MAPKAT
UM FRAMEWORK LÓGICO PARA PLANEJAMENTO MULTI-AGENTE

Luis Roberto Medeiros Lopes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

[illegible]

Prof. Mario Roberto Folhadela Benevides, Ph. D.

[illegible]

Prof^{a}. Inês de Castro Dutra, Ph. D.

[illegible]

Prof. Edward Hermann Haeusler, D. Sc.

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 2006

LOPES, LUIS ROBERTO MEDEIROS

MAPKAT - Um Framework Lógico para Planejamento Multi-Agente [Rio de Janeiro] 2006

IX, 134 p. 29,7cm (COPPE/UFRJ, M. Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Planejamento Automático Multi-Agente

I. COPPE/UFRJ II. Título (série)

Dedicado aos meus pais Tereza e Francisco e à minha irmã Carina,

por todo o apoio, carinho e confiança que sempre me deram.

Agradecimentos

Ao professor *Mario Benevides*, por sua dedicação e paciência em me ajudar a superar os obstáculos encontrados ao longo de nossos anos de estudo, pesquisa e convivência.

À professora *Inês Dutra*, por ter me apresentado de forma brilhante a área de Inteligência Artificial.

A *todos os colegas e amigos* que fiz durante meu mestrado na COPPE/Sistemas, cuja companhia e camaradagem aliviaram em muito a jornada que foi a preparação deste trabalho.

À *minha família*, sem a qual eu não poderia ter sequer começado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

MAPKAT
UM FRAMEWORK LÓGICO PARA PLANEJAMENTO MULTI-AGENTE

Luis Roberto Medeiros Lopes

Março/2006

Orientador: Mario Roberto Folhadela Benevides

Programa: Engenharia de Sistemas e Computação

A crescente importância dos chamados *sistemas distribuídos* por redes de computadores levaram-nos a evoluir de formas que os reuníram com os *sistemas multi-agente* e que motivaram a criação do domínio de pesquisa chamado *Planejamento Automático Multi-Agente.* Esse texto apresenta o framework MAPKAT, desenvolvido para permitir *model-checking* e planejamento automáticos e semanticamente corretos em problemas multi-agente envolvendo conhecimento individual.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

MAPKAT
A LOGICAL FRAMEWORK FOR MULTI-AGENT PLANNING

Luis Roberto Medeiros Lopes

March/2006

Advisor: Mario Roberto Folhadela Benevides

Department: Systems Engineering and Computer Science

The growing importance of *distributed systems* over computer networks caused them to evolve in ways that brought them and the *multi-agent systems* together and motivated the creation of the *Multi-Agent Planning* domain of research. This text presents the *MAPKAT* framework designed to support automatic model-checking and planning in dynamic multi-agent environments involving individual knowledge.

Sumário

Capítulo 1

Introdução

Planejamento Automático é uma das principais áreas de estudo do domínio da Inteligência Artificial. Como tal, ela cresceu e se desenvolveu através de anos de estudo e pesquisa. Naturalmente, essa área evoluiu para cobrir problemas mais sofisticados.

Alguns desses problemas mais recentes surgiram pela importância obtida pelos chamados *sistemas distribuídos*. Esses sistemas evoluíram de forma que os reuniu aos *sistemas multi-agentes*, levando a sistemas mais complexos compostos por agentes mais sofisticados. Tal crescimento em complexidade motivou a criação do domínio de pesquisa de *Planejamento Multi-Agente*.

Foi mostrado, em [van der Hoek and Wooldridge, 2002], que problemas de planejamento multi-agente podem ser abordados como problemas de lógica modal (através da lógica conhecida como ATEL). Porém, formular um problema desse tipo através de lógica modal pode ser problemático, uma vez que esse não é o propósito original da lógica usada. Além disso, falta a essa abordagem estruturas semânticas adequadas para formular planos facilmente compreensíveis pelos usuários humanos.

KCTL é uma lógica modal desenvolvida originalmente para permitir raciocinar sobre tempo, o conhecimento de um conjunto de agentes e a evolução desse conhecimento [Benevides et al., 2004]. Ela foi baseada na lógica modal temporal clássica CTL [Ben-Ari et al., 1981] e, portanto, seu foco primário é model-checking.

Esse texto propõe a evolução das estruturas semânticas da KCTL para obter uma abordagem mais apropriada de problemas de planejamento multi-agente.

1.1 Planejamento Clássico "Mono-Agente"

Antes de abordar planejamento multi-agente adequadamente, é necessário considerar o "caso básico": Planejamento Automático Clássico. Conforme mencionado, ele compõe uma das áreas primordiais da Inteligência Artificial. Essa área se dedica a resolver o chamado Problema de Planejamento Automático, que consiste em tomar o modelo de um determinado sistema e levá-lo de sua configuração inicial até outra configuração que atenda a uma determinada propriedade.

A área de Planejamento Automático vem sendo estudada e pesquisada há anos, atualmente apresentando diversas abordagens para a resolução de seus problemas. Porém, por mais variadas que sejam, todas essas abordagens apresentam os mesmos elementos fundamentais:

Estados As diferentes configurações que um modelo pode assumir;

Ações As formas pelas quais uma configuração é transformada em outra;

Objetivos Propriedades às quais se deseja satisfazer.

Neste trabalho, quatro abordagens foram estudadas:

- Planejamento por Cálculo de Situação;
- Planejamento por Lógica Modal;
- Planejamento STRIPS;
- Planejamento por Análise de Grafos.

Cada abordagem é apresentada brevemente a seguir. Maiores detalhes são dados no anexo A, página 32.

1.1.1 Planejamento por Cálculo de Situação

Lógica de primeira ordem foi largamente usada em planejamento automático. Sua técnica mais bem-sucedida foi o chamado *Cálculo de Situação*, proposta por MacCarthy.

No Cálculo de Situação, a especificação de um problema de planejamento é dada por um *sistema de axiomas*. As propriedades de cada estado são representadas por *predicados*, sendo cada estado composto por fórmulas de primeira ordem. Ações são representadas por *funções* que mapeiam estados em outros estados. Objetivos são descritos por fórmulas de primeira ordem.

Essa abordagem foi bastante difundida por ser baseada em um sistema lógico conhecido. Portanto, sua semântica era clara e comprovadamente correta. Pelo mesmo motivo, podia-se utilizar provadores automáticos de teoremas já disponíveis, sendo desnecessário o desenvolvimento de ferramentas específicas para esse fim.

Em compensação, ela sofre do chamado *problema de frame*, uma dificuldade oriunda da necessidade de axiomas para especificar não só o que caa ação modificava, mas também o que permanecia inalterado. Rapidamente verificou-se serem necessários muito mais axiomas para descrever o que cada ação *não* mudava. Como resultado, as provas eram longas e lentas demais, limitando em muito a sua aplicação a problemas reais.

1.1.2 Planejamento por Lógica Modal

Lógica Modal de Ações ou Lógica Dinâmica é uma lógica modal onde os operadores modais de necessidade $\Box$ e possibilidade $\Diamond$ são substituídos por uma família de operadores $[a]$ e $< a >$.

Para cada ação a, existe o par operadores modais $[a]$ e $< a >$. A semântica dessa lógica é a semântica dos mundos possíveis, onde para cada ação existe uma relação de acessibilidade: Uma fórmula $[a]\alpha$ é válida em um estado s se α é verdadeira em todos os possíveis estados resultantes da execução da ação a no estado s.

