Um provador de teoremas multi-estratégia

Adolfo Gustavo Serra Seca Neto

Tese apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título de Doutor
em
Ciências

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Marcelo Finger

Durante a elaboração deste trabalho o autor recebeu auxílio financeiro da CAPES.

São Paulo, março de 2007.

Um provador de teoremas multi-estratégia

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida por Adolfo Gustavo Serra Seca Neto e aprovada pela Comissão Julgadora.

São Paulo, 1º de março de 2007.

Banca Examinadora:

Prof. Dr. Marcelo Finger (orientador) - IME-USP
Profa. Dra. Renata Wassermann - IME-USP
Prof. Dr. Walter Alexandre Carnielli - CLE-IFCH-UNICAMP
Prof. Dr. Guilherme Bittencourt - DAS-UFSC
Prof. Dr. Mario Benevides - COPPE-UFRJ

Resumo

Nesta tese apresentamos o projeto e a implementação do **KEMS**, um provador de teoremas multi-estratégia baseado no método de tablôs **KE**. Um provador de teoremas multi-estratégia é um provador de teoremas onde podemos variar as estratégias utilizadas sem modificar o núcleo da implementação. Além de multi-estratégia, **KEMS** é capaz de provar teoremas em três sistemas lógicos: lógica clássica proposicional, **mbC** e **mCi**.

Listamos abaixo algumas das contribuições deste trabalho:

- um sistema **KE** para **mbC** que é analítico, correto e completo;
- um sistema **KE** para **mCi** que é correto e completo;
- um provador de teoremas multi-estratégia com as seguintes características:
  - aceita problemas em três sistemas lógicos: lógica clássica proposicional, **mbC** e **mCi**;
  - tem seis estratégias implementadas para lógica clássica proposicional, duas para **mbC** e duas para **mCi**;
  - tem treze ordenadores que são usados em conjunto com as estratégias;
  - implementa regras simplificadoras para lógica clássica proposicional;
  - possui uma interface gráfica que permite a visualização de provas;
  - é de código aberto e está disponível na Internet em http://kems.iv.fapesp.br;
- *benchmarks* obtidos através da comparação das estratégias para lógica clássica proposicional resolvendo várias famílias de problemas;
- sete famílias de problemas para avaliar provadores de teoremas paraconsistentes;
- os primeiros *benchmarks* para as famílias de problemas para avaliar provadores de teoremas paraconsistentes.

Abstract

In this thesis we present the design and implementation of **KEMS**, a multi-strategy theorem prover based on the **KE** tableau inference system. A multi-strategy theorem prover is a theorem prover where we can vary the strategy without modifying the core of the implementation. Besides being multi-strategy, **KEMS** is capable of proving theorems in three logical systems: classical propositional logic, **mbC** and **mCi**.

We list below some of the contributions of this work:

- an analytic, correct and complete **KE** system for **mbC**;
- a correct and complete **KE** system for **mCi**;
- a multi-strategy prover with the following characteristics:
  - accepts problems in three logical systems: classical propositional logic, **mbC** and **mCi**;
  - has 6 implemented strategies for classical propositional logic, 2 for **mbC** and 2 for **mCi**;
  - has 13 sorters to be used alongside with the strategies;
  - implements simplification rules of classical propositional logic;
  - provides a proof viewer with a graphical user interface;
  - it is open source and available on the internet at http://kems.iv.fapesp.br;
- benchmark results obtained by **KEMS** comparing its classical propositional logic strategies with several problem families;
- seven problem families designed to evaluate provers for logics of formal inconsistency;
- the first benchmark results for the problem families designed to evaluate provers for logics of formal inconsistency.

A Deus, por tudo.

Agradecimentos

Ao professor Marcelo Finger, que me orientou, apoiou e encorajou durante todo este processo.

Aos professores Walter Carnielli, Marcelo Coniglio e Ítala D'Ottaviano, que tão bem me acolheram no CLE-Unicamp. Em particular ao Walter pelas críticas e sugestões feitas à tese e ao Coniglio pelos comentários feitos durante o desenvolvimento do trabalho.

À professora Renata Wassermann, pelas críticas e sugestões feitas à tese e durante o desenvolvimento do trabalho.

Aos professores Mario Benevides e Guilherme Bittencourt, pelas críticas e sugestões feitas à tese.

A todos os professores da pós-graduação de Departamento de Ciência da Computação do IME-USP, em especial a Flávio Soares, Fabio Kon, Carlos Eduardo Ferreira e Leliane Nunes. Também ao professor Jacques Wainer, do IC-Unicamp.

Aos que me estimularam e me ajudaram a iniciar o doutorado: Evandro Costa, Guilherme Ataíde, Valdemar Setzer, Liliane, Luiz Elias, Ruy de Queiroz e os amigos e ex-colegas de trabalho do CEFET-AL.

Aos colegas (do CLE, IME, IC-Unicamp e de outros ambientes por onde circulei nestes anos) com quem aprendi bastante: Helio Martins, Ronaldo Silva, Samir, Rodrigo Freire, Juliana Bueno, Juan Carlos, Patrick, André Atanásio, Augusto Devegili, Eduardo Guerra, Eudenia, Mehran, Rudini, Ricardo Damm, João Marcos, Francisco Antonio Doria, Eleonora Moraes, Wellington, Plínio, Marcelo Couto, Leonardo, Vagner, Helio Dias, Eric, Gil e Alexandre Junqueira.

Aos irmãos da Igreja Cristã Maranata, em especial ao Adailton, por me mostrarem o caminho a seguir nos momentos difíceis.

Aos meus pais, Albino e Lídice, aos meus sogros Walter e Waldtraut, à minha tia Eli, aos meus irmãos Albino Júnior e Antônio Augusto, ao meu cunhado Walter Filho, e a todos os meus familiares pelo apoio e encorajamento.

E, finalmente, à minha amada esposa Wiviane, pelo estímulo, apoio, paciência e por ter dado a luz à nossa querida filha Noara, que é motivo de grande alegria para nós.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Um Provador de Teoremas Multi-Estratégia

1.1 Motivação

A Dedução Automática tem sido uma área de pesquisa ativa desde os anos 50 [69]. Os esforços iniciais na área tiveram um efeito profundo no domínio da Inteligência Artificial (IA) e em toda a Ciência da Computação [70]. A Prova Automática de Teoremas (PAT) lida com o desenvolvimento de programas de computador que demonstram que alguma sentença (a conjetura) é uma conseqüência lógica de um conjunto de sentenças (os axiomas e as hipóteses). Sistemas de PAT são utilizados em uma grande variedade de domínios [110], como matemática, inteligência artificial, geração e verificação de software, verificação de protocolos de segurança e verificação de hardware.

A maior parte dos provadores automáticos de teoremas hoje em dia é baseada ou no princípio da resolução [100] ou no procedimento de Davis-Logemann-Loveland[1] [32]. Porém, outros métodos também podem ser utilizados. Os métodos baseados em tablôs são particularmente interessantes para a PAT por existirem em diferentes variedades e para várias lógicas [52]. Além disso, estes métodos não exigem a conversão dos problemas para a forma clausal. Tablôs podem ser utilizados para desenvolver procedimentos de prova para lógica clássica assim como para vários tipos de lógicas não clássicas, como

[1] Também conhecido como procedimento de Davis-Putnam e que é uma forma restrita de resolução.

Lógica Nebulosa [68], *Residuated Logic* [71], lógicas modais e de descrição [46], lógicas sub-estruturais [30], lógicas multi-valoradas [13], e Lógicas de Inconsistência Formal [18].

As regras de inferência dos sistemas de dedução automática, em geral, e dos provadores baseados em métodos de tablôs, em particular, são tipicamente não-determinísticas. Elas dizem o que *pode* ser feito, não o que *deve* ser feito [52]. Logo, para obter um procedimento automatizável, as regras de inferência precisam ser complementadas por um outro componente, geralmente chamado *estratégia* ou *plano de busca*, que fica responsável pelo *controle* da aplicação das regras de inferência [6]. Isto é, as regras de inferência de um método de prova definem um algoritmo não determinístico para encontrar uma prova; uma estratégia é um algoritmo determinístico para encontrar provas neste método. Para cada método de prova, muitas estratégias podem ser definidas. O tamanho das provas assim como o tempo gasto pelo procedimento de prova pode variar imensamente quando são usadas estratégias diferentes.

Algoritmos não determinísticos são utilizados em diversas áreas da ciência da computação como PAT [94], sistemas de reescrita de termos [114], especificação de protocolos [59], especificação formal [81], otimização [10], reconhecimento de padrões [61], e tomada de decisões [84]. Um algoritmo é uma seqüência de passos computacionais que recebe um valor (ou conjunto de valores) como entrada e produz um valor (ou conjunto de valores) como saída [26]. Um algoritmo não determinístico é um algoritmo com um ou mais *pontos de escolha* onde várias continuações diferentes são posssíveis, sem qualquer especificação de qual será escolhida. Uma execução particular de um tal algoritmo escolhe uma das continuações sempre que chega a um ponto de escolha. Portanto, diferentes caminhos de execução do algoritmo aparecem quando ele é aplicado à mesma entrada, e estes caminhos, quando terminam, geralmente produzem diferentes saídas [115].

Algoritmos não determinísticos computam a mesma classe de funções que os algoritmos determinísticos, mas sua complexidade pode ser menor. Qualquer algoritmo não determinístico (AND) pode ser transformado num algoritmo determinístico (AD), possivelmente com uma redução de eficiência exponencial em tempo. Isto é, um AD que percorre todos os caminhos de execução possíveis de um AND polinomial pode ter com-

plexidade de tempo exponencial. Um dos mais importantes problemas em aberto na pesquisa em computação atualmente é a questão "P=NP?" [20, 21]. Informalmente, a resposta a esta questão corresponde a saber se problemas de decisão que podem ser resolvidos em tempo polinomial por um AND podem também ser resolvidos em tempo polinomial por um AD.

O problema da satisfatibilidade (SAT) para lógica proposicional clássica foi o primeiro problema declarado como NP-completo. Um problema de decisão é *NP-completo* se (1) ele está em NP, e (2) qualquer problema em NP pode ser reduzido em tempo polinomial a ele. SAT pode ser descrito como "dada uma fórmula proposicional, decida se ela é ou não satisfatível". Muitos outros problemas de decisão, como problemas de coloração de grafos, problemas de planejamento e problemas de agendamento podem ser codificados em instâncias de SAT.

Um dentre os muitos métodos lógicos que podem ser utilizados para resolver o problema da satisfatibilidade é o sistema **KE**. É um método de tablôs originalmente desenvolvido para lógica clássica por Marco Mondadori e Marcello D'Agostino [31], mas que foi estendido para outros sistemas lógicos. O sistema **KE** foi apresentado como uma melhoria, no aspecto computacional, em relação ao sistema de tablôs analíticos [106]. Apesar de parecido com o sistema de tablôs analíticos, o sistema **KE** é um sistema refutacional que não é afetado pelas anomalias dos sistemas livres de corte [29].

Nós projetamos e implementamos **KEMS**, um provador de teoremas multi-estratégia baseado no método **KE** para lógicas proposicionais clássicas e não-clássicas. Um provador de teoremas multi-estratégia é um provador de teoremas onde podemos variar as estratégias utilizadas sem modificar o núcleo da implementação. Um provador de teoremas multi-estratégia pode ser usado com três objetivos: educacional, exploratório e adaptativo. Com fins educacionais, pode ser utilizado para ilustrar como a escolha de uma estratégia pode afetar a performance de um provador de teoremas. Como uma ferramenta exploratória, um provador de teoremas multi-estratégia pode ser usado para testar novas estratégias e compará-las com outras já existentes. Por fim, podemos ainda imaginar um provador de teoremas multi-estratégia adaptativo, que modifica a estratégia utilizada de

acordo com as características do problema que é submetido ao provador.

A versão atual do **KEMS** implementa estratégias para três sistemas lógicos: lógica clássica proposicional, **mbC** e **mCi**. **mbC** e **mCi** são lógicas paraconsistentes. As lógicas paraconsistentes podem ser usadas para representar teorias inconsistentes porém não triviais [42]. Essas duas lógicas são de uma classe especial de lógicas paraconsistentes, as Lógicas de Inconsistência Formal [18], uma família de lógicas paraconsistentes que internalizam as noções de consistência e inconsistência no nível da linguagem-objeto. Esta família de lógicas tem algumas propriedades interessantes em sua teoria de prova e foi utilizada em algumas aplicações em ciência da computação, como na integração de informação inconsistente em bases de dados [34].

1.1.1 Um Exemplo de Aplicação de Lógicas de Inconsistência Formal

Vamos exibir agora uma plausível aplicação prática (ainda não implementada nem completamente especificada) de lógicas de inconsistência formal[2]. Em primeiro lugar apresentaremos o problema e depois a solução proposta. O problema que queremos ajudar a resolver está relacionado ao atendimento ao parto. De acordo com [93, 39], a cesariana é um tipo de parto que só deve acontecer quando há uma indicação médica correta. Porém, o que se observa na realidade é que algumas vezes ela acontece sem que haja indicação médica, seja porque é mais cômodo para o médico obstetra (que não precisa esperar por um longo trabalho de parto), seja por escolha da própria parturiente (que tem medo da dor). O problema é que nestes casos, quando realizada sem justificativa médica, a cesárea traz mais riscos do que benefícios tanto para o bebê quanto para a parturiente.

Estamos preocupados aqui apenas com o primeiro caso, que ocorre quando o médico obstetra indica uma cesariana sem que haja uma justificativa médica correta. Quando isto acontece, o médico apresenta como justificativa para a realização da operação uma série de motivos que na verdade não são suficientes para justificar a realização de uma cesariana. E algumas parturientes, geralmente por desinformação, aceitam esta indicação incorreta.

[2]Outras aplicações de lógicas paraconsistentes podem ser encontradas em [16]

Nestes casos, dizemos que aconteceu uma 'cesárea desnecessária'. Diversos casos reais de situações como a descrita acima podem ser encontrados em listas de discussão sobre parto na Internet [41, 45, 35].

Para evitar que isto aconteça é necessário que as mulheres informem-se melhor sobre quais são as condições que justificam ou não a realização de uma cesárea. Os livros e listas de discussão citados acima são excelentes fontes de consulta sobre este assunto. Além destes, [44] traz uma abordagem baseada em evidências científicas sobre esta e outras questões relacionadas à gravidez e ao nascimento.

Nossa proposta aqui é um sistema para auxiliar a parturiente a tomar uma decisão informada sobre a finalização do próprio parto. Um sistema que a ajude compreender melhor o diagnóstico médico e, se for o caso, a procurar uma segunda opinião.

Suponha a seguinte situação: a parturiente recebe, antes de entrar em trabalho de parto, o diagnóstico (informal) do médico de que uma cesárea é necessária. Neste diagnóstico o médico elenca uma ou mais justificativas para a realização da cesariana. E algumas vezes (conforme relatado em [41, 45, 35]) estas justificativas não são suficientes para que uma cesárea seja realizada.

O sistema que estamos propondo atuaria da seguinte forma: a parturiente acessaria o sistema, informaria ao sistema as justificativas oferecidas pelo médico e o sistema responderia se a decisão de fazer cesárea é (a) realmente necessária, (b) se é uma decisão questionável ou (c) se a cesárea naquele caso é desnecessária.

A base de regras para o sistema seria algo como:

Bebê em posição pélvica não é indicação absoluta de cesárea.

Bebê pesando mais de 5kg e em posição pélvica é indicação absoluta de cesárea.

Bebê em posição transversa é indicação absoluta de cesárea.

Placenta prévia é indicação absoluta de cesárea.

Gravidez com mais de 40 semanas não é indicação absoluta de cesárea.

Circular de cordão não é indicação absoluta de cesárea.

$\vdots$

Esta base de regras (que chamaremos de **BR**) pode ser representada por fórmulas

lógicas como as abaixo:

$$\text{BEBE\_PELVICO} \rightarrow \neg\ \text{FAZER\_CESAREA}$$
$$(\text{BEBE\_GRANDE} \wedge \text{BEBE\_PELVICO}) \rightarrow \text{FAZER\_CESAREA}$$

Se usarmos **BR** e as justificativas fornecidas pela parturiente ao sistema como premissas, podemos verificar a que conclusões é possível chegar. Vejamos alguns exemplos:

$$\mathbf{BR}, \text{BEBE\_PELVICO} \vdash \neg\ \text{FAZER\_CESAREA}$$
$$\mathbf{BR}, \text{PLACENTA\_PREVIA} \vdash \text{FAZER\_CESAREA}$$
$$\mathbf{BR}, \text{BEBE\_GRANDE}, \text{BEBE\_PELVICO} \vdash \text{FAZER\_CESAREA} \wedge \neg$$
$$\text{FAZER\_CESAREA}.$$

Em lógica clássica, a última dedução acima levaria à seguinte conclusão:

$$\mathbf{BR}, \text{BEBE\_GRANDE}, \text{BEBE\_PELVICO} \vdash X$$

para qualquer fórmula X, pois para quaisquer fórmulas C e X, $(C \wedge \neg C) \vdash X$ em lógica clássica. Esta característica é chamada de *explosividade (explosiveness)* [18]. Isto é, a partir de uma contradição 'A e $\neg A$' tudo é derivável. Em outras palavras, as teorias contraditórias são triviais. Em lógicas de inconsistência formal (e em lógicas paraconsistentes em geral) esta característica é controlada, ou seja, nem toda teoria contraditória é trivial.

Portanto, quando se utiliza lógica clássica para a construção de uma base de conhecimento [99] como esta, se ela contiver uma contradição, todas as fórmulas seguem trivialmente desta base, o que torna extremamente difícil a construção da base. Porém, se utilizarmos um sistema lógico que tolera contradições para descrever uma base de conhecimento, esta pode produzir resultados úteis mesmo que contenha contradições. No sistema em questão, naturalmente podem aparecer contradições entre as diferentes fontes que podemos consultar para criar a base de regras do sistema. Portanto, lógicas de inconsistência formal seriam bastante adequadas para a construção desta base.

Utilizando uma lógica de inconsistência formal, um base de regras como a descrita acima não iria tornar-se inútil na ocorrência de uma contradição. Um sistema baseado em **BR** pode indicar que encontrou uma contradição, e exibir o que o levou a encontrar

esta contradição. E não deixaria de chegar a outras conclusões a partir das justificativas. Deste modo, este sistema orientaria a parturiente sobre como informar-se mais sobre a sua situação, e ela (a parturiente) teria como discutir melhor esta situação ao procurar uma segunda opinião.

Este é apenas um exemplo de um possível uso de lógicas de inconsistência formal. Pode-se pensar em outros. Não conhecemos, porém, nenhuma aplicação de lógicas de inconsistência formal em uso na prática.

1.2 Apresentação e Resumo dos Apêndices

Esta tese contém este capítulo escrito em português e vários apêndices escritos em inglês. Abaixo descrevemos cada um dos apêndices, exceto o Apêndice A que é apenas uma introdução em inglês para os outros apêndices.

1.2.1 Tablôs para Lógica Clássica e Lógicas Paraconsistentes

O Apêndice B apresenta as lógicas com as quais iremos trabalhar: lógica clássica proposicional (em inglês, *classical propositional logic* — **CPL**), **mbC** e **mCi**. As duas últimas são lógicas paraconsistentes, da família de lógicas de inconsistência formal. Em seguida, exibimos alguns sistemas de tablôs relevantes para a nossa análise: o sistema de tablôs analíticos para **CPL**, o sistema de tablôs **KE** para **CPL** e os sistemas de tablôs **KE** que desenvolvemos para **mbC** e **mCi**, entre outros. Por fim, discutimos brevemente a questão da complexidade computacional de alguns sistemas de prova.

1.2.2 Projeto e Implementação do KEMS

No Apêndice C apresentamos o projeto e a implementação do **KEMS**. Primeiramente discutimos provadores de teoremas baseados em tablôs e as idéias que estão por trás do desenvolvimento do **KEMS**. Depois exibimos algumas extensões dos métodos **KE** discutidos no Apêndice B, extensões estas cujo desenvolvimento foi motivado por questões de implementação. Em seguida apresentamos uma breve descrição do sistema, mostrando

sua arquitetura e alguns diagramas de classe. Por fim, cada uma das estratégias implementadas é discutida.

1.2.3 Avaliação do KEMS

Provadores de teoremas são geralmente comparados usando-se *benchmarks* [112]. SATLIB [102] and TPTP [111] são dois exemplos de sítios na Internet que contêm problemas para avaliar provadores de teoremas. Nós avaliamos o **KEMS** para **CPL** usando como benchmarks algumas famílias de problemas difíceis encontradas na literatura [12, 95]. Além disso, desenvolvemos algumas novas famílias especialmente para avaliar o nosso provador em suas estratégias para lógicas paraconsistentes, para as quais não encontramos famílias de problemas na literatura. Apresentamos no Apêndice D todas estas famílias além dos resultados da avaliação.

1.2.4 Conclusão

Desenvolvemos um provador de teoremas multi-estratégia onde podemos variar a estratégia sem modificar o núcleo da implementação. **KEMS** permite descrever várias estratégias de prova para o mesmo sistema lógico, além de permitir implementar diferentes sistemas lógicos.

Avaliamos o **KEMS** com várias instâncias de famílias de problemas (Apêndice D) e nenhuma configuração do **KEMS** obteve resultados incorretos. Algumas destas instâncias, como as de PHP e ST (veja Apêndice D) são reconhecidamente bastante difíceis de provar. Para várias destas instâncias, mesmo algumas de tamanho bem pequeno, o procedimento de busca de prova não terminou no tempo limite estabelecido para nenhum par estratégia-ordenador utilizado nos testes. Algumas instâncias de outras famílias foram difíceis apenas para alguns dos pares estratégia-ordenador.

Executamos os testes com problemas para os três sistemas lógicos implementados. Para lógica clássica proposicional, *Memory Saver Strategy* foi a melhor estratégia para a maior parte das famílias. Porém, nenhum ordenador se destacou como tendo desempenho consistentemente melhor do que os outros. E para as lógicas de inconsistência formal

(**mbC** e **mCi**) nenhuma estratégia ou ordenador se destacou. É importante observar que os resultados obtidos pelo **KEMS** para as famílias de problemas para lógicas de inconsistência formal são os primeiros resultados de avaliação para estas famílias.

Todos os resultados usados nesta tese estão disponíveis em [92]. Estas e outras conclusões, além de uma revisão das contribuições desta tese e algumas sugestões de trabalhos futuros são apresentados no Apêndice E.

1.2.5 Manual do Usuário Simplificado

No Apêndice F descrevemos o procedimento de instalação do **KEMS** (disponível em [92]), além de alguns cenários para sua utilização. Na apresentação dos cenários, as funcionalidades básicas do sistema ficam claras.

1.3 Contribuições

Os seguintes resultados foram obtidos e estão sendo apresentados nesta tese:

- desenvolvemos sistemas **KE** para duas lógicas paraconsistentes: **mbC** e **mCi**. Provamos que os dois são corretos e completos. Além disso, demonstramos que o primeiro é analítico (ver Apêndice B);
- implementamos um provador de teoremas multi-estratégia, chamado **KEMS**, de código aberto e dispondo de uma interface gráfica que permite a visualização de provas. O provador tem 10 estratégias implementadas (seis para **CPL**, duas para **mbC** e duas para **mCi**) além de 13 ordenadores de fórmulas (ver Apêndice C);
- comparamos as estratégias para lógica clássica proposicional observando os resultados da avaliação do **KEMS** com várias famílias de problemas;
- desenvolvemos sete famílias de problemas para avaliar provadores de teoremas paraconsistentes (ver Seção D.1.2) e obtivemos os primeiros benchmarks para sete destas famílias.

1.3.1 Publicações e Submissões

Resultados parciais obtidos durante a elaboração desta tese foram divulgados nas seguintes publicações:

- *Effective Prover for Minimal Inconsistency Logic* [91] – artigo sobre a implementação das estratégias do **KEMS** para **mbC** e sobre os problemas para avaliar provadores para **mbC** e outras lógicas de inconsistência formal;
- *Implementing a Multi-Strategy Theorem Prover* [89] – artigo descrevendo uma versão inicial do **KEMS**;
- *Using Aspect-Oriented Programming in the Development of a Multi-Strategy Theorem Prover* [90] – artigo contendo considerações preliminares sobre o uso de programação orientada a aspectos no desenvolvimento de provadores de teorema multi-estratégia;
- duas versões do artigo *A Multi-Strategy Tableau Prover* [87, 88], artigo este que mostra uma visão geral do **KEMS**.

E os seguintes artigos estão sendo preparados para em breve serem submetidos a conferências e/ou revistas:

- *A KE Tableau for a Logic of Formal Inconsistency* – artigo sobre a implementação das estratégias do **KEMS** para **mCi** e sobre os problemas desenvolvidos para avaliar provadores para **mCi**;
- *Implementing Backjumping in a Multi-Strategy Tableau Prover* – artigo sobre a implementação de uma estratégia com a técnica chamada 'retrossalto' (*backjumping*);
- *Implementing Learning in a a Multi-Strategy Tableau Prover* – artigo sobre a implementação de duas estratégias com a técnica chamada 'aprendizado' (*learning*);
- *A* **KE***-based Multi-Strategy Tableau Prover* – um artigo mais longo contendo os resultados desta tese.

Adolfo Gustavo Serra Seca Neto

A Multi-Strategy Tableau Prover

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science to the Institute of Mathematics and Statistics of the University of São Paulo

Advisor: Prof. Dr. Marcelo Finger

São Paulo

2007

Apêndice A

Introduction

Automated deduction has been an active area of research since the 1950s [69]. Early developments within the automated deduction field have had a profound effect on the Artificial Intelligence (AI) domain, and, indeed, all of Computer Science [70]. Automated Theorem Proving (ATP) deals with the development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). ATP systems are used in a wide variety of domains [110], such as mathematics, AI, software generation, software verification, security protocol verification, and hardware verification.

Most automated theorem provers nowadays are based either on the resolution principle [100] or on the Davis-Logemann-Loveland (DLL) procedure[1] [32], but other methods can also be used. Tableau methods are particularly interesting for theorem proving since they exist in many varieties and for several logics [52]. Besides that, they do not require conversion of input problems to clausal form. Tableaux can be used for developing proof procedures in classical logic as well as several kinds of non-classical logics, such as Fuzzy Logic [68], Residuated Logic [71], Modal and description logics [46], Substructural logics [30], Many Valued Logics [13], and Logics of Formal Inconsistency [18].

The inference rules of automated deduction systems in general and tableau provers in particular are typically non-deterministic in nature. They say what can be done, not what must be done [52]. Thus, in order to obtain a mechanical procedure, inference rules

[1]Which is a restricted form of resolution also known as Davis-Putnam procedure.

need to be complemented by another component, usually called *strategy* or *search plan*, which is responsible for the *control* of the inference rules [6]. That is, the inference rules of a proof method (or of an automated deduction system based on this proof method) define a nondeterministic algorithm for finding a proof; a strategy is a (deterministic) algorithm for finding proofs in this method. For each proof method, many strategies can be defined. The size of proofs as well as the time spent by the proof procedure can vary greatly as different strategies are used.

Nondeterministic algorithms are used in several areas in computer science such as automated theorem proving [94], term-rewriting systems [114], protocol specification [59], formal specification [81], optimization [10], pattern recognition [61], and decision making [84]. An algorithm is a sequence of computational steps that takes a value (or set of values) as input and produces a value (or set of values) as output [26]. A nondeterministic algorithm is an algorithm with one or more choice points where multiple different continuations are possible, without any specification of which one will be taken. A particular execution of such an algorithm picks a choice whenever such a point is reached. Thus, different execution paths of the algorithm arise when it is applied to the same input, and these paths, when they terminate, generally produce different output [115].

Nondeterministic algorithms compute the same class of functions as deterministic algorithms, but the complexity may be lower. Every nondeterministic algorithm can be turned into a deterministic algorithm, possibly with exponential slow down. That is, a deterministic algorithm that traces all possible execution paths of a polynomial time nondeterministic algorithm may have exponential time complexity. One of the most important open research problems in computer science nowadays is the "P=NP?" question [20, 21]. Informally speaking, the answer to this question corresponds to knowing if decision problems that can be solved by a polynomial-time nondeterministic algorithm can also be solved by polynomial-time deterministic algorithm.

The satisfiability problem (SAT) for classical propositional logic was the first known NP-complete problem. A decision problem is NP-complete if (1) it is in NP, and (2) any problem in NP can be reduced in polynomial time to it. SAT can be described as "given

a propositional formula, decide whether or not it is satisfiable". Many other decision problems, such as graph coloring problems, planning problems, and scheduling problems can be encoded into SAT.

One of many logical methods that can be used to solve the satisfiability problem is the **KE** system. It is a tableau method originally developed for classical logic by Marco Mondadori and Marcello D'Agostino [31], but that has been extended for other logical systems. The **KE** system was presented as an improvement, in the computational sense, over traditional Analytic Tableaux [106]. It is a refutation system, that though close to the analytic tableau method, is not affected by the anomalies of cut-free systems [29].

We have designed and implemented **KEMS**, a multi-strategy theorem prover based on the **KE** method for propositional logics. A multi-strategy theorem prover is a theorem prover where we can vary the strategy without modifying the core of the implementation. A multi-strategy theorem prover can be used for three purposes: educational, exploratory and adaptive. For educational purposes, it can be used to illustrate how the choice of a strategy can affect performance of the prover. As an exploratory tool, a multi-strategy theorem prover can be used to test new strategies and compare them with others. And we can also think of an adaptive multi-strategy theorem prover that changes the strategy used according to features of the problem presented to it.

KEMS current version implements strategies for three logics: classical propositional logic and two paraconsistent logics: **mbC** and **mCi**. Paraconsistent logics are tools for reasoning under conditions which do not presuppose consistency [18]. We have chosen to implement **KEMS** for **mbC** and **mCi** because these two logics are the simplest logics from the family of Logics of Formal Inconsistency [18], a family of paraconsistent logics that internalize the notions of consistency and inconsistency at the object-language level. This family of logics has some nice proof-theoretic features and have been used in some computer science applications such as the integration of inconsistent information in multiple databases [34].

A.1 Overview

Appendix B discusses the logical systems implemented in **KEMS**, their tableau methods, and their complexity. It also provides the necessary background and the basic notions which will be used in the rest of the document. Appendix C presents **KEMS** design and implementation, discussing in more detail the implemented strategies. In Appendix D we exhibit the problems used to evaluate **KEMS** as well as the results obtained in the evaluation. Concluding remarks, a review of this thesis contributions and some suggestions for future works are presented in Appendix E. Next, a brief **KEMS** user manual is shown in Appendix F.

Apêndice B

Tableaux for Classical and Paraconsistent Logics

The method of tableaux is a formal proof procedure existing in many varieties and for several logics [52]. It is a refutation procedure — that is, in order to prove that a formula X is valid we try to invalidate its negation. Besides that, tableau systems are *expansion systems*, i.e., they are systems that contain a finite set of expansion rules [29]. A *tableau* is a tree[1] [26] with one or more branches[2], whose nodes are lists of formulas. The proof search procedure of each specific tableau method describes how to construct a tableau proof (for a list of formulas that one wants to refute) using the expansion rules.

In this appendix we briefly present two tableau systems for classical propositional logic: analytic tableaux [106] and the **KE** inference system [29]. After that, we show **KE** systems we have developed for **mbC** and **mCi**, two paraconsistent logics, and prove some properties of these systems. We finish by discussing the complexity of some of the logical systems and proof methods presented.

[1] In fact, it is better to represent a tableau as a sequence of trees, as it is done in [17], to describe the history of the derivation.

[2] Some of the terms used in this appendix will become clear only in Section C.2.1 when we will describe the **KE** Proof Search Procedure.

B.1 Logical Systems

In this section we present some notions[3] about logical systems which will be used in the rest of this thesis. We assume familiarity with the syntax and semantics of *classical propositional logic*, from now on denoted by **CPL**.

The language of every logic **L** is defined over a propositional *signature* $\Sigma = \{\Sigma_n\}_{n\in\omega}$ such that Σ_n is the set of connectives of arity n. As can be seen in [19], 'ω' is the smallest infinite ordinal, which is the order type of the natural numbers; it can be identified with the set of natural numbers. Therefore, each Σ_n is at most enumerable. The cardinality of each Σ_n is less than or equal to $\aleph_0$ (the cardinal of the set of natural numbers $\mathbb{N}$ [79]); in fact, each Σ_n can be written as a family with indices in ω [74].

The set $\mathcal{P} = \{p_n : n \in \omega\}$ is the set of *propositional variables* (or *atomic formulas*) from which we freely generate the algebra *For* of formulas using Σ. From here on, Σ will denote the signature containing the binary connectives '$\wedge$', '$\vee$', '$\rightarrow$', and the unary connective '$\neg$'. By *For* we will denote the set of formulas freely generated by $\mathcal{P}$ over Σ.

In the same spirit, Σ° ($\Sigma^\bullet$) will denote the signature obtained by the addition of a new unary connective '$\circ$' ('$\bullet$') to the signature Σ, and For° ($For^\bullet$) will denote the algebra of formulas for the signature Σ° ($\Sigma^\bullet$). '$\circ$' and '$\bullet$' are called, respectively, the 'consistency' and 'inconsistency' connectives.

The other connectives' names are the following: '$\wedge$' is the 'and' connective, also called 'conjunction'; '$\vee$' is the 'or' connective, also called 'disjunction'; '$\rightarrow$' is the 'implies' connective, also called 'implication'; and '$\neg$' is the 'not' connective, also called 'negation'.

Given a formula A, the set of its *subformulas*, $sf(A)$, is defined recursively in the following way:

- If p is a propositional atom, then $sf(p) = \{p\}$;
- If A is $\odot A_1$, where $\odot$ is a unary connective, then $sf(A) = \{A\} \cup sf(A_1)$;
- If A is $A_1 \oslash A_2$, where $\oslash$ is a binary connective, then $sf(A) = \{A\} \cup sf(A_1) \cup sf(A_2)$.

[3] Most of the notions presented here were taken from [18].

