << modal_system() << endl << endl;

        while (myfile.good())
        {
            string formula;
            getline(myfile, formula);

            // Trim white spaces from the line we just read
            formula.erase(0, formula.find_first_not_of(' '));
            formula.erase(formula.find_last_not_of(' ')+1);

            // Skip blank lines
            if (formula == "")
            {
                continue;
            }

            formula_number++;

            // Print separator if this is not the first formula
            if ((verbosity > 0) && (formula_number > 1))
            {
                for (int i = 0; i < 80; i++)
                {
                    cout << "=";
                }
                cout << endl << endl;
            }

            if (verbosity > 0)
            {
                cout << "Formula f" << formula_number << ": " << formula
```

```
                    << endl << endl;
            }

            prove_formula(formula, formula_number);
        }

        // Execution time
        //show_execution_time(t1);

    } catch (char* e) {
        cerr << e << endl;
    }

    return 0;
}
```