Parece bastante intuitivo utilizar lógica dinâmica na representação de problemas de planejamento. Ações são representados pelos operadores modais descritos e estados são descritos por mundos possíveis. O estado inicial é o mundo onde o sistema começa e o problema é especificado por um conjunto de axiomas.

As vantagens do uso da lógica modal são similares às do uso do cálculo de situações: a abordagem é baseada em um sistema lógico bastante conhecido e estudado, com uma semântica comprovadamente forte e permitindo o uso de provadores de teorema genéricos de lógica modal. Além disso, a representação é mais clara e elegante que a do cálculo de situações.

Infelizmente, essa abordagem também sofre do problema de frame, resultando em provas muito lentas e em limitações na sua aplicação a problemas reais.

1.1.3 Planejamento STRIPS

STRIPS é um solucionador automático de problemas de planejamento proposto por Fikes e Nilson. Os estados são representados por modelos de mundo, que por sua vez são conjuntos de fórmulas de lógica de primeira ordem. Cada ação é definida por um operador consistindo de uma descrição dos seus efeitos no modelo de mundo. O operador também descreve as condições necessárias para a execução de sua ação.

A especificação de um problema de planejamento consiste de um modelo inicial do mundo, um conjunto de operadores e um objetivo (que, por sua vez, é uma fórmula de primeira ordem). Um provador de teoremas por resolução é usado para encontrar uma sequência de operadores que transforma o modelo inicial em um modelo final, onde o objetivo é satisfeito.

STRIPS foi, durante anos, a abordagem mais popular para a representação de problemas de planejamento automático. Além de uma representação clara e flexível, mais fácil de ser compreendida por leigos, STRIPS faz uso de um mecanismo próprio para evitar o problema de frame. Dessa forma, seu processo de prova e planejamento é rápido e eficiente, podendo lidar com problemas reais.

Essa eficiência, porém, tinha um preço: sua especificação inicial deu liberdade suficiente para a implementação de planejadores automáticos que, entre outras coisas, permitiam a manutenção de estados inconsistentes durante o processo de planejamento. Apesar de garantirem a eficiência do planejamento, esses planejadores passaram a apresentar uma semântica duvidosa e fraca, o que possibilitava o surgimento de planos errados ou simplesmente impossíveis.

Estudos sobre a semântica da linguagem STRIPS foram apresentados por Lifschitz, incluindo sugestões para corrigir a fraqueza da linguagem. Infelizmente, essas sugestões impuseram séries restrições à linguagem, conforme exposto na seção A.5.

1.1.4 Planejamento por Análise de Grafos

Graphplan é um planejador automático apresentado por Blum e Furst. Ele introduz uma nova abordagem ao planejamento em domínios STRIPS-like. Esse paradigma se baseia em construir e analisar uma estrutura compacta chamada *Grafo de Planejamento*.

Um grafo de planejamento codifica o problema de planejamento de forma que várias restrições úteis e inerentes ao problema se tornem explícitos, reduzindo a quantidade de buscas necessárias para alcançar uma conclusão - seja ela um plano ou a certeza de que não existe um plano possível. Esse grafo *não é* o grafo do espaço de estados. Enquanto um plano em um grafo de estados é um *caminho*, um plano em um grafo de planejamento é um *fluxo*, no sentido de fluxo de redes.

O planejador Graphplan usa o grafo de planejamento que ele cria para guiar sua busca por um plano. Essa busca combina aspectos tanto de planejadores de ordem total quanto dos de ordem parcial. Como os de ordem total, Graphplan faz fortes restrições em sua busca - quando ele considera uma ação, ele a considera em um ponto específico do tempo. Por outro lado, como planejadores de ordem parcial, Graphplan gera planos parcialmente ordenados: ações no mesmo passo de tempo podem ser executadas em qualquer ordem - é um tipo de plano paralelizado.

Um grafo de planejamento é um grafo nivelado. Ele possui dois tipos de nós: nós de proposição, cada um rotulado com uma proposição do problema, e nós de ação, cada um rotulado com um operador do problema. Ele tem dois tipos de níveis: níveis de proposição, compostos apenas por nós de proposição, e níveis de ação, compostos apenas por nós de ação.

O primeiro nível é composto pelos nós de proposição correspondentes ao estado inicial do problema. O segundo nível é composto pelos nós de ação correspondentes a todos os operadores aplicáveis ao estado inicial. A partir de então se alternam níveis de proposição e de ação, onde um nível de proposição é composto pela união do nível de proposição anterior com o ADD-LIST dos operadores do nível de ação anterior.

Muito mais detalhes sobre essa abordagem são apresentados na seção A.6, página 56. Por agora, é necessário destacar:

- Estados são representados subconjuntos de níveis de proposição;
- Ações são denotadas por nós de ação;
- Objetivos são conjuntos de nós de proposição que precisam pertencer a um mesmo nível de proposição;
- A especificação do problema é o próprio grafo de planejamento.

Essa nova abordagem oferece uma semântica correta, fortemente baseada em causalidade, e processamento rápido, devido à um processo de busca direcionada no grafo de planejamento. Em compensação, sua representação é inusitada, lembrando uma árvore de computação compactada. Sua semântica também pode ser difícil de compreender a princípio, por ser baseada em possibilidades ao invés de certezas. Como resultado, com exceção do estado inicial, todos os estados que o sistema pode alcançar estão implícitos nos níveis de proposição.

1.2 Planejamento Multi-Agente

O planejamento multi-agente é uma evolução do caso clássico. Além dos elementos mencionados (operadores, estados e objetivos), um problema de planejamento multi-agente inclui os próprios agentes em sua descrição. Os operadores agora "pertencem" aos agentes que os eecutam e os estados do sistema modelado agora é derivado dos estados dos agentes que o compõem.

Além dessas adaptações menores, a inclusão de agentes implica na consideração de determinados aspectos:

- Relacionamento Inter-agentes;
- Conhecimento Distribuído;
- Influência Distribuída.

O Relacionamento entre os agentes de um determinado problema pode ser classificado como Cooperativo (os agentes têm objetivos similares e trabalham juntos para alcançá-los), Competitivo (cada agente tem objetivos conflitantes com os outros e deve se esforçar para os outros falharem, de forma que possa alcançar seus próprios objetivos) ou Neutro (os agentes têm objetivos independentes e não precisam interferir uns com os outros).

O Conhecimento distribuído em um sistema descrito por um problema multi-agente também tem classificações: Local (apenas um agente sabe de determinado fato), Geral (todos sabem de um certo fato) ou Comum (não só todos os agentes sabem de um fato como também sabem que os outros sabem desse fato).

A Influência exercida por um agente ou por um grupo de agentes (coalisão) são caracterizados pelos conceitos de Capacidade (um agente ou uma coalisão é capaz de levar o sistema a um estado tal que uma determinada propriedade seja satisfeita) e Controle (um agente ou uma coalisão não pode ser impedida de levar o sistema a um estado tal que uma determinada propriedade seja satisfeita).

Cada uma dessas considerações adiciona uma dimensão nova aos problemas de planejamento. Como resultado, esses problemas se tornam muito mais complexos, levando a processamentos muito mais longos na tarefa de planejamento. Mais que isso, os planos gerados tendem a ser mais sofisticados, dificultando em muito qualquer verificação humana da corretude desses planos - aumentando a chance de erros passarem despercebidos!