Let $\wp(X)$ be the powerset of a set X. As usual, given a set *For* of formulas, we say that $\vdash$ defines a *(Tarskian) consequence relation* on *For*, where $\vdash \subseteq \wp(For) \times For$, if the following clauses hold, for any formulas A and B, and subsets Γ and Δ of *For* (formulas and commas at the left-hand side of $\vdash$ denote, as usual, sets and unions of sets of formulas) [18]:

$A \in \Gamma$ implies $\Gamma \vdash A$ (reflexivity)

($\Delta \vdash A$ and $\Delta \subseteq \Gamma$) implies $\Gamma \vdash A$ (monotonicity)

($\Delta \vdash A$ and $\Gamma, A \vdash B$ implies $\Gamma, \Delta \vdash B$ (cut)

A *(Tarskian) logic* **L** is a structure of the form $\langle For, \vdash \rangle$, containing a set of formulas and a consequence relation defined on this set [18]. A logical system is a pair $(\vdash, \mathrm{S}_{\vdash})$, where $\vdash$ is a consequence relation (a set of pairs) and $\mathrm{S}_{\vdash}$ is an algorithmic system for generating all the pairs in that relation [54]. For a given consequence relation, there can be many algorithmic systems. For **CPL**, for instance, we have axiomatic systems (see Section B.1.1), the Davis-Putnam procedure [33, 32], the Resolution method [100] and many others.

B.1.1 Classical Propositional Logic

According to [17], the *axiomatic method* is the oldest known proof method. An *axiomatic system* [78] is composed of a set of axioms and a set of inference rules. An *axiom* is a starting point for deducing logically valid propositions, a formula which is taken for granted as valid. And an *inference rule* is a relation between formulas. As stated in [78], for each inference rule $\mathcal{R}$, there is a unique positive integer j such that for every set of j formulas and each formula C, one can effectively decide whether the given j formulas and each C are related to C in $\mathcal{R}$, and, if so, C is said to *follow from* or to be a *direct consequence of* the given formulas by virtue of $\mathcal{R}$. In an inference rule, the j formulas are called *premises* and C is called the *conclusion* of the inference rule.

An *axiom schema* is a formula such that any formula obtained by one or more substitutions in it is taken to be an axiom [67]. Similarly, an *inference rule schema* is an inference rule where one or more substitutions can be performed on the formulas in the

inference rule relation (see [17]).

The following definitions were taken from [78]: a *proof* in an axiomatic system AS is a sequence $F_1, \ldots, F_n$ of formulas such that, for each i, either F_i is an axiom of AS or F_i is a direct consequence of some of the preceding formulas in the sequence by virtue of one of the inference rules of AS. A *theorem* of AS is a formula F such that F is the last formula of some proof in AS. A formula C is said to be a *consequence* in AS of a set Γ of formulas if and only if there is a sequence $F_1, \ldots, F_k$ of formulas such that C is F_k and, for each i, either F_i is an axiom or F_i is in Γ, or F_i is a direct consequence by some inference rule of some of the preceding formulas in the sequence. Such a sequence is called a *proof* (or *deduction*) *of* C *from* Γ. The members of Γ are called the *hypotheses* or *premises* of the proof.

Given an axiomatic system for a logic **L,** we write $\Gamma \vdash_{\mathbf{L}} A$ to say that there is proof in **L** of A from the premises in Γ [18]. Axiomatization is the process of defining an axiomatic system for a given logical system. Classical propositional logic can be axiomatized in several ways. The following is an axiomatization for **CPL** [18] that contains axiom and inference rule schemas in the signature Σ:

Axiom schemas:

(Ax1) $A \to (B \to A)$

(Ax2) $(A \to B) \to ((A \to (B \to C)) \to (A \to C))$

(Ax3) $A \to (B \to (A \wedge B))$

(Ax4) $(A \wedge B) \to A$

(Ax5) $(A \wedge B) \to B$

(Ax6) $A \to (A \vee B)$

(Ax7) $B \to (A \vee B)$

(Ax8) $(A \to C) \to ((B \to C) \to ((A \vee B) \to C))$

(Ax9) $A \vee (A \to B)$

(Ax10) $A \vee \neg A$

(exp) $A \to (\neg A \to B)$

Inference rule schema:

(MP) $$\frac{A, A \to B}{B}$$

Let $\mathbf{2} \stackrel{\text{def}}{=} \{0, 1\}$ be the set of truth-values, where 1 denotes the 'true' value and 0 denotes the 'false' value. By defining a valuation, we can inductively define the truth-value of a formula from the truth-value of its propositional variables. Below we present a definition of a valuation for **CPL**:

Definition B.1.1. *[18] A* **CPL**-valuation *is any function* v : For $\longrightarrow$ **2** *subject to the following clauses:*

(v1) $v(A \wedge B) = 1$ *iff* $v(A) = 1$ *and* $v(B) = 1$*;*

(v2) $v(A \vee B) = 1$ *iff* $v(A) = 1$ *or* $v(B) = 1$*;*

(v3) $v(A \rightarrow B) = 1$ *iff* $v(A) = 0$ *or* $v(B) = 1$*;*

(v4) $v(\neg A) = 1$ *iff* $v(A) = 0$.

A formula X is said to be *satisfiable* if truth-values can be assigned to its propositional variables in a way that makes the formula true, i.e. if there is at least one valuation such that $v(X) = 1$. A formula is a *tautology* if all possible valuations make the formula true. In **CPL**, for instance, $A \vee B$ is satisfiable, but it is not a tautology, while $A \vee \neg A$ is a tautology.

Let Γ be a set of formulas in *For*, and A a formula in *For*. We say that A is a semantical consequence of Γ (denoted by $\Gamma \models A$) if for any valuation v we have the following [17]:

$$\text{if } v(B) = 1 \text{ for all } B \text{ in } \Gamma, \text{ then } v(A) = 1.$$

The **CPL** axiomatization we have presented above is sound and complete with respect to the semantical consequence relation presented above. That is, for any Γ and A, $\Gamma \vdash_{\mathbf{CPL}} A$ implies $\Gamma \models_{\mathbf{CPL}} A$ (soundness [17]). And $\Gamma \models_{\mathbf{CPL}} A$ implies $\Gamma \vdash_{\mathbf{CPL}} A$ (strong completeness [17]).

B.1.2 Logics of Formal Inconsistency

Logics of Formal Inconsistency (**LFI**s) are a class of paraconsistent logics. Below we reproduce a short definition of paraconsistent logics taken from [42]:

A theory T is said to be inconsistent (contradictory) if it has as theorems a formula and its negation; otherwise, T is consistent (non-contradictory). A theory T is said to be trivial if every formula of its language is a theorem; otherwise T is non-trivial.

If a theory T has as its underlying logic classical logic, the deduction of a contradiction leads to its trivialization.

A logic is paraconsistent if it can be used as the underlying logic to inconsistent but non-trivial theories, which we call paraconsistent theories.

Logics of Formal Inconsistency are paraconsistent logics that internalize the notions of consistency and inconsistency at the object-language level (read more about the foundations of **LFI**s in [14]). Below we present some definitions (taken from [18]) which are necessary to give a formal characterization of **LFI**s.

The *Principle of Explosion* states that a logic **L** is explosive when $\forall\Gamma\forall A\forall B(\Gamma, A, \neg A \vdash B)$. It is well known that **CPL** is explosive. To define a Gentle Principle of Explosion, we first have to define *Gently Explosive Theories*. Consider a (possibly empty) set $\bigcirc(A)$ of formulas which depends only on the formula A. This is the set of formulas that, along with A and $\neg A$, makes a given theory Γ explode. We will call a theory Γ *gently explosive* (*with respect to* $\bigcirc(A)$) if there are formulas A and B such that the following hold:

1. $\bigcirc(A), A \nvdash B$;
2. $\bigcirc(A), \neg A \nvdash B$; and
3. $\forall A\forall B(\Gamma, \bigcirc(A), A, \neg A \vdash B)$.

The *Gentle Principle of Explosion* states that a logic **L** is said to be *gently explosive* when there is a set $\bigcirc(A)$ such that all of the theories of **L** are gently explosive (with respect to $\bigcirc(A)$). Finally, a Logic of Formal Inconsistency is defined as any logic in which the Principle of Explosion does not hold, but the Principle of Gentle Explosion does.

B.1.3 mbC, A Fundamental LFI

The logic **mbC** is the weakest **LFI** based on classical logic [18]. Any **LFI** based on classical logic can be axiomatized starting from positive classical logic ($\mathbf{CPL}^{+}$), whose axiomatization is that of **CPL** without the **(exp)** axiom schema. **mbC** is the weakest of such logics because all other **LFI**s based on classical logic presented in [18] prove more theorems. **mbC** axiomatization is obtained from $\mathbf{CPL}^{+}$'s axiomatization, over the signature Σ°, by adding the following axiom schema:

(bc1) $\circ A \rightarrow (A \rightarrow (\neg A \rightarrow B))$

The following valuation for **mbC** was presented in [18]:

Definition B.1.2. *An* **mbC***-valuation is any function* $v : \text{For}^{\circ} \longrightarrow \mathbf{2}$ *subject to* **(v1)-(v3)** *from Definition B.1.1 and the following clauses:*

(v4') $v(\neg A) = 0$ *implies* $v(A) = 1$*;*

(v5) $v(\circ A) = 1$ *implies* $v(A) = 0$ *or* $v(\neg A) = 0$*.*

The definition of satisfiability in Section B.1.1 also holds for **mbC**. For instance, $A \vee \neg A$ is a tautology in **CPL**, but not in **mbC**. The formula $\neg(A \wedge \neg A \wedge \circ A)$ is an **mbC**-tautology. The intended reading of $\circ A$ is 'A is consistent'. In **mbC**, $\circ A$ is logically independent from $\neg(A \wedge \neg A)$, that is, $\circ$ is a primitive unary connective, not an abbreviation depending on conjunction and negation, as it happens in da Costa's C_n hierarchy of paraconsistent logics [27].

If $\vdash_{\mathbf{mbC}}$ denotes the consequence relation of **mbC**, then we obtain, by **(MP)**:

$$\circ A, A, \neg A \vdash_{\mathbf{mbC}} B \tag{B.1}$$

The *Finite Gentle Principle of Explosion* says that a logic **L** will be said to be *finitely gently explosive* when there is a *finite* set $\bigcirc(A)$ such that all of the theories of **L** are *finitely* gently explosive (with respect to $\bigcirc(A)$). According to [18], Rule (B.1) can be read as saying that 'if A is consistent and contradictory, then it explodes', and amounts to a realization of the Finite Gentle Principle of Explosion.

B.1.4 The mCi Logic

The **mCi** logic is another **LFI** presented in [18]. The motivation for its development was to enrich **mbC** so as to be able to define an inconsistency connective by the direct use of the paraconsistent negation, that is, by setting $\bullet A \stackrel{\text{def}}{=} \neg \circ A$. In **mCi**, $\bullet A$ and $\neg \bullet A$ are logically indistinguishable from $\neg \circ A$ and $\circ A$, respectively.

The logic **mCi** is obtained from **mbC** by the addition of the following axiom schemas:

(ci) $\neg \circ A \rightarrow (A \wedge \neg A)$;

(cc)$_n$ $\circ \neg^n \circ A \quad (n \geq 0)$.

To the above axiomatization is added the definition of an inconsistency connective $\bullet$ by setting $\bullet A \stackrel{\text{def}}{=} \neg \circ A$.

It is easy to verify (and it was shown in [18]) that $A \wedge \neg A \vdash_{\text{mbC}} \neg \circ A$. The converse property does not hold in **mbC** and it was the first additional axiom **(ci)** added to obtain **mCi**. So $\neg \circ A$ and $(A \wedge \neg A)$ are equivalent in **mCi**. To make formulas of the form $\neg \circ A$ 'behave classically', and to obtain a logic that is controllably explosive in contact with formulas of the form $\neg^n \circ A$, where $\neg^0 A \stackrel{\text{def}}{=} A$ and $\neg^{n+1} A \stackrel{\text{def}}{=} \neg \neg^n A$, axioms **(cc)**$_n$ were added to obtain **mCi**. With these axioms added, any formula of the form $\neg^n \circ A$ 'behaves classically' and $\{\neg^n \circ A, \neg^{n+1} \circ A\}$ in an explosive theory in **mCi**. Much more about **mCi** can be found in [18].

The following valuation for **mCi** was presented in [18]:

Definition B.1.3. *An* **mCi***-valuation is an* **mbC***-valuation* $v : \text{For}^\circ \longrightarrow \mathbf{2}$ *(see Definition B.1.2) satisfying, additionally, the following clauses:*

(v6) $v(\neg \circ A) = 1$ *implies* $v(A) = 1$ *and* $v(\neg A) = 1$*;*

(v7.n**)** $v(\circ \neg^n \circ A) = 1$ *for* $n \geq 0$.

The definition of satisfiability in Section B.1.1 also holds for **mCi**. In **mCi**, $(\bullet A) \rightarrow (A \wedge \neg A)$, which is by definition equal to $(\neg \circ A) \rightarrow (A \wedge \neg A)$, is a tautology. In **mbC**, the second formula is not a tautology.

B.2 Tableau Systems

In this section we will present some tableau systems for the logical systems presented in Section B.1. We will discuss their origin and motivation, present their rules, as well as prove some properties. We will not discuss here some aspects of the tableau systems which are relevant only for implementation. These aspects will be discussed in Appendix C.

B.2.1 Analytic Tableaux for CPL

The *analytic tableau* (**AT**) method, also known as *semantic tableaux*, is surely the most studied tableau method. It was presented in [106] as "an extremely elegant and efficient proof procedure for propositional logic". According to [52], this method is a variant of Beth's *"semantic tableaux"* [5], and of Hintikka methods [60].

The **AT** for **CPL** is a sound and complete proof system for **CPL** [106]. Its expansion rules (for the connectives in Σ) are presented in Figure B.1. This is the signed formula version of the **AT** for **CPL** (an unsigned version is also presented in [106]). A *signed formula* is an expression $\mathcal{S}\,X$ where $\mathcal{S}$ is called the *sign* and X is a propositional *formula*. The symbols $\mathtt{T}$ and $\mathtt{F}$, respectively representing the 'true' and 'false' truth-values, can be used as signs. The *conjugate* of a signed formula $\mathtt{T}\,A$ ($\mathtt{F}\,A$) is $\mathtt{F}\,A$ ($\mathtt{T}\,A$). Following [29], we shall call *subformulas* of a signed formula $\mathcal{S}A$ (where $\mathcal{S} \in \{\mathtt{T}\,, \mathtt{F}\,\}$) all the formulas of the form $\mathtt{T}\,B$ or $\mathtt{F}\,B$ where B is a subformula of A.

B.2.2 A KE System for CPL

The **KE** inference system is a more recent tableau method [31]. It was developed by Marco Mondadori and discussed in detail in several works authored or co-authored by Marcello D'Agostino [28, 8, 29]. The **KE** system was presented as an improvement, in the computational efficiency sense, over Analytic Tableaux. It is a refutation system that, though close to the analytic tableau method, is not affected by the anomalies of cut-free systems [29].

The **KE** system for **CPL** is a refutation system that is sound and complete. The

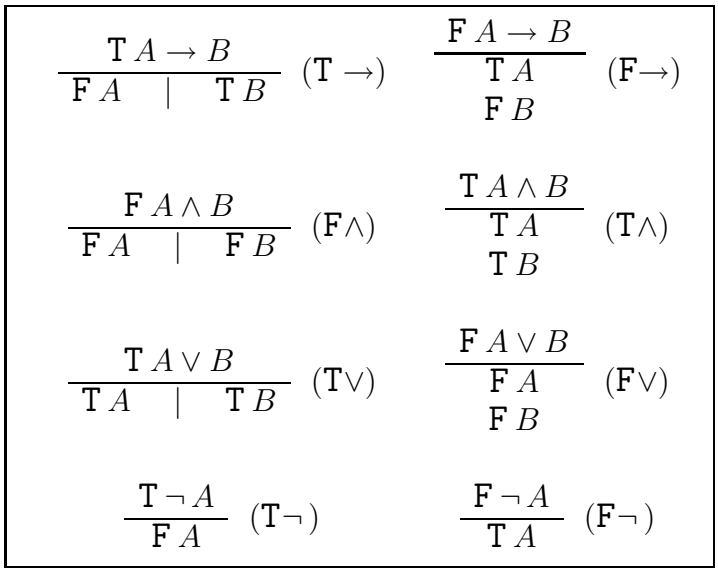


Figure B.1: **CPL AT** rules.

first motivation for its development was to obtain a tableau method inline with classical principles. In [29], D'Agostino argues that analytic tableau rules for **CPL** are not really classical since one of the two principles that form the basis of the classical notion of truth, the Principle of Bivalence, is not clearly present in that tableau system. The Principle of Bivalence (also known as the Principle of the Excluded Middle) states that every proposition is either true or false, and there are no other possibilities. The other principle, the Principle of Non-contradiction, which asserts that no proposition can be true and false at the same time, is embodied in the definition of closed branches in analytic tableaus together with the rules for negation.

The second motivation for **KE** development was to design a computationally more efficient system. Analytic tableau refutations are intrinsically redundant because, depending on the problem being tackled, it is necessary to prove again the same subproblem one or more times [29]. This difficulty is not related to any intrinsic difficulty of the problem considered but only to the redundant behavior of analytic tableau rules. Analytic tableaux correspond to cut-free sequent calculus [56] while **KE** corresponds to sequent calculus with *Cut* rule. Several families of problems have only exponential size proofs in cut-free sequent calculus, but can have polynomial size proofs in sequent calculus with Cut. Therefore, the **KE** system solves this limitation of analytic tableaux by presenting

a set of rules that does not have **AT** rules' redundant behavior.

So what exactly is the main difference between the **KE** system and **AT**? A **KE** system is a tableau system with *only one* branching rule, the Principle of Bivalence (PB) rule. And although this rule resembles Gentzen's sequent calculus [56] *Cut* rule, which is not essential for the sequent calculus proof system, (PB) is not eliminable from **KE**.

The **KE** expansion rules for **CPL** are presented in Figure B.2. Notice that some rules have two premises, some have one premise and the (PB) rule is a zero premise rule. We say that the rules with two premises have a *main premise* (the more complex) and an *auxiliary premise*. For rules with only one premise, this premise is also referred as main premise. A linear rule is a rule which does not force branching. All **KE** rules are linear, except the (PB) rule.

Some rules in Figure B.2 can be derived from others. For instance, if we have ($\mathtt{T}\vee_1$) and (PB) we can derive ($\mathtt{T}\vee_2$). It is clear that the opposite is true: if we have ($\mathtt{T}\vee_2$) and (PB) we can derive ($\mathtt{T}\vee_1$). This relationship also happens between ($\mathtt{F}\wedge_1$) and ($\mathtt{F}\wedge_2$), and between ($\mathtt{T}\rightarrow_1$) and ($\mathtt{T}\rightarrow_2$).

$$\frac{\mathtt{T}\,A\rightarrow B \quad \mathtt{T}\,A}{\mathtt{T}\,B}\ (\mathtt{T}\rightarrow_1) \qquad \frac{\mathtt{T}\,A\rightarrow B \quad \mathtt{F}\,B}{\mathtt{F}\,A}\ (\mathtt{T}\rightarrow_2) \qquad \frac{\mathtt{F}\,A\rightarrow B}{\begin{array}{c}\mathtt{T}\,A\\ \mathtt{F}\,B\end{array}}\ (\mathtt{F}\rightarrow)$$

$$\frac{\mathtt{F}\,A\wedge B \quad \mathtt{T}\,A}{\mathtt{F}\,B}\ (\mathtt{F}\wedge_1) \qquad \frac{\mathtt{F}\,A\wedge B \quad \mathtt{T}\,B}{\mathtt{F}\,A}\ (\mathtt{F}\wedge_2) \qquad \frac{\mathtt{T}\,A\wedge B}{\begin{array}{c}\mathtt{T}\,A\\ \mathtt{T}\,B\end{array}}\ (\mathtt{T}\wedge)$$

$$\frac{\mathtt{T}\,A\vee B \quad \mathtt{F}\,A}{\mathtt{T}\,B}\ (\mathtt{T}\vee_1) \qquad \frac{\mathtt{T}\,A\vee B \quad \mathtt{F}\,B}{\mathtt{T}\,A}\ (\mathtt{T}\vee_2) \qquad \frac{\mathtt{F}\,A\vee B}{\begin{array}{c}\mathtt{F}\,A\\ \mathtt{F}\,B\end{array}}\ (\mathtt{F}\vee)$$

$$\frac{\mathtt{T}\,\neg A}{\mathtt{F}\,A}\ (\mathtt{T}\neg) \qquad \frac{\mathtt{F}\,\neg A}{\mathtt{T}\,A}\ (\mathtt{F}\neg)$$

$$\frac{}{\mathtt{T}\,A \quad | \quad \mathtt{F}\,A}\ (\mathrm{PB})$$

Figure B.2: **CPL KE** rules.

B.2.3 A KE System for mbC

In [11], Caleiro et alli exhibit a way of effectively constructing the two-valued semantics of any logic that has a truth-functional finite-valued semantics and a sufficiently expressive language. From there, one can provide those logics with adequate canonical systems of sequents or tableaux. The method permits one to obtain a complete tableau system for any propositional logic which has a complete semantics given through the so-called 'dyadic valuations'. In [18], sound and complete tableau systems for **mbC** and **mCi** obtained by using this general method are presented[4]. Let us call these systems $\mathbf{C^3M}$ tableau systems.

The $\mathbf{C^3M}$ tableau system rules for **mbC** are shown in Figure B.3. It is easy to notice that the rules for the binary connectives in Σ are the same as that from **AT** (see Figure B.1). It also has **AT** ($\mathtt{F}\neg$) rule but does not have **AT** ($\mathtt{T}\neg$) rule. To compensate for this, it has two additional rules: a branching rule similar to **KE** (PB) rule, and a ($\mathtt{T}\circ$) rule. In total, this tableau system has 5 branching rules.

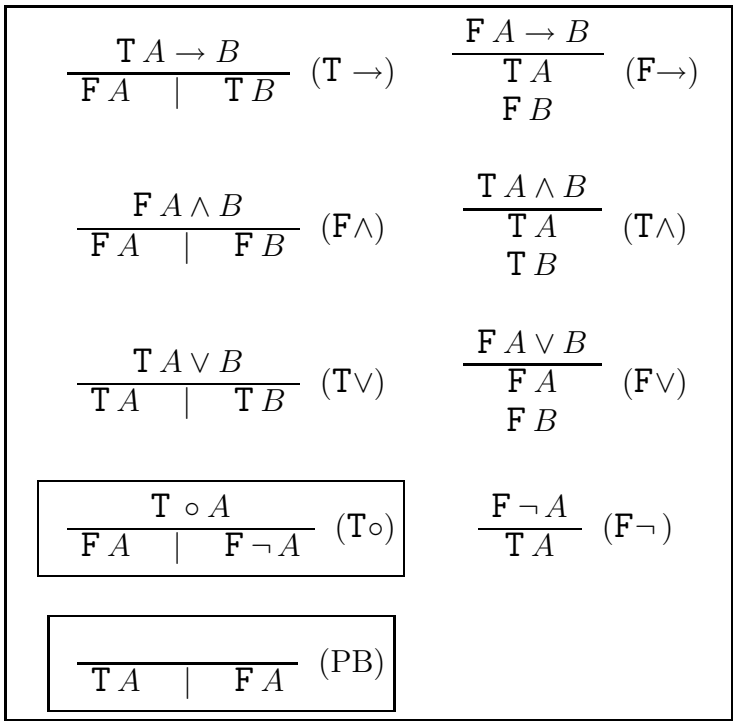


Figure B.3: **mbC** $\mathbf{C^3M}$ tableau rules.

As explained in [29], branching rules lead to inefficiency. To obtain a more efficient proof system, we used the $\mathbf{C^3M}$ tableau system for **mbC** as a basis to devise an original

[4]Some tableau systems for logics of formal inconsistency had already been presented in [15].

mbC KE system. The rules are presented in Figure B.4. The only difference between this system and the **KE** system for **CPL** is the replacement of the **CPL KE** ($\mathtt{T}\neg$) rule by the **KE** ($\mathtt{T}\neg'$) rule. Notice that the **KE** ($\mathtt{T}\neg'$) rule is a **LFI** version of **CPL KE** ($\mathtt{T}\neg$). It states clearly that besides $\mathtt{T}\,\neg A$, we need to have $\mathtt{T}\,\circ A$ to obtain $\mathtt{F}\,A$.

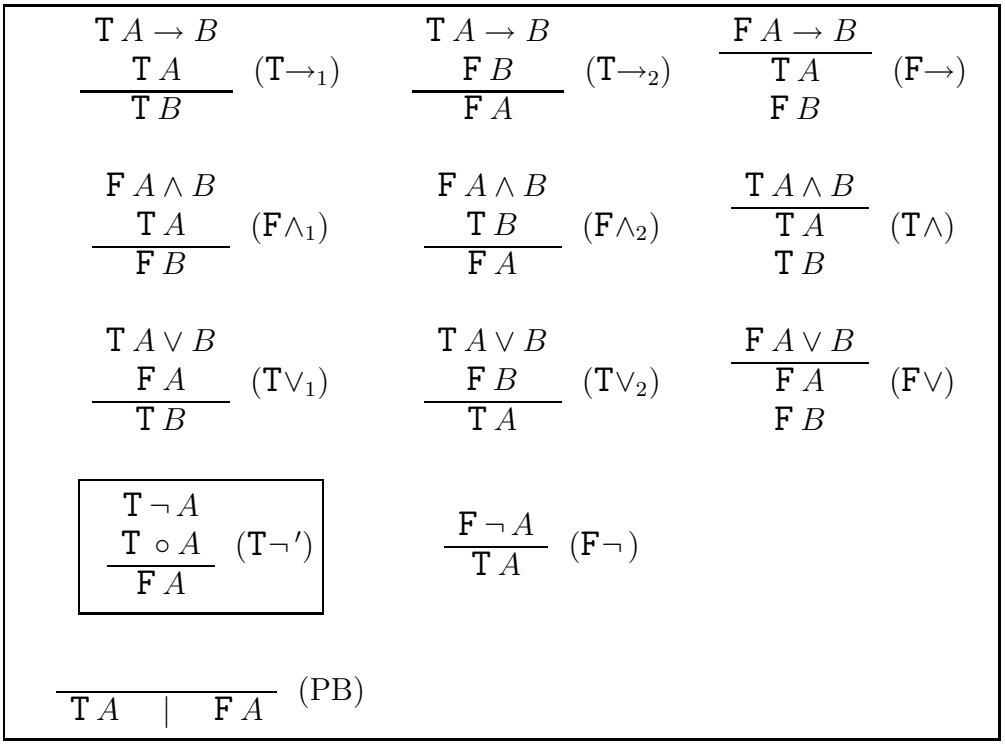


Figure B.4: **mbC KE** rules.

Example B.2.1. In Figure B.5 we show a proof of $\circ A, \circ C, A \to \circ B, B \to C, (\neg B) \to (D \to \neg A) \vdash \neg(A \wedge \neg C \wedge D)$. Notice that to obtain 17 from 16 we used the optional rule $\mathtt{F}_{\text{for}}$ (a derived rule presented in Section C.2.4).

Analyticity, Correctness and Completeness Proof

A tableau proof enjoys the *subformula property* if every signed formula in the proof tree is a subformula of some formula in the list of signed formulas to be proved. Let us call *analytic* the applications of (PB) which preserve the subformula property. And the *analytic restriction* of a tableau system is the system obtained by restricting (PB) to analytic applications.

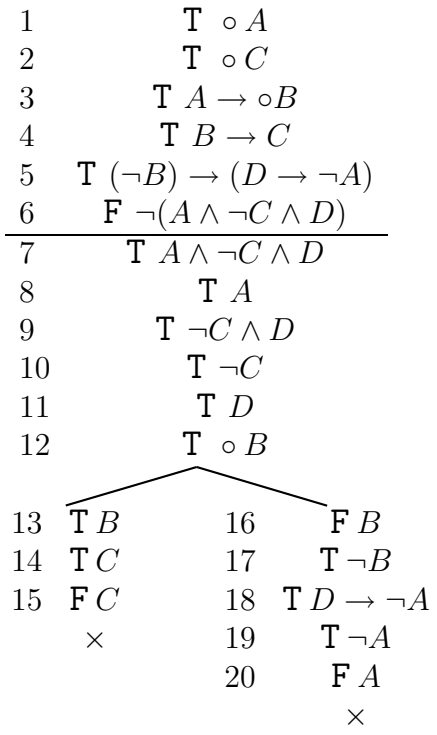


Figure B.5: An **mbC KE** proof.

Given a rule R of an expansion system **S,** we say that an application of R to a branch θ is *analytic* when it has the *subformula property,* i.e. if all the new signed formulas appended to the end of θ are subformulas of signed formulas occurring in θ. According to [29], a *rule* R is *analytic* if every application of it is analytic. When the analytic restriction of a tableau system is sound and complete we say that this system is analytic (and that we have proved the system's analyticity).

Our intention here is to prove that the **mbC KE** system is analytic, sound and complete. As we have seen in Section B.2.3, the **mbC KE** system originated from the **CPL KE** system and the $\mathbf{C^3M}$ tableau system. It is easy to notice that all **CPL KE** rules, except (PB), are analytic. And although (PB) is not analytic, the **KE** system for **CPL** was proven to be analytic, sound and complete [29]. On the other hand, the $\mathbf{C^3M}$ tableau system for **mbC** has two non-analytic rules: (PB) and (T∘). It is sound and complete [18, 11] but there is no proof either that it is analytic or not analytic.

It is easy to show a procedure that transforms any proof in the $\mathbf{C^3M}$ tableau system for **mbC** in an **mbC KE** proof, thus proving that **mbC KE** system is also sound and complete. We will not do this here. Instead, we will prove directly that the **mbC KE**

system is sound, complete and also analytic.

Proving that analytic restriction of **mbC KE** is sound and complete is a little bit more difficult than proving that **CPL KE** is analytic, because **mbC KE** has a two-premise rule, the ($\mathtt{T}\neg'$) rule, where neither premise is a subformula of the other premise, a condition satisfied by all **CPL KE** two-premise rules.

Because of this feature of the ($\mathtt{T}\neg'$) rule, it could be necessary to have non-analytic applications of the (PB) rule. But that is not the case: when performing an **mbC KE** proof we can restrict ourselves to analytic applications of (PB), applications which do not violate the subformula property, without affecting completeness. In this way, we demonstrate that even the analytic restriction of **mbC KE** is sound and complete.

The proof will be as follows. First we will define the notion of downward saturatedness for **mbC**. Then we will prove that every downward saturated set is satisfiable. The **mbC KE** proof search procedure for a set of signed formulas S either provides one or more downward saturated sets that give a valuation satisfying S or finishes with no downward saturated set.

Therefore, if an **mbC KE** tableau for a set of formulas S closes, then there is no downward saturated set that includes it, so S is unsatisfiable. However, if the tableau is open and completed, then any of its open branches can be represented as a downward saturated set and be used to provide a valuation that satisfies S. By construction, downward saturated sets for open branches are analytic, i.e. include only subformulas of S. We then conclude that the **mbC KE** system is analytic. As a corollary, it is also sound and complete.

Note: some concepts used in this proof are defined in Section C.2.1.

Definition B.2.1. A set of **mbC** signed formulas DS is *downward saturated* if

1. whenever a signed formula is in DS, its conjugate[5] is *not* in DS;

2. when all premises of any **mbC KE** rule (except (PB)) are in DS, its conclusions are also in DS;

[5]For any formula A The conjugate of $\mathtt{T}\ A$ is $\mathtt{F}\ A$, and vice-versa.

3. when the major premise of a two-premise **mbC KE** rule is in DS, either its auxiliary premise or its conjugate is in DS. For **mbC KE**, this item is valid for every rule except ($\mathtt{T}\neg'$). In this case, if $\mathtt{T}\,\neg X$ (which we define as the major premise in ($\mathtt{T}\neg'$)) is in DS, either $\mathtt{T}\circ X$ or $\mathtt{F}\circ X$ can be in DS, but only if $\circ X$ is a subformula of some other formula in DS. If $\circ X$ is *not* a subformula of some other formula in DS, neither $\mathtt{T}\circ X$ nor $\mathtt{F}\circ X$ are in DS.

We can extend valuations to signed formulas in an obvious way: $v(\mathtt{T}\,A) = v(A)$ and $v(\mathtt{F}\,A) = 1 - v(A)$. A set of signed formulas L is satisfiable if it is not empty and there is a valuation such that for every formula $\mathcal{S}X \in L$, $v(\mathcal{S}X) = 1$. Otherwise, it is unsatisfiable.

Lemma B.2.2. *(Hintikka's Lemma for* **mbC***) Every* **mbC** *downward saturated set is satisfiable.*

Proof. For any downward saturated set DS, we can easily construct an **mbC** valuation v such that for every signed formula $\mathcal{S}X$ in the set, $v(\mathcal{S}X) = 1$. How can we guarantee this is in fact an **mbC** valuation? First, we know that there is no pair $\mathtt{T}\,X$ and $\mathtt{F}\,X$ in DS. Second, all premised **mbC KE** rules preserve **mbC** valuations. That is, if $v(\mathcal{S}X_i) = 1$ for every premise $\mathcal{S}X_i$, then $v(\mathcal{S}C_j) = 1$ for all conclusions C_j. And if $v(\mathcal{S}X_1) = 1$ and $v(\mathcal{S}X_2) = 0$, where X_1 and X_2 are, respectively, major and minor premises of an **mbC KE** rule, then $v(\mathcal{S}'X_2) = 1$, where $\mathcal{S}'X_2$ is the conjugate of $\mathcal{S}X_2$. For instance, suppose $\mathtt{T}\,A \wedge B \in DS$, then $v(\mathtt{T}\,A \wedge B) = 1$. In accordance with the definition of downward saturated sets, $\{\mathtt{T}\,A, \mathtt{T}\,B\} \subseteq DS$. And by the definition of **mbC** valuation, $v(\mathtt{T}\,A \wedge B) = 1$ implies $v(\mathtt{T}\,A) = v(\mathtt{T}\,B) = 1$. □

Theorem B.2.3. *Let DS' be a set of signed formulas. DS' is satisfiable* if and only if *there exists a downward saturated set DS" such that DS' ⊆ DS".*

Proof. ($\Leftarrow$) First, let us prove that if there exists a downward saturated set DS" such that DS' $\subseteq DS$", then DS' is satisfiable. This is obvious because from DS" we can obtain a valuation that satisfies all formulas in DS", and DS' $\subseteq DS$".