Por esse motivo a seção de abertura deste capítulo menciona a necessidade de métodos próprios para a análise e planejamento automáticos em sistemas multiagentes. Esses métodos devem não só serem mais eficientes, como precisam ter uma semântica forte, clara e rigorosa.

1.3 Organização do Texto

O texto é organizado conforme descrito abaixo.

O primeiro capítulo demonstra as motivações atrás do estudo descrito e a estrutura usada para apresentar o estudo em si.

O segundo capítulo apresenta o *framework* **MAPKAT**, descrevendo os elementos e conceitos envolvidos para cumprir seu papel como estrutura semântica.

O terceiro capítulo comenta como o *framework* apresentado pode ser utilizado como a estrutura lógica do processo de *model-checking* da lógica **KCTL**.

O capítulo de conclusão encerra o texto expondo de forma resumida os frutos do estudo realizado e propõe temas para trabalhos futuros.

Os apêndices apresentam em maior detalhe todos os aspectos envolvidos nesse estudo. Eles se encontram em inglês.

O apêndice A apresenta diversas abordagens conhecidas para o problema de planejamento mono-agente clássico, incluindo um estudo do STRIPS e um resumo do artigo que apresentou os grafos de planejamento.

O apêndice B comenta alguns aspectos próprios do problema de planejamento multi-agente e apresenta um resumo do artigo que apresentou a lógica KCTL.

O apêndice C é uma versão detalhada do segundo capítulo, tomando o cuidado de ser rigoroso e não ambíguo.

Similarmente, o apêndice D é a versão mais extensa do terceiro capítulo. Por questão de completude, ele apresenta muitos detalhes que, em nome da clareza, não foram apresentados no terceiro capítulo. Ele também apresenta todos os algoritmos necessários para efetuar o *model-checking* de uma fórmula KCTL em um modelo MAPKAT.

Capítulo 2

O Framework Semântico MAPKAT

Conforme descrito em [van der Hoek and Wooldridge, 2002], a lógica modal pode ser utilizada para planejamento automático. Ao estender a semântica da lógica **KCTL**, o *framework* **MAPKAT** se esforça em adequá-la como uma ferramenta não só de *model-checking*, mas também de planejamento automático multi-agente.

A principal motivação para essa tarefa vem do fato que planejamento automático pode sofrer com dificuldades semânticas no cálculo de soluções. Lógica modal, em contraste, sempre garantiu corretude semântica em suas soluções, porém dificilmente se aproximava da eficiência dos planejadores automáticos. Portanto, o estudo descrito focou o desenvolvimento de um framework "híbrido", unindo as vantagens de ambas as áreas e minimizando suas fraquezas e permitindo sua aplicação em ambos os tipos de problemas.

O framework MAPKAT também foi projetado para superar a principal limitação da lógica KCTL original: as estruturas lógicas da versão original da KCTL, os autômatos, não têm como sustentar a propriedade de "*perfect recall*".

[van der Meyden, 1994] define *perfect recall* como a propriedade de um sistema distribuído em manter um registro em cada processador tal que cada processador registre todos os estados em que ele já esteve. Podemos interpretar essa definição como a propriedade de um sistema garantir que cada processador "se lembre" de todos os estados que teve desde o início da execução do sistema. Para um sistema multi-agente, isso significa que cada agente precisa "se lembrar" de todos os estados que assumiu.

A abordagem oferecida pelo MAPKAT sustenta essa propriedade, visto que ele precisa manter tal registro para poder formular suas respostas: cada resposta é composta por uma ou mais "estratégias" capazes de levar o modelo a um estado que atenda a uma dada propriedade, se existir. Logo, MAPKAT faz o modelo se lembrar de cada passo necessário e, portanto, pode reconhecer quando um dado estado já foi alcançado.

Esse capítulo apresentará a semântica lógica por trás de um modelo MAPKAT. Explicações de como essa semântica se aplica a um problema de planejamento automático será exposto no capítulo 3.

2.1 Conceitos Básicos

MAPKAT usa uma estrutura híbrida que combina os conceitos semânticos da KCTL com os conceitos lógicos dos grafos de planejamento. Essa seção fornece uma descrição de cada conceito envolvido.

Os modelos MAPKAT são baseados em proposições. Porém isso não se torna um limitador de sua aplicabilidade: planejadores automáticos baseados em grafos de planejamento traduzem domínios *STRIPS-like*[1] em grafos similares aos usados pelo MAPKAT, conforme visto em [Blum and Furst, 1995]. Portanto já é assumida a aplicabilidade do framework MAPKAT a domínios STRIPS-like.

[1]Um domínio *STRIPS-like* é a descrição de um problema ou sistema que utiliza uma sintaxe muito parecida com a linguagem STRIPS e inclui variáveis discretas finitas. Durante a produção deste texto, domínios STRIPS-like se mostraram bastante populares na área de planejamento automático.

A seguir são dadas definições simplificadas dos conceitos envolvidos no framework. Para maiores detalhes, consulte o apêndice C, página 76.

Definição 2.1.1 Operador.

Um operador *é uma estrutura* $\mathcal{O} = \langle \mathbb{C}, \mathbb{A}, \mathbb{D} \rangle$ *tal que:*

- $\mathbb{C}$ *é o conjunto das proposições que formam as* pré-condições *do operador - denotado* precond($\mathcal{O}$)*;*
- $\mathbb{A}$ *é o conjunto das proposições que o operador torna verdadeiras após sua execução. Ele também é chamado de* Add-List *do operador e é denotado por* addlist($\mathcal{O}$)*;*
- $\mathbb{D}$ *é o conjunto das proposições proposições que o operador torna falsas após sua execução. Ele também é chamado de* Del-List *do operador e é denotado por* dellist($\mathcal{O}$)*.*

Se existir um conjunto $\mathbb{F}$ *tal que* $\mathbb{C} \subset \mathbb{F}$*,* $\mathbb{A} \subset \mathbb{F}$ *e* $\mathbb{D} \subset \mathbb{F}$*, então diz-se que o operador é* descrito por $\mathbb{F}$ ■

Existe uma família de operadores chamada "operadores nulos". Existe um operador nulo para cada proposição p e ele tem a forma "Δ_p". Cada Δ_p é tal que precond(Δ_p) =addlist(Δ_p) = $\{p\}$ e dellist(Δ_p) = $\emptyset$. Sua semântica é "fazer nada a p".

Definição 2.1.2 Agente.

Um agente é uma estrutura $\langle \mathbb{F}, \mathbb{O}, S_0 \rangle$ *tal que:*

- $\mathbb{F}$ *é o seu conjunto de* proposições exclusivas*;*
- $\mathbb{O}$ *é o seu conjunto de operadores exclusivos, todos descritos por* $\mathbb{F}$ *e incluindo um operador nulo* Δ_p *para cada* $p \in \mathbb{F}$*;*
- $S_0 \subseteq \mathbb{F}$ *representa o seu* estado inicial.

■

Os operadores de um agente podem ser descritos por um conjunto que contenha proposições fora do conjunto exclusivo desse agente. Isso é permitido para que haja comunicação entre os agentes. Porém, as pré-condições de um operador não podem conter nenhuma proposição que não pertença ao conjunto exclusivo do agente.

Definição 2.1.3 Estado Local.

Um Estado Local é o conjunto de proposições que descreve a configuração de um agente. Por isso, todo estado local de um agente é um subconjunto de seu conjunto de proposições exclusivas.

■

O conjunto de todos os estados locais possíveis de um agente $\mathcal{X}$ é denotado por $\mathbb{S}^{\mathcal{X}}$.

Um operador é aplicável a um estado local somente se o estado local contiver todas as proposições que compõem as pré-condições desse operador.

Definição 2.1.4 Sistema Multi-Agente

Um Sistema Multi-Agente (ou SMA*) é uma estrutura* $\mathfrak{M} = \langle \mathbb{X}, \mathbb{N}, \mathbb{Q} \rangle$ *tal que:*

- $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *é o conjunto de agentes do sistema;*
- $\mathbb{N} = \bigcup_{i=1}^{n} \mathbb{O}^i \Big| \mathbb{O}^i$ *é o conjunto de operadores do agente* $X^i \in \mathbb{X}$*;*
- $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{F}^i \Big| \mathbb{F}^i$ *é o conjunto de proposições exclusivas do agente* $X^i \in \mathbb{X}$.

Um SMA é bem-formado *somente se todo operador* $\mathcal{O} \in \mathbb{N}$ *for descrito por* $\mathbb{Q}$. *Assume-se que* $\mathbb{Q}$ *é uma união de conjuntos disjuntos.*■

Definição 2.1.5 Estado Global e Verdade Global

Seja $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *o conjunto de agentes de um SMA bem-formado* $\mathfrak{M}$. *Um* estado global *é uma* composição paralela $\mathcal{G} = \langle S^1 \parallel S^2 \parallel \ldots \parallel S^n \rangle$ *onde* S^i *é o estado local de* X^i.

Um estado global $\mathcal{G}$ *representa uma configuração do SMA e define um conjunto* $\mathcal{G}^p$ *de proposições tal que* $\mathcal{G}^p = \bigcup_{i=1}^{n} S^i$. *Esse conjunto é chamado de* Verdade Global do Estado Global $\mathcal{G}$.

O Estado Global Inicial *de* $\mathfrak{M}$ *é a composição paralela dos estados locais iniciais dos agentes do SMA.*■

Um operador é aplicável a um estado global somente se a verdade global correspondente contiver todos os elementos de sua pre-condição.

MAPKAT seria muito limitado se considerássemos apenas a utilização de ações isoladas. Usando uma abordagem inspirada em [Blum and Furst, 1995], utilizamos "*passos*". Intuitivamente, um *passo* é um conjunto de ações que podem ser aplicadas em paralelo a um dado estado global.

Definição 2.1.6 Passos

Seja $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *o conjunto de agentes de um SMA bem-formado* $\mathfrak{M}$ *e seja* G *um estado global de* $\mathfrak{M}$.

Um Passo aplicável a G, *ou simplesmente* passo, *é a composição paralela* $m = \langle A_G^1 \parallel A_G^2 \parallel \ldots \parallel A_G^n \rangle$ *aonde* A_G^i *é um conjunto de operadores do agente* X^i *aplicáveis a* G. ■

Definição 2.1.7 Produção

Seja G *um estado global e seja* m *um passo aplicável a* G.

O resultado da aplicação de m *a* G *é um novo estado global* G' *tal que*

$$G' = G \cup ADD_m \setminus DEL_m$$

onde ADD_m *é a união das add-lists dos operadores que compõem* m *e* DEL_m *é a união de seus del-lists.*

Dizemos que G' é produzido *pelo passo m aplicada a G. Essa relação é denotada $h(G, m) = G'$.*■

Um *passo* só é válido se sua composição não incluir nenhum par de operadores *mutuamente exclusivos* - ver definição 2.1.10, página 16.

O que nos leva a um importante conceito trazido dos grafos de planejamento: Exclusão Mútua. Em essência, dois elementos mutuamente exclusivos não podem coexistir: eles são conflitantes. Esse conceito pode ser aplicado tanto a operadores quanto a proposições. A definição adequada, mais uma vez, é dada na página 16.

Definição 2.1.8 Execuções

Uma execução - ou "run" *- de $\mathfrak{M}$ é uma sequência $R = m_1m_2m_3\ldots$ de passos ordenados no tempo. Essa sequência pode ser finita ou infinita. Se for uma sequência finita de comprimento k, ela também é chamada de k-execução.*

Seja $\mathfrak{M}$ um SMA bem-formado. Uma execução só é válida *em $\mathfrak{M}$ - ou simplesmente* válida *- se:*

1. m_1 é aplicável ao estado global inicial de $\mathfrak{M}$;

2. Para todo $i > 1$, m_i é aplicável ao estado global produzido por m_{i-1}.

O produto do j-ésimo passo *de R é denotada por $\pi_j(R)$ e o* estado inicial *é denotado por $\pi_0(R)$. O* produto de uma k-execução *R_k é $\pi_k(R_k)$.*

A parte de uma execução R formada pelos operadores de um agente X^i é a execução local de *X^i e é denotada por R^i.* ■

São as execuções que fazem o SMA evoluir no tempo, um passo de cada vez. Por isso, os passos também podem ser usados como uma referência de tempo, apesar de não terem uma duração específica nem qualquer forma direta de serem medidos.

Definição 2.1.9 Níveis de Ação e Níveis de Proposição

O nível de ação AL_t de um SMA bem-formado, para algum $t \geq 1$, é a união de todos os passos que podem figurar como o t-ésimo elemento de uma execução válida.

Também pode-se considerar AL_t *como o conjunto de todos os operadores que podem ser executados no tempo* t.

O nível de proposição PL_t *para um SMA bem-formado* $\mathfrak{M}$ *e algum* $t \geq 0$, *é o conjunto de todas as proposições que podem ser verdadeiras no tempo* t. *Portanto,* PL_0 *corresponde à verdade global do estado global inicial do SMA* e PL_t, *para* $t \geq 1$, *é a união das verdades globais de todos os estados globais que podem ser produzidos por alguma* t*-execução.* ■

Definição 2.1.10 Exclusão Mútua

- Entre Operadores

  Sejam $\mathcal{O}_1$ *e* $\mathcal{O}_2$ *dois operadores diferentes do nível de ação* AL_k *de algum SMA. Eles são considerados* mutuamente exclusivos em k *se:*

  - *Para* $k \geq 1$, *o Del-List de um dos operadores contém ao menos uma proposição que também pertence ao Add-List ou às Pré-Condições do outro (caso base); ou*
  - *Para* $k > 1$, *uma das proposições pertencente às Pré-Condições de um operador é mutuamente exclusivo em* $k-1$ *de uma das proposições pertencentes às Pré-Condições do outro operador.*

- Entre Proposições

  Sejam $p, q \in \mathbb{Q}$ *duas proposições pertencentes ao nível de proposição* k *de um SMA bem-formado. Eles são* mutuamente exclusivos em k *se todo operador* $\mathcal{O}_1 \in AL_k$ *com* p *em seu Add-List for mutuamente exclusivo de todo operador* $\mathcal{O}_2 \in AL_k$ *com* q *em seu Add-List.*

■

Esse conceito pode ser estendido a estados locais, estados globais, passos e execuções. No caso de estados locais: se as proposições p e q forem mutuamente exclusivas em k, então qualquer estado local que inclua p é mutuamente exclusivo de qualquer estado local que inclua q. Para estados globais, passos e execuções, os critérios são análogos. Uma definição mais detalhada é fornecida no apêndice C.