($\Rightarrow$) Now, let us prove that if DS' is satisfiable, there exists a downward saturated set

DS" such that DS' $\subseteq DS$".

So, suppose that DS' is satisfiable and that there is no downward saturated set DS" such that DS" $\subseteq DS$'. Using items (2) and (3) of (B.2.1), we can obtain a family of sets of signed formulas DS'_i ($i \geq 1$) that include DS'. If none of them is downward saturated, it is because for all i, $\{\mathtt{T}\,X, \mathtt{F}\,X\} \in DS'_i$ for some X. But all rules are valuation-preserving, so this can only happen if DS is unsatisfiable, which is a contradiction. □

Corollary B.2.4. *DS' is an unsatisfiable set of formulas if and only if there is no downward saturated set DS" such that DS' $\subseteq$ DS".*

Theorem B.2.5. *The* **mbC KE** *system is analytic.*

Proof. The **mbC KE** proof search procedure for a set of signed formulas S either provides one or more downward saturated sets that give a valuation satisfying S or finishes with no downward saturated set. If an **mbC KE** tableau for a set of formulas S closes, then there is no downward saturated set that includes it, so S is unsatisfiable. If the tableau is open and completed, then any of its open branches can be represented as a downward saturated set and be used to provide a valuation that satisfies S. By construction, downward saturated sets for open branches are analytic, i.e. include only subformulas of S. Therefore, the **mbC KE** system is analytic. □

Corollary B.2.6. *The* **mbC KE** *system is sound and complete.*

Proof. The **mbC KE** system is a refutation system, as most tableau systems. The **mbC KE** system is sound because if an **mbC KE** tableau for a set of formulas S closes, then S is unsatisfiable. And if the tableau is open and completed, S is satisfiable. This has been shown in the proof of the theorem above. It is complete because if S is satisfiable, no **mbC KE** tableau for a set of formulas S closes. And if S is unsatisfiable, all completed **mbC KE** tableau for S close. □

B.2.4 A KE System for mCi

A tableau system for **mCi**, the $\mathbf{C^3M}$ system for **mCi**, was presented in [18] as an extension of the $\mathbf{C^3M}$ **mbC** system. Its rules are shown in Figure B.6. This system has a new rule called $(\mathtt{T}\neg\circ)$ that corresponds to the axiom **(cc)** (see **mCi** axiomatization in Section B.1.4) and rules $(\mathtt{T}\circ\neg^n\circ)$, for $n \geq 0$, that correspond to axioms $\mathbf{(cc)}_n$.

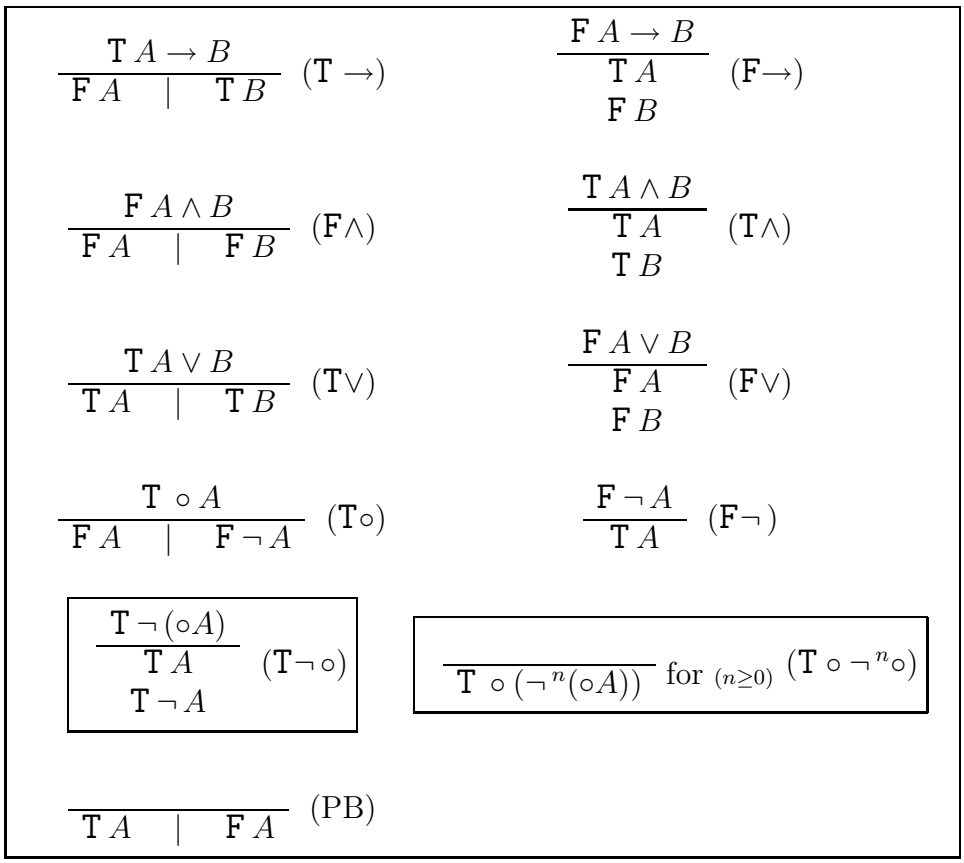


Figure B.6: **mCi** $\mathbf{C^3M}$ tableau rules.

As we have done for **mbC**, we use the $\mathbf{C^3M}$ tableau system for **mCi** as a basis to devise an original **mCi KE** system. **mCi KE** rules are presented in Figure B.7. We can see this system as an extension of the **mbC KE** system, where we include the $(\mathtt{T}\neg\circ)$ rule and the new $(\mathtt{F}\circ\neg^n\circ)$ rules, for $n \geq 0$. These $(\mathtt{F}\circ\neg^n\circ)$ rules were motivated by the same axioms that motivated $(\mathtt{T}\circ\neg^n\circ)$ rules, with the advantage of being analytic. The $(\mathtt{T}\neg\circ)$ rule, however, is not analytic. So neither $\mathbf{C^3M}$ for **mCi** nor **mCi-KE** are analytic proof systems.

Example B.2.2. In Figure B.8 we show an example of an **mCi KE** non analytic proof:

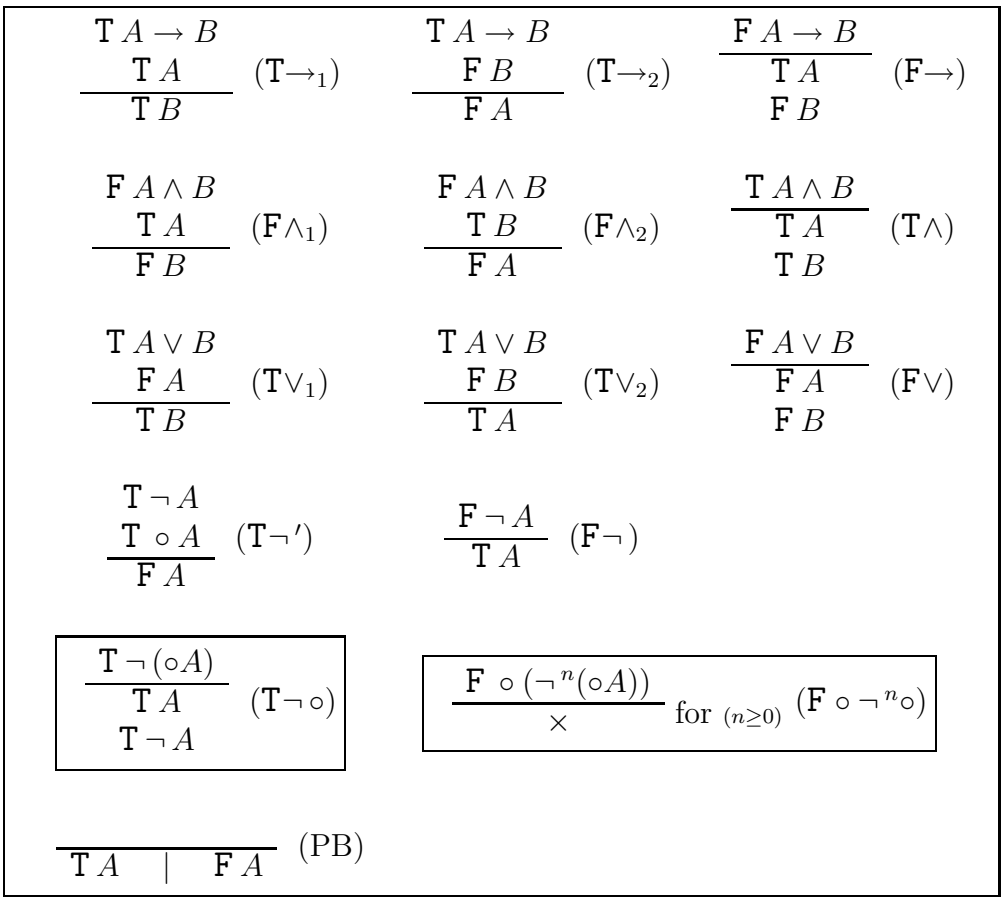


Figure B.7: **mCi KE** rules.

a proof of $\circ A \vdash \neg\neg \circ A$. The formula number 5 is not a subformula of any formula in the sequent being proved.

Example B.2.3. In Figure B.9 we show another example of a non analytic proof of $\neg\neg \circ A \vdash \circ A$. In fact, it is not possible to prove this sequent in **mCi KE** without a non analytic application of the (PB) rule.

$$\begin{array}{ll} 1 & \mathtt{T}\ \circ A \\ 2 & \mathtt{F}\ \neg\neg \circ A \\ \hline 3 & \mathtt{T}\ \neg \circ A \\ 4 & \mathtt{T}\ A \\ 5 & \mathtt{T}\ \neg A \\ 6 & \mathtt{F}\ A \\ & \times \end{array}$$

Figure B.8: An **mCi KE** proof of $\circ A \vdash \neg\neg \circ A$.

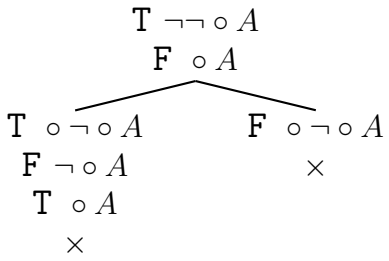


Figure B.9: An **mCi KE** proof of $\neg\neg \circ A \vdash \circ A$.

Correctness and Completeness Proof

Our intention here is to prove that the **mCi KE** system is sound and complete. It seems clear to us that the **mCi KE** system is not analytic, because of its ($\mathtt{T}\neg\circ$) rule, which is not analytic. We will follow the same schema used in Section B.2.3.

Definition B.2.7. A set of **mCi** signed formulas DS is *downward saturated*:

1. whenever a signed formula is in DS, its conjugate is *not* in DS;
2. when all premises of any **mCi KE** rule (except (PB) and ($\mathtt{F} \circ \neg^n \circ$), for $n \geq 0$) are in DS, its conclusions are also in DS;

3. when the major premise of a two-premise **mCi KE** rule is in DS, either its auxiliary premise or its conjugate is in DS. And the same condition for the $(\mathtt{T}\neg')$ rule that holds in Definition B.2.1 also holds here: if $\mathtt{T}\,\neg X$ is in DS, either $\mathtt{T}\circ X$ or $\mathtt{F}\circ X$ can be in DS, but only if $\circ X$ is a subformula of some other formula in DS. If $\circ X$ is *not* a subformula of some other formula in DS, neither $\mathtt{T}\circ X$ nor $\mathtt{F}\circ X$ are in DS;

4. if a signed formula $\mathcal{S}\ X$ is in DS, then for any sign $\mathcal{S}$, for any formula X, for all subformulas Y of X and for all $n \geq 0$, the signed formula $\mathtt{T}\circ\neg^n\circ Y$ is in DS.

The Hintikka's Lemma also holds for **mCi** downward saturated sets:

Lemma B.2.8. *(Hintikka's Lemma for* **mCi***) Every* **mCi** *downward saturated set is satisfiable.*

Proof. For any downward saturated set DS, we can easily construct an **mCi** valuation v such that for every signed formula $\mathcal{S}X$ in the set, $v(\mathcal{S}X) = 1$. How can we guarantee this is in fact a valuation? First, we know that there is no pair $\mathtt{T}\,X$ and $\mathtt{F}\,X$ in DS. Second, all premised **mCi KE** rules (except $(\mathtt{F}\circ\neg^n\circ)$ rules) preserve valuations. Note that $(\mathtt{F}\circ\neg^n\circ)$ rules are taken into account by the last clause in Definition B.2.7. That is, if we have a set of signed formulas that contains $\mathtt{F}\circ\neg^n\circ X$, every downward saturated set that contains this set should also contain $\mathtt{T}\circ\neg^n\circ X$. Therefore it is not downward saturated. To be downward saturated a set DS must contain, for all its subformulas[6] X, $\mathtt{T}\circ\neg^n\circ X$ (and must not contain any $\mathtt{F}\circ\neg^n\circ X$). As we can see in clause **(v7.n)** of the **mCi** valuation definition (see Definition B.1.3), $v(\mathtt{T}\circ\neg^n\circ X) = 1$ for all X. Therefore, DS is satisifable. □

Theorem B.2.3 and Corollary B.2.4 also hold for **mCi** downward saturated sets.

Theorem B.2.9. *The* **mCi KE** *system is sound and complete.*

Proof. The **mCi KE** proof search procedure for a set of signed formulas S either provides

[6]To be precise, by the subformulas of a set of signed formulas $\{\mathcal{S}_iF_i\}$, where $\mathcal{S}_i$ is a sign and F_i is an unsigned formula, we mean the set of subformulas of $\{F_i\}$.

one or more downward saturated sets that give a valuation satisfying S or finishes with no downward saturated set. The **mCi KE** system is a refutation system. The **mCi KE** system is sound because if an **mCi KE** tableau for a set of formulas S closes, then there is no downward saturated set that includes it, so S is unsatisfiable. If the tableau is open and completed, then any of its open branches can be represented as a downward saturated set and be used to provide a valuation that satisfies S (in other words, S is satisfiable).

The **mCi KE** system is complete because if S is satisfiable, no **mCi KE** tableau for a set of formulas S closes. And if S is unsatisfiable, all completed **mCi KE** tableau for S close. □

B.3 Complexity of Logical Systems

In this section, we are going to discuss some issues related to the complexity of logical systems. We begin with the complexity of decision problems.

B.3.1 Complexity of Decision Problems

The **CPL** satisfiability problem (known as 'SAT') is a decision problem studied in complexity theory. A decision problem is a problem that can be answered by 'yes' or 'no'. SAT can be described as "given a propositional formula, decide whether or not it is satisfiable". Many other decision problems, such as graph coloring problems, planning problems, and scheduling problems can be encoded into SAT.

SAT was the first known NP-complete problem. The class of NP-complete problems is a subclass of NP. While P is the class of decision problems that can be solved in polynomial time by a *deterministic* algorithm, NP is the class of decision problems that can be solved in polynomial time by a *nondeterministic* algorithm. Therefore P $\subseteq$ NP. The problems in NP are such that positive solutions can be verified in polynomial time. NP-complete problems are the most difficult problems in NP, the ones most likely not to be in P. If we find a polynomial time algorithm for any NP-complete problem, we can solve all problems in NP in polynomial time, because there is a polynomial time reduction

from any NP problem into any NP-complete problem.

The *complement* of a decision problem is the decision problem resulting from reversing the 'yes' and 'no' answers. We can generalize this to the complement of a complexity class, called the complement class, which is the set of complements of every problem in the class. co-NP is the complement of the complexity class NP. It is the class of problems for which a 'no' answer can be verified in polynomial time. And co-NP-complete is the complement of the class of NP-complete problems.

mCi is co-NP-complete

The **CPL** decision problem ("given a propositional formula, decide whether or not it is a *tautology*") is co-NP-complete, because a formula in **CPL** is a tautology if and only if its negation is unsatisfiable. In [18], it was shown that the decision problem for **mbC** is also co-NP-complete.

As the **mbC** decision problem is co-NP-complete and **mCi** extends **mbC**, the **mCi** decision problem is co-NP-hard. To prove that the **mCi** decision problem is also co-NP-complete (as suggested in [18]) we need a NP algorithm for the complement of **mCi** decision: the falsification of a formula. That is, we must show that given a formula A and a **mCi**-valuation v it is possible to verify if $v(A) = 0$ in polynomial time.

Let A be an **mCi** formula. We show below how to construct an **mCi**-valuation v for A. This is here only to show that it is more difficult to build an **mCi**-valuation than a **CPL**-valuation.

Let $\mathrm{SSF}(A)$ be the set of all strict subformulas of A. A *strict subformula* of A is any subformula of A except A itself. Then we construct a new set $\mathrm{ESSF}(A)$, such that for all $X \in \mathrm{SSF}(A)$, X, $\circ X$ and $\neg X$ belong to $\mathrm{ESSF}(A)$.

If n is the size of A, then the size of $\mathrm{ESSF}(A)$ is at most $3(n-1)$. To build a valuation v for A we must, for any $X \in \mathrm{ESSF}(A)$, set $v(X)$ either to 0 or to 1, obbeying the **mCi**-valuation clauses presented in Definition B.1.3.

Up to now, we have only $v(X)$ for all $X \in \mathrm{ESSF}(A)$ (not necessarily a value for $v(A)$). The following algorithm allows us to find a value for $v(A)$:

1. if, for some X, A is $\circ X$, then:
   (a) if $v(\neg X) = 0$, then $v(X) = 1$ and $v(A)$ can be set either to 0 or to 1;
   (b) if $v(X) = 1$ and $v(\neg X) = 1$, then $v(A) = 0$;
   (c) if $v(X) = 0$ and $v(\neg X) = 1$, then $v(A)$ can be set either to 0 or to 1;
2. if, for some X, A is $\neg X$, then:
   (a) if $v(X) = 1$ then:
      i. if $v(\circ X) = 1$ then $v(A) = 0$;
      ii. if $v(\circ X) = 0$ then $v(A)$ can be set either to 0 or to 1;
   (b) if $v(X) = 0$ then $v(A) = 1$;
3. if, for some X, Y, A is $X \wedge Y$, then:
   (a) if $v(X) = 1$ and $v(Y) = 1$, then $v(A) = 1$;
   (b) otherwise, $v(A) = 0$;
4. if, for some X, Y, A is $X \vee Y$, then:
   (a) if $v(X) = 0$ and $v(Y) = 0$, then $v(A) = 0$;
   (b) otherwise, $v(A) = 1$;
5. if, for some X, Y, A is $X \rightarrow Y$, then:
   (a) if $v(X) = 0$ and $v(Y) = 1$, then $v(A) = 0$;
   (b) otherwise, $v(A) = 1$.

Therefore, it is more difficult to build an **mCi**-valuation than a **CPL**-valuation[7]. But, given a formula A, if we have a valuation for all formulas in ESSF(A), it is easy to verify that $v(A)$ can be 0. The algorithm above is clearly polynomial in time (and also in space). As the NP class contains the problems that can be verified in polynomial time [26], the complement of the decision problem (falsification) for **mCi** is in NP. Therefore, the decision problem for **mCi** is co-NP-complete.

[7] A **CPL**-valuation can be built by setting values only to atomic formulas (see Definition B.1.1).

B.3.2 Complexity of Theorem-Proving Procedures

Besides the complexity of decision problems for logics, the complexity of theorem-proving procedures [23] and the length of proofs in **CPL** [22] has also been extensively studied. Given a possible tautology, we are faced with the problem of finding a proof, if one exists [7]. Then we encounter two additional problems: the first is concerned with *the complexity of the proof search* while the second with *the complexity of the smallest possible proof*, which might be exponential in the size of the tautology.

The complexity of proof search algorithms is obviously related to the complexity of the smallest possible proof. For instance, if the smallest proof is exponential in size, the proof search has to be exponential. But sometimes even when the proof system has polynomial size proofs for some classes of problems, current algorithms can be exponential, because it is harder to find the smallest proofs in stronger proof systems.

To study the length of **CPL** proofs, some families of problems that are known to be difficult are used. The pigeon hole principle (PHP) family of problems is probably the most studied of such families. Some works have shown that there are polynomial size proofs of this problem in some propositional proof systems [9, 24].

The resolution method [100] is the most widely studied propositional proof system. It can also be used for first-order classical logic and is implemented by many theorem provers, such as OTTER [77] and Vampire [98]. Resolution has exponential lower bounds for PHP and other classes of formulas [58, 113]. That is, all resolution proofs of PHP are exponential in length.

Another famous and successful decision procedure for **CPL** is the Davis-Logemann-Loveland (DLL) procedure [32]. Its modern variants, such as Chaff [83], are used in the most efficient SAT provers. However, these provers perform poorly on many important families of problems, including PHP. The most competitive SAT solvers show exponential scaling on these simple structured problems. This happens because DLL is based on tree resolution, a variant of resolution. Because of that, the proof search procedure of DLL also has exponential complexity.

Some extensions of DLL (see [40] for a good coverage) make it stronger than tree

resolution, but not stronger than resolution. Therefore, one solution to achieve shorter PHP proofs is to use stronger proof systems. However, as we said before, sometimes even when the proof system has polynomial size proofs for some classes of problems, current algorithms can be exponential, because it is harder to find proofs in stronger proof systems. In [40], a DLL style satisfiability solver that uses pseudo-Boolean representation and automates cutting plane [25], an inference system properly stronger than resolution, is presented. This pseudo-Boolean solver allowed exponential speedups over traditional methods on PHP problems.

Complexity of Tableau Methods

The complexity of Analytic Tableaux (**AT**) has been much studied [22, 2, 76]. For instance, Cook and Reckhow established in [22] that the family Σ_n of unsatisfiable formulas gives a lower bound of $2^{\Omega(2^n)}$ on the proof size with **AT**. Later, Massacci [76] exhibited an **AT** proof for Σ_n for whose size he proved an *upper bound* of $O(2^{n^2})$, which, although not polynomial in the size $O(2^n)$ of the input, is exponentially shorter than the claimed lower bound.

The **KE** system was proven to be more efficient than Smullyan's tableaux in [29]. There, a simple refutation procedure for **KE** was defined and called *canonical procedure*. And the *canonical restriction of* **KE** was defined as **KE** used with this canonical proof search procedure. The canonical restriction of **KE** can polinomially simulate the analytic tableau method, but the tableau method *cannot* polinomially simulate the canonical restriction of **KE**. In other words, for each analytic tableau proof of size n, there is a **KE**-Tableau proof with size polynomial in n. But there is at least one proof in **KE**-Tableau of size n whose corresponding proof in Analytic Tableaux has size superpolynomial in n. Besides that, the **KE** system polinomially simulates the truth-table procedure, although the analytic tableau method does not.

The **KE** system is more efficient than **AT** because it is based on Sequent Calculus with Cut, while the analytic tableau method is based on cut-free Sequent Calculus. It is well known that several families of problems have only exponential size proofs in cut-free

Sequent Calculus, but can have polynomial size proofs in Sequent Calculus with Cut. The set of rules for the **KE** system has only one rule of the branching type, the (PB) rule. Even if we add this rule to the analytic tableau method, it is not difficult to construct with this extended system short refutations of the 'hard examples' for Smullyan's tableaux [29]. However, in this system the (PB) rule is dispensable, while in **KE** formulation it is essential.

The complexity of **KE** deserves more study. According to [29], "a detailed study of proof-search in the **KE** system will have to involve more sophisticated criteria for the choice of the (PB)-formulae. A good choice may sometimes be crucial for generating essentially shorter proofs than those generated by the analytic tableau method".

Apêndice C

KEMS Design and Implementation

In this appendix we will discuss **KEMS** design and implementation. We first present tableau provers and the basic ideas behind **KEMS**, a multi-strategy tableau prover. Then we show some extensions to the **KE** methods discussed in Appendix B that were motivated by implementation issues. After that we present a brief description of the system, discussing its architecture and showing some class diagrams. Finally we briefly discuss each of the implemented **KEMS** strategies.

C.1 Tableau Provers

Theorem provers are computer programs that prove formal theorems, and *tableau provers* are theorem provers based on tableau methods (see Appendix B). Theorem provers receive a problem as input, where a *problem* [95] is a list of logic formulas (or, for signed tableau methods, a list of signed formulas) that represents a sequent.

A tableau prover output can be a 'closed' or an 'open' answer. A 'closed' answer means that the refutation of the problem sequent was successful and a closed tableau (see Section C.2.1) was obtained; an 'open' means that the sequent refutation failed and an open and completed tableau (see Section C.2.1) was obtained. Some tableau provers, besides this closed-open answer, provide a proof tree and maybe a countermodel. Let us explain this in more detail. The prover uses tableau expansion rules on problem formulas (and on formulas later generated by these rules) to construct a *proof tree* (also called *proof*

object). If the prover cannot close a tableau branch, the search for a refutation fails, and the proof tree represents an open tableau from which we can obtain a *countermodel* for the problem. Otherwise, if the prover closes all tableau branches, the search for a refutation succeds and no countermodel can be given. The resulting *closed tableau* is a refutation for the sequent, that is, an object that explains why the sequent is not valid[1]. Therefore an 'open' answer may be accompanied by an open proof tree and a countermodel. And a 'closed' answer may be accompanied by a closed proof tree.

In other words, tableau methods can be seen as search procedures for countermodels meeting certain conditions [52]. If we use a tableau prover to search for a model in which a sequent X is false, and we produce a closed tableau, no such model exists, so X must be valid. Tableau methods can be used to generate counter-examples: if we do not produce a closed tableau, then we have a countermodel for X.

Many tableau provers for several logics were described in the literature [104]: leanTAP [4], leanKE [96], linTAP [73], LOTREC [46], and jTAP [3], among others. According to [75], "tableau and sequent calculi are the basis for most popular interactive theorem provers for hardware and software verification. Yet, when it comes to decision procedures or automatic proof search, tableaux are orders of magnitude slower than Davis-Putnam, SAT based procedures or other techniques based on resolution." But tableau provers have two advantages over the Resolution method and the DLL procedure. First, they usually do not require conversion to any normal form. Most implementations of Resolution and DLL require problem formulas to be in clausal form. Second, there are tableau systems available for several non-classical logics, while the Resolution method and the DLL procedure are deeply linked to classical logic.

C.2 KEMS—A Multi-Strategy Tableau Prover

In this thesis we investigate the construction of **KEMS**, a **KE**-based multi-strategy tableau prover. In a multi-strategy theorem prover we can vary the strategy without

[1]Many SAT provers, zChaff [53] for instance, do not give any justification when they found a problem to be unsatisfiable.

modifying the core of the implementation. Our main objective was to be able to test and compare strategies with respect to the time spent by the proof search and the size of the proof obtained. In **KEMS**, a *strategy* will be responsible, among other things, for: *(i)* choosing the next rule to be applied, *(ii)* choosing the formula on which to apply the (PB) rule, and *(iii)* verifying branch closure.

In **KEMS**, we are able to implement different strategies for the same logical system. Then we use benchmarks to compare the results obtained by these strategies. A secondary objective was to investigate if proof strategies for tableau provers could be well modularized by using object-oriented and aspect-oriented programming.

The first step towards **KEMS** construction was to study and make some modifications (discussed in [86]) on an object-oriented framework for **KE**-based provers [38]. The second step was the implementation of a single-strategy **KE**-based object-oriented prover [85]. After that, we implemented a multi-strategy **KE**-based object-oriented prover for an extended **CPL KE** system [89]. In this system, besides using object orientation, we implemented some aspects [43], a new programming construct. Finally, we extended this system to deal with **mbC** and **mCi**, two logics of formal inconsistency, and implemented strategies for the three logical systems. Here we describe **KEMS** design and implementation as well as some related issues.

C.2.1 KE Proof Search Procedure

As **KEMS** is a **KE**-based prover, we describe here the proof search procedure for this system. This procedure builds a **KE** *tableau* (also called **KE** *proof tree*) for a target sequent $A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$ (the sequent we want to prove or refute). A *sequent* is an expression of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of formulas. The symbols $\bigwedge \Gamma$ and $\bigvee \Gamma$ stand for, respectively, the conjunction and the disjunction of all formulas in Γ. That is, if $\Gamma = \{A_1, A_2, \ldots, A_n\}$, then $\bigwedge \Gamma = (A_1 \wedge (A_2 \wedge (\ldots \wedge (A_{n-1} \wedge A_n))))$ and $\bigvee \Gamma = (A_1 \vee (A_2 \vee (\ldots \vee (A_{n-1} \vee A_n))))$, keeping in mind that $\wedge$ and $\vee$ are left-associative. So, a sequent should be read as "from $\bigwedge \Gamma$ we can deduce $\bigvee \Delta$". A sequent $\Gamma \vdash \Delta$ is *valid* when the formula "$\bigwedge \Gamma \rightarrow \bigvee \Delta$" is a tautology. A sequent $\Gamma \vdash \Delta$ is

satisfiable when "$\bigwedge \Gamma \rightarrow \bigvee \Delta$" is satisfiable.

The **KE** tableau for $A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$ is an ordered binary tree whose nodes contain finite *sets* of signed formulas. The proof search procedure starts by placing the following signed formulas

$$\mathtt{T}\, A_1, \mathtt{T}\, A_2, \ldots, \mathtt{T}\, A_m, \mathtt{F}\, B_1, \mathtt{F}\, B_2, \ldots, \mathtt{F}\, B_n$$

in the root node. These formulas represent the falsification of the target sequent.

The proof search proceeds by expanding the tableau. The **KE** method is an *expansion system* whose rules for **CPL** are presented in Figure B.2. An *expansion rule R of type* $\langle n \rangle$, with $n \geq 1$, is a computable relation between sets of signed formulas and n-tuples of sets of signed formulas satisfying the following condition:

$$R(S_o, (S_1, \ldots, S_n)) \Longrightarrow \text{ for every truth-set } S, \text{ if } S_0 \subseteq S, \text{ then } S_i \subseteq S \text{ for } 1 \leq i \leq n.$$

A *truth-set* or *saturated set* [29] is a set of signed formulas corresponding to **CPL** valuations. Given any **CPL** valuation v, there exists a saturated set $\mathcal{S}_v$ such that, for any formula A, if $v(A) = 1$ then $A \in \mathcal{S}_v$. For instance, if v is a valuation such that $v(A \wedge B) = 1$, then $\mathcal{S}_v = \{A, B, A \wedge B\}$ is the truth-set corresponding to v, because of **(v1)** in Definition B.1.1.

The **KE** expansion rules define what one can do, not what one must do. That is, at a given time during the construction of the tree one may have several rules that can be applied. To introduce signed formulas in a node, we can apply linear expansion rules that take as premises one or more signed formulas that already appear in that node or in some other node of the same branch. These new signed formulas are obviously logical consequences of the premises. We can always adjoin two nodes as successors of a given node, by applying the (PB) rule, which is a branching rule without premises. We only have to choose the formula to be used in (PB).

The proof search terminates when the tableau is closed or completed. A **KE** tableau is *closed* when all its branches are closed. We say that a branch is *closed* if, for some

formula X, $\mathtt{T}\ X$ and $\mathtt{F}\ X$ appear in the same branch, possibly not in the same node. Otherwise it is *open*. That is, a branch is closed when we arrive at a contradiction and a tableau is closed when we arrive at a contradiction in all branches of the generated tree. If this happens, the sequent we were trying to falsify is valid. Therefore, the resulting **KE** tableau is a **KE**-*refutation* (or *proof*) of $A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$.

We say that a signed formula $\mathcal{S}A$ was *analyzed* in a branch θ when:

- A is an atomic formula or
- $\mathcal{S}A$ was used as the main premise in the application of some rule in θ.

A branch is *completed* when all its signed formulas have been analyzed. A **KE** tableau is *completed* when at least one of its branches is completed and open.

When a tableau is completed, the sequent we were trying to falsify is not valid. In the **KE** systems presented here, as in most (if not all) tableau systems, there is a method for obtaining a counter-model of the target sequent $(A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n)$ from a completed tableau. A counter-model for a sequent is a valuation that assigns true to all formulas in the left side and false to the formulas in the right side. By analyzing any completed open branch[2] it is possible to obtain a valuation such that for $1 \leq i \leq m$, $v(A_i) = 1$ and for $1 \leq j \leq n$, $v(B_j) = 0$.

We define the *size* of a tableau proof as the sum of the sizes of all its nodes. The size of a node is the sum of the size of all its signed formulas. Besides that, the size of a list of signed formulas is the sum of its components' sizes. The size of a signed formula $\mathcal{S}A$ is defined as the size of A. And finally, the *size* $s(A)$ of a formula A is defined as [51]:

- $s(A) = 1$ if A is a propositional atom;
- $s(\odot A) = 1 + s(A)$, where A is a formula and $\odot$ is a unary connective;
- $s(A \oslash B) = 1 + s(A) + s(B)$, where $\oslash$ is a binary connective, and A and B are formulas.

[2] A branch is a sequence of nodes that goes from the root branch to a leaf node (a node without successors).

The *height* of the proof tree and the *number of nodes* in the tree are other important dimensions for evaluating the efficiency of a proof search procedure. These are defined as usually for trees [26].

Example C.2.1. In Figures C.1 and C.2, we present two different proofs of the same sequent: the third instance of the Γ family [12] of problems (see Section D). The Γ_3 problem instance is represented by the following valid sequent:

$$(p_1 \vee q_1), (p_1 \rightarrow (p_2 \vee q_2)), (q_1 \rightarrow (p_2 \vee q_2)), (p_2 \rightarrow (p_3 \vee q_3)), (q_2 \rightarrow (p_3 \vee q_3)),$$

$$(p_3 \rightarrow (p_4 \vee q_4)), (q_3 \rightarrow (p_4 \vee q_4)) \vdash (p_4 \vee q_4)$$

In both proofs, the first step was to include the signed formulas[3] numbered 1 to 8 (representing the falsification of the sequent) in the origin. In Figure C.1, the next step was to apply all linear rules that could be applied. This generated formulas 9 to 12. Then, we had to choose the first formula to apply the (PB) rule. In this case, we would do better by choosing a formula that could be used as an auxiliary premise with one of the five formulas (1-5) that were not yet used as main premises. By choosing the left subformula of 2, the best result is a proof with size 71 and 31 nodes.

In Figure C.2, we used a different strategy. We did not apply all linear rules that could be applied (formula 8 was not expanded), generating only 9 and 10. After that, we chose the left subformula of 4 to apply the (PB) rule, and the result was a proof with size 61 and 25 nodes.

C.2.2 Extended CPL KE System

Instead of the original **CPL KE** system (see Section B.2.2), **KEMS** implements an extended **CPL KE** system, which we present here. First let us introduce four symbols to the **CPL** language (**L**): $\top$ (called 'top'), $\bot$ (named 'bottom'), $\leftrightarrow$ (called 'bi-implication') and $\oplus$ (named 'exclusive or'). Σ^s will denote the signature obtained by the addition of

[3] From now on we will use the term *s-formula* to refer to signed formulas.

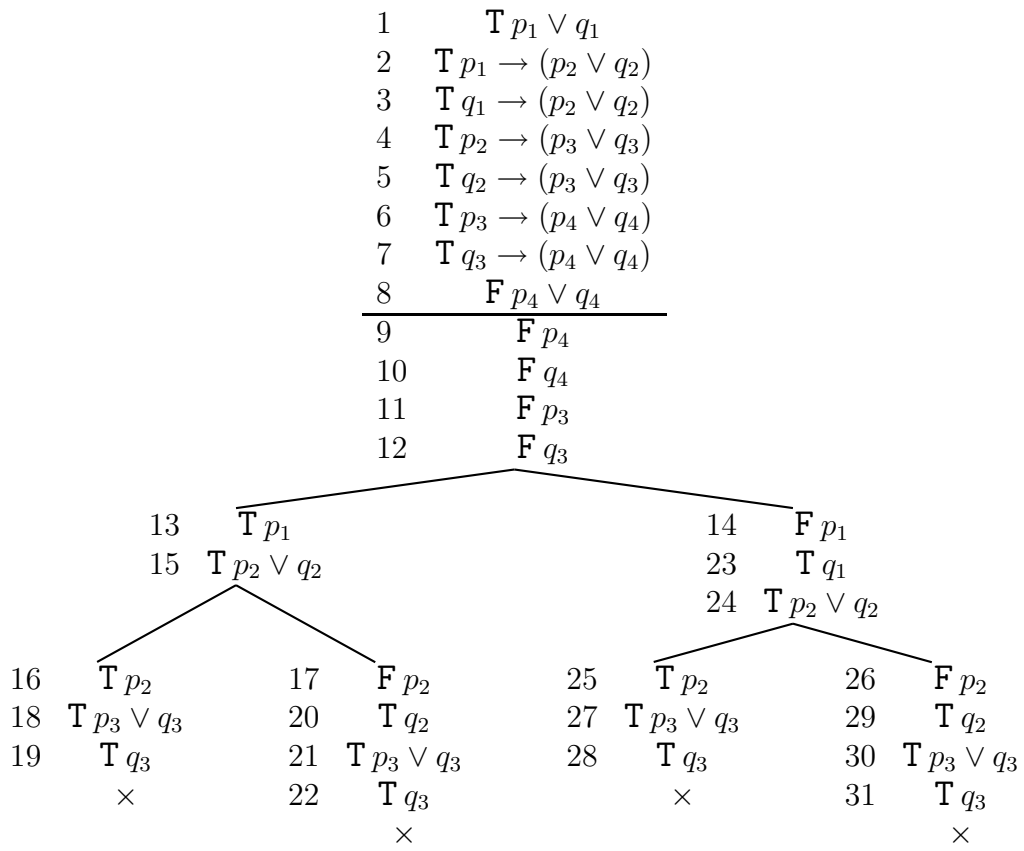


Figure C.1: A **CPL KE** proof of Γ_3.

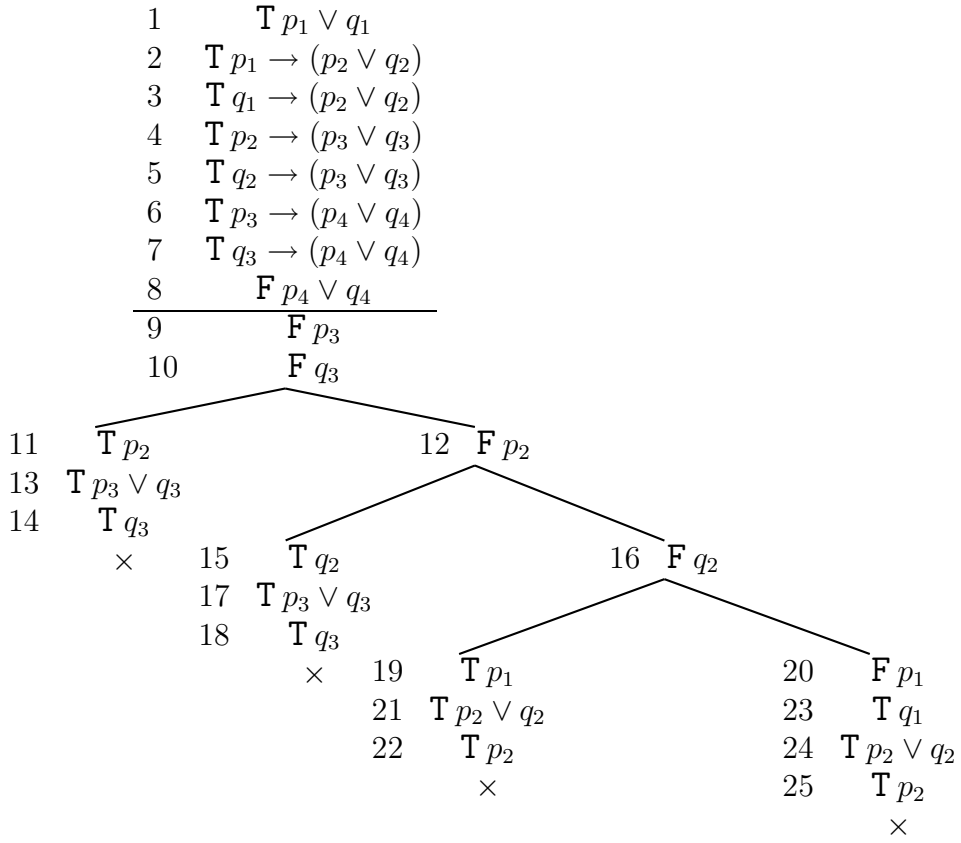


Figure C.2: A smaller **CPL KE** proof of Γ_3.

these two zeroary ($\top$ and $\bot$) and two binary ($\leftrightarrow$ and $\oplus$) connectives to the original **CPL** signature (Σ), and For^s will denote the algebra of formulas for this signature.

The following axioms have to be added to **CPL** axiomatization (see Section B.1.1) to deal with the new connectives:

(Ax12) $(A \leftrightarrow B) \rightarrow ((A \wedge B) \vee ((\neg A) \wedge (\neg B)))$;

(Ax13) $(A \oplus B) \rightarrow ((A \wedge (\neg B)) \vee ((\neg A) \wedge B))$;

(Ax14) $\bot \rightarrow A$;

(Ax15) $A \rightarrow \top$.

Then we extend **CPL**-valuations (see Definition B.1.1) by adding the following clauses:

(v5) $v(A \leftrightarrow B) = 1$ if and only if $v(A) = v(B)$;

(v6) $v(A \oplus B) = 1$ if and only if $v(A) = 1$ and $v(B) = 0$, or $v(A) = 0$ and $v(B) = 1$.

(v7) $v(\top) = 1$;

(v8) $v(\bot) = 0$.

Finally, we have to add the rules shown in Figure C.3, Figure C.4 and Figure C.5 to the original set of **CPL KE** rules (see Figure B.2). We call **e-CPL-KE** this extended **CPL KE** system .

$$\frac{}{\mathtt{T}\top}\ (\mathtt{T}\top) \qquad \frac{}{\mathtt{F}\bot}\ (\mathtt{F}\bot)$$

Figure C.3: 'Top' and 'bottom' **KE** rules.

C.2.3 Simplification Rules

To obtain a more efficient system, we can add a set of simplification rules [75] to the extended **CPL KE** system. These simplification inference rules do not cause branching and in some cases may even prevent it. They play for tableau methods the same role of unit propagation for DLL and subsumption for resolution. We adapt Massacci's simplification

$$\frac{\mathtt{T}\, A \leftrightarrow B \quad \mathtt{T}\, A}{\mathtt{T}\, B}\ (\mathtt{T} \leftrightarrow_1) \qquad \frac{\mathtt{T}\, A \leftrightarrow B \quad \mathtt{T}\, B}{\mathtt{T}\, A}\ (\mathtt{T} \leftrightarrow_2)$$

$$\frac{\mathtt{T}\, A \leftrightarrow B \quad \mathtt{F}\, A}{\mathtt{F}\, B}\ (\mathtt{T} \leftrightarrow_3) \qquad \frac{\mathtt{T}\, A \leftrightarrow B \quad \mathtt{F}\, B}{\mathtt{F}\, A}\ (\mathtt{T} \leftrightarrow_4)$$

$$\frac{\mathtt{F}\, A \leftrightarrow B \quad \mathtt{T}\, A}{\mathtt{F}\, B}\ (\mathtt{F} \leftrightarrow_1) \qquad \frac{\mathtt{F}\, A \leftrightarrow B \quad \mathtt{T}\, B}{\mathtt{F}\, A}\ (\mathtt{F} \leftrightarrow_2)$$

$$\frac{\mathtt{F}\, A \leftrightarrow B \quad \mathtt{F}\, A}{\mathtt{T}\, B}\ (\mathtt{F} \leftrightarrow_3) \qquad \frac{\mathtt{F}\, A \leftrightarrow B \quad \mathtt{F}\, B}{\mathtt{T}\, A}\ (\mathtt{F} \leftrightarrow_4)$$

Figure C.4: 'Bi-implication' **KE** rules.

$$\frac{\mathtt{T}\, A \oplus B \quad \mathtt{T}\, A}{\mathtt{F}\, B}\ (\mathtt{T} \oplus_1) \qquad \frac{\mathtt{T}\, A \oplus B \quad \mathtt{T}\, B}{\mathtt{F}\, A}\ (\mathtt{T} \oplus_2)$$

$$\frac{\mathtt{T}\, A \oplus B \quad \mathtt{F}\, A}{\mathtt{T}\, B}\ (\mathtt{T} \oplus_3) \qquad \frac{\mathtt{T}\, A \oplus B \quad \mathtt{F}\, B}{\mathtt{T}\, A}\ (\mathtt{T} \oplus_4)$$

$$\frac{\mathtt{F}\, A \oplus B \quad \mathtt{T}\, A}{\mathtt{T}\, B}\ (\mathtt{F} \oplus_1) \qquad \frac{\mathtt{F}\, A \oplus B \quad \mathtt{T}\, B}{\mathtt{T}\, A}\ (\mathtt{F} \oplus_2)$$

$$\frac{\mathtt{F}\, A \oplus B \quad \mathtt{F}\, A}{\mathtt{F}\, B}\ (\mathtt{F} \oplus_3) \qquad \frac{\mathtt{F}\, A \oplus B \quad \mathtt{F}\, B}{\mathtt{F}\, A}\ (\mathtt{F} \oplus_4)$$

Figure C.5: 'Exclusive or' **KE** rules.

rule definition in [75] and define the following schema for **KE** simplification rules:

$$\frac{\begin{array}{c}\mathcal{S}_1\ \Phi(\Theta(X)) \\ \mathcal{S}_2\ X\end{array}}{\mathcal{S}_1\ \Phi(\Theta(X)/\mathrm{simpl}(\Theta(X), \mathcal{S}_2\ X))}$$

where:

1. $\mathcal{S}_1\ \Phi(\Theta(X))$ is the major premise;
2. $\mathcal{S}_2\ X$ is the minor premise;
3. $\mathcal{S}_1\ \Phi(\Theta(X)/\mathrm{simpl}(\Theta(X), \mathcal{S}_2\ X))$ is the conclusion

and

- $\mathcal{S}_1$ and $\mathcal{S}_2$ are signs;
- for any Z, $\Phi(Z)$ is a formula where Z appears (one or more times) as a subformula;
- $\Theta(X)$ is either $\neg X$ or $X \oslash Y$ or $Y \oslash X$, for any X and Y, where $\oslash \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$;
- $\Phi(\Theta(X)/\mathrm{simpl}(\Theta(X), \mathcal{S}_2\ X))$ means that we substitute every occurrence of $\Theta(X)$ in $\Phi(\Theta(X))$ by $\mathrm{simpl}(\Theta(X), \mathcal{S}_2\ X)$.

Finally, '$\mathrm{simpl}(\mathcal{F}_1, \mathcal{S}\ \mathcal{F}_2)$', where $\mathcal{F}_2$ if a subformula of $\mathcal{F}_1$ and $\mathcal{S}$ is a sign, is the following rewrite function:

1. $\mathrm{simpl}(\neg X, \mathtt{T}\ X) \rightsquigarrow \bot$;
2. $\mathrm{simpl}(\neg X, \mathtt{F}\ X) \rightsquigarrow \top$;
3. $\mathrm{simpl}(X \wedge Y, \mathtt{T}\ X) = \mathrm{simpl}(Y \wedge X, \mathtt{T}\ X) \rightsquigarrow Y$;
4. $\mathrm{simpl}(X \wedge Y, \mathtt{F}\ X) = \mathrm{simpl}(Y \wedge X, \mathtt{F}\ X) \rightsquigarrow \bot$;
5. $\mathrm{simpl}(X \vee Y, \mathtt{T}\ X) = \mathrm{simpl}(Y \vee X, \mathtt{T}\ X) \rightsquigarrow \top$;
6. $\mathrm{simpl}(X \vee Y, \mathtt{F}\ X) = \mathrm{simpl}(Y \vee X, \mathtt{F}\ X) \rightsquigarrow Y$;

7. $\text{simpl}(X \to Y, \mathtt{T}\ X) \rightsquigarrow Y$;
8. $\text{simpl}(X \to Y, \mathtt{F}\ Y) \rightsquigarrow \neg X$;
9. $\text{simpl}(X \to Y, \mathtt{F}\ X) \rightsquigarrow \top$;
10. $\text{simpl}(X \to Y, \mathtt{T}\ Y) \rightsquigarrow \top$;
11. $\text{simpl}(X \leftrightarrow Y, \mathtt{T}\ X) \rightsquigarrow Y$;
12. $\text{simpl}(X \leftrightarrow Y, \mathtt{T}\ Y) \rightsquigarrow X$;
13. $\text{simpl}(X \leftrightarrow Y, \mathtt{F}\ X) \rightsquigarrow \neg Y$;
14. $\text{simpl}(X \leftrightarrow Y, \mathtt{F}\ Y) \rightsquigarrow \neg X$;
15. $\text{simpl}(X \oplus Y, \mathtt{T}\ X) \rightsquigarrow \neg Y$;
16. $\text{simpl}(X \oplus Y, \mathtt{T}\ Y) \rightsquigarrow \neg X$;
17. $\text{simpl}(X \oplus Y, \mathtt{F}\ X) \rightsquigarrow Y$;
18. $\text{simpl}(X \oplus Y, \mathtt{F}\ Y) \rightsquigarrow X$.

For instance, below we have an application of a simplification rule:

$$\frac{\begin{array}{c}\mathtt{T}\ D \to ((A \to B) \wedge (C \to E)) \\ \mathtt{T}\ A\end{array}}{\mathtt{T}\ D \to (B \wedge (C \to E))}$$

where

- $\Theta(A) = A \to B$;
- $\Phi(Z) = D \to (Z \wedge (C \to E))$, therefore, $\Phi(\Theta(A)) = D \to ((A \to B) \wedge (C \to E))$;
- $\text{simpl}(A \to B, \mathtt{T}\ A) \rightsquigarrow B$; and
- '$D \to (B \wedge (C \to E))$' is the result of substituting '$A \to B$' by 'B' in '$D \to ((A \to B) \wedge (C \to E))$'.

Given this schema for simplification rules we can define an extension of **e-CPL-KE** that includes simplification rules for every **CPL** connective. As an example we show in Figure C.6 the simplification rules for conjunction. We call this system **s-CPL-KE**.

$$\frac{\mathcal{S}_1\ \Phi(A \wedge B) \quad \mathtt{T}\ A}{\mathcal{S}_1\ \Phi(B)} \qquad \frac{\mathcal{S}_1\ \Phi(B \wedge A) \quad \mathtt{T}\ A}{\mathcal{S}_1\ \Phi(B)}$$

$$\frac{\mathcal{S}_1\ \Phi(A \wedge B) \quad \mathtt{F}\ A}{\mathcal{S}_1\ \Phi(\bot)} \qquad \frac{\mathcal{S}_1\ \Phi(B \wedge A) \quad \mathtt{F}\ A}{\mathcal{S}_1\ \Phi(\bot)}$$

Figure C.6: Simplification **CPL KE** rules for the conjunction connective.

A logical system must have some properties, such as the replacement property, to accept the definition of such rules. In **mbC** and **mCi**, for instance, the replacement property is not valid [18], so it is not possible to have general simplification rules for these logical systems.

C.2.4 Extended mbC and mCi KE Systems

As we have done with **CPL**, in **KEMS** we work with extended versions of the **mbC KE** (see Section B.2.3) and **mCi KE** (see Section B.2.4) systems. For the extended versions of these systems, called respectively **e-mbC-KE** and **e-mCi-KE**, we only introduced the '$\top$' and '$\bot$' connectives; we did not include '$\leftrightarrow$' and '$\oplus$'. For this inclusion we followed the same steps presented in Section C.2.2, restricting ourselves to '$\top$' and '$\bot$' axioms, valuation clauses and rules.

Derived Rules

To achieve better performance with some problems, we can add derived rules to the two extended systems for **mbC** and **mCi**. In Figure C.7 we show some of the rules that can be derived from $\mathbf{C^3M}$ **mbC** rules (see Figure B.3) and that can be added to the extended systems.

Note that the ($\mathtt{T}\circ'$) rule was in fact presented in [18] as a $\mathbf{C^3M}$ $\mathbf{C}_1$ rule. The ($\mathtt{F}_{\text{formula}}$)

rule can be used to shorten proofs when we have, for instance, $\mathtt{F}\ A$ and $\mathtt{T}\ (\neg A) \rightarrow X$. Without this rule, in such a situation we must apply (PB) on $\{\mathtt{F}\ \neg A, \mathtt{T}\ \neg A\}$. From $\mathtt{F}\ \neg A$ we obtain $\mathtt{T}\ A$ and close the left branch. So we proceed in the right branch with $\mathtt{T}\ \neg A$, exactly like the ($\mathtt{F}_{\text{formula}}$) rule does without branching.

The current **KEMS** version has **mbC** and **mCi** strategies that use only ($\mathtt{T}\circ''$) and ($\mathtt{T}\neg''$) as additional rules, but strategies can be implemented to use the other two rules, or even other derived rules.

$$\frac{\mathtt{T}\circ A \quad \mathtt{T}\,A}{\mathtt{F}\neg A}\ (\mathtt{T}\circ'') \qquad \frac{\mathtt{T}\neg A \quad \mathtt{T}\,A}{\mathtt{F}\circ A}\ (\mathtt{T}\neg'')$$

$$\frac{\mathtt{F}\,A}{\mathtt{T}\neg A}\ (\mathtt{F}_{\text{formula}}) \qquad \frac{\mathtt{T}\circ A \quad \mathtt{T}\,B \rightarrow A \quad \mathtt{T}\,B \rightarrow \neg A}{\mathtt{F}\,B}\ (\mathtt{T}\circ')$$

Figure C.7: Derived **mbC KE** rules.

C.3 System Description

In this section we present a description of the implemented **KEMS** system. We start with the system architecture which is presented in Figure C.8. The digram describes that the user presents as input to **KEMS** a problem instance and a prover configuration. The prover configuration must contain values for four major **KEMS** parameters:

1. the logical system for the proof search procedure (as we see in Appendix D, some problems can be submitted to provers for different logical systems);
2. the analyzer used to lexically analyse and parse the problem;
3. the strategy chosen to search a proof for the problem;
4. the sorter chosen to be used with the strategy.

Besides that, the prover configuration can be used to give values to other four minor parameters:

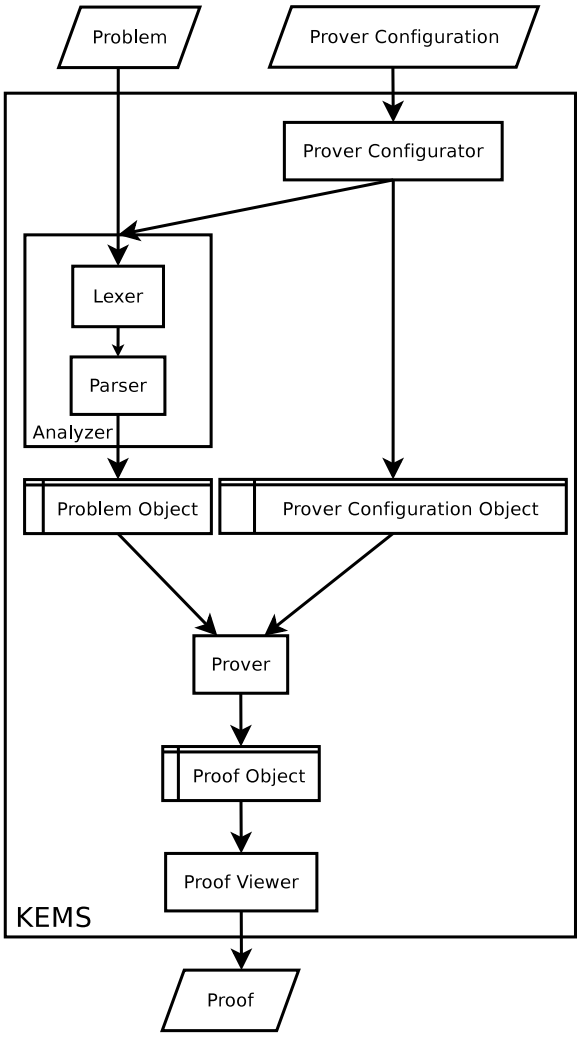


Figure C.8: System architecture.

- the number of times the prover must run the proof search procedure with the given problem[4];
- the time limit for the proof search procedure (if this time limit is exceeded, the prover is interrupted);
- a boolean option to determine if the prover must save s-formula origins or not;
- a boolean option to determine if the prover must discard closed branches or not;
- a boolean option to determine if the prover must save discarded branches in disk (to be restored after the proof procedure finishes) or not.

Given these inputs, the system outputs a proof that contains, among other things:

- the open/closed final status of the tableau;
- the tableau proof tree, that can be partial if the discard closed branches option is set to true;
- the problem size;
- the time spent by the prover while building a proof for the input problem;
- the proof size;
- a counter-model valuation, if the tableau is open.

Let us see an example. The user can present a PHP_4 instance (see Section D.1.1) with a prover configuration establishing '**mbC**' (see Section B.1.3) as the logical system, '**LFI** analyzer'[5] as the problem analyzer, '**mbC** Simple Strategy' (see Section C.4.4) as the strategy, and 'Insertion Order' (see Section C.4.2) as the sorter. Then **KEMS** uses the chosen analyzer (which contains a lexer and a parser) to build a problem object. And it builds a prover configuration object from user options. These two objects are used by the prover module to build a proof object. This proof object is given as input to the proof viewer. Finally, the proof viewer shows the proof to the user.

[4] This is useful for prover evaluation, in order to get a better estimate of the time spent to find a proof.

[5] A problem analyzer for **mbC** and **mCi** that we implemented.

C.3.1 Class Diagrams

Here we present and discuss some simplified class diagrams for **KEMS**. The first of these diagrams is depicted in Figure C.9. It describes the classes we implemented to represent formulas and signed formulas. The Formula class uses the Composite design pattern [55]: a formula can be either an AtomicFormula or a composition of AtomicFormula objects. A SignedFormula object contains references of one FormulaSign object and one Formula object.

The FormulaFactory and SignedFormulaFactory classes use the Flyweight design pattern [55]. This pattern prevents the multiplication of objects representing formulas and signed formulas as well as makes it easier to implement rule choice and application. That is, there is only one instance of each formula and each signed formula. This allows us to save space as well as serves to simplify the search for subformulas of a formula, and for signed formulas where a formula appears. Using this pattern, when we want to compare two formulas, we have only to compare pointers instead of character strings or formula structure.

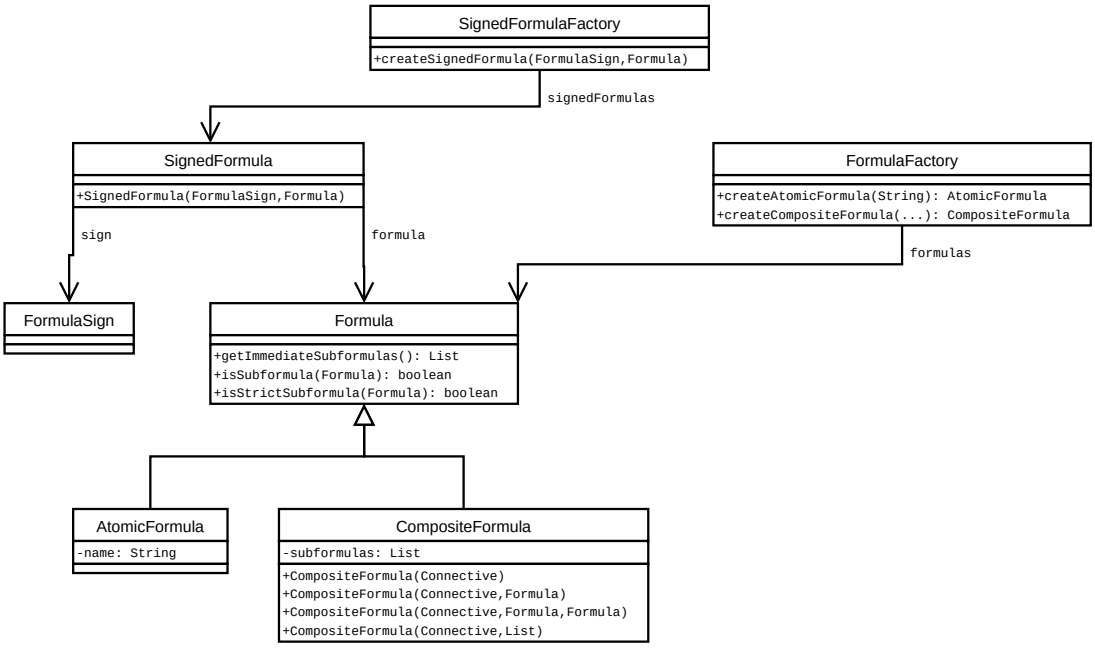


Figure C.9: Formula and Signed Formula class diagram.

The diagram in Figure C.10 shows that in the ProverFacade class we have used the

Facade design pattern [55] to provide a higher-level interface to the classes that implement prover functionality. The SignedFormulaCreator class uses instances of subclasses of Lexer and Parser classes to build a Problem object. The Prover class has a method that receives as input a Problem object and outputs a Proof object. And the ProofVerifier class has a method that gets as input a Proof object and outputs an ExtendedProof object, containing a tableau proof tree and additional information about the proof and the proof search procedure.

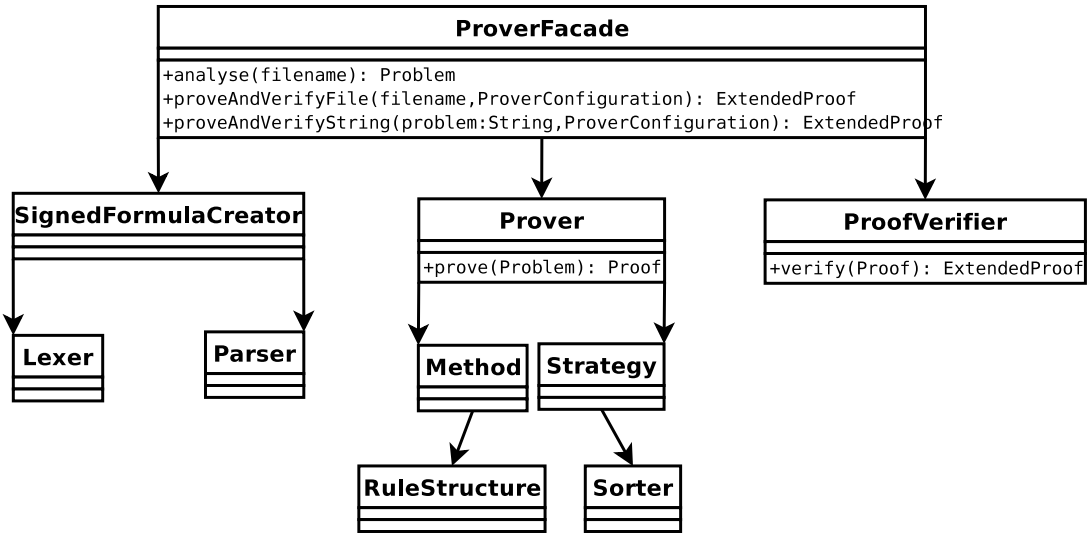


Figure C.10: Prover class diagram.

The Prover class uses the Strategy design pattern [55] to be able to make proof strategies interchangeable. In Figure C.11 we see that an IStrategy interface was defined. All strategies must implement this interface. Besides that we defined an ISimpleStrategy that defines several methods that are common to all strategies implemented in **KEMS** current version (but that future strategies need not implement). We implemented the functionality which is common to all implemented **KEMS** strategies in AbstractSimpleStrategy. This class has three subclasses: SimpleStrategy, MemorySaverStrategy and ConfigurableSimpleStrategy. These three classes define three strategies whose features we discuss in Section C.4.3. And we used SimpleStrategy as a basis for several other strategies that are discussed in Sections C.4.4, C.4.5 and also in Section C.4.3.

A subset of the classes and interfaces written to implement **KE** rules is shown in

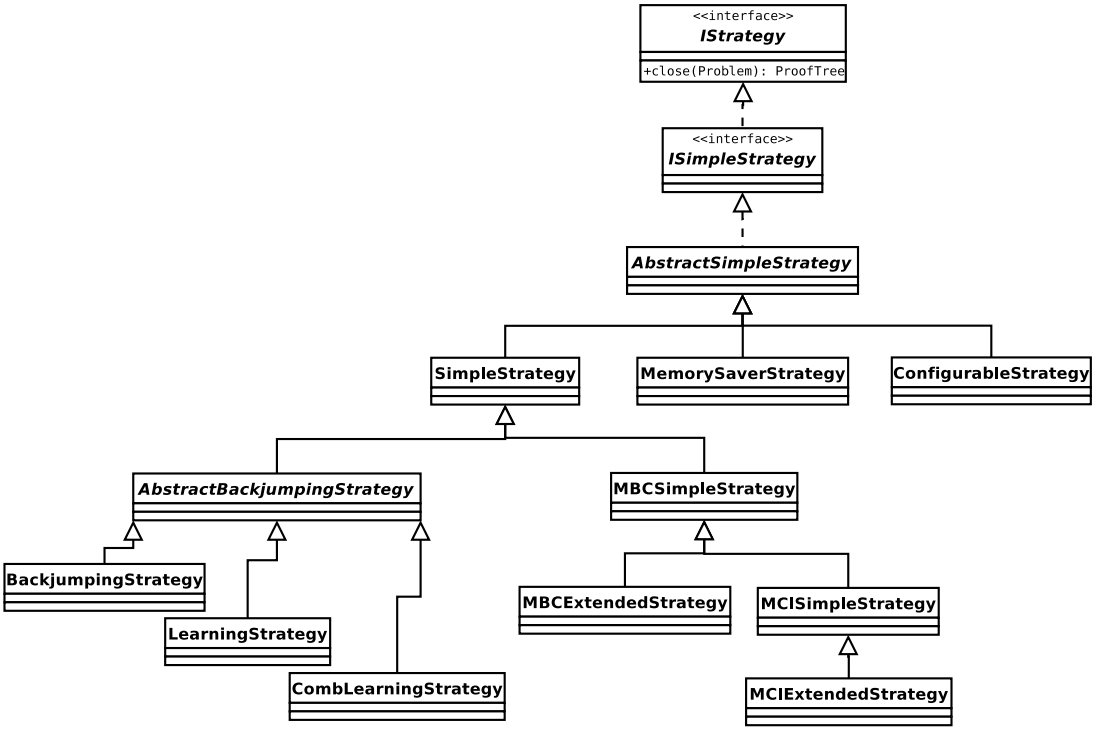


Figure C.11: Strategy class diagram.

Figure C.12. We have classes for one-premise one-conclusion rules, two-premise one-conclusion rules and one-premise two-conclusion rules. All of these classes are subclasses of an abstract class Rule that implements the IRule interface. A RuleStructure contains one or more lists of rules and there is a map that assigns a name to each RuleList object. In this way, a RuleStructure can have subsets of the chosen **KE** system rules so that a strategy can choose which subset to apply first. Besides these classes, we have several classes to implement rule premise patterns and rule conclusion actions.

C.3.2 Programming Languages Used

Current **KEMS** version is written in Java 1.5 [57] with some aspects written in AspectJ 1.5 [65, 64]. Java was chosen because it is a well established object-oriented programming (OOP) language for which there is an extension, called AspectJ, that supports a new software development paradigm: aspect-orientation. At the time of our choice, 2003, AspectJ was the best aspect-oriented language. As we had a plan to make use of aspects in our implementation, we chose to work with Java and AspectJ.