Definição 2.1.11 Relações de Possibilidade *Sejam* $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *o conjunto de agentes de um SMA,* R_a *a execução válida que produz o estado global* G_a *e* R_b *a execução válida que produz o estado global* G_b*.*

Então, para $1 \geq i \geq n$*, escrevemos* $G_a \sim_i G_b$ *se e somente se a execução local* R^i_a *for igual à execução local* R^i_b*.* ■

A relação $G_a \sim_i G_b$ significa que o agente X^i não consegue distinguir G_a de G_b. Portanto, quando considerarmos o que o agente X^i realmente sabe depois de executar sua run local R^i, devemos considerar as propriedades comuns a todos os estados globais que podem ser alcançados por runs contendo R^i como i-ésimo componente.

2.2 Definição de Modelo

A definição de um modelo MAPKAT é muito parecida com a definição de um domínio STRIPS-like. De fato, conforme mencionado antes, MAPKAT foi desenvolvido para utilizar tais domínios. Portanto, ao invés de uma especificação rigorosa de sintaxe, essa seção especificará o que é necessário incluir na definição de um modelo MAPKAT correto.

Existem dois elementos essenciais em uma especificação de modelo: os *agentes*, cuja função é definir o contexto do modelo, e os *operadores*, cuja função é definir a evolução desse contexto. Juntos, os agentes definem as proposições que o Sistema Multi-Agente tem e o estado global inicial. Já os operadores definem as ações que podem ser executadas.

Obviamente, há alguns detalhes importantes: um agente deve especificar quais operadores pode usar, enquanto os operadores devem se restringir a manipular apenas as proposições dos agentes. Por esse motivo, e pelo bem de uma melhor organização, um modelo pode ser definido apenas pelo seu conjunto (não vazio) de agentes, os quais incluirão as descrições de seus operadores.

O conjunto de agentes é composto por estruturas com o formato apresentado na figura 1.

Figura 1 Definição de Agente

1. Agente: *Identificador do agente*
2. Conjunto de Proposições: Não vazio.
3. Conjunto de Operadores: lista não vazia de definições de operador.
4. Estado Inicial: Sub-Conjunto não vazio do parâmetro 2.

O conjunto de operadores de um agente. por sua vez, é composto por elementos com o formato da figura 2.

Figura 2 Definição de Operador

1. Operador: *Identificador de Operador*
2. Pré-condições: conjunto não vazio composto por proposições exclusivas do agente que define o operador.
3. Add-List: conjunto composto por proposições do sistema.
4. Del-List: conjunto composto por proposições do sistema.

2.3 Linguagem de Verificação

O formato de especificação de modelos apresentado na seção 2.2 é um produto direto da literatura de planejamento. Já a linguagem de verificação de propriedades reflete a influência da literatura de *model-checking*; de fato, ela é a linguagem da lógica KCTL.

2.3.1 Sintaxe

Nessa seção haverá referências a um SMA $\mathfrak{M}$ composto pelo conjunto de agentes $\mathbb{X} = \{X^1, \ldots, X^n\}$. Também será usado o conjunto $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{P}^i$, onde $\mathbb{P}^i$ é o conjunto de preposições do agente X^i.

Definição 2.3.1 A linguagem KCTL

O conjunto de fórmulas KCTL para o SMA $\mathfrak{M}$, *denotado por* ForMAPKAT$(\mathfrak{M})$, *é o menor conjunto* For *tal que:*

1. $p \in \mathbb{Q} \leftrightarrow p \in \text{For}$;
2. $\Big\{\neg\phi_1, \phi_1 \vee \phi_2\Big\} \subseteq \text{For} \leftrightarrow \Big\{\phi_1, \phi_2\Big\} \subseteq \text{For}$;
3. $\Big\{\exists \bigcirc \phi_1, \exists\Box\phi_1, \exists\big[\phi_1 \mathsf{U} \phi_2\big]\Big\} \subseteq \text{For} \leftrightarrow \Big\{\phi_1, \phi_2\Big\} \subseteq \text{For}$;
4. $\mathcal{K}_i\phi_1 \in \text{For} \leftrightarrow \Big((\phi_1 \in \text{For}) \wedge (1 \leq i \leq n)\Big)$;

■

A primeira forma caracteriza a classe das proposições. Os operadores definidos na segunda forma compõem a classe de *operadores booleanos*. A terceira forma define a classe dos *operadores temporais*. Finalmente, a última forma caracteriza o *operador epistêmico*.

Além das definições apresentadas, existem algumas "*abreviações*" úteis. Obviamente, a abreviação de um operador pertence à mesma classe do operador original. Elas incluem:

- $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$;
- $\phi_1 \longrightarrow \phi_2 = \phi_2 \vee \neg\phi_1$;

- $\phi_1 \longleftrightarrow \phi_2 = (\phi_1 \longrightarrow \phi_2) \land (\phi_2 \longrightarrow \phi_1)$;
- $\exists\Diamond\phi = \exists[\mathit{true}\mathsf{U}\phi]$;
- $\forall \bigcirc \phi = \neg\exists \bigcirc \neg\phi$;
- $\forall\Box\phi = \neg\exists\Diamond\neg\phi$;
- $\forall\Diamond\phi = \neg\exists\Box\neg\phi$;
- $\forall[\phi_1\mathsf{U}\phi_2] = \neg\exists[\neg\phi_2\mathsf{U}(\neg\phi_1 \land \neg\phi_2)]$
- $\mathcal{K}_i\phi = \neg\mathcal{B}_i\neg\phi$

Uma fórmula φ é chamada uma *fórmula de KCTL sobre* $\mathfrak{M}$ somente se $\varphi \in$ *ForMAPKAT*$(\mathfrak{M})$.

2.3.2 Semântica

A semântica de uma fórmula $\phi \in$ *ForMAPKAT*$(\mathfrak{M})$ é dada naturalmente em termos do modelo MAPKAT construído para $\mathfrak{M}$.

Seja $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ o conjunto de agentes de $\mathfrak{M}$.

Seja também $G = \langle\ S^1\ \|\ S^2\ \|\ \ldots\ \|\ S^n\ \rangle$ um estado global de $\mathfrak{M}$.

Uma fórmula KCTL φ sobre $\mathfrak{M}$ é sempre testada para satisfabilidade em relação a algum estado global G de $\mathfrak{M}$. Escrevemos $\mathfrak{M}, G\ \models\ \varphi$ para alguma fórmula φ somente se φ é verdadeiro no estado G do sistema $\mathfrak{M}$.

Existem algumas variantes válidas para a sintaxe da *relação de satisfabilidade* $\models$:

- $G \models \varphi$, para algum estado global G do SMA $\mathfrak{M}$, é o mesmo que $\mathfrak{M}, G\ \models\ \varphi$;
- $\mathfrak{M} \models \varphi$ é o mesmo que $\mathfrak{M}, G_0\ \models\ \varphi$, onde G_0 é o estado global inicial de $\mathfrak{M}$;
- $\models\ \varphi$ significa que φ é um axioma, uma fórmula válida para qualquer estado de qualquer SMA.