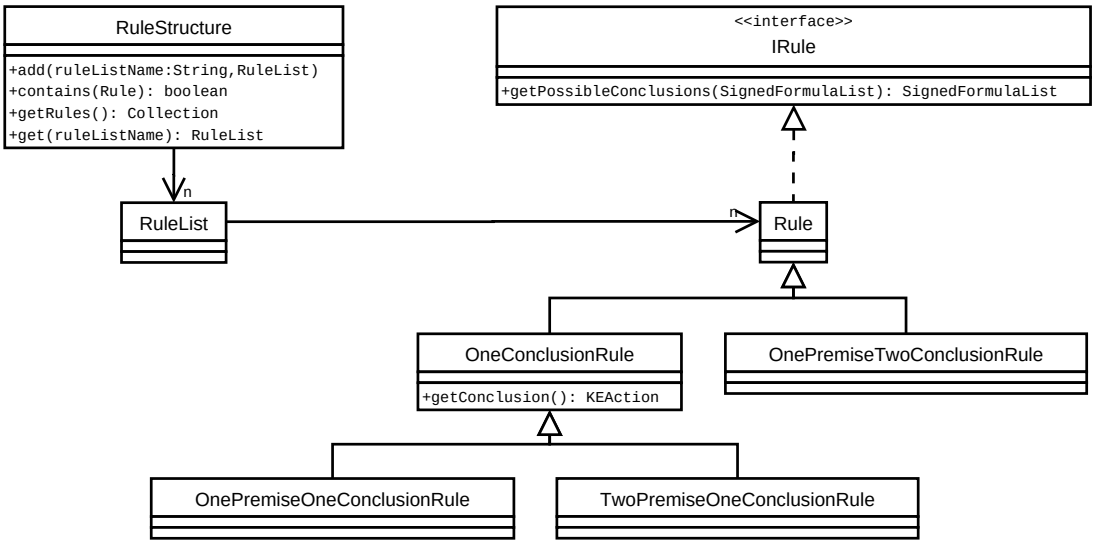


Figure C.12: Rule class diagram.

In aspect-oriented systems, classes are blueprints for the objects that represent the main concerns of a system while aspects represent concerns that are orthogonal to the main concerns and that may have impact over several classes in different places in the class hierarchy. The use of aspects, among other advantages, leads to less scattered code. That is, lines of code that implement a given feature of the system can rest in the same file. Next we present in more detail AspectJ and Aspect-oriented programming (AOP), and discuss briefly some aspects we have implemented in **KEMS**.

AspectJ

Although the object oriented paradigm is dominant nowadays, it has some limitations. For instance, in object oriented systems, code with different purposes can become scattered and tangled. Part of these limitations can be overcome with the use of design patterns [55] or traits [103]. Aspect oriented programming [43] is an attempt to solve these and other problems identified in the object oriented paradigm. It is a technique that intends to achieve more modularity in situations where object orientation and the use of design patterns is not enough.

The main motivation for the development of AOP was the alleged inability of OOP and other current programming paradigms to fully support the *separation of concerns*

principle [65]. According to AOP proponents, AOP solves some problems of OOP by allowing an adequate representation of the so-called *crosscutting concerns* [107]. With AOP, code that implements crosscutting concerns, i.e. that implements specific functions that affect different parts of a system and would be scattered and tangled in an OOP implementation, can be localized, increasing modularity. With this increase in modularity, one can achieve software that is more adaptable, maintainable and evolvable in the face of changing requirements [97]. Other expected benefits of using AOP are a more readable and reusable code and a more natural mapping of system requirements to programming constructs.

To work with AOP one needs an aspect-oriented programming language or an aspect-oriented extension/framework to an existing language. AspectJ is a general purpose, seamless aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns. We have chosen to use AspectJ because it seems to be the most promising approach to aspect orientation. It adds support for aspects to the well-established Java language and is regarded as the most mature approach to AOP at the time of writing.

An AspectJ program is a Java program with some extra constructs. These included language constructs allow the implementation of AOP features such as *pointcuts*, *advice*, *inter-type declarations* and *aspects*. Pointcuts and advice dynamically affect program flow, while inter-type declarations statically affect the class hierarchy of a program. Aspects are the modularization unit in AspectJ, just as a common concern's implementation in OOP is called a class. They are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations [65]. Aspects are defined by aspect declarations, which have a form similar to that of class declarations.

Implemented Aspects

In **KEMS**, we have a few aspects implementing secondary functionalities:

- the Complexity aspect introduces an `int getComplexity()` method in the classes that represent formulas and their lists and factories, and signed formulas and their

lists and factories. This method calculates the size of the objects of these classes;

- the Formula Parent Introduction (FPI) aspect adds data structures (as attributes) and methods to the Formula class. With these structures and methods, a given formula can hold references to its parents (i.e to all formulas of which this formula is a subformula) and to its signed counterparts (i.e to all signed formulas that have this formula). This aspect works together with the Formula Parents aspect, which intercepts Formula and Signed Formula object creations and uses FPI aspect methods to add those extra references to all formulas;

- the Memory Usage Tracker aspect inspects memory usage and when necessary forces the java virtual machine to perform a garbage collection on its heap memory;

- the Proof Tree Size aspect includes attributes and methods in the Proof Tree class to get the size, the number of branches and the number of nodes of a proof tree;

- finally, the Prover Thread aspect interrupts running problems. That is, when a problem is being runned, there is always a thread of the prover which was initiated only for this purpose. When the user asks **KEMS** to interrupt a running problem, or when a prespecified time limit is reached, the Prover Thread aspect captures the runnning thread and interrupts it.

We had planned to make more use of AOP in **KEMS**, but we find out it was not as productive as we thought it would be (in line with what was described in [109]). This happened, in part, due to the evolutionary nature of our development and to the lack of adequate tools for refactoring aspects (to aspects and to classes). Therefore we decided to use it only on secondary features. But as it was described in [90], it is still possible to use AOP to design future **KEMS** strategies.

C.4 Strategies

As we have said in Section C.2, a **KEMS** strategy is responsible, among other things, for: *(i)* choosing the next rule to be applied, *(ii)* choosing the formula on which to apply

the (PB) rule, and *(iii)* verifying branch closure. In other provers, these features can be scattered in several prover modules if strategies are not designed as first-class citizens. In **KEMS**, strategies are first-class citizens. This is **KEMS** most important feature. In **KEMS**, the core of the implementation is shared by all strategies and each strategy is defined in one main class and possibly some auxiliary classes and aspects. In this way we can prove the same problem with several different strategies and compare the proofs obtained.

The idea of having several strategies implemented in the same prover and being able of varying the strategy used is not new: in [72], in the context of first-order theorem proving, this idea is clearly present. **KEMS** is a multi-strategy tableau prover for **KE** systems. As **KE** is a tableau method that is available for several logics, **KEMS** can be used to provide effective provers for many different logical systems, as well as to enable the comparison between strategies for these systems.

Let us see how some other object-oriented tableau-based provers represent strategies. The prover developed by Wagner Dias, which we call WDTP [38], was written in C++ and implements Analytic [106] and KE propositional tableaux methods. jTAP [3] is a propositional tableau prover, written in Java, based on the method of signed Analytic Tableaux. Both systems have some strategies implemented and can be extended with new ones, but strategies are not well modularized since one has to create subclasses of one or more classes of the system, as well as modify existing ones, to implement a new strategy. LOTREC [46] is a generic tableau prover for modal and description logics (MDLs). It aims at covering all logics having possible worlds semantics, in particular MDLs. It is implemented in Java. Logic connectives, tableau rules and strategies are defined in a high-level language specifically designed for this purpose. In LOTREC, strategies are described using a very simple language, not in a programming language. They are limited to establishing the order and the number of times the rules will be applied.

C.4.1 Strategy Implementation

As we have said above, in **KEMS** a strategy is implemented by writing a main class and possibly some auxiliary classes and aspects. We will describe later each strategy implemented in **KEMS**. Let us discuss now some features that are common to all strategies.

We are going to see now the basic procedure for applying rules in a node. The procedure which is the basis for all strategies is the **KE** canonical procedure described in [29]. This generic procedure establishes the following order for **KE** rule applications:

1. all one-premise rules;
2. all two-premise rules;
3. (PB) rule.

This sequence is rather obvious since (PB) rule applications are computationally more expensive.

Another important feature of **KEMS** strategies is that they choose which rules are going to be applied by analyzing signed formula structure. For instance, suppose a **CPL KE** strategy is analyzing a node that has the following list of signed formulas: [$\mathtt{T}\ A \rightarrow B$, $\mathtt{T}\ C \vee A$, $\mathtt{F}\ C$, $\mathtt{T}\, D \rightarrow E$]. By iterating over this list we first analyse '$\mathtt{T}\ A \rightarrow B$' and see that it can be used as the main premise in application of the ($\mathtt{T}{\rightarrow}_1$) and ($\mathtt{T}{\rightarrow}_2$) rules[6]. These rules tell us that the minor premise could be either '$\mathtt{T}\ A$' or '$\mathtt{F}\ B$', but as none of these signed formulas in our list, we cannot apply any of these rule.

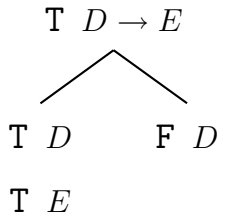
Next the strategy analyses the '$\mathtt{T}\ C \vee A$' s-formula and discovers that it can be the major premise of ($\mathtt{T}{\vee}_1$) and ($\mathtt{T}{\vee}_2$) rule applications. These rules tells us that the minor premise could be either '$\mathtt{F}\ C$' or '$\mathtt{F}\ A$'. As '$\mathtt{F}\ C$' is in our list, we apply the ($\mathtt{T}{\vee}_1$) rule and include '$\mathtt{T}\ A$' in our list. Now we can apply ($\mathtt{T}{\rightarrow}_1$) rule and obtain '$\mathtt{T}\ B$'.

Some s-formulas can be the main premise of two-premise rule applications, some cannot. In **CPL KE** all s-formulas whose connective is binary can be the main premise of a two-premise rule application, but that can vary from logic to logic. When we start the proof search procedure for a problem, we create a list of *not analyzed main candidates*

[6]Because the analyzed s-formula's sign is $\mathtt{T}$ and its main connective is '$\rightarrow$'.

(namc) that contains all problem s-formulas that can be main premise. Every time we use one of these s-formulas as main premise in some rule application, we remove it from the list. If, after we have applied all possible linear rules, the tableau is still open, and the *namc* list is not empty, we can choose one of the s-formulas on the list to serve as a basis for a (PB) application. In **KEMS**, after choosing a s-formula $\mathcal{S}\ \mathcal{F}$ from the list, where $\mathcal{F}$ is something like '$A \oslash B$' and '$\oslash$' is a binary connective, all strategies apply the (PB) rule by branching on '$\mathcal{S}_1\ A$' and '$\mathcal{S}_2\ A$'[7]. And $\mathcal{S}_1$ is the sign that allows the application of a two-premise rule on the left successor node.

Continuing our example, after iterating over the list we have only one formula in our *namc* list: '$\mathtt{T}\ D \rightarrow E$'. By following the procedure described in the previous paragraph, we apply (PB) with $\{\mathtt{T}\ D, \mathtt{F}\ D\}$ and the result is the following:



Therefore, from the node we were analyzing we created two new successor node. Now let us describe how strategies deal with nodes. Every strategy has to keep a stack of open nodes. When a strategy starts the proof search procedure for a problem, the root branch, containing the s-formulas of the problem, is put on the top of this stack. Only one node is being expanded at a given time. We call this node the *current* node.

The procedure for dealing with nodes is as follows:

1. Remove the node which is on the top of the open node stack and make it the current node. If the stack is empty, finish the procedure;

2. Apply all possible linear rules to the current node. If the node closes, go to the first step. If it remains open, apply the (PB) rule, put the two newly created nodes on the stack (first the right node and after that the left, so that the left goes to the top) and go back to the first step. If no (PB) rule can be applied, finish the procedure.

[7]That is, they always choose the left subformula to be the motivation for (PB) application, that is, the auxiliary formula of a two-premise rule application.

When the procedure finishes, the strategy checks if the root node is closed. If it is, that is because all of its child nodes are also closed. Therefore the tableau is closed. If the root node is not closed, this happened because at least one of its child nodes remained open and completed, thus the tableau is open.

C.4.2 Sorters

The order in which s-formulas are analyzed is an aspect that can have a strong influence on a strategy performance. As we have already said, at every moment in the proof search procedure, the prover has a list of not-analyzed s-formulas in the current node from which it is going to choose the next formula to be analyzed. As the s-formulas in the beginning of the list are analyzed first, if we sort the s-formulas we can change the strategy behavior.

In **KEMS**, the s-formulas are sorted before rules are applied by one *signed formula sorter*. Let us describe each of the thirteen sorters we have implemented[8]. The first two sorters are related to the order in which signed formulas are inserted in **KE** tableau nodes' signed formula list:

1. insertion order - most recently inserted s-formulas go to the end of the list;

2. reverse order - most recently inserted s-formulas go to the beginning of the list.

The five connective sorters behave similarly: the s-formulas where a given connective appears as the main connective are put in the beginning of list:

1. and connective;

2. or connective;

3. implication connective;

4. bi-implication connective;

5. exclusive or connective.

[8]In future versions we may have more sorters and even combine two or more in a prover configuration.

The two sign sorters behave in a similar way: the s-formulas with a given sign are put in the beginning of list:

1. true sign;

2. false sign.

In the two complexity sorters, the s-formulas are sorted according to their size:

1. increasing complexity - the smaller s-formulas appear first;

2. decreasing complexity - the more complex s-formulas appear first.

In the two string sorters, the s-formulas are sorted according to the string that represents them:

1. string order - sorts s-formulas in alphabetical order;

2. reverse string order - sorts s-formulas in reverse alphabetical order.

The two sorters above were implemented only to be compared with others.

The results presented in Section D.2 will illustrate the impact of sorters on **KEMS** prover configurations performance.

C.4.3 CPL Strategies

Here we will discuss the six strategies implemented for **CPL**.

Simple Strategy

The first implemented strategy is called *Simple Strategy* because it uses simplification rules (it implements the **s-CPL-KE** system). This is the order of rule applications in Simple Strategy:

1. all one-premise rules;

2. all simplification rules in which '`T` $\top$' or '`F` $\bot$' is the minor premise;

3. all other simplification rules[9];

4. (PB) rule.

Tableau proof trees are represented by the ProofTree class. A ProofTree contains a list of nodes and three references to other ProofTree objects: a reference to its left child, a reference to its right child, and a reference to its parent. Each of these references may be null. Only the root proof tree does not have a parent. Every ProofTree either has two children or none.

ClassicalProofTree is a subclass of ProofTree in which each node contain a list that we call s-formula container. A s-formula container includes a s-formula, a state and an origin. The state of a signed formula container (sf-container) can be either *not analyzed*, *analyzed*, and *fulfilled*. The first state is for those containers that contain signed formulas that can be used as the major premise in some rule application, but were not yet used. The second is for the containers that contain signed formulas that were used as major premise. And the last state is for the containers that contain signed formulas that cannot be used as major premise. A container origin may contain references to the rule that originated that container as well as the premises used in that rule application. Besides that, a ClassicalProofTree contains a *namc* list and has additional methods for dealing with node closure.

To apply simplification rules, we designed a subclass of the ClassicalProofTree class called FormulaReferenceClassicalProofTree. This class contains data structures that keep track of all references between signed formula containers. For instance, if we have a sf-container that contains the '$\mathtt{T}\ \ A \rightarrow B$' s-formula, we can find all s-formulas that have '$A \rightarrow B$' as subformula so that we can apply a simplification rule. And all this data is kept in memory, what makes this strategy consume a lot of memory.

Let us see an example of this strategy in action. In this example we will work with labelled signed formulas '$l : \mathcal{S}\ \mathcal{F}$' where l is a label that contains the '$\mathcal{S}\ \mathcal{F}$' signed formula's origin. Signed formulas which come from the problem are labelled p_i, where i is an index. $\mathrm{app}_c^p(\mathrm{R}, \mathcal{S}_1\ \mathcal{F}_1, t)$ is the label associated with the application of a basic (non simplification)

[9]Notice that no non-simplification two-premise rule is used in this strategy.

rule R (that has p premises and c conclusions), where '$\mathcal{S}_1$ $\mathcal{F}_1$' is the main premise and the third parameter (t) can either not appear, or be '$\mathcal{S}_2$ $\mathcal{F}_2$', the auxiliary premise (for two-premise rules), or even be an i which is the number of the conclusion associated with the label (for two-conclusion rules).

To indicate when a node is closed, we have the close(sfl_1, sfl_2) label, where sfl_1 and sfl_2 are the labels of the s-formulas that justify the closure. Finally,

$$\text{simplSubst}(sfl_m, sfl_a, \Theta(X))$$

is the label of the s-formula which is the result of applying a simplification rule, where sfl_m is the main premise label, sfl_a is the auxiliary premise label, and $\Theta(X)$ is the subformula of the main premise to which the simplification rule is applied (see Section C.2.3 for simplification rule definition).

In the examples below, the '$\times$' symbol denotates that a node is closed. The '$\blacksquare$' symbol is used to state that a node is open and completed, that is, no further rule can be applied and from that node we can find a valuation that falsifies the sequent. The '$\square$' symbol is used to state that a node is open but not completed. When another (a previous) node is found to be open and completed, that is, when we find a valuation that falsifies, we no longer need to analyze other nodes that are in the open node stack. Therefore the other nodes remain open but usually are not completed.

The first example only illustrates the use of labels in a proof of $A, A \to B \vdash B$:

$$\begin{array}{rl} p_1: & \mathtt{T}\, A \\ p_2: & \mathtt{T}\, A \to B \\ p_3: & \mathtt{F}\, B \\ \hline g_1 := \text{app}_1^2(\mathtt{T} \to_1, p_2, p_1): & \mathtt{T}\, B \\ \text{close}(g_1, p_3): & \times \end{array}$$

The second example shows three applications of simplification rules. We used the $g_i := l$ notation, stating that g_i abbreviates label l, to simplify the labels in the following open tableau for $A, D \to ((A \to B) \wedge (C \to A)) \vdash B$:

$$
\begin{array}{rrc}
 & p_1: & \mathtt{T}\,\top \\
 & p_2: & \mathtt{F}\,\bot \\
 & p_3: & \mathtt{T}\,A \\
 & p_4: & \mathtt{T}\,D \rightarrow ((A \rightarrow B) \wedge (C \rightarrow A)) \\
 & p_5: & \mathtt{F}\,B \\
\hline
g_1 := & \mathrm{simplSubst}(p_5, p_3, A \rightarrow B): & \mathtt{T}\,D \rightarrow (B \wedge (C \rightarrow A)) \\
g_2 := & \mathrm{simplSubst}(g_1, p_5, B \wedge (C \rightarrow A)): & \mathtt{T}\,D \rightarrow \bot \\
g_3 := & \mathrm{simplSubst}(g_2, p_2, D \rightarrow \bot): & \mathtt{T}\,\neg D \\
 & \mathrm{app}_1^1(\mathtt{T}\neg, g_3): & \mathtt{F}\,D \\
 & & \blacksquare
\end{array}
$$

Our last example uses the $PB(sfl_1, s)$ label to indicate that the signed formula to which this label is associated is the result of applying the (PB) rule to the signed formula whose label is sfl_1. The b parameter indicates if this is the label associated with the left successor node (when $s = l$) or right successor node ($s = r$). The example below is an open and completed tableau for $E \vee C \vdash A \rightarrow \neg(C \vee D)$:

$$
\begin{array}{rrcc}
 & p_1: & \mathtt{T}\,E \vee C & \\
 & p_2: & \mathtt{F}\;\; A \rightarrow \neg(C \vee D) & \\
 & \mathrm{app}_2^1(\mathtt{F}\rightarrow, p_2, 1): & \mathtt{T}\,A & \\
g_1 := & \mathrm{app}_2^1(\mathtt{F}\rightarrow, p_2, 2): & \mathtt{F}\,\neg(C \vee D) & \\
g_2 := & \mathrm{app}_1^1(\mathtt{F}\neg, g_1): & \mathtt{T}\,C \vee D & \\
 & \swarrow & & \searrow \\
g_3 := & PB(g_2, l): & \mathtt{F}\;\; C & PB(g_2, r): \;\; \mathtt{T}\;\; C \\
 & \mathrm{app}_1^2(\mathtt{T}\vee, g_2, g_3) & \mathtt{T}\;\; D & \square \\
 & \mathrm{app}_1^2(\mathtt{T}\vee, p_1, g_3) & \mathtt{T}\;\; E & \\
 & & \blacksquare &
\end{array}
$$

Memory Saver Strategy

The Memory Saver Strategy implements almost the same algorithm implemented by Simple Strategy but keeps the minimum amount of data structures in memory. For

instance, instead of using the FormulaReferenceClassicalProofTree class for keeping references to formulas in memory, this strategy uses an OptimizedClassicalProofTree class (that does not keep those references). And it uses a ReferenceFinder class that has methods for searching the same references stored in a FormulaReferenceClassicalProofTree whenever they are needed.

Backjumping Simple Strategy

Backjumping is a technique used in backtracking algorithms that allows for efficient pruning of search spaces. It has been proposed in the early 1990s, and has been extensively applied in constraint propagation [36], but it has hardly ever been applied to theorem provers [40], specially tableau based ones [63]. Backjumping can be very useful in tableau based theorem provers because it can be used to prevent repeatedly re-solving the same subproblem.

The Backjumping Simple Strategy implements the backjumping technique and is an extension of Simple Strategy. The only difference occurs when nodes are closed. Recall that a **KE**-tableau tree is a tree whose nodes contain a list of signed formulas. A left (right) node is a node which is the left (right) child of another node. The first signed formula of a left node is called a decision. A node that contains a decision is called a *decision node*. In the Backjumping Simple Strategy, whenever a node closes, the decision nodes used (as premises) to close that node must be marked as used. And whenever a right node closes, if any of its grandparents is a not used decision, its sibling can be closed by backjumping.

In Section D.1.1 we show a family of problems used to test the Backjumping Simple Strategy. The tableau proofs (for instances of this family) generated by strategies that do not use backjumping may include redundant sub-trees. This is clear in Figure C.13 where we show an example of a B_PHP^2_n proof (see Section D.1.1). The sub-tree containing the **KE** PHP$_n$ proof ($\mathcal{T}_1$) appears four times.

In Figure C.14 we show a proof of the same problem using backjumping. The $\mathcal{T}_1$ sub-tree now appears only once because the nodes containing $\mathtt{F}\ A_{1,1}$ and $\mathtt{F}\ A_{2,1}$ are declared

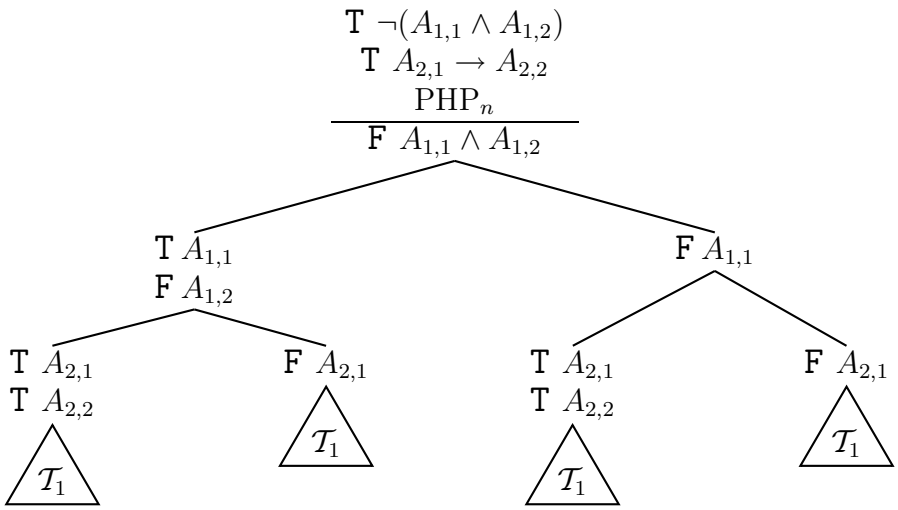


Figure C.13: A proof of B_PHP^2_n.

closed by the backjumping strategy when it is found that the T $A_{1,1}$ and T $A_{2,1}$ decisions were not used to close $\mathcal{T}_1$. That is, the $\mathcal{T}_1$ sub-proof now needs to be generated only once, in the leftmost leaf node. When the strategy verifies that the two open nodes came from unused decisions, it closes these nodes.

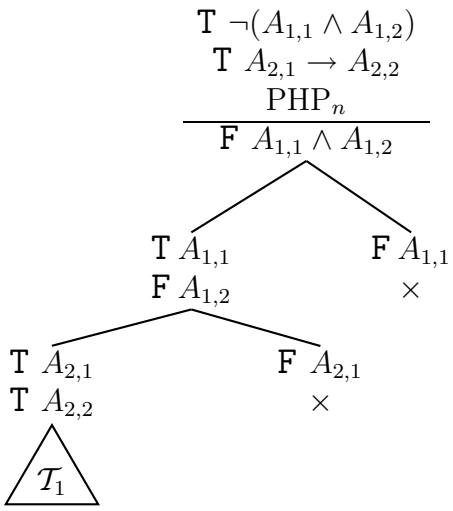


Figure C.14: A proof of B_PHP^2_n using backjumping.

Learning Strategy

In the Learning Strategy we implemented the learning technique [40] used in SAT solvers. The idea is the following: whenever a left node closes we look for the reasons for this closure. The reasons are two s-formulas: T X and F X. Then we look for the

binary s-formulas that were used as main premise to obtain the closing reasons. Then we apply general resolution [1] to these two formulas and include this *learned formula* in the parent node.

For instance, suppose we close a node of an instance of PHP_3 with $\mathtt{F}\ p_{3,1}$ and $\mathtt{T}\ p_{3,1}$ (see Figure C.15). The s-formulas that gave origin to these formulas are $\mathtt{F}\ p_{2,1} \wedge p_{3,1}$ and $\mathtt{T}\ p_{3,0} \vee p_{3,1}$. Suppose we apply $(\mathtt{F}\wedge_2)$ rule to $\mathtt{F}\ p_{2,1} \wedge p_{3,1}$ and $\mathtt{T}\ p_{3,1}$; the result would be $\mathtt{F}\ p_{2,1}$. And suppose we apply $(\mathtt{T}\vee_2)$ to $\mathtt{T}\ p_{3,0} \vee p_{3,1}$ and $\mathtt{F}\ p_{3,1}$; the result would be $\mathtt{T}\ p_{3,0}$. We take these two results ($\mathcal{S}_1\ \mathcal{F}_1$ and $\mathcal{S}_2\ \mathcal{F}_2$) and create a learned formula (lf) is the following way:

- $\mathrm{lf}_i = \mathcal{F}_i$ if $\mathcal{S}_i = \mathtt{T}$, otherwise $\mathrm{lf}_i = \neg(\mathcal{F}_i)$;
- $\mathrm{lf} = \mathtt{T}\ (\mathrm{lf}_1 \vee \mathrm{lf}_2)$.

This '$\mathtt{T}\ \neg(p_{2,1}) \vee p_{3,0}$' learned formula is then included the node which is the parent of the current node (see Figure C.16) so that it can be used in the proof search on the open nodes.

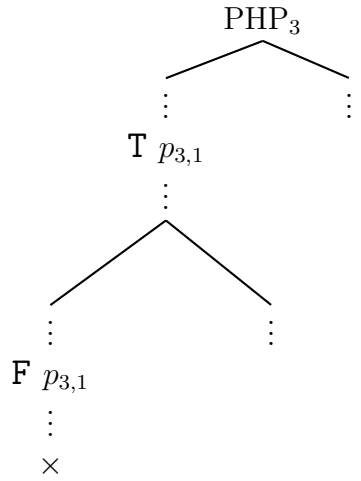


Figure C.15: A proof of PHP_3.

Comb Learning Strategy

This strategy produces a proof whose left branches have a height equal or less than 1, that is, a comb-like proof (see Figure C.17). The idea is the following: whenever we close

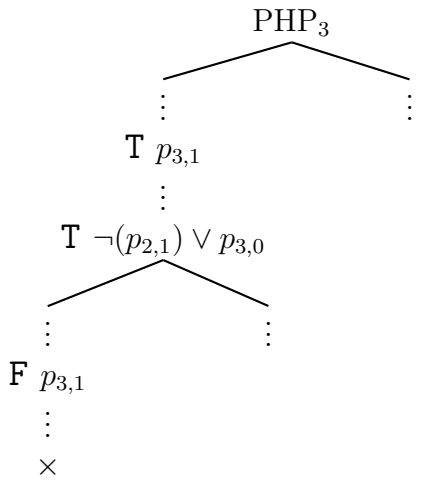
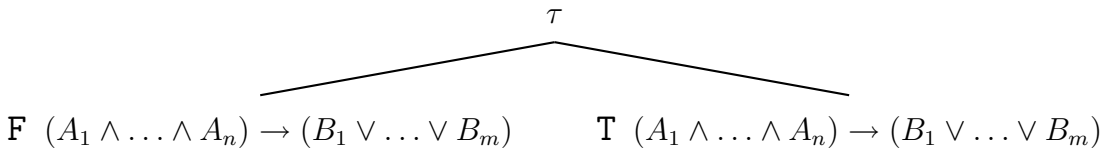


Figure C.16: An example of learning in a proof of PHP$_3$.

a left node, the strategy gathers the used decisions and separates them in two groups: those with a T sign ($\{\mathtt{T}\ A_1, \ldots \mathtt{T}\, A_n\}$) and those with a F sign ($\{\mathtt{F}\ B_1, \ldots \mathtt{F}\, B_m\}$). Then it discards the whole left node from the root node (τ) and creates two new child nodes for τ:



The left node does not have to be expanded, because it will surely close since the '$\mathtt{F}\ (A_1 \wedge \ldots \wedge A_n) \rightarrow (B_1 \vee \ldots \vee B_m)$' learned formula will be decomposed into the decisions that led the original left branch to close. So the proof can continue with the right node. It is important to notice that this is a naïve kind of learning that usually produces proofs which are bigger than other strategies' proofs.

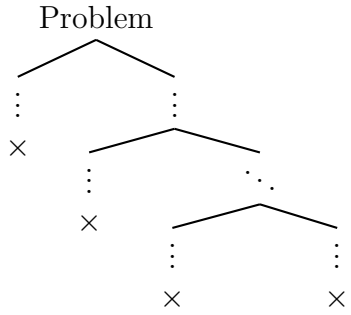


Figure C.17: Sketch of a Comb Learning Strategy proof.

Configurable Strategy

All previous **CPL** strategies implement the **s-CPL-KE** system. This strategy implements the **e-CPL-KE** system. This is the order of rule applications:

1. all one-premise rules;
2. all two-premise rules;
3. (PB) rule.

The other features are equal to Simple Strategy features.

This strategy is useful because, by using sorters, it allows a higher control over which formulas are analyzed first. Besides that, it is interesting to compare this strategy with the strategies that implement the **s-CPL-KE** system, because the comparison may show when simplification rules are most useful.

C.4.4 mbC Strategies

For **mbC**, we have implemented the following two strategies.

mbC Simple Strategy

This is an extension of **CPL** Simple Strategy for **mbC**. It implements the **e-mbC-KE** system – therefore it does not use any simplification rule. This is the order of rule applications:

1. all **mbC** one-premise rules;
2. all **mbC** two-premise rules;
3. (PB) rule.

An important difference here is that in **mbC**'s ($\mathtt{T}\neg'$) rule the two premises have the same size (in **CPL** two-premise rules the major premise is always bigger than the minor premise). We have proved (see Section B.2.3) that we only need to branch on a '$\circ A$' formula if it already appears as subformula of some formula in this branch. So we had

to add this check before applying the (PB) rule. The other features are equal to Simple Strategy features.

mbC Extended Strategy

This is an extension of the previous strategy. It implements the **e-mbC-KE** system with the ($\mathtt{T}\circ''$) and ($\mathtt{T}\neg''$) additional rules (see Section C.2.4). These rules are applied after all other **mbC** two-premise rules and before (PB) rule. The other features are equal to **mbC** Simple Strategy features.

C.4.5 mCi Strategies

For **mCi**, we have implemented the following two strategies.

mCi Simple Strategy

This strategy is very similar to **mbC** Simple Strategy. It implements the **e-mCi-KE** system. This is the order of rule applications:

1. all **mCi** one-premise rules;
2. all **mCi** two-premise rules;
3. (PB) rule.

But here we do not have to restrict the application of the (PB) rule because of the ($\mathtt{T}\neg'$) rule. All other features of this strategy are equal to **mbC** Simple Strategy features.

mCi Extended Strategy

This is an extension of the previous strategy. It implements the **e-mCi-KE** system with two additional rules: ($\mathtt{T}\circ''$) and ($\mathtt{T}\neg''$) (see Section C.2.4). These rules are applied after all other **mCi** two-premise rules and before (PB) rule. The other features are equal to **mCi** Simple Strategy features.

Strategy	Main feature
CPL Simple Strategy	Keeps formula reference data structures in memory
CPL Memory Saver Strategy	Does not keep formula reference data structures in memory
CPL Backjumping Simple Strategy	Implements the backjumping technique
CPL Learning Strategy	Implements a learning technique
CPL Comb Learning Strategy	Implements the comb learning technique
CPL Configurable Strategy	Allows a higher control over which formulas are analyzed first
mbC Simple Strategy	An extension of **CPL** Simple Strategy for **mbC**
mbC Extended Strategy	An extension of **mbC** Simple Strategy that applies derived rules before applying (PB)
mCi Simple Strategy	An extension of **CPL** Simple Strategy for **mCi**
mCi Extended Strategy	An extension of **mCi** Simple Strategy that applies derived rules before applying (PB)

Table C.1: Overview of **KEMS** Strategies.

C.5 Conclusion

KEMS current version implements strategies for three logics: **CPL**, **mbC** and **mCi**. An overview of the implemented strategies is presented in Table C.1. We have shown that to solve a problem with a strategy we must choose a sorter. The implemented sorters are described in Table C.2. In Appendix D we present the results obtained by **KEMS** with several families of problems using these strategies and sorters.

Sorter	Description
Insertion Order	least recently inserted s-formulas are analyzed first
Reverse Order	most recently inserted s-formulas are analyzed first
And	s-formulas with '$\wedge$' as main connective are analyzed first
Or	s-formulas with '$\vee$' as main connective are analyzed first
Implication	s-formulas with '$\rightarrow$' as main connective are analyzed first
Bi-implication	s-formulas with '$\leftrightarrow$' as main connective are analyzed first
Exclusive Or	s-formulas with '$\oplus$' as main connective are analyzed first
True	s-formulas with `T` as sign are analyzed first
False	s-formulas with `F` as sign are analyzed first
Increasing	smaller s-formulas are analyzed first
Decreasing	bigger s-formulas are analyzed first
String Order	s-formulas whose representation as a string of characters come first in alphabetical order are analyzed first
Reverse String Order	s-formulas whose representation as a string of characters come last in alphabetical order are analyzed first

Table C.2: Overview of **KEMS** sorters.

Apêndice D

KEMS Evaluation

Theorem provers are usually compared by using benchmarks [112]. SATLIB [102] (for **CPL**) and TPTP [111] (for first-order classical logic) are two web sites that contain benchmark problems to evaluate theorem provers. We have chosen to evaluate **KEMS** using as benchmarks some families of difficult problems [12, 95], some of which well known and some new families we developed to test **KEMS**. We present below the families we used to evaluate **CPL**, **mbC** and **mCi** strategies.

D.1 Problem Families

A *problem family* is a set of problems that we know, by construction, whether they are valid, satisfiable or unsatisfiable. For any family f we have a procedure such that, for any $n \in \mathbb{N}$, $n \geq 1$, we construct an instance f_n of this family.

In the problem families described below, formulas such as $\bigwedge_{i=m}^{n} A_i$ (an iterated conjunction) and $\bigvee_{i=m}^{n} A_i$ (an iterated disjunction) may appear, where $m, n \in \mathbb{N}$. If $m = n$, then both reduce to A_m. If $m < n$, then the first reduces to $(A_m \wedge (A_{m+1} \wedge (\ldots \wedge (A_{n-1} \wedge A_n))))$ and the second to $(A_m \vee (A_{m+1} \vee (\ldots \vee (A_{n-1} \vee A_n))))$. And if $m > n$, then the first formula is an empty conjunction (which corresponds to the $\top$ formula) and the second is an empty disjunction (which corresponds to the $\bot$ formula). Besides that, whenever we have $(A \wedge \top)$, $(\top \wedge A)$, $(A \vee \bot)$ or $(\bot \vee A)$, for any formula A, we replace any of these formulas by A.

D.1.1 CPL Problem Families

The problem families used to evaluate **KEMS CPL** strategies contain explicit propositional valid sequents whose proofs in Sequent Calculus [56], Analytic Tableaux [106] or Resolution [100] can be exponential. For instance, zChaff [53], one of the fastest SAT solvers available, takes a couple of days to prove the instance number 14 of the PHP family on a personal computer.

Γ Problems

The n-th instance of the Γ family [12] has $2n$ propositional variables and $2n + 2$ formulas. It is possible to find both exponential and non-exponential (in n) proofs of instances of this family using **AT**. For the n-th instance (Γ_n), the sequent to be proved is:

$$p_1 \vee q_1, C_n \vdash p_{n+1} \vee q_{n+1}$$

where

$$C_n = \{p_i \rightarrow (p_{i+1} \vee q_{i+1}), q_i \rightarrow (p_{i+1} \vee q_{i+1}) | 1 \leq i \leq n\}$$

H Problems

For this family, the sequent to be proved for the n-th instance is

$$\vdash H_n$$

where H_n is constructed in the following way:

$$\begin{aligned}
H_1 &= p_1 \vee \neg p_1 \\
H_2 &= (p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2) \vee (\neg p_1 \wedge \neg p_2) \\
H_3 &= (p_1 \wedge p_2 \wedge p_3) \vee \ldots \vee (\neg p_1 \wedge \neg p_2 \wedge \neg p_3) \\
&\vdots
\end{aligned}$$

These formulas (also called “truly fat” formulas) were defined in [28] and used to prove that the Analytic Tableaux method cannot polynomially simulate the truth-table

method[1]. Each instance of this family has n propositional variables, but the size of the formula H_n grows exponentially in n (because any instance has 2^n clauses $q_1 \wedge \ldots \wedge q_n$, where each q_i is either equal to p_i or $\neg p_i$).

Statman Problems

Statman formulas [108] can be constructed as follows. Consider

$$A_k = \bigwedge_{j=1}^{k}(p_j \vee q_j)$$
$$B_1 = p_1$$
$$C_1 = q_1$$

and, inductively:

$$B_{i+1} = A_i \rightarrow p_{i+1} \qquad C_{i+1} = A_i \rightarrow q_{i+1}$$

The sequent to be proved for the n-th instance of this family is:

$$B_1 \vee C_1, \ldots, B_n \vee C_n \vdash p_n \vee q_n$$

Statman has proved that this family of formulas has polynomial proofs in sequent calculus if we use the cut rule, but only exponential size cut-free proofs [12].

Pigeon Hole Principle Problems

The Pigeon Hole Principle (PHP) [95, 12] states that given $n-1$ pigeon holes and n objects to be put in these holes, there is always one hole that will receive at least two objects.

The sequent to be proved is

$$\vdash \mathrm{PHP}_n$$

where

$$\mathrm{PHP}_n = A_n \rightarrow B_n$$

[1]And this problem family has a structure which is very similar to the structure of the problems used in [22] (and commented in [76]) to analyze the complexity of Analytic Tableaux.

and

$$A_n = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n-1} p_{i,j}$$

$$B_n = \bigvee_{i=1}^{n-1} \bigvee_{k=i}^{n} \bigvee_{j=1}^{n-1} (p_{i,j} \wedge p_{k,j})$$

where $p_{i,j}$ expresses that object number i was inserted in hole number j. The A_n formula states that every object goes to some hole and the B_n formula that at least one hole receives 2 objects.

This problem leads to a lot of branching in Analytic Tableaux and Sequent Calculus. In Sequent Calculus, cut-free proofs are exponential but there is a very complicated polynomial proof with cut [9].

U Problems

The U family was described in [95]. It is stated there that resolution system proofs of this problem instances increase exponentially with n. The sequent to be proved for the n-th instance of this family is $\vdash U_n$, where U_n is defined as follows:

$$\begin{aligned}
U_1 &: \quad (P_1 \leftrightarrow P_1) \\
U_2 &: \quad (P_1 \leftrightarrow (P_2 \leftrightarrow (P_1 \leftrightarrow P_2))) \\
U_3 &: \quad (P_1 \leftrightarrow (P_2 \leftrightarrow (P_3 \leftrightarrow (P_1 \leftrightarrow (P_2 \leftrightarrow P_3))))) \\
&\vdots \\
U_n &: \quad (P_1 \leftrightarrow (P_2 \leftrightarrow (P_3 \leftrightarrow \ldots \leftrightarrow (P_n \leftrightarrow (P_1 \leftrightarrow P_2 \ldots \leftrightarrow P_n) \ldots)
\end{aligned}$$

Square Tseitin Problems

We present below a description (found in [95]) of the arbitrary graph problems due to Tseitin:

> Consider a graph with the edges labelled. (...) Assign 0 or 1 arbitrarily to nodes of the graph. For each node of the graph, we associate a set of clauses as follows:

1. every label of an edge emanating from that node will occur in each clause (of the set of clauses generated from that node);
2. if the node is assigned 0, then the number of negated literals in each of the generated clauses is to be odd. Generate all such clauses for that node;
3. if the node is assigned 1, then the number of negated literals in each of the generated clauses is to be even. Generate all such clauses for that node.

Tseitin's result is this: the sum (mod 2) of the 0's and 1's assigned to the nodes of the graph equals 1 if and only if the set of all generated clauses is inconsistent.

A rather obvious subset of Tseitin's problems is what we call here Square Tseitin (ST) problems. The ST_n instance is constructed (using Tseitin procedure) from a graph that resembles a matrix, with n lines and n columns of nodes. Every node is connected with its four closest neighbors: left, right, top and bottom[2]. The top left node is assigned 0 and the other nodes are assigned 1. The result is an inconsistent set of clauses. By negating this set of clauses we obtain a valid sequent. We will not describe the exact procedure here; we only present ST_1, which is the following sequent: $\vdash (V_{00} \leftrightarrow H_{00}) \vee ((V_{01} \oplus H_{00}) \vee ((V_{00} \oplus H_{10}) \vee (V_{01} \oplus H_{10})))$. An alternative version of this instance is: $(V_{00} \oplus H_{00}) \vdash (V_{01} \oplus H_{00}), (V_{00} \oplus H_{10}), (V_{01} \oplus H_{10})$.

Backjumping PHP Problems

To test the performance of strategies in presence of irrelevant premises, we devised a schema to generate what we called backjumping versions of any family. For instance, for each PHP family instance we can generate one or more backjumping versions (B_PHP) of these problems. The idea is to include in the beginning of each instance one or more

[2] Obviously, some nodes do not have one or more of these neighbors.

signed formulas that are not relevant, such as:

$$A_{i,1} \oslash A_{i,2}$$

or

$$\neg(A_{i,1} \oslash A_{i,2})$$

where $i \in \mathbb{N}^+$ and $\oslash \in \{\wedge, \vee, \rightarrow\}$.

For instance, if we add three irrelevant formulas to PHP_4, we obtain the following B_PHP_4^3 instance:

$$\neg(A_{1,1} \wedge A_{1,2}), (A_{2,1} \vee A_{2,2}), (A_{3,1} \rightarrow A_{3,2}) \vdash \text{PHP}_4$$

where we know that no $A_{i,j}$ for any i or j appears in PHP_4.

We used backjumping versions of PHP to evaluate **KEMS**.

Random SAT Problems

Random K-SAT formulas were presented in [82] as a class of random formulas to be used as a benchmark for **CPL** satisfiability-testing procedures. To generate a problem instance of this class one must provide values for three parameters: number of propositional symbols, number of clauses and length of each clause. The generated set of clauses can be satisfiable or not. We have generated six instances of random K-SAT problems (see Table D.1) to test **KEMS**. To verify unsatisfiability of a set of clauses $\{C_1, C_2 \ldots C_n\}$ we try to prove whether $(C_1 \wedge C_2 \wedge \ldots \wedge C_n) \vdash \perp$ is valid.

name	propositional variables	clauses	clause length	size	**CPL** satisfiability
cnf_1	10	30	3	223	satisfiable
cnf_2	10	70	3	526	unsatisfiable
cnf_3	20	70	3	517	satisfiable
cnf_4	10	140	3	1042	unsatisfiable
cnf_5	20	140	3	1043	unsatisfiable
cnf_6	30	140	3	1066	unsatisfiable

Table D.1: Random K-SAT problems.

D.1.2 LFI Problem Families

We present below the problem families we devised to evaluate **mbC** and **mCi** theorem provers. We had two objectives in mind. First, to obtain families of valid problems whose **KE** proofs were as complex as possible. And second, to devise problems which required the use of many, if not all, **KE** rules. These families are not classically valid, since their formulas are in $For^{\circ\bullet}$. However, if we define $\circ X \stackrel{\text{def}}{=} \top$ and $\bullet X \stackrel{\text{def}}{=} \bot$ in **CPL**[3], then all families become **CPL**-valid and can be used for evaluating a **CPL** prover.

The **LFI** families can be divided into two groups: first to fourth families are valid in **mbC** (and also in **mCi**, since this logic extends **mbC**) while seventh to ninth families were designed as valid **mCi** problems. This strange numbering is due to historical reasons (fifth and sixth families are not presented here).

Note: in [18] we can find many simple problems that can be used to evaluate the correctness of provers for **LFI**s, including the so-called contraposition rules. We have used all these problems to evaluate **KEMS**. They were useful to evaluate correctness, but not to evaluate performance since they are rather small.

First family

Here we present the first family (Φ^1) of valid sequents for **mbC**. In this family all **mbC** connectives from the Σ° signature are used. The sequent to be proved for the n-th instance (Φ^1_n) of this problem is:

$$\bigwedge_{i=1}^{n}(\neg A_i), \bigwedge_{i=1}^{n}((\circ A_i) \rightarrow A_i), [\bigvee_{i=1}^{n}(\circ A_i)] \vee (\neg A_n \rightarrow C) \vdash C \tag{D.1}$$

The sequent in (D.1) means that, for every $1 \leq i \leq n$, the following set of assumptions has C as a logical consequence:

1. all $\neg A_i$ are true;
2. if $\circ A_i$ is true, then A_i is also true;
3. either one of the $\circ A_i$s or $\neg A_n \rightarrow C$ is true.

[3] A natural way of extending **CPL** presented in [18].

The explanation for this family is the following: suppose we are working with a system that allows inconsistent information representation. The A_i fact means that someone expressed an opinion A about an individual i and $\neg A_i$ means that someone expressed an opinion $\neg A$ about this same individual. For instance, if A means that a person is nice, $\neg A_3$ means that at least one person finds 3 is not nice, and A_4 means that at least one person finds 4 nice. Then $\circ A_i$ means that either all people think i is nice, or all people think i is not nice, or there is no opinion A recorded about i. '$\circ A_i \rightarrow A_i$' means that if all opinions about a person are the same, then that opinion is A.

For a subset of individuals numbered 1 to n, we have $\neg A_i$ and $\circ A_i \rightarrow A_i$ for all of them. From the fact that either $\neg A_n \rightarrow C$ or for one of them we have $\circ A_i$, we can conclude C.

It is easy to obtain polynomial **mbC KE** proofs for this family of problems. The proofs use most (but not all) **mbC KE** rules and have a comb-like form (see Figure D.1). In this figure, the shown proof of Φ^1_n closes left branches with the following sequence of signed formulas, which we call $\Phi^1_C(i)$:

$$\begin{array}{l} \mathtt{T}\ \circ A_i \\ \mathtt{T}\ A_i \\ \mathtt{F}\ A_i \\ \quad \times \end{array}$$

Thus, a proof of Φ^1_n is a $\Phi^1_p(1)$ structure, where $\Phi^1_p(i)$ is depicted below:

1. $\Phi^1_p(1)$ is:

$$\begin{array}{c}
\Phi^1_n \\
\vdots \\
\mathtt{T}\ \neg A_1 \\
\vdots \\
\mathtt{T}\ \neg A_n \\
\mathtt{T}\ (\circ A_1) \rightarrow A_1 \\
\vdots \\
\mathtt{T}\ (\circ A_n) \rightarrow A_n \\
\hline
\Phi^1_C(1) \qquad \Phi^1_p(2)
\end{array}$$

2. For $1 < i \leq n$, $\Phi^1_p(i)$ is defined as:

$$\begin{array}{c}
\mathtt{F}\ \circ A_{i-1} \\
\mathtt{T}\ [\bigvee_{j=i}^{n}(\circ A_j)] \vee ((\neg A_n) \rightarrow C) \\
\hline
\Phi^1_C(i) \qquad \Phi^1_p(i+1)
\end{array}$$

3. And when $i = n + 1$, $\Phi^1_p(i)$ is:

$$\begin{array}{c}
\mathtt{F}\ \circ A_n \\
\mathtt{T}\ (\neg A_n) \rightarrow C \\
\mathtt{F}\ \neg A_n \\
\times
\end{array}$$

Second Family

The second family of problems (Φ^2) is a variation over the first family whose proofs can be exponential. The sequent to be proved for the n-th instance of this family (Φ^2_n) is:

$$\bigwedge_{i=1}^{n}(\neg A_i),\ [\bigwedge_{i=1}^{n}[(\circ A_i) \rightarrow ([\bigvee_{j=i+1}^{n} \circ A_j] \vee ((\neg A_n) \rightarrow C))]],\ [\bigvee_{i=1}^{n}(\circ A_i)] \vee (\neg A_n \rightarrow C) \vdash C$$

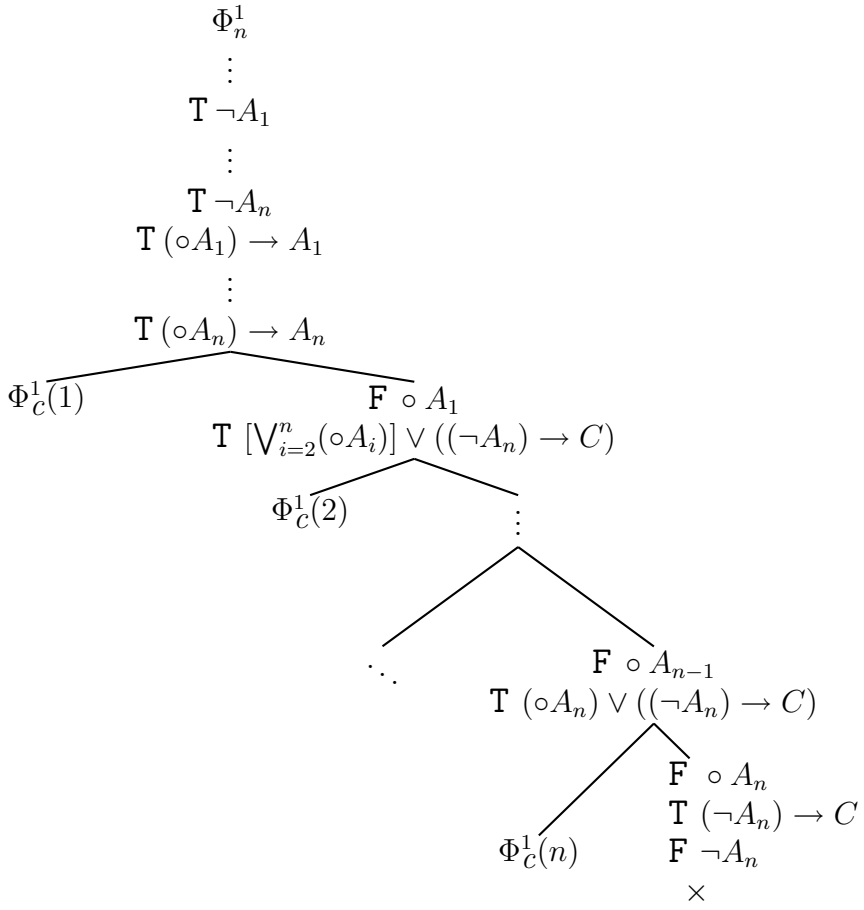
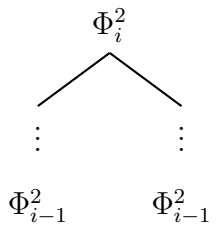


Figure D.1: A proof of Φ^1_n.

In this family, instead of closing left branches easily on $\Phi^1_C(i)$, we obtain two child branches with the same difficulty. Our objective when designing this family was to obtain a problem whose proof contained two proofs of the immediately smaller instance of the same problem:



To achieve this, we had to replace the conjunction of $(\circ A_i) \to A_i$ in (D.1) by a conjunction of

$$(\circ A_i) \to ([\bigvee_{j=i+1}^{n} \circ A_j] \vee ((\neg A_n) \to C)) \tag{D.2}$$

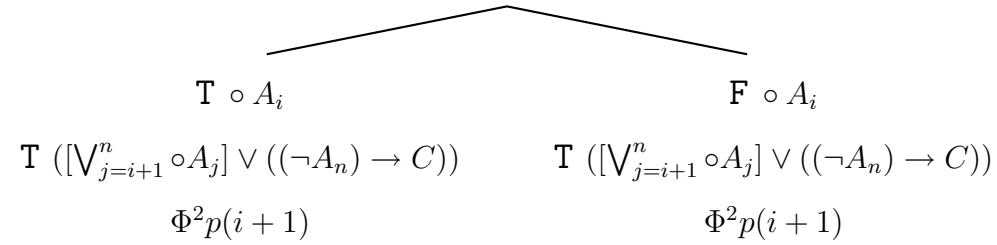
This means that for every person numbered 1 to n, if all opinions about a person are the

same, then either all opinions about some other person with a higher index are the same or $(\neg A_n) \rightarrow C)$ is true.

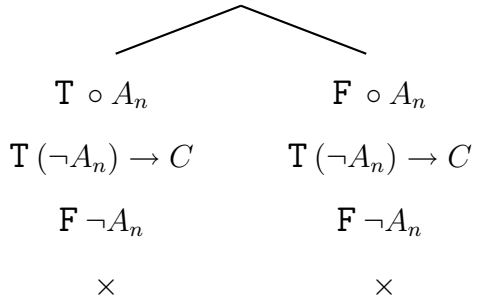
The proof of Φ^2_n is

$$\Phi^2_n$$
$$\mathtt{T}\,(\neg A_1) \wedge (\neg A_2) \wedge \ldots \wedge (\neg A_n)$$
$$\Phi^2 p(1)$$

Where, for $1 \leq i < n$, $\Phi^2 p(i)$ is:



Finally, $\Phi^2 p(n)$ is



In Figure D.2 we show a proof of Φ^2_3. The right branch (which we omitted) is a copy of the left branch except that, in the beginning, instead of $\mathtt{T} \circ A_1$ we have $\mathtt{F} \circ A_1$.

Third Family

With the third family of problems we intended to develop a family whose instances required the application of all **mbC KE** rules. To obtain an instance of the third family (Φ^3), we have to make the following changes to an instance of the second family:

1. replace C in the right-side of the sequent by $C' \rightarrow (C'' \vee C)$;

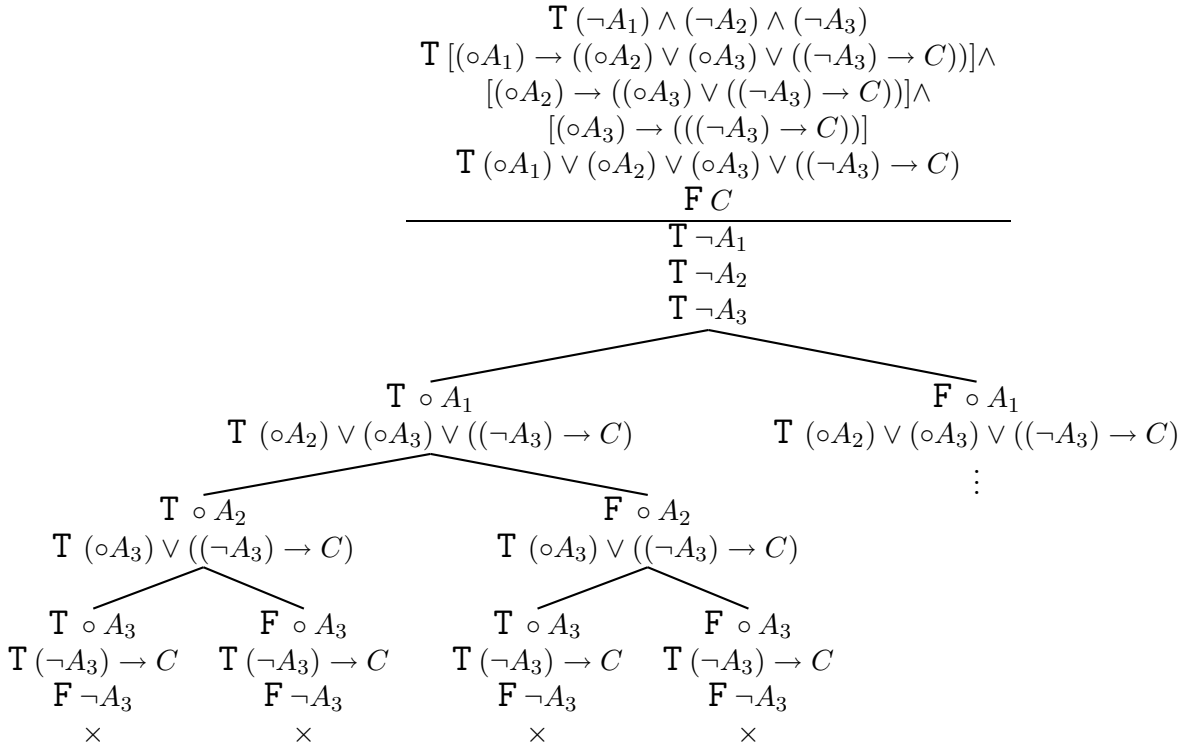


Figure D.2: A proof of Φ^2_3.

2. replace $[\bigvee^n_{j=i+1} \circ A_j] \vee ((\neg A_n) \rightarrow C)$ in $(\bigwedge^n_{i=1}((\circ A_i) \rightarrow ([\bigvee^n_{j=i+1} \circ A_j] \vee ((\neg A_n) \rightarrow C)))$ (which is left-associated on $\vee$) by a right-associated $(((\neg A_n) \wedge U_l) \rightarrow C) \vee [\bigwedge^n_{j=i+1} \circ A_j]$;

3. in (D.2), replace $(\neg A_n) \rightarrow C$ by $(U_r \wedge (\neg A_n)) \rightarrow C$;

4. add $U_l \wedge U_r$ to the left-side of the sequent.

The result is the following sequent (Φ^3_n):

$$
\begin{aligned}
& U_l \wedge U_r, \\
& \textstyle\bigwedge^n_{i=1}(\neg A_i), \\
& \textstyle\bigwedge^n_{i=1}[(\circ A_i) \rightarrow \ ((((\neg A_n) \wedge U_l) \rightarrow C) \vee \bigvee^n_{j=i+1} \circ A_j)], \\
& (\textstyle\bigvee^n_{i=1} \circ A_i) \vee \ ((U_r \wedge (\neg A_n)) \rightarrow C) \\
& \vdash \ C' \rightarrow (C'' \vee C)
\end{aligned}
$$

Fourth Family

This is a family where negation appears only in the conclusion. The sequent to be proved (Φ^4_n) is:

$$\bigwedge_{i=1}^{n}(A_i), \bigwedge_{i=1}^{n}((A_i \vee B_i) \rightarrow (\circ A_{i+1})), [\bigwedge_{i=2}^{n}(\circ A_i)] \rightarrow A_{n+1} \vdash \neg\neg A_{n+1}$$

The formulas of this family can be explained as follows. We have two formulas to represent two types of opinion: A and B. First we assume A_i for every i from 1 to n. Then we suppose for all j from 1 to n that ($A_i \vee B_j$ implies that $\circ A_{j+1}$. And finally we assume that for every k from 2 to n the conjunction of $\circ A_k$'s implies A_{n+1}. It is easy to see that from these assumptions we can deduce A_{n+1}. So we can also deduce its double negation: $\neg\neg A_{n+1}$.

Seventh family

The seventh family (Φ^7) was designed for testing **mCi** provers. Notwithstanding, if we use the definition $\bullet A \stackrel{\text{def}}{=} \neg \circ A$, it is also valid in **mbC**. The sequent to be proved (Φ^7_n) is:

$$\bigwedge_{i=1}^{n}(A_i), \bigwedge_{i=1}^{n}(B_i \rightarrow (\neg A_i)), \bigvee_{i=1}^{n}(\circ A_i) \vdash \bigwedge_{i=1}^{n}((\bullet A_i) \vee (\neg B_i))$$

In this sequent, the $\bigvee_{i=1}^{n}(\circ A_i)$ formula is actually not essential to arrive at the conclusion. Therefore, we can define a variant (called $\Phi^{7'}$) of this family where this formula does not appear. The Φ^7 family is probably more difficult to prove because of the irrelevant premise.

Eighth family

The eighth family (Φ^8) was also designed as a valid **mCi** problem family. This family is not valid for **mbC**[4]. The sequent to be proved (Φ^8_n) is the following:

$$\bigvee_{i=1}^{n}(\bullet A_i), \bigwedge_{i=1}^{n}(A_i \rightarrow (\neg B_i)), \bigwedge_{i=1}^{n}((\neg A_i) \rightarrow C_{(n-i+1)}) \vdash \bigwedge_{i=1}^{n}((\neg B_i) \wedge C_{(n-i+1)})$$

[4]Not even if we define the inconsistency ('$\bullet$') connective in the standard way.

Ninth family

The ninth family (Φ^9) is also a **mCi** valid family which is not **mbC**-valid[5]. The sequent to be proved (Φ^9_n) is:

$$\bigvee_{i=1}^{n}(\circ A_i), \bigwedge_{i=1}^{n}(B_i \rightarrow (\bullet A_i)) \vdash \bigvee_{i=1}^{n} \neg((\circ(\circ A_i)) \rightarrow B_i)$$

We can have several valid variations of this family, for $m \geq 0$ and $p \geq 0$:

$$\bigvee_{i=1}^{n}(\neg^m(\circ A_i)), \bigwedge_{i=1}^{n}(B_i \rightarrow (\neg^m(\bullet A_i))) \vdash \bigvee_{i=1}^{n} \neg((\circ(\neg^p(\circ A_i))) \rightarrow B_i)$$

where $(\neg^1 A) \stackrel{\text{def}}{=} (\neg A)$ and $(\neg^n A) \stackrel{\text{def}}{=} (\neg(\neg^{n-1} A))$.

D.2 Results Obtained

In this section we exhibit the results obtained by **KEMS** on the problem families we just presented. All results were obtained on a Pentium IV machine with a 3.20G Hz processor and 3775MB memory running Linux version 2.6.15-26-386. The `java -jar kems.jar` command[6] was issued with the `-Xms200m -Xmx2048m` options to set the initial and maximum heap sizes. This allowed **KEMS** to use more of the computer's main memory than the default memory allocated by the java virtual machine.

As we have pointed out in Section C.3, the user can present to **KEMS** a problem and a prover configuration. For the purpose of evaluation, we have developed a command-line version of **KEMS** that accepts a sequence of problems and a set of prover configurations to be run on the problems. The so-called *sequence files* used to evaluate **KEMS** as well as all results obtained are available on [92].

For each prover configuration, the first two parameters are the logical system and the analyzer. In our sequence files, the logical system could be **CPL**, **mbC** or **mCi**, the three logical systems implemented in **KEMS** current version.

[5]Also not even if we give the standard definition of the inconsistency connective in **mbC**.

[6]The command used to execute `kems.jar`, which is the java archive that contains **KEMS** executable version.

First, we have evaluated **CPL** prover configurations with all **CPL** problems presented in Section D.1.1 and also with **LFI** problems families[7] (Section D.1.2). For **CPL**, we had problems written in the format defined in [38] and problems written in SATLIB SAT Format [101]. For the format defined in [38] we used the `sats5` analyzer, an analyzer implemented using JFlex [66] for lexical analysis and CUP [62] for syntactical analysis. And for the SATLIB SAT Format we used the `satcnf2` analyzer, implemented using the same technologies.

After that, we evaluated prover configurations for **mbC** and **mCi** with all **LFI** problems families which are valid in at least one of the two logical systems. We extended the format defined in [38] to deal with **LFI** connectives and implemented the `satlfiinconsdef` analyzer, also using JFlex and CUP.

The most important prover configuration parameters for our evaluation were the strategy and the sorter. The reason is that we have noticed through our experiments that these two are the parameters that most affect a prover configuration performance. For this reason, in the following we will refer to a prover configuration as a strategy-sorter pair, or simply a pair.

As we had a lot of problems to evaluate, we fixed the 'number of times the prover must run the proof search procedure with a given problem' parameter to one, and the 'time limit for the proof search procedure' parameter to three minutes or 180.000 milliseconds (the time spent is measured by **KEMS** in milliseconds). We set to true the 'save formula origin' option, and to false the 'discard closed branches' option and the 'save discarded branches' option.

In the tables we display below each prover configuration will be represented by a binary tuple:

```
<strategyId,sorterId>
```

where strategyId and sorterId can vary.

These are the ids for strategies:

SS - **CPL** Simple Strategy;

[7] Except the fifth and the sixth.

MSS - **CPL** Memory Saver Strategy;

BSS - **CPL** Backjumping Simple Strategy;

LS - **CPL** Learning Strategy;

CLS - **CPL** Comb Learning Strategy;

CS - **CPL** Configurable Strategy;

MBCSS - **mbC** Simple Strategy;

MBCES - **mbC** Extended Strategy;

MCISS - **mCi** Simple Strategy;

MCIES - **mCi** Extended Strategy.

And these are the ids for sorters:

ins - insertion order;

rev - reverse order;

and - 'and' connective;

or - 'or' connective;

imp - 'implication' connective;

bi-impli - 'bi-implication' connective;

xor - 'exclusive or' connective;

`T` - 'true' sign;

`F` - 'false' sign;

inc - increasing complexity;

dec - decreasing complexity;

nfo - string order;

rfo - reverse string order.

D.2.1 Gamma Family Results

The biggest Γ family instance solved within the time limit was Γ_{360}, whose size is 3606. In Table D.2 we can see the results obtained by some selected pairs with this instance. The worst pair for this instance lasts approximately the same than the best ones. And its proof size is exactly the same size of the best pairs. In fact, several pairs obtained results similar to the time and size results obtained by the best pairs. In Table D.3 we show two results obtained with the biggest instance solved by all pairs (Γ_{10}, whose size

is 106). Several other pairs obtained results very similar to the one obtained by the best and worst pair. It is easy to see that the differences in time (more that 150 times) and size (more that 25 times) between the best and the worst pair are big.

Pair	Time spent	Proof Size	Comments
<LS,inc>	166068	7926	best in time and size
<BSS,inc>	166120	7926	second best in time
<CLS,rfo>	166631	7926	third best in time
<SS,dec>	174465	7926	worst in time

Table D.2: Γ_{360} results table.

Pair	Time spent	Proof Size	Comments
<BSS,T>	7	226	best in time and size
<CLS,and>	7	226	best in time and size
<CLS,imp>	7	226	best in time and size
<CS,rev>	669	3950	an intermediate result
<CS,ins>	1061	5735	worst in time and size

Table D.3: Γ_{10} results table.

D.2.2 H Family Results

The biggest H family instance solved within the time limit was H_6 (whose size is 954) and it was solved by all pairs. The best results were obtained by MSS pairs. In Table D.4 we can see the results obtained by some selected pairs with the biggest instance solved. The worst pair for this instance lasts approximately two times more than the best ones. And its size is approximately two times higher than the size of best pairs.

Pair	Time spent	Proof Size	Comments
<MSS,F>	29313	33345	best in time and size
<MSS,ins>	29314	33345	second best in time
<MSS,dec>	29321	33345	third best in time
<CLS,rfo>	79890	75887	worst in time and size

Table D.4: H_6 results table.