Suponha que $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{P}^i$ — onde $\mathbb{P}^i$ é o conjunto de proposições do agente X^i — é o conjunto de proposições de $\mathfrak{M}$ e que $\mathbb{N} = \bigcup_{i=1}^{n} \mathbb{O}^i$ — onde $\mathbb{O}^i$ é o conjunto de operadores do agente X^i — representa o conjunto de operadores de $\mathfrak{M}$. A semântica das fórmulas KCTL obre $\mathfrak{M}$ são:

1. $\mathfrak{M}, G \models$ *true*, para todo SMA $\mathfrak{M}$ e todo estado global G;

2. $\mathfrak{M}, G \models p$ se e somente se $p \in G^P$, onde G^P é a verdade global de G;

3. $\mathfrak{M}, G \models \neg\varphi$ se e somente se $\mathfrak{M}, G \not\models \varphi$;

4. $\mathfrak{M}, G \models \varphi_1 \vee \varphi_2$ se e somente se $\mathfrak{M}, G \models \varphi_1$ ou $\mathfrak{M}, G \models \varphi_2$;

5. $\mathfrak{M}, G \models \exists \bigcirc \varphi$ se e somente se existe um passo m aplicável a G tal que $\mathfrak{M}, g(G, m) \models \varphi$;

6. $\mathfrak{M}, G \models \exists\Box\varphi$ se e somente se existe uma execução infinita $R = m_1 m_2 m_3 \ldots$ que começa no estado G tal que $\mathfrak{M}, \pi_k(R) \models \varphi$, para $k \geq 1$;

7. $\mathfrak{M}, G \models \exists[\varphi_1 \mathsf{U} \varphi_2]$ se e somente se existe uma h-execução $h \geq 1$, $R^h = m_1 m_2 \ldots m_h$, tal que $\mathfrak{M}, \pi_h(R^h) \models \varphi_2$ e que, para cada inteiro $1 \leq k < h$, $\mathfrak{M}, \pi_k(R^h) \models \varphi_1$;

8. $\mathfrak{M}, G \models \mathcal{K}_i\varphi$ se e somente se $\mathfrak{M}, G' \models \varphi$ para cada estado global G' tal que, se $\pi_k(R_k) = G$ para algum $k \geq 1$ e R_k^i é a k-execução local em R_k de X^i, então $G' = \pi_k(R'_k)$ para alguma k-execução R'_k que também contenha a k-execução local R_k^i.

Capítulo 3

MAPKAT como um Framework Lógico

O capítulo anterior apresentou o framework MAPKAT e como ele pode suprir a semântica da lógica modal KCTL. Usando uma estrutura inspirada nos grafos de planejamento, ele consegue descrever sistemas dinâmicos de forma adequada.

O presente capítulo demonstra como é possível utilizar esse framework como uma estrutura lógica para permitir o *model-checking* de sistemas multi-agente (SMA). Essa função é executada pelos algoritmos a serem apresentados na seção 3.2. Porém, para melhorar a eficiência dos algoritmos mencionados, algumas pequenas adaptações são necessárias. Essas adaptações são explicadas na próxima seção.

3.1 Do ideal para o concreto

Da forma como foi definido no capítulo 2, o framework MAPKAT toma algumas liberdades para permitir uma melhor caracterização da semântica desejada para a lógica KCTL. Por exemplo, a possibilidade de se lidar com execuções infinitas. Pórém, tais liberdades devem ser estudadas e melhor modeladas para que o framework possa ser efetivo como uma estrutura de *model-checking* e planejamento.

Utilizar um grafo de planejamento (ver seção A.6.3, página 58) como estrutura lógica é a abordagem mais natural. Porém uma estrutura lógica exige mais informações que uma estrutura de planejamento. É por esse motivo que as seguintes tarefas são necessárias:

- Apresentar *observações* sobre o modelo construído;
- Dividir os *nós de proposição* entre *nós positivos* e *nós negativos*;
- Relacionar as *adaptações* do modelo com a *semântica* dos operadores.

Explicar cada um dos itens acima é o foco desta seção.

3.1.1 Observações sobre o modelo

A estrutura básica do framework MAPKAT é fortemente ligada aos grafos de planejamento. Sendo assim, é necessário mencionar algumas de suas propriedades antes de explicar como adaptá-las e utilizá-las. Esta seção irá mencionar o essencial para garantir o entendimento. Para maiores detalhes, consulte a seção A.6.3 ou o artigo original ([Blum and Furst, 1995]).

Um grafo de planejamento é dividido em níveis, cada nível formado exclusivamente por um de dois tipos de nós: nós de proposição ou nós de ação. Cada nó de proposição representa uma proposição do sistema. Analogamente, cada nó de ação representa um operador do sistema.

Um grafo de planejamento alterna níveis de proposição com níveis de ação. Ele começa com um nível de proposição e termina com outro nível de proposição. O primeiro nível de proposição, demarcado PL_0 corresponde ao estado inicial do modelo - ou, seguindo o framework MAPKAT, a verdade global do estado global inicial do Sistema Multi-Agente. Seguindo essa camada vem o primeiro nível de ação, denominado AL_1, composto por todos os operadores aplicáveis a PL_0. Cada nó de operador é ligado aos nós de proposição que representam suas pré-condições.

O modelo é então construído iterativamente, camada a camada. Para $k \geq 1$, AL_k é examinada para se determinar os pares de nós que representam operadores mutuamente exclusivos. Então PL_k é formado pela união $\mathrm{PL}_{k-1} \cup \mathrm{ADD}_k$, onde ADD_k é a união das *Add-Lists* de todos os operadores pertencentes a AL_k. Cada nó de AL_k é ligado, com as chamadas *arestas de Add-effect*, aos nós de PL_k que representam os elementos da Add-list do operador. Similarmente, cada nó de AL_k é ligado aos nós de PL_k que representam os elementos da Del-list do operador, com as chamadas *arestas de Del-effect.*

PL_k deve então sofrer uma varredura imediata para se demarcar os pares de elementos que representam proposições mutuamente exclusivas - isto é, as proposições que obrigam a reordenação de ações. Em seguida, uma nova camada AL_{k+1} é definida, composta por todos os operadores aplicáveis a PL_k, começando uma nova iteração para $k+1$[1].

Eventualmente o modelo chegará a um nível de proposição PL_t na qual todas as proposições do sistema estão representadas. O processo continua até que se encontre um nível PL_f, $f > t$, com os mesmos pares de proposições mutuamente exclusivas que PL_{f-1}. Nesse ponto, a construção do modelo se encerra e o valor f é considerado o *comprimento do modelo.*

O motivo para esse encerramento vem do fato que o grafo *nivelou.* Isto é, ele alcançou uma profundidade tal que qualquer $\mathrm{PL}_f = \mathrm{PL}_x, f < x$. Consequentemente, $\mathrm{AL}_f = \mathrm{AL}_x, f < x$. Ou seja, um modelo maior não incluirá mais informações sobre a evolução do sistema do que as já obtidas.

Um modelo lógico MAPKAT usa um processo de construção similar. Porém, ele difere na presença de *nós positivos* e *negativos* (ver a seção seguinte) e nos critérios de parada: a construção de um modelo MAPKAT somente se encerra quando no

[1]Note que em nenhum momento são utilizados os Del-lists para REMOVER proposições. Isso se deve ao fato que estão sendo modelados todas as proposições que podem ser verdadeiras a cada momento. Como execuções podem incluir operadores nulos, qualquer proposição anterior pode ser verdadeira a qualquer momento futuro.

modelo nivela. Porém, isso não significa que estaremos restringindo nosso sistema a execuções de comprimento f ou menor. No caso de uma execução de comprimento $k > f$, basta considerar $\mathrm{PL}_x = \mathrm{PL}_f$ e $\mathrm{AL}_x = \mathrm{AL}_f$, para cada $f < x \leq k$.

3.1.2 Nós Positivos e Nós Negativos

Os algoritmos de verificação utilizam princípios de planejamento para acelerar a descoberta de soluções. Esses princípios incluem a definição de "objetivos" adequados. Muitas fórmulas exigem não só que alguns fatos sejam verdadeiros, mas também que outros não o sejam. Para representar esse fato adequadamente e de forma genérica, incluiu-se o conceito de *nós positivos* e *nós negativos*.

Os nós positivos são utilizados de forma igual aos nós de proposição dos grafos de planejamento. Cada um deles significa que a proposição representada é verdadeira. Eles são inseridos no modelo quando suas proposições aparecem na Add-List de um operador. São ligados por arestas de Add-Effect aos operadores que contenham suas proposições na Add-list e por arestas de Del-Effect aos operadores que contenham suas proposições na Del-list.

Os nós negativos são utilizados de forma inversa. Cada um deles significa que a proposição representada é falsa. Eles são inseridos no modelo quando suas proposições aparecem na Del-List de algum operador. São ligados por arestas de Add-Effect aos operadores que contenham suas proposições na Del-list e por arestas de Del-Effect aos operadores que contenham suas proposições na Add-list.

Obviamente, o nó positivo relacionado a uma proposição e o nó negativo relacionado à mesma proposição sempre são mutuamente exclusivos.

3.1.3 Reflexos das Adaptações

Devido à relação com grafos de planejamento, MAPKAT explora o princípio de busca por objetivos. Para cada fórmula é calculado um conjunto de soluções, cada solução composta de uma execução que leva a um estado que satisfaça a fórmula e por um conjunto de proposições que precisam ser verdadeiras ou falsas para a execução ser aplicável. Os algoritmos de todos os operadores informam algum conjunto de proposições necessárias. Os operadores temporais utilizam os conjuntos de proposições necessárias como objetivos dos algoritmos de planejamento para produzirem as execuções de suas respostas.

Devido às adaptações apresentadas e à natureza dos algoritmos utilizados, há uma pequena modificação na semântica de dois operadores: o operador de negação "$\neg$" e o operador epistêmico "$\mathcal{K}$".

Idealmente, o operador de negação procuraria pelo conjunto de estados onde uma dada fórmula não fosse satisfeita. Infelizmente, uma busca direta dessas alcançaria complexidade exponencial. A solução adotada é o motivador da existência de nós positivos e negativos: o algoritmo simplesmente calcula os conjuntos de nós positivos e negativos necessários para negar uma determinada fórmula e os apresenta como sua solução.

O operador epistêmico "$\mathcal{K}$" teve sua semântica alterada. A semântica original exige uma busca iterativa entre todas as runs que incluam uma determinada run local. Devido à natureza dos algoritmos utilizados, que produzem "objetivos" para acelerar o processo, essa semântica não é compatível.

Ao invés disso, o operador "$\mathcal{K}$" exige que um conjunto de proposições seja feito "*evidente*": MAPKAT supõe que todos os agentes conhecem as "regras do jogo", ou seja, quais as pré-condições e os efeitos de todos os operadores do sistema, e que nenhuma ação é "executada em segredo", ou seja todos sabem quando um determinado operador é aplicado. Deixar uma proposição em evidência tão somente significa exigir que um operador que torne tal proposição evidente seja aplicado, e que nenhum operador que negue tal proposição seja executado mais tarde. Seguindo essas regras, o operador alcança um subconjunto correto da semântica original.

3.2 Verificação de Fórmulas

Essa seção descreve os fundamentos dos algoritmos utilizados para a verificação de fórmulas KCTL no framework MAPKAT. Para maiores detalhes, favor consultar o apêndice D, página 107.

Os algoritmos de verificação calculam os requisitos necessários para o modelo de um dado Sistema Multi-Agente (SMA) satisfazer uma dada fórmula KCTL φ. Ele o faz calculando as pré-condições de φ e verificando se o estado global inicial satisfaz essas pré-condições.

As pré-condições de uma fórmula φ são as proposições necessárias para satisfazer φ. Se φ for da forma $T\psi$, onde T se trata de um operador temporal, a execução que leva o modelo ao estado que satisfaz a ψ é retornada e as pré-condições dessa execução se tornam as pré-condições de φ. Caso contrário, nenhuma execução é necessária, sendo retornada uma execução nula junto dos pré-requisitos de φ. A tupla formada pelo conjunto de proposições necessárias e a execução associada é chamada de *solução*.

Uma solução é calculada detectando-se o que é necessário para satisfazer as porções atômicas (ou "mais internas") da fórmula e então, passo-a-passo, calcula-se o que é necessário para satisfazer os operadores gradativamente mais externos, combinando-se as soluções até se alcançar a solução da fórmula como um todo.

Em outras palavras, cada passo do processo calcula o conjunto de soluções possíveis para um operador da fórmula. Cada solução é uma *possibilidade* e tem a forma de uma tupla que inclui um conjunto de proposições necessárias e uma execução necessária. Pode-se considerar o conjunto de proposições de cada resposta como uma conjunção booleana, e o conjunto de respostas alternativas como uma disjunção booleana.

O primeiro passo é baseado no estado inicial do sistema dado pela configuração do modelo. Esse é o primeiro nível de proposição, chamado PL_0. Quando o conjunto de soluções possíveis de φ como um todo é calculado, retornamos apenas as soluções cujos conjuntos de proposições necessárias estiverem contidos em PL_0.

O *model-checking* de operadores temporais toma o conjunto de pré-condições de cada solução da fórmula interna e o utiliza como o conjunto de objetivos de um algoritmo de planejamento adaptado do algoritmo original dos grafos de planejamento. Esse algoritmo toma dois níveis de proposições - o atual (origem) e o seguinte (destino). Se o algoritmo não encontrar ao menos uma execução entre a origem e o destino que sirva como resposta, tenta-se de novo tomando como novo destino o nível de proposição seguinte ao destino anterior, e de novo com o seguinte se necessário, desistindo apenas se o algoritmo tiver tentado com um número de níveis igual ao comprimento do modelo.

Operadores booleanos são calculados no nível de proposição corrente. Suas soluções possíveis envolvem apenas as proposições para esse nível e não manipulam execuções de forma alguma.

O operador epistêmico segue uma semântica adaptada, conforme explicado na seção 3.1.3. Essa semântica envolve o uso de *evidências*, proposições específicas que são tratadas com mais cuidado. Quando o processo examina uma fórmula da forma "$\mathcal{K}_i\psi$", ele verifica, para cada resposta de ψ, as proposições necessárias que pertencem ao conjunto exclusivo do agente X^i. Essas figurarão na resposta de $\mathcal{K}_i\psi$ correspondente como proposições necessárias normais. As outras proposições da resposta de ψ serão tratadas como "proposições que exigem evidências" e, portanto, colocadas em um conjunto especial.

Quando uma resposta de uma fórmula epistêmica é avaliada por um operador temporal, ambos os conjuntos de proposições, o normal e o especial, são tomados como objetivos. Porém, as proposições do conjunto especial não poderão figurar nas pré-condições das respostas fornecidas, que terão o conjunto especial esvaziado.