D.2.3 Statman Family Results

The biggest Statman family instance solved within the time limit was Statman_{29}, whose size is 3338. The best results were obtained by MSS pairs; all of them solved the biggest solved instance. In Table D.5 we can see the results obtained by some selected pairs with the biggest solved instance. The worst pair for this instance lasts approximately three times more than the best ones. And the proof sizes of all pairs that solved the biggest instance are equal. In Table D.6 we show the results obtained with the biggest instance solved by all pairs (Statman_9, whose size is 318). It is easy to see that the differences in time (more that 600 times) and size (more that 150 times) are huge.

Pair	Time spent	Proof Size	Comments
<MSS,F>	5838	34366	best in time and size
<MSS,or>	5840	34366	second best in time
<MSS,xor>	5859	34366	third best in time
<LS,rev>	15452	34366	worst in time

Table D.5: Statman_{29} results table.

Pair	Time spent	Proof Size	Comments
<MSS,or>	37	1256	best in time and size
<MSS,xor>	37	1256	second best in time
<MSS,dec>	38	1256	third best in time
<CS,rfo>	22722	179991	worst in time
<CS,or>	21494	192983	worst in size

Table D.6: Statman_9 results table.

D.2.4 PHP Family Results

The best results for the PHP_6 instance (the biggest PHP instance solved within a 3-minute time limit) were obtained by the <MSS,or> and <MSS,T> pairs, with approximately 9 seconds and a proof size of 21273. No strategy could solve PHP_7 within the time limit. PHP_6 was solved by 43 pairs, 10 of each had MSS, SS, LS and SS as strategy. Only 3 pairs had CS as strategy and none had CLS as strategy. PHP_4 was the biggest instance that was solved by all pairs.

In Table D.7 we can see the results obtained by some selected pairs with the biggest solved instance (PHP_6, whose size is 455). The worst pair for this biggest instance lasts ten times more than the best ones. And its proof size is approximately 7 times bigger than the best sizes.

Pair	Time spent	Proof Size	Comments
<MSS,or>	9067	21273	best in time and size
<MSS,T>	9086	21273	second best in time
<SS,T>	9543	21273	third best in time
<LS,xor>	101522	148743	worst in time and size

Table D.7: PHP_6 results table.

We also tried PHP_7 (whose size is 692) with a 20-minute time limit. The results are presented in Table D.8. Surprisingly, the best pairs solved the biggest solved instance in less than three minutes. Again the best pair was <MSS,or>.

Pair	Time spent	Proof Size	Comments
<MSS,or>	150007	153988	best in time and size
<MSS,rfo>	154587	153988	second best in time and size
<SS,T>	162933	153988	third best in time and size
<BSS,T>	169517	153988	worst in time
<LS,or>	166666	181704	worst in size

Table D.8: PHP_7 results table.

In Table D.9 we can see the results obtained by two pairs with the biggest instance solved by all pairs (PHP_4, whose size is 155). The worst pair for this instance lasts 43 times more than the best ones. And its proof size is approximately 13 times bigger than the best sizes.

Pair	Time spent	Proof Size	Comments
<BSS,or>	70	720	one of the several best in time and size
<CS,F>	3024	9689	worst in time and size

Table D.9: PHP_4 results table.

D.2.5 U Family Results

The biggest U family instance solved within the time limit was U_{13} (whose size is 51). The best results were obtained by MSS pairs; all of them solved the biggest solved instance.

In Table D.10 we can see the results obtained by some selected pairs with the biggest solved instance. The worst pair for this instance lasts approximately 1.5 times more than the best ones. And the proof sizes of all pairs that solved the biggest instance are equal. In Table D.11 we show the results obtained with the biggest instance solved by all pairs (U_8, whose size is 31). It is easy to see how big are the differences in time (more than 400 times) and size (more than 18 times) between the best and worst pairs.

Pair	Time spent	Proof Size	Comments
<MSS,bi-impli>	33686	483158	best in time and size
<MSS,rfo>	33790	483158	second best in time
<MSS,T>	33793	483158	third best in time
<SS,rev>	49541	483158	worst in time

Table D.10: U_{13} results table.

Pair	Time spent	Proof Size	Comments
<MSS,inc>	314	9554	best in time and size
<MSS,F>	319	9554	second best in time
<MSS,ins>	319	9554	third best in time
<CLS,rev>	134411	174624	worst in time
<CLS,rfo>	127820	174880	worst in size

Table D.11: U_8 results table.

D.2.6 Square Tseitin Family Results

The biggest ST family instance solved within the time limit was ST_4, whose size is 80. The best time results were obtained by MSS pairs. Besides that, all of them solved the biggest solved instance. Only CLS pairs could not solve this instance.

In Table D.12 we can see the results obtained by some selected pairs with the biggest solved instance. The worst pair for this instance lasts approximately 3 times more than

the best ones. And its proof size is approximately 3 times biggest than the best pair size. In Table D.13 we show the results obtained with the biggest instance solved by all pairs (ST_3, whose size is 39). It is easy to see that the differences in time (more than 10 times) and size (approximately 4 times) between the best and worst pairs are not so big.

Pair	Time spent	Proof Size	Comments
<MSS,rev>	1087	13676	best in time
<MSS,inc>	1156	14661	second best in time
<CS,dec>	1454	9216	best in size
<CS,ins>	1503	12712	second best in size
<CS,rev>	3933	30690	worst in time and size

Table D.12: ST_4 results table.

Pair	Time spent	Proof Size	Comments
<BSS,T>	17	407	best in time
<BSS,F>	17	409	best in time
<BSS,ins>	17	410	best in time
<CS,dec>	22	283	best in size
<CLS,nfo>	188	1110	worst in time
<CLS,rev>	113	1123	worst in size

Table D.13: ST_3 results table.

D.2.7 Backjumping Family Results

As expected, the Backjumping Simple Strategy achieved the best results with B_PHP family instances. The biggest instance solved within the time limit was $B\_PHP^3_6$, whose size is 465; no strategy could solve $B\_PHP^3_7$ within the time limit. $B\_PHP^3_6$ was solved by 25 pairs. Of these, in 10 pairs the strategy was the Backjumping Simple Strategy. The other fifteen pairs had Simple Strategy, Learning Strategy and Memory Saver Strategy (5 each) as strategy. $B\_PHP^3_4$ was the biggest instance that was solved by all pairs.

In Table D.14 we can see the results obtained by some selected pairs with the biggest solved instance ($B\_PHP^3_6$). And in Table D.15 we present the results obtained by some selected pairs with the biggest instance solved by all pairs ($B\_PHP^3_4$, whose size is 165).

It is interesting to compare these results: the worst pair that solved B_PHP$_4^3$ took more time than the best pair for B_PHP$_6^3$ and produced a bigger proof.

Pair	Time spent	Proof Size	Comments
<BSS,or>	11006	21291	best in time and size
<BSS,T>	11116	21296	second best in time and size
<BSS,rfo>	11705	21296	third best in time and size
<BSS,dec>	21425	39483	fourth best in time and size
<BSS,bi-impli>	105920	110306	worst in time
<LS,nfo>	99591	186361	worst in size

Table D.14: B_PHP$_6^3$ results table.

Pair	Time spent	Proof Size	Comments
<BSS,T>	91	743	best in time
<BSS,or>	93	738	best in size
<CS,and>	3615	9702	an intermediate result
<CS,inc>	11870	26683	worst in time and size

Table D.15: B_PHP$_4^3$ results table.

It is also interesting to compare B_PHP with PHP results: the results obtained by Backjumping Simple Strategy pairs with B_PHP instances were only slightly worst than those obtained with PHP instances. But the results obtained by other strategies were much worst (for example, the time spent by <MSS,T> with B_PHP was more than four times the time spent by the same pair with PHP).

D.2.8 Random SAT Family Results

CPL strategies were able to give an answer in the time limit only for random K-SAT (cnf) instances 1 to 3 (see Table D.1). The results are exhibited in Table D.16. These results are much worse than those obtained by state-of-the-art SAT solvers.

Instance	Pair	Time spent	Proof size	Comments
cnf_1	<MSS,inc>	444	3778	best in time
cnf_1	<CS,inc>	969	3698	best in size
cnf_1	<CS,rfo>	1684	4654	worst in size and time
cnf_2	<MSS,nfo>	8252	19438	best in size and time
cnf_2	<CS,rfo>	53208	29428	worst in time and size
cnf_3	<MSS,rev>	8540	19243	best in time
cnf_3	<CS,nfo>	18308	19155	best in size
cnf_3	<CS,rfo>	73356	31204	worst in time
cnf_3	<CS,bi-impli>	60204	31988	worst in size

Table D.16: Random K-SAT results table.

D.2.9 First family results

mbC

The biggest first family (see Section D.1.2) instance solved within the time limit was Φ^1_{90}, whose size is 993. And this instance was solved by all pairs. In Table D.17 we can see the results obtained by some selected pairs with the biggest solved instance. The worst pair in time for this instance lasts approximately 1.4 times more than the best one. And the worst pair in size (which is the best in time) has almost the same size of the best pairs.

Pair	Time spent	Proof Size	Comments
<MBCES,dec>	66955	58681	best in time and worst in size
<MBCSS,ins>	67258	58591	second best in time
<MBCSS,inc>	78871	58416	one of the best in size
<MBCES,nfo>	97387	58416	worst in time

Table D.17: **mbC** Φ^1_{90} results table.

mCi

The biggest instance solved by **mCi** pairs with first family (see Section D.1.2) instances within the time limit was Φ^1_{90}. And this instance was solved by all pairs. In Table D.18 we can see the results obtained by some selected pairs with the biggest solved instance.

The worst pair in time for this instance lasts approximately 1.4 times than the best one. And the worst pair in size (which is the best in size) has almost the same size of the best pairs.

Pair	Time spent	Proof Size	Comments
`<MCIES,dec>`	67042	58681	best in time and worst in size
`<MCISS,ins>`	67145	58591	second best in time
`<MCISS,and>`	67199	58591	third best in time
`<MCIES,nfo>`	96930	58416	worst in time and one of the best in size

Table D.18: **mCi** Φ^1_{90} results table.

CPL

We also posed instances of the first family to **CPL** pairs. The The biggest instance solved within the time limit was Φ^1_{120} (size 1083 for **CPL**[8]). And Φ^1_{90} (size 813 for **CPL**) was the biggest instance solved within the time limit by all **CPL** pairs. Φ^1_{120} was solved by `<MSS,bi-impli>` (the best pair for this instance) in 42008 milliseconds and with a proof size of 51550. The best pair for Φ^1_{90} was `<MSS,rfo>`, which solved this instance in 12980 milliseconds and produced a proof with a size of 29209. These results make it very clear that this family is much easier for **CPL** than for **mbC** and **mCi**. The same observation applies to all other **LFI** families submitted to **CPL** pairs.

D.2.10 Second family results

mbC

The biggest second family instance solved by **mbC** pairs within the time limit was Φ^2_{14}, whose size is 472. And the biggest instance solved by all pairs was Φ^2_{10} (whose size is 278). In Table D.19 we can see the results obtained by the only two pairs that solved the biggest solved instance. The interesting fact is that both use the same sorter: Reverse Insertion Order. In Table D.20 we show some results obtained with the biggest instance solved by all pairs (Φ^2_{10}). The difference in time between the best and worst pairs is more

[8]If we define $\circ X \stackrel{\text{def}}{=} \top$ and $\bullet X \stackrel{\text{def}}{=} \bot$, as we have shown in Section D.1.2, **CPL** versions of **LFI** problems have a smaller size.

than 12 times. And the difference in size between the best and worst pairs is more than 15 times.

Pair	Time spent	Proof Size	Comments
<MBCSS,rev>	25213	57827	best in time
<MBCES,rev>	26297	57827	second best in time

Table D.19: **mbC** Φ^2_{14} results table.

Pair	Time spent	Proof Size	Comments
<MBCES,rev>	2397	9769	best in time
<MBCSS,rev>	2401	9769	second best in time
<MBCSS,inc>	10984	26565	third best in time
<MBCES,and>	30573	116037	worst in time
<MBCSS,imp>	23767	149504	worst in size
<MBCES,imp>	24401	149504	worst in size

Table D.20: **mbC** Φ^2_{10} results table.

mCi

The biggest second family instance solved by **mCi** pairs within the time limit was Φ^2_{17}, whose size is 649. And the biggest instance solved by all pairs was Φ^2_{11} (whose size is 322). In Table D.21 we can see the results obtained by the only two pairs that solved the biggest solved instance. The interesting fact is that both use the same sorter: Reverse Insertion Order. In Table D.22 we show some results obtained with the biggest instance solved by all pairs (Φ^2_{11}). The difference in time between the best and worst pairs is more than 19 times. And the difference in size between the best and worst pairs is more than 22 times.

Pair	Time spent	Proof Size	Comments
<MCISS,rev>	112586	181333	best in time
<MCIES,rev>	116751	181333	second best in time

Table D.21: **mCi** Φ^2_{17} results table.

Pair	Time spent	Proof Size	Comments
<MCISS,rev>	4376	15785	best in time and size
<MCIES,rev>	4550	15785	second best in time
<MCISS,inc>	27681	52226	third best in time
<MCIES,and>	86858	285642	worst in time
<MCISS,imp>	65654	360515	worst in size
<MCIES,imp>	66834	360515	worst in size

Table D.22: **mCi** Φ^2_{11} results table.

CPL

We also posed instances of the second family to **CPL** pairs. The biggest instance solved within the time limit was Φ^2_{20} (whose size is 623), and it was solved by all pairs. It was solved by <MSS,T> (the best pair for this instance) in 2123 milliseconds and with a proof size of 5784.

D.2.11 Third family results

mbC

The biggest instance solved by some **mbC** pairs with third family instances within the time limit was Φ^3_{14}, whose size is 509. And the biggest instance solved by all pairs was Φ^3_{11} (size 353). In Table D.23 we can see the results obtained by the only two pairs that solved the biggest solved instance. The interesting fact is that both use the same sorter: Reverse Insertion Order. In Table D.24 we show some results obtained with the biggest instance solved by all pairs. The difference in time between the best and worst pairs is more than 10 times. And the difference in size between the best and worst pairs is more than 8 times.

Pair	Time spent	Proof Size	Comments
<MBCSS,rev>	96055	163470	best in time and size
<MBCES,rev>	102098	163470	second best in time

Table D.23: **mbC** Φ^3_{14} results table.

Pair	Time spent	Proof Size	Comments
<MBCSS,rev>	7358	21915	best in time and size
<MBCES,rev>	7798	21915	second best in time
<MBCSS,dec>	10715	27942	third best in time
<MBCES,ins>	75160	115467	worst in time
<MBCSS,imp>	49971	192386	worst in size
<MBCES,imp>	52308	192386	worst in size

Table D.24: **mbC** Φ^3_{11} results table.

mCi

The biggest instance solved by **mCi** pairs with third family instances. within the time limit was Φ^3_{12} (size 402). And the biggest instance solved by all pairs was Φ^3_{10} (size 307).

In Table D.25 we can see the results obtained by some pairs that solved the biggest solved instance. The interesting fact is that the two best pairs use the same sorter: Reverse Insertion Order. The difference in time between the best and worst pairs is approximately two times. And the difference in size between the best and worst pairs is less than 1.5 times.

In Table D.26 we show some results obtained with the biggest instance solved by all pairs. The difference in time between the best and worst pairs is more than 7 times. And the difference in size between the best and worst pairs is approximately 8 times.

Pair	Time spent	Proof Size	Comments
<MCISS,rev>	16925	42590	best in time and in size
<MCIES,rev>	18063	42590	second best in time
<MCISS,or>	30769	54903	worst in size
<MCIES,or>	32555	54903	worst in time and in size

Table D.25: **mCi** Φ^3_{14} results table.

CPL

We also posed instances of the second family to **CPL** pairs. The biggest instance solved within the time limit was Φ^3_{20} (size 672), and it was solved by all pairs. It was solved by <MSS,F> (the best pair for this instance) in 2513 milliseconds and with a proof size of 6311.

Pair	Time spent	Proof Size	Comments
<MCISS,rev>	3103	11786	best in time and size
<MCIES,rev>	3338	11786	second best in time
<MCIES,T>	28029	52540	worst in time
<MCISS,imp>	19634	87029	worst in size
<MCIES,imp>	20557	87029	worst in size

Table D.26: **mCi** Φ^3_{10} results table.

D.2.12 Fourth family results

mbC

The biggest instance solved by **mbC** pairs with fourth family instances within the time limit was Φ^4_{90} (size 1079). And the biggest instance solved by all pairs was Φ^4_{80} (size 959).

In Table D.27 we can see the results obtained by some pairs that solved the biggest solved instance. In Table D.28 we show some results obtained with the biggest instance solved by all pairs. In both cases, the best pairs' results are only slightly better than the worst pair results. The interesting fact is that the two best pairs use the same sorter: Reverse String Order.

Pair	Time spent	Proof Size	Comments
<MBCES,rfo>	55417	40142	best in time
<MBCSS,rfo>	55486	40140	second best in time and best in size
<MBCSS,dec>	74809	51202	worst in time and in size

Table D.27: **mbC** Φ^4_{90} results table.

Pair	Time spent	Proof Size	Comments
<MBCSS,rfo>	36033	31865	best in time and in size
<MBCES,rfo>	36315	31869	second best in time
<MBCSS,imp>	57975	40480	worst in time
<MBCSS,dec>	48496	40712	worst in size

Table D.28: **mbC** Φ^4_{80} results table.

mCi

The biggest instance solved by **mCi** pairs with fourth family instances within the time limit was Φ^4_{90}. And the biggest instance solved by all pairs was Φ^4_{80}.

In Table D.29 we can see the results obtained by some pairs that solved the biggest solved instance. The interesting fact is that the two best use the same sorter: Reverse String Order. In Table D.30 we show some results obtained with the biggest instance solved by all pairs. In both cases, the best pairs' results are only slightly better than the worst pair results. The interesting fact is that the two best pairs use the same sorter: Reverse String Order.

Pair	Time spent	Proof Size	Comments
<MCISS,rfo>	55026	40140	best in time and size
<MCIES,rfo>	55542	40142	second best in time
<MCISS,dec>	75296	51202	worst in time and size

Table D.29: **mCi** Φ^4_{90} results table.

Pair	Time spent	Proof Size	Comments
<MCISS,rfo>	35624	31865	best in time and size
<MCIES,rfo>	36177	31869	second best in time
<MCISS,imp>	58403	40480	worst in time
<MCISS,dec>	48500	40712	worst in size

Table D.30: **mCi** Φ^4_{80} results table.

CPL

We also posed instances of the fourth family to **CPL** pairs. The biggest instance solved within the time limit was Φ^4_{100} (size 1000). And Φ^4_{90} was the biggest instance solved within the time limit by all **CPL** pairs. Φ^4_{100} was solved by <MSS,F> (the best pair for this instance) in 39236 milliseconds and with a proof size of 41006. The best pair for Φ^4_{90} was <MSS,rfo>, which solved this instance in 26162 milliseconds and produced a proof with a size of 33306.

D.2.13 Seventh family results

mbC

The biggest instance solved by **mbC** pairs with seventh family instances within the time limit was Φ^7_{20} (size 336). And the biggest instance solved by all pairs was Φ^7_7 (size 115).

In Table D.31 we can see the results obtained by some pairs that solved the biggest solved instance. The best pair in time is more than 45 times faster than the worst pair. And the proof produced by the worst pair is more than 35 times bigger than the proof produced by the pair with the smallest proof.

In Table D.32 we show some results obtained with the biggest instance solved by all pairs. Here the best pair in time is more than 1200 times faster than the worst pair. And the proof produced by the worst pair is more than 480 times bigger than the proof produced by the pair with the smallest proof.

Pair	Time spent	Proof Size	Comments
<MBCES,F>	845	3524	best in time
<MBCES,and>	847	3524	second best in time
<MBCSS,nfo>	875	3504	best in size
<MBCSS,F>	951	3504	best in size
<MBCSS,and>	957	3504	best in size
<MBCES,rev>	40319	105872	worst in time and size

Table D.31: **mbC** Φ^7_{20} results table.

Pair	Time spent	Proof Size	Comments
<MBCES,F>	30	586	best in time
<MBCES,nfo>	32	586	second best in time
<MBCES,and>	32	586	second best in time
<MBCSS,and>	33	579	best in size
<MBCSS,nfo>	34	579	best in size
<MBCSS,F>	35	579	best in size
<MBCSS,rfo>	38006	279070	worst in time and size

Table D.32: **mbC** Φ^7_7 results table.

mCi

The biggest instance solved by **mCi** pairs with seventh family instances within the time limit was Φ^7_{20}. And the biggest instance solved by all pairs was Φ^7_8 (size 132).

In Table D.33 we can see the results obtained by some pairs that solved the biggest solved instance. The best pair in time is more than 45 times faster than the worst pair. And the proof produced by the worst pair is more than 30 times bigger than the proof produced by the pair with the smallest proof.

In Table D.34 we show some results obtained with the biggest instance solved by all pairs. Here the best pair in time is more than 2400 times faster than the worst pair. And the proof produced by the worst pair is more than 870 times bigger than the proof produced by the pair with the smallest proof.

Pair	Time spent	Proof Size	Comments
<MCIES,and>	849	3524	best in time
<MCISS,nfo>	883	3504	best in size
<MCISS,F>	963	3504	best in size
<MCISS,and>	964	3504	best in size
<MCIES,rev>	40445	105872	worst in time and size

Table D.33: **mCi** Φ^7_{20} results table.

Pair	Time spent	Proof Size	Comments
<MCIES,F>	45	728	best in time
<MCISS,rfo>	7209	720	best in size
<MCISS,nfo>	49	720	best in size
<MCISS,F>	51	720	best in size
<MCISS,and>	51	720	best in size
<MCISS,rfo>	110192	629303	worst in time and size

Table D.34: **mCi** Φ^7_8 results table.

CPL

We also posed instances of the second family to **CPL** pairs. The biggest instance solved within the time limit was Φ^7_{25} (size 272), and it was solved by all pairs. It was

solved by <MSS,or> (the best pair for this instance) in 317 milliseconds and with a proof size of 3246.

D.2.14 Eighth family results

mbC

As all eighth family instances are not valid for **mbC**, the biggest instance we submitted for **mbC** pairs was Φ^8_{10} (size 186). This was found to be not valid by all pairs. All proofs took approximately the same time and have almost the same size. The best result for the biggest instance was obtained by the <MBCSS,dec> pair which took 148 milliseconds and produced a refutation of size 1257.

mCi

The biggest instance solved by **mCi** pairs with eighth family instances within the time limit was Φ^8_{50} (size 946). And the biggest instance solved by all pairs was Φ^8_7 (size 129).

In Table D.35 we can see the results obtained by some pairs that solved the biggest solved instance. The time and size of best and worst pairs are almost the same.

In Table D.36 we show some results obtained with the biggest instance solved by all pairs. Here the best pair in time is more than 2000 times faster than the worst pair. And the proof produced by the worst pair is more than 530 times bigger than the proof produced by the pair with the smallest proof.

Pair	Time spent	Proof Size	Comments
<MCISS,or>	18430	25407	best in time and in size
<MCISS,dec>	18442	25407	second best in time
<MCIES,or>	18635	25507	worst in size
<MCIES,dec>	18969	25507	worst in time and size

Table D.35: **mCi** Φ^8_{50} results table.

Pair	Time spent	Proof Size	Comments
<MCISS,or>	39	682	best in time and in size
<MCISS,dec>	40	682	second best in time and best in size
<MCIES,dec>	41	696	third best in time
<MCIES,and>	78662	365630	worst in time and one of the worst in size

Table D.36: **mCi** Φ_7^8 results table.

CPL

We also posed instances of the eighth family to **CPL** pairs. The biggest instance we submitted was Φ_{10}^8 (size 166), and it was solved by all pairs. It was solved by <MSS,rev> (the best pair for this instance) in 17 milliseconds and with a proof size of 1006.

D.2.15 Ninth family results

mbC

As all ninth family instances are not valid for **mbC**, the biggest instance we submitted for **mbC** pairs was Φ_{25}^9 (size 397). This was found to be not valid by all pairs. All proofs took approximately the same time and have almost the same size. The best result in time for the biggest instance was obtained by the <MBCSS,nfo> pair which took 1807 milliseconds and produced a refutation of size 6219. And the best result in size for the biggest instance was obtained by the <MBCES,or> pair which took 1897 milliseconds and produced a refutation of size 5740.

mCi

The biggest instance solved by **mCi** pairs with ninth family instances within the time limit was Φ_{75}^9 (size 1197). And the biggest instance solved by all pairs was Φ_{40}^9 (size 637).

In Table D.37 we can see the results obtained by some pairs that solved the biggest solved instance. The size of best and worst pairs are almost the same, but the time taken to finish the worst pair was approximately 1.6 higher than the time taken by the best pair.

In Table D.38 we show some results obtained with the biggest instance solved by all

pairs. Here the best pair in time is more than 9 times faster than the worst pair. And the proof produced by the worst pair is approximately 1.8 times bigger than the proof produced by the pair with the smallest proof.

Pair	Time spent	Proof Size	Comments
<MCIES,inc>	59644	47691	best in time and size
<MCIES,imp>	59818	47691	second best in time
<MCISS,rev>	100869	48142	worst in time and size

Table D.37: **mCi** Φ^9_{75} results table.

Pair	Time spent	Proof Size	Comments
<MCIES,imp>	7547	14231	best in time and size
<MCIES,inc>	7549	14231	second best in time
<MCIES,rev>	68886	25534	worst in time and size

Table D.38: **mCi** Φ^9_{40} results table.

CPL

We also posed instances of the second family to **CPL** pairs. The biggest instance we submitted was Φ^9_{25} (size 272), and it was solved by all pairs. It was solved by <MSS,T> (the best pair for this instance) in 209 milliseconds and with a proof size of 3246.

Apêndice E

Conclusion

E.1 Test Conclusions

Let us discuss the results presented in Section D.2. First let us state that no **KEMS** prover configuration obtained incorrect results with the evaluation problem instances. Some of these instances, such as the ones from PHP and ST families, were very difficult to prove. That is, the proof search procedure for some very small instances did not finish for any strategy-sorter pair. Some other families were difficult only for some strategy-sorter pairs.

For **CPL** tests, a pair with Memory Saver Strategy (MSS) as the strategy achieved the best time results for all problem families except for Γ and B_PHP families (see Table E.1). The best pair for Γ had Learning Strategy (LS) as the strategy, but several other pairs had equally good results. And the best pair for B_PHP used, as expected, Backjumping Simple Strategy (BSS). Therefore, in general our first option when choosing a strategy should be Memory Saver Strategy.

The results for sorters were not as conclusive. We can see in Table E.1 that almost all sorters were present in a best pair for some family. Thus, the choice of the best sorter to run a problem with will depend on the problem being tackled. If we already know the the problem belongs to a given family, we can choose a sorter that worked well for that family. Otherwise, we cannot suggest any sorter.

Now let us analyse **mbC** and **mCi** tests. Both **mbC** strategies, **mbC** Simple Strategy

(MBCSS) and **mbC** Extended Strategy (MBCES), achieved equivalent results with valid problems (see Table E.2)[1]. MBCSS was slightly better with non-valid problem families. And for the other implemented **LFI**, **mCi** Simple Strategy (MCISS) and **mCi** Extended Strategy (MCIES) also achieved comparable results (see Table E.3). Therefore, if we are going to run a problem that does not belong to one of the families we have used for evaluating **KEMS**, we cannot indicate a specific strategy to be tried first. Nor any sorter.

In all **LFI** tests the sorter in the best pairs varied according to the problem family. And it was interesting to notice that for some families the sorter choice was almost as important as the strategy choice. For instance, with second family instances only one sorter was able to prove the biggest problem solved, independently of the strategy.

These results we obtained by **KEMS** with the **LFI** families are the first benchmark results for these families. All results obtained with the three logical systems can be compared with other provers for these logics and are available at [92].

Bigger instance solved	Problem size	Best time pair	Best size pair
Γ_{360}	3606	<LS,inc>	<LS,inc>
H_6	959	<MSS,F>	<MSS,F>
Statman_{29}	3338	<MSS,F>	<MSS,F>
PHP_7	692	<MSS,or>	<MSS,or>
U_{13}	51	<MSS,bi-impli>	<MSS,bi-impli>
ST_4	80	<MSS,rev>	<CS,dec>
B_PHP_6^3	465	<BSS,or>	<BSS,or>
cnf_2	526	<MSS,nfo>	<MSS,nfo>
Φ_{130}^1	1173	<MSS,F>	<MSS,rfo>
Φ_{20}^2	623	<MSS,T>	<CS,rfo>
Φ_{20}^3	672	<MSS,F>	<CS,or>
Φ_{100}^4	1000	<MSS,F>	<MSS,F>
Φ_{20}^7	296	<MSS,xor>	<MSS,xor>
Φ_{10}^8	166	<MSS,and>	<MSS,and>
Φ_{25}^9	272	<MSS,T>	<MSS,T>

Table E.1: Best **CPL** strategy-sorter pairs.

[1]Actually this is not clear in the table, where we show only one of the best when more than one pair tied, but can be seen when we examine the evaluation files produced by **KEMS**.

Bigger instance solved	Problem size	Best time pair	Best size pair
Φ^1_{90}	993	<MBCES,dec>	<MBCSS,inc>
Φ^2_{14}	472	<MBCSS,rev>	<MBCSS,rev>
Φ^3_{14}	509	<MBCSS,rev>	<MBCSS,rev>
Φ^4_{90}	1079	<MBCES,rfo>	<MBCSS,rfo>
Φ^7_{20}	336	<MBCES,F>	<MBCSS,nfo>

Table E.2: Best **mbC** strategy-sorter pairs.

Bigger instance solved	Problem size	Best time pair	Best size pair
Φ^1_{90}	993	<MCIES,dec>	<MCISS,inc>
Φ^2_{17}	649	<MCISS,rev>	<MCISS,rev>
Φ^3_{12}	402	<MCISS,rev>	<MCISS,rev>
Φ^4_{90}	1079	<MCISS,rfo>	<MCISS,rfo>
Φ^7_{20}	336	<MCIES,and>	<MCISS,nfo>
Φ^8_{50}	946	<MCISS,or>	<MCISS,or>
Φ^9_{75}	1197	<MCIES,inc>	<MCIES,inc>

Table E.3: Best **mCi** strategy-sorter pairs.

E.2 Thesis Conclusions and Contributions

We succeeded in developing a multi-strategy theorem prover where we can vary the strategy without modifying the core of the implementation. **KEMS** allows us to describe several proof strategies for the same logical system, and to implement different logical systems.

We list below some of the contributions of this work:

- an analytic, correct and complete **KE** system for **mbC**;
- a correct and complete **KE** system for **mCi**;
- a multi-strategy prover with the following characteristics:
  - accepts problems in three logical systems: **CPL**, **mbC** and **mCi**[2];
  - has 6 implemented strategies for **CPL**, 2 for **mbC** and 2 for **mCi**;
  - has 13 sorters to be used alongside with the strategies;
  - implements simplification rules of **CPL**;

[2]We know of no other prover for **mbC** and **mCi**.

  - provides a proof viewer with a graphical user interface;
  - it is open source and available at [92].

- benchmark results obtained by **KEMS** comparing its **CPL** strategies with several problem families;
- seven problem families designed to evaluate **LFI** provers;
- the first benchmark results for **LFI** families obtained with several **KEMS mbC**, **mCi** and **CPL** strategies.

E.3 Future Works

On the logical side, it would be useful to have a general procedure for automatically generating correct and complete **KE** systems for **LFI**s and other logical systems, similar to the procedure for generating tableau systems presented in [11]. This could help us to extend **KEMS** to other logical systems. Marcelo Coniglio (one of the authors of [11]) informed us that they have already thought of adapting their method that obtains what we called here $\mathbf{C^3M}$ systems (which for **LFI**s are a kind of mixture of **AT** and **KE**) to be able to produce **KE** systems.

On the implementation side, from the results we have obtained[3], we can see that it would be very useful to have an adaptive strategy that changes its behavior according to features of the problem presented to it. This strategy can behave as other implemented strategies and it will be able to vary its actions not only for different problems but also for different subproblems of the same problem given as input.

Next, we plan to implement new strategies making heavy use of Aspect Orientation. The objective is to easily mix strategy features to produce new strategies. For instance, we have a Backjumping Simple Strategy for **CPL** but not for **LFI**s. Without using aspect orientation, in **KEMS** we would have to create a new class for implementing a Backjumping Simple Strategy for **LFI**s. If we use aspects we could have strategies for

[3]That is, from the conclusion that there is no strategy-sorter pair which is the best for every problem family.

CPL, **mbC**, and **mCi** and we would be able to obtain Backjumping versions of them by implementing a Backjumping Aspect that changes the behavior of these strategies.

An obvious **KEMS** extension is to develop strategies for $\mathbf{C}_1$, the simplest in da Costa's C_n hierarchy of paraconsistent logics [27]. To achieve this, we first have to provide a **KE** system for $\mathbf{C}_1$. We have already developed the rules of this system and soon we will publish them and prove that this $\mathbf{C}_1$ **KE** system is correct and complete. Another possible future work is to develop strategies for the other **LFI**s presented in [18]. Several **LFI**s are built there by adding/removing axioms to other **LFI**s. It does not seem to be difficult to implement a module in **KEMS** where we could build a particular **LFI** by choosing some features and then adapt some general predefined **LFI** strategies for this specific system. To evaluate strategies for these **LFI**s we would have to design new problem families which are valid in these logics.

Another extension would be to implement some restricted simplification rules for **LFI**s to obtain more efficient strategies. **KEMS** could also be improved by developing strategies for other propositional logics that have **KE** systems. It would be interesting to extend **KEMS** to first-order logics and then use some of the ideas described in [72] to vary strategies.

Logical systems for approximate reasoning are presented in [48, 49, 50], and in [47] the relationship between some of these systems and paraconsistency is discussed. It would be interesting to try to implement these logical systems in **KEMS** to evaluate how easy is to adapt **KEMS** to other logics. We know that it is not so easy to implement different logical systems in **KEMS**, therefore in the future we plan to document how this can be done.

Finally, we plan to implement in **KEMS** a probabilistic strategy such as GSAT [105]. This strategy, given a valid problem, would not always find a **KE** proof. However, it would be able in most cases to find fast valuations for satisfiable problems. Yet another idea is to allow the dynamic building of strategies either from some predefined options or from some strategy-definition language, in a way which is similar to what is done in [37].