Quando uma resposta de uma fórmula epistêmica é avaliada por operadores booleanos, os elementos do conjunto especial são tratados como proposições normais, mas ainda não são misturados com os do conjunto normal e nem são retirados do conjunto especial.

Se, no final do processo, uma resposta tiver algum elemento no conjunto especial no ato de avaliação em PL_0, ela é descartada: essa resposta não permitiu ao modelo suprir o agente X^i com todas as evidências necessárias para saber ψ.

Resumindo, Cada passo corresponde ao tratamento de um operador lógico e é sempre baseado em algum nível de proposição do SMA. A resposta de cada passo é sempre um conjunto de tuplas (C, E, r) tais que:

- C e E são conjuntos de proposições e r é uma execução;
- C é o conjunto de pré-condições de r;
- E é o conjunto de evidências requeridas para que algum agente tenha o conhecimento requerido pela fórmula sendo verificada.

Mais detalhes, incluindo todos os algoritmos envolvidos, são fornecidos no apêndice D.2.

Por fim, resta o fato de que um modelo MAPKAT pode ser utilizado para planejamento automático. De fato, algoritmos de planejamento foram adaptados para apoiarem o processo de verificação de fórmulas de lógica modal. O problema de "encontrar um plano que torne verdadeiro um conjunto de proposições" é simplesmente outra formulação para a semântica do operador temporal "$\exists\Diamond\phi$", onde ϕ é uma fórmula conjunção de proposições "$p \wedge q \wedge \ldots \wedge v$".

Porém, quando utilizamos o operador $\exists\Diamond$ do framework MAPKAT para solucionar problemas de planejamento, somos capazes de especificar objetivos com formas diferentes de conjunções de proposições: de fato, qualquer propriedade representável por um fórmula da lógica KCTL pode ser especificada como o objetivo do planejamento, provando que o framework MAPKAT expande o poder expressivo dessas estruturas.

Capítulo 4

Conclusão

4.1 Contribuição

Este texto apresentou um estudo da área de planejamento multi-agente para a expansão da lógica **KCTL**, cuja versão original é apoiada por modelos Kripke estendidos (e autômatos correspondentes) conforme introduzida por [Benevides et al., 2004]. Sua principal contribuição foi o **framework MAPKAT**, desenvolvido para reforçar o valor semântico para a lógica **KCTL** Como estrutura básica, o *framework* **MAPKAT** substitui os modelos Kripke extendidos por adaptações dos grafos de planejamento apresentados por [Blum and Furst, 1995].

A maior vantagem dessa substituição é uma representação mais adequada da natureza dinâmica e paralelizada de um sistema multi-agente. Essa abordagem permite modelos e respostas mais compactos pois, entre outros motivos, o *framework* **MAPKAT** se baseia não em sequências de ações mas em sequências de *moves.*

O poder desse novo *framework* foi ainda mais reforçado ao se apresentar algoritmos que permitem o *model-checking* da lógica **KCTL** no modelo semântico em si. Isso permite aplicá-lo a uma função similar aos autômatos usados pelo **KCTL** original. Em outras palavras, o *framework* semântico mostrou também ser um *framework* lógico.

A vantagem dessa característica é trivial: não é necessária uma etapa de tradução, diminuindo não só o processo de *model-checking* em si como também diminuindo as chances de erros e confusões. Os algoritmos podem ser muito mais próximos à semântica exata de cada operador. Finalmente, por serem aplicados a grafos de planejamento, os algoritmos se beneficiam dos mesmos princípios que os métodos de planejamento originais, podendo chegar a índices de eficiência similares. É importante lembrar que grafos de planejamento são conhecidos por suas respostas rápidas.

Uma última vantagem é a capacidade que **MAPKAT** dá à lógica **KCTL** de trabalhar em domínios *STRIPS-like*. Por se basear em grafos de planejamento adaptados, **MAPKAT** pode lidar com domínios *STRIPS-like* da mesma forma que *Graphplan*.

4.2 Trabalhos Futuros

Não foi possível desenvolver uma implementação da lógica **KCTL** usando o *framework* **MAPKAT**. Essa tarefa é a primeira dentre as tarefas propostas para ser possível analisar a eficiência real do *framework* descrito.

A versão do MAPKAT apresentado nesse trabalho lida com o caso geral e genérico de sistemas assíncronos. O estudo de variantes que incluam sincronicidade e outras políticas de funcionamento figura como outra tarefa importante, visto que o caso genérico não inclui mecanismos simples para tal adaptação direta.

Appendix A

Planning

The roots of AI planning lie partly in problem solving through state-space search (and associated techniques such as problem reduction and means-end analysis) and partly in theorem proving and situation calculus [Russel and Norvig, 1995].

Planning agents (or *planners*) differ from problem-solving agents in the representations of goals, states and actions. Inspired by situation calculus, planners use explicit, logical representations, so they can direct their reasoning much more sensibly.

Unfortunately, situation calculus planning using a general-purpose theorem prover is very inefficient. Using a restricted language and special-purpose algorithms, though, allows planning to solve quite complex problems.

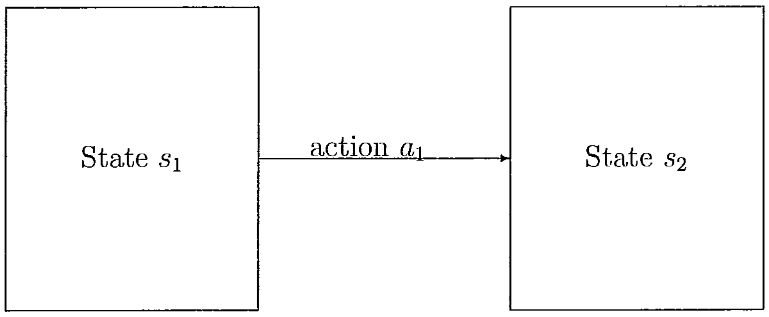
However, before we can debate on an adequate language for the planning agents and their plans, we need to define the scope of their reasoning - in other words, this section will start by addressing the *planning problem.* The discussion on languages, or *planning representation*, comes in the next subsection.

A.1 Planning Problem

The aim of this section is not only to provide a characterization of the classical planning problem, but also to introduce some concepts related to it.

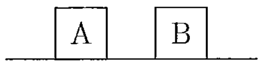
A mini-world or simply a world is the part of the universe being modeled, as for example a table with some wood blocks on it.

A *state* is a description of the world in a formal language at a point of time. In general, there is an infinite number of states in which the world could be, but at a certain time, it can only be in one state. An *action* is an operator that when performed changes the state of the world. The only way of changing the world from one state to another is by performing an action, otherwise it remains static.

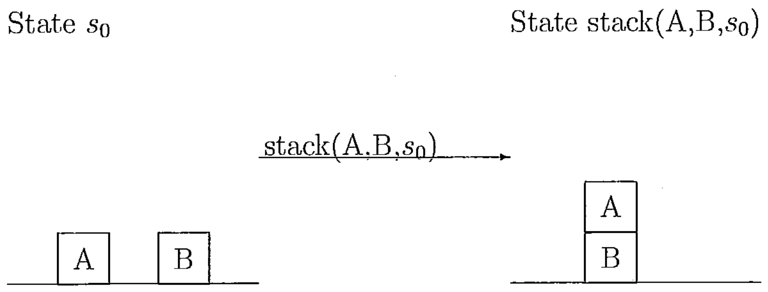