Apêndice F

Brief User Manual

Here we present a simple user manual. Firstly we show its installation procedure. After that we describe some scenarios for using **KEMS**, where we will see its functionalities.

F.1 Installation

As **KEMS** was implemented in Java, it can be run on any platform for which there is a Java Runtime Environment (JRE), version 5.0. Therefore the only requirement is to have a JRE installed on your computer. The JRE is available for several operating systems (see [80] for more details on how to install a JRE for a specific system).

The following instructions are specific for the Linux platform, but it is easy to adapt them for Windows and other operating systems. First we assume you have placed the `kems.zip` file (available at [92]) in the directory `/home`. Unzip the file with the

```
unzip kems.zip
```

command. The `kems.export` directory should be created. Henceforth we refer to

```
/home/kems.export
```

as `KEMS.HOME`.

If the JRE is in the `PATH`[1] and you are in `KEMS.HOME`, then you can issue the

[1]The JRE installation should put the JRE directory (`$JAVA_HOME/bin`) in the `PATH`, but if that is not the case you can see how to set the `JAVA_HOME` and PATH environment variables in [80].

```
java -jar kems.jar
```

command. This is going to start **KEMS** graphical interface. If you want to determine the amount of heap memory to be used by **KEMS**[2], you can issue the

```
java -Xms size -Xmx size -jar kems.jar
```

command. '`-Xmx`' and '`-Xms`' are `java` command options. The `-Xmx size` option sets the maximum heap size to size. Kilobytes are indicated by the letters k or K and megabytes by m or M. Similarly, `-Xms size` determines the initial heap size. For instance,

```
java -Xms200m -Xmx800m -jar kems.jar
```

sets the minimum heap size to 200 megabytes and the maximum heap size to 800 megabytes.

F.2 Scenarios

Here we will present some scenarios for using **KEMS**. Let us enumerate the main ones:

1. configure the prover;
2. run a problem (that is in a file) with one prover configuration;
3. edit a problem and run it with one prover configuration;
4. run a sequence of problems with a list of prover configurations.

F.2.1 Configuring the Prover

Several scenarios require the prover to be previously configured. To configure the prover we need to:

1. open the Prover Configurator (see Figure F.2) by choosing on main window (see Figure F.1) menu bar the `Configure` option and then clicking on the `Prover` option;

[2]This can be necessary to run some difficult problems.

2. choose a logical system;

3. choose a strategy;

4. set the number of times to run each problem;

5. set the maximum number of minutes to run each problem (with a prover configuration);

6. choose the analyzer name;

7. choose a sorter;

8. mark/unmark the 'save formula origin', 'discard closed branches', and 'save discarded branches' options.

Figure F.1: **KEMS** main window.

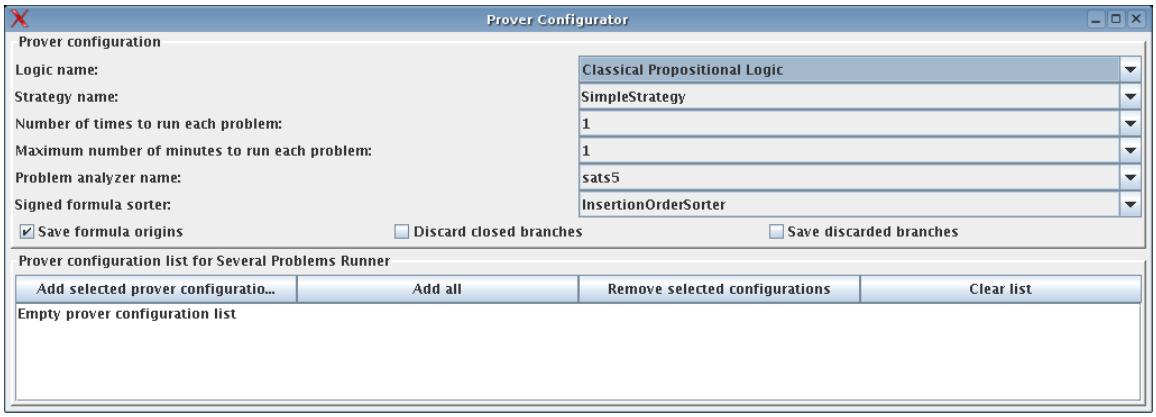


Figure F.2: Prover Configurator window.

F.2.2 Choosing and Running a Problem

To run the first scenario (run a problem that is in a file with one prover configuration) you must first configure the prover. After that you must choose on main window menu bar the `Problem` option and then click on the `Instance - Choose, Run and View Proof of a Problem Instance` option. Next you perform the following actions:

1. choose a problem file (you can browse directories on the `Open` window until you find the desired file);
2. run the problem (by clicking on the `Run` button);
3. browse the proof.

Later we will describe how the user can browse the proof.

F.2.3 Editing and Running a Problem

The second scenario, edit a problem and run it with one prover configuration, comprises the following:

1. open the Problem Editor (see Figure F.3) by choosing main window menu bar `Problem` option and after that clicking on `Editor`;
2. type the problem on (or load it from a file to) the Problem Editor window;
3. configure the prover;
4. run the problem (choose Problem Editor menu bar `Run` option and after that click on `Run this problem`);
5. browse the proof.

The user can enter a problem either using SATLIB CNF format [101] or in an extension of the format used in [38]. To enter a problem (either in the Problem Editor window or when editing a problem file in any text editor) using the extension of the format of [38], one must provide a list of signed formulas. Each line can contain at most one s-formula.

Comment lines start with a `#`. A signed formula is a sign followed by a formula. The allowed signs are `T` (true) and `F` (false).

Formulas can be atomic or composite. An atomic formula is a string of letters and numbers initiated by a letter. Composite formulas have a connective and zero, one or two subformulas (which are themselves formulas). The notation for composite formulas is the common infix notation. For zeroary connectives, a formula is its `Connective`. For unary connectives, `Connective Formula`. And for binary connectives, `Formula Connective Formula`. Parentheses are used when necessary to establish precedence.

The allowed connectives are:

zeroary - `TOP` and `BOTTOM`;

unary - `!` (not), `@` (consistency) and `*` (inconsistency);

binary - `&` (and), `|` (or), `->` (implication), `<=>` (bi-implication) and `+` (exclusive or).

The following is an example of a **CPL** problem:

```
T  A <=> (B&!C)
T  TOP-> !(A|D)
F  BOTTOM | (!D)
```

And here we can see an example of a **LFI** problem:

```
T  A <=> @(B&!*C)
T  TOP-> !(A|@D)
F  BOTTOM | @(!D)
```

In **mbC** and **mCi**, '`*A`' is translated into '`!@A`'. The previous problem can also be submitted to a **CPL** strategy: '`@A`' formulas will be translated into '`TOP`' and '`*A`' into '`BOTTOM`'.

F.2.4 Running a Problem Sequence

For the third scenario, running a sequence of problems with a list of prover configurations, we have the following steps:

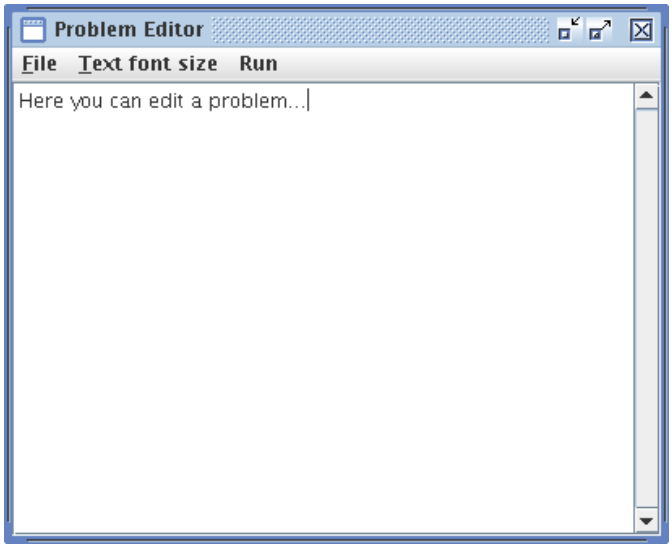


Figure F.3: Problem Editor.

1. configure the prover (here it is sometimes useful to create a list of prover configurations in the bottom part of the Prover Configurator window[3]);

2. open the Several Problems Runner window (see Figure F.4) by choosing main window menu bar `Problem` option and after that clicking on `Several - Choose and Run Several Problems`;

3. choose problem files (see below) one or more times;

4. run the problem sequence (choose Several Problems Runner menu bar `Run` option and after that click on `Run problems`);

5. wait until **KEMS** finishes all problems with all selected prover configurations[4].

The results window show some extra information about the proof such as proof size, time spent, proof tree height and problem size.

To choose one or more problem files you must:

1. click on the `Choose one or more problem instances` button;

[3]For other scenarios only the currently selected prover configuration (PC) is used — the list of PCs is not taken into consideration.

[4]You can also choose Several Problems Runner menu bar `Results window` option and after that click on `Show current results` option to show a partial results window (more recent results appear on the top) that will be updated as new results are obtained.

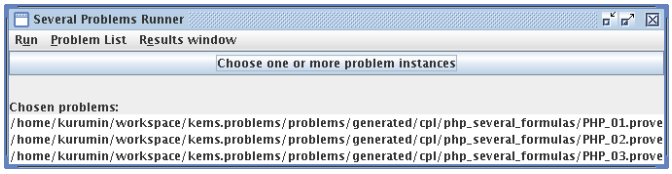


Figure F.4: Several Problems Runner window.

2. browse directories on the `Open` window until you find the desired files. If you hold the `ctrl` key you can choose more than one file;

3. click on the `Choose` button to add the selected files to the problem list.

F.2.5 Browsing a Proof

After a proof is obtained, a window such as the one in Figure F.5 is opened. On the left side there is an interactive proof viewer. On the right side we have a graphical proof viewer.

Interactive Proof Viewer

The interactive proof viewer (IPV) allows the user to see one branch at a time. The IPV shows a list of signed formulas. The currently selected branch is highlighted in the graphical proof viewer and identified in a button on the top of this list. By clicking on this button it one can see more information about the branch.

The window that is opened to show more information about the currently selected branch displays the prover configuration used to run the problem, a valuation (when the proof tree is open), and statistics about the proof.

If the user clicks with the left mouse button on one signed formula button, that s-formula is highlighted as well as all other s-formulas that gave origin to that s-formula. If the user clicks with the right mouse button on one signed formula button that s-formula's immediate origin (rule, major premise and minor premise) is shown.

Whenever there is a (PB) rule application, instead of one s-formula button, two s-formula buttons appear on one line: one for each (PB) s-formula. One of the two s-formula buttons is highlighted – the one which is in current branch – and the other is not

highlighted. If the user clicks on the unselected s-formula button, the IPV then shows the leftmost branch that includes that s-formula.

Whenever IPV shows a closed branch, a button that shows an '×' symbol is included as the last button of the list of s-formula buttons of that branch.

Graphical Proof Viewer

The graphical proof viewer (GPV) starts by showing a graphical representation of the proof. Initially, circles are presented in the place of s-formulas. This graphic gives us an idea of the proof form, how many applications of (PB) were necessary, how many s-formulas were included in the proof, and so on. The '×', '■' and '□' symbols that appear in the end of every branch denotate, respectively, that a branch is either 'closed', 'open and completed', or 'open and not completed'.

If the user clicks on GPV's area with the right mouse button a menu will appear allowing the user to set or unset some options and to perform one action:

- if the 'show circles' option is set, GPV shows circles as elements of tableau tree nodes. If it is unset, GPV shows s-formulas;
- if the 'show circles' option is unset and the 'show numbers' option is set, a unique sequential number is assigned to each s-formula (and show at the right of the s-formula);
- if the 'show circles' option is unset, the 'show sign marking used formulas' option is set, and the tableau is closed, a '*' sign is assigned to each s-formula effectively used to close the tableau (and displayed at the right of the s-formula);
- if the 'change parameters' action is selected, a window with some GPV and IPV parameters is exhibited. If the user changes some parameter(s) and clicks the 'Update' button, both viewers are refreshed with the new values for the changed parameters.

After setting parameters the user can scroll the proof using the horizontal and vertical scroll bars (that appear in GPV depending on GPV's width and height parameter values). It would be interesting if these parameters could automatically adjust themselves

according to the proof object and other GPV parameters, but that does not seem to be a trivial task and was not implemented in **KEMS** current version.

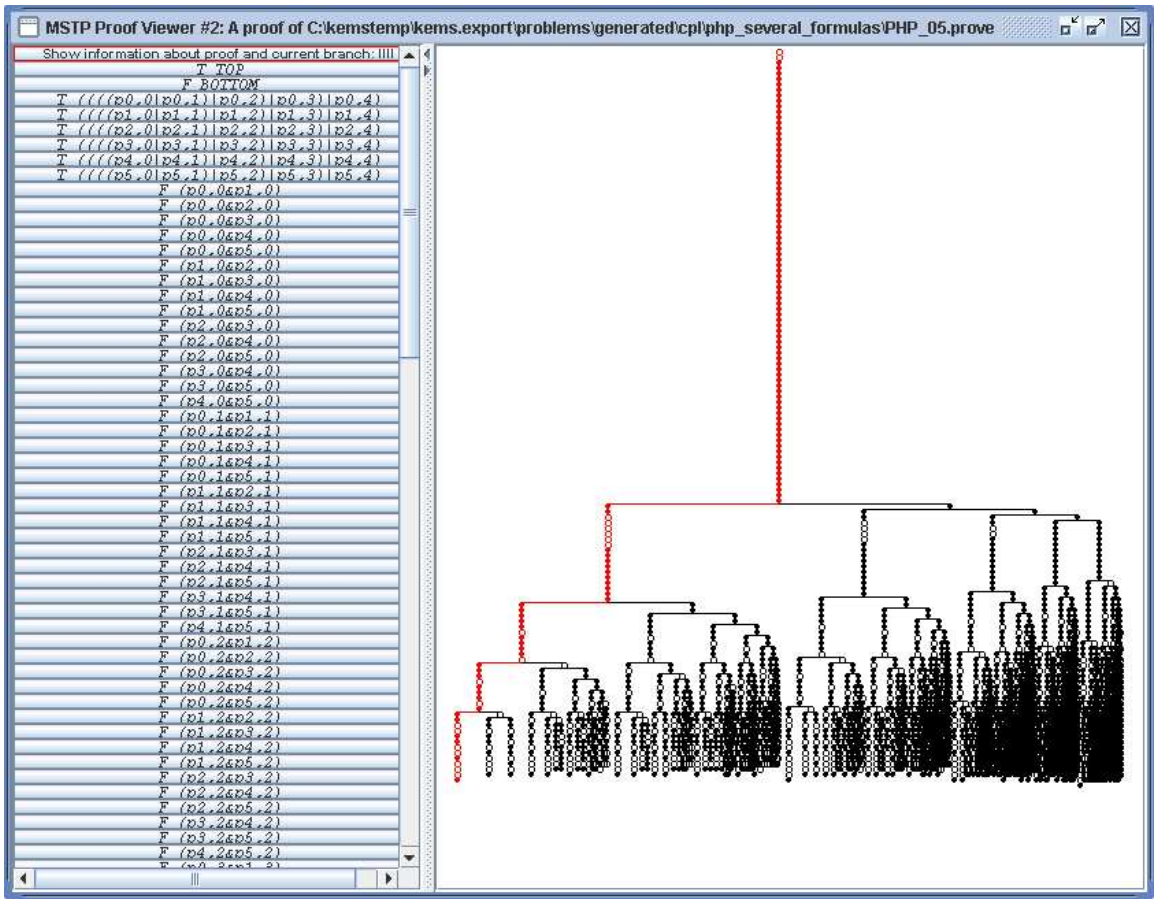


Figure F.5: A Proof Viewer window.

F.2.6 Command-line Sequence Runner

We will not describe in detail here but it is possible to run a sequence of problems from command line without opening **KEMS** graphical interface. A sequence file such as the following must be given as input:

```
parser=sats5

saveOrigin=false

discardClosedBranches=true

saveDiscardedBranches=false

times=1
```

```
timeLimit=3


problems=
problems/generated/cpl/php_several_formulas/PHP_08.prove
problems/generated/cpl/php_several_formulas/PHP_09.prove
problems/generated/cpl/php_several_formulas/PHP_10.prove



strategies=
MemorySaverStrategy
SimpleStrategy
#BackjumpingSimpleStrategy

comparators=
ReverseInsertionOrderComparator
OrComparator
TrueComparator

run
```

The sequence file above tells **KEMS** to run 3 PHP instances with 6 strategy-sorter pairs (all possible combinations of the 2 strategies and 3 comparators listed). One strategy name has a '#' character before it. This character is used to mark a comment so this strategy name will not be used by **KEMS**.

Referências Bibliográficas

[1] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 448–456, New York, NY, USA, 2002. ACM Press.

[2] Noriko Arai, Toniann Pitassi, and Alasdair Urquhart. The complexity of analytic tableaux. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 356–363. ACM Press, 2001.

[3] Bernhard Beckert, Richard Bubel, Elmar Habermalz, and Andreas Roth. jTAP - a Tableau Prover in Java, February 1999. Universitat Karlsruhe. Available at `http://i12www.ira.uka.de/~aroth/jTAP/`. Last accessed, November 2006.

[4] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[5] Evert W. Beth. *The Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1959.

[6] Maria Paola Bonacina and Thierry Boy de la Tour. Fifth Workshop on Strategies in Automated Deduction - Workshop Programme, 2004. `http://tinyurl.com/y8dkbj`. Last accessed, November 2006.

[7] Maria Luisa Bonet and Nicola Galesi. A study of proof search algorithms for resolution and polynomial calculus. In *FOCS '99: Proceedings of the 40th Annual*

Symposium on Foundations of Computer Science, page 422, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Krysia Broda, Marcello D'Agostino, and Marco Mondadori. A Solution to a Problem of Popper. In *The Epistemology of Karl Popper*. Kluwer, 1995. `http://citeseer.nj.nec.com/broda95solution.html`. Last accessed, November 2006.

[9] S. R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.

[10] Liming Cai. *Nondeterminism and optimization*. PhD thesis, Texas A&M University, USA, 1994.

[11] Carlos Caleiro, Walter Carnielli, Marcelo E. Coniglio, and Joao Marcos. Two's company: "The humbug of many logical values". In *Logica Universalis*, pages 169–189. Birkhäuser Verlag, Basel, Switzerland, 2005. Pre-print available at `http://tinyurl.com/yb5qbz`. Last accessed, November 2006.

[12] Alessandra Carbone and Stephen Semmes. *Graphic Apology for Symmetry and Implicitness*. Oxford University Press, 2000.

[13] W. A. Carnielli. Systematization of the finite many-valued logics through the method of tableaux. *The Journal of Symbolic Logic*, 52:473–493, 1987.

[14] W.A. Carnielli and J. Marcos. Ex Contradictione Non Sequitur Quodlibet. *Bulletin of Advanced Reasoning and Knowledge*, 1:89–109, 2001.

[15] W.A. Carnielli and J. Marcos. Tableau systems for logics of formal inconsistency. *Proceedings of the International Conference on Artificial Intelligence (IC-AI'2001)*, pages 848–852, 2001.

[16] Walter Carnielli. How to build your own paraconsistent logic: an introduction to the Logics of Formal (In)Consistency. *Proceedings of the Workshop on Paraconsistent Logic (WoPaLo)*, 2002.

[17] Walter Carnielli, Marcelo Coniglio, and Ricardo Bianconi. *Logic and Applications: Mathematics, Computer Science and Philosophy (in Portuguese).* Unpublished, 2005. Preliminary version. Chapters 1 to 5.

[18] Walter Carnielli, Marcelo E. Coniglio, and Joao Marcos. Logics of Formal Inconsistency. In *Handbook of Philosophical Logic*, volume 12. Kluwer Academic Publishers, 2005. To appear. Pre-print available at `http://tinyurl.com/ybn4yw`. Last accessed, November 2006.

[19] Walter A. Carnielli and Richard L. Epstein. *Computabilidade – Funções Computáveis, Lógica e os Fundamentos da Matemática*. Editora da Unesp, 2006.

[20] Stephen Cook. The P versus NP problem, 2000. `http://tinyurl.com/n5thm`. Last accessed, May 2005.

[21] Stephen Cook. The importance of the P versus NP question. *J. ACM*, 50(1):27–29, 2003.

[22] Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 135–148, New York, NY, USA, 1974. ACM Press.

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

[24] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976.

[25] W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Appl. Math.*, 18(1):25–38, 1987.

[26] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms - Second Edition*. MIT Press, 2001.

[27] Newton C. A. da Costa, Décio Krause, and Otávio Bueno. Paraconsistent logics and paraconsistency: Technical and philosophical developments. *CLE e-prints (Section Logic)*, 4(3), 2004. Pre-print available at `http://tinyurl.com/yxhon7`. Last accessed, November 2006.

[28] Marcello D'Agostino. Are Tableaux an Improvement on Truth-Tables? Cut-Free proofs and Bivalence, 1992. Available at `http://citeseer.nj.nec.com/140346.html`. Last accessed, May 2005.

[29] Marcello D'Agostino. Tableau methods for classical propositional logic. In Marcello D'Agostino et al., editor, *Handbook of Tableau Methods*, chapter 1, pages 45–123. Kluwer Academic Press, 1999.

[30] Marcello D'Agostino, Dov Gabbay, and Krysia Broda. Tableau methods for substructural logics. In Marcello D'Agostino et al., editor, *Handbook of Tableau Methods*, chapter 6, pages 397–467. Kluwer Academic Press, 1999.

[31] Marcello D'Agostino and Marco Mondadori. The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, pages 285–319, 1994.

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

[33] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[34] Sandra de Amo, Walter Carnielli, and João Marcos. A Logical Framework for Integrating Inconsistent Information in Multiple Databases. In Thomas Eiter and Klaus-Dieter Schewe, editors, *Lecture Notes in Computer Science*, volume 2284, pages 67–84. Springer-Verlag, Berlim., 2002.

[35] Eleonora de Moraes. *Lista de discussão gestar bem interior-sp*, 2004. `http://br.groups.yahoo.com/group/gestarbeminterior-sp`. Last accessed, December 2006.

[36] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[37] Luis Fariñas del Cerro, David Fauthoux, Olivier Gasquet, Andreas Herzig, Dominique Longin, and Fabio Massacci. Lotrec: The generic tableau prover for modal and description logics. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 453–458. Springer-Verlag, 2001.

[38] Wagner Dias. Tableaux implementation for approximate reasoning (in portuguese). Master's thesis, Computer Science Department, Institute of Mathematics and Statistics, University of São Paulo, 2002.

[39] Simone Grilo Diniz and Ana Cristina Duarte. *Parto normal ou cesárea? O que toda mulher deve saber (e todo homem também)*. Editora da Unesp, São Paulo, 2004.

[40] Heidi Dixon. *Automating Pseudo-Boolean Inference Within a DPLL Framework*. PhD thesis, University of Oregon, USA, Dec 2004. Available at `http://www.cirl.uoregon.edu/dixon/dixonDissertation.pdf`. Last accessed, August 2005.

[41] Amigas do Parto. *Lista de discussão Parto Nosso*, 2003. `br.groups.yahoo.com/group/partonosso`. Last accessed, December 2006.

[42] Itala M. Loffredo D'Ottaviano and Milton Augustinis de Castro. Analytical Tableaux for da Costa's Hierarchy of Paraconsistent Logics $C_n, 1 \leq n \leq \omega$. *Journal of Applied Non-Classical Logics*, 15(1):69–103, 2005.

[43] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44, 2001.

[44] M. Enkin, M. Skiers, J. Nelson, C. Crowder, L. Duly, and E. Hodnett. *A guide to effective care during pregnancy and childbirth*. Oxford, UK: Oxford University Press, 2000.

[45] Fadynha. *Lista de discussão partonatural*, 1999. `br.groups.yahoo.com/group/partonatural`. Last accessed, December 2006.

[46] Luis Fariñas del Cerro, David Fauthoux, Olivier Gasquet, Andreas Herzig, Dominique Longin, and Fabio Massacci. Lotrec: the generic tableau prover for modal and description logics. In *International Joint Conference on Automated Reasoning*, LNCS, page 6. Springer Verlag, 18-23 juin 2001.

[47] M. Finger and R. Wassermann. Approximate Reasoning and Paraconsistency-Preliminary Report. *Proceedings of the Eigth Workshop on Logic, Language, Information and Comunication (WoLLIC), Braslia, Brazil*, 2001.

[48] M. Finger and R. Wassermann. The universe of approximations. *Electronic Notes in Theoretical Computer Science*, 84:1–14, 2003.

[49] M. Finger and R. Wassermann. Anytime Approximations of Classical Logic from Above. *Journal of Logic and Computation*, 2006.

[50] M. Finger and R. Wassermann. The universe of propositional approximations. *Theoretical Computer Science*, 355(2):153–166, 2006.

[51] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., 1996.

[52] Melvin Fitting. Introduction. In Marcello D'Agostino et al., editor, *Handbook of Tableau Methods*, chapter 1, pages 1–43. Kluwer Academic Press, 1999.

[53] Zhaohui Fu, Yogesh Mahajan, and Sharad Malik. New Features of the SAT'04 versions of zChaff, 2004. `http://www.princeton.edu/~chaff/zchaff/sat04.pdf`. Last accessed, September 2005.

[54] Dov M. Gabbay. *Labelled Deductive Systems, Volume 1.* Oxford University Press, Oxford, 1996.

[55] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Adisson-Wesley, 1994.

[56] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Works of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[57] J. Gosling, B. Joy, and G. Steele. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[58] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[59] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, 1996.

[60] J. Hintikka. Form and content in quantification theory. *Acta Philosophica Fennica*, 8:7–55, 1955.

[61] Jan Holub and Borivoj Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In *WIA '98: Revised Papers from the Third International Workshop on Automata Implementation*, pages 92–99. Springer-Verlag, 1999.

[62] Scott E. Hudson, Frank Flannery, C. Scott Anaian, Dan Wang, and Andrew Appel. *CUP Parser Generator for Java*, 1999. `http://www2.cs.tum.edu/projects/cup`. Last accessed, November 2006.

[63] Ullrich Hustadt and Renate A. Schmidt. Simplification and backjumping in modal tableau. In *Lecture Notes in Computer Science*, volume 1397 of *Lecture Notes in Computer Science*, pages 187–201, 1998.

[64] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44:59–65, 2001.

[65] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[66] Gerwin Klein. *JFlex: the fast lexical analyzer generator for Java*, 1998. `http://jflex.de`. Last accessed, November 2006.

[67] S. Kundu and J. Chen. Fuzzy logic or Lukasiewicz logic: A clarification. *Fuzzy Sets and Systems*, 95(3):369–379, 1998.

[68] L. Di Lascio. Analytic fuzzy tableaux. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 5(6):434–439, Dec 2001.

[69] D. W. Loveland. Automated deduction: Some achievements and future directions. Technical report, National Science Foundation, 1997. Available at `http://tinyurl.com/y7mb6p`. Last accessed, May 2005.

[70] D. W. Loveland. Automated deduction: achievements and future directions. *Commun. ACM*, 43(11es):10, 2000.

[71] Wendy MacCaull. Tableau method for residuated logic. *Fuzzy Sets Syst.*, 80(3):327–337, 1996.

[72] Alistair Manning, Andrew Ireland, and Alan Bundy. Increasing the Versatility of Heuristic Based Theorem Provers. In *LPAR'93*, 1993.

[73] Heiko Mantel and Jens Otten. lintap: A tableau prover for linear logic. In *TABLEAUX '99: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 217–231, London, UK, 1999. Springer-Verlag.

[74] João Marcos. Personal communication by email, October 2006.

[75] Fabio Massacci. Simplification: A general constraint propagation technique for propositional and modal tableaux. In *TABLEAUX '98: Proceedings of the Inter-*

national Conference on Automated Reasoning with Analytic Tableaux and Related Methods, pages 217–231. Springer-Verlag, 1998.

[76] Fabio Massacci. The proof complexity of analytic and clausal tableaux. *Theor. Comput. Sci.*, 243(1-2):477–487, 2000.

[77] William McCune and Larry Wos. Otter - The CADE-13 Competition Incarnations. *J. Autom. Reason.*, 18(2):211–220, 1997.

[78] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, London, UK, fourth edition, 1997.

[79] Paulo Blauth Menezes. *Linguagens Formais e Autômatos*. Instituto de Informática da UFRGS : Editora Sagra Luzzatto, Porto Alegre, 232p., 2005.

[80] Sun Microsystems. Java Runtime Environment (JRE) 5.0 Installation Notes, 2006. `http://java.sun.com/j2se/1.5.0/jre/install.html`. Last accessed, November 2006.

[81] S. H. Mirian and M. Mousavi. Nondeterminism in set-theoretic specifications (in persian). In *Proceedings of Iranian Computer Society Annual Conference (CSICC'02)*, feb 2002.

[82] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[83] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[84] John K. Myers. An introduction to planning and meta-decision-making with uncertain nondeterministic action using 2nd-order probabilities. In *Proceedings of the first*

international conference on Artificial intelligence planning systems, pages 297–298. Morgan Kaufmann Publishers Inc., 1992.

[85] Adolfo Neto. An Object-Oriented Implementation of a KE Tableau Prover, nov 2003. Avaliable at `http://tinyurl.com/y3qmkx`. Last accessed, November 2006.

[86] Adolfo Neto. Modifications on the implementation of a framework for tableau methods, jul 2003. Avaliable at `http://tinyurl.com/yjgjnq`. Last accessed, November 2006.

[87] Adolfo Neto and Marcelo Finger. A Multi-Strategy Tableau Prover. In *I Simpósio de Iniciação Científica e Pós-Graduação do IME-USP*. University of São Paulo, 2005. Available at `http://tinyurl.com/tbdd6`. Last accessed, November 2006.

[88] Adolfo Neto and Marcelo Finger. A Multi-Strategy Tableau Prover. In *SeMe-2005. Workshop "Semantics and Meaning"*, IFIP International Federation for Information Processing. Unicamp. Campinas-SP., 2005. Available at `http://tinyurl.com/yzx8ve`. Last accessed, November 2006.

[89] Adolfo Neto and Marcelo Finger. Implementing a multi-strategy theorem prover. In Ana Cristina Bicharra Garcia and Fernando Santos Osório, editors, *Proceedings of the V ENIA (Encontro Nacional de Inteligência Artificial), held in São Leopoldo-RS, Brazil, July 22-29 2005*, 2005. Available at `http://tinyurl.com/yd6n6n`. Last accessed, November 2006.

[90] Adolfo Neto and Marcelo Finger. Using Aspect-Oriented Programming in the Development of a Multi-Strategy Theorem Prover. In *Anais da II Jornada do Conhecimento e da Tecnologia do Univem*, Marília-SP, 2005. Available at `http://www.ime.usp.br/~adolfo/trabalhos/jornada2005.pdf`. Last accessed, November 2006.

[91] Adolfo Neto and Marcelo Finger. Effective Prover for Minimal Inconsistency Logic. In *Artificial Intelligence in Theory and Practice*, IFIP International Federation for Information Processing, pages 465–474. Springer Verlag, 2006. Available at

`http://www.springerlink.com/content/b80728w7m6885765`. Last accessed, November 2006.

[92] Adolfo Neto and Marcelo Finger. *KEMS - A* **KE** *Multi-Strategy Tableau Prover*, 2006. `http://kems.iv.fapesp.br`. Last accessed, November 2006.

[93] Michel Odent. *The Caesarean*. Free Association Books, 2004.

[94] Lawrence C. Paulson. *Handbook of logic in computer science (vol. 2): background: computational structures*, chapter Designing a theorem prover, pages 415–475. Oxford University Press, Inc., 1992.

[95] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *J. Autom. Reason.*, 2(2):191–216, 1986.

[96] J. V. Pitt and R. J. Cunningham. Theorem proving and model building with the calculus ke. *Journal of the IGPL*, 4(1):129–150, 1996.

[97] Awais Rashid and Lynne Blair. Editorial: Aspect-oriented Programming and Separation of Crosscutting Concerns. *The Computer Journal*, 46(5):527–528, 2003.

[98] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 376–380. Springer-Verlag, 2001.

[99] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, 2006.

[100] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[101] Satisfiability suggested format, 1993. `http://www.satlib.org`. Last accessed, March 22, 2005.

[102] Satisfiability library, 2003. `http://www.satlib.org`. Last accessed, March 22, 2005.

[103] N. Scharli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable units of behaviour. *Proc. of ECOOP*, 2743:248–274, 2003.

[104] J. Schumann. Tableau-based theorem provers: Systems and implementations. *Journal of Automated Reasoning*, 13(3):409–421, 1994. `http://www.springerlink.com/content/k182u80451306371`. Last accessed, November 4th, 2006.

[105] Bart Selman, Hector J. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[106] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968.

[107] Sérgio Soares and Paulo Borba. AspectJ - Programação orientada a aspectos em Java. *Tutorial no SBLP 2002, 6o. Simpósio Brasileiro de Linguagens de Programação. 5 a 7 de Junho, PUC-Rio, Rio de Janeiro, Brasil*, pages 39–55, 2002.

[108] R. Statman. Bounds for proof-search and speed-up in the predicate calculus. *Annals of Mathematical Logic*, pages 225–287, 1978.

[109] Friedrich Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, 2006.

[110] Geoff Sutcliffe. An overview of automated theorem proving, 2001. `http://www.cs.miami.edu/~tptp/OverviewOfATP.html`. Last accessed, March 2005.

[111] Geoff Sutcliffe. Thousands of problems for theorem provers, 2001. `http://www.cs.miami.edu/~tptp`. Last accessed, March 2005.

[112] Geoff Sutcliffe and Christian Suttner. The CADE ATP System Competition, 2003. `http://www.cs.miami.edu/~tptp/CASC`. Last accessed, March 2005.

[113] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

[114] Marian Vittek. A compiler for nondeterministic term rewriting systems. In *RTA '96: Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 154–167. Springer-Verlag, 1996.

[115] Wikipedia. *Nondetermnistic algorithm*, 2007. `http://en.wikipedia.org/wiki/Nondeterministic`. Last accessed, February 2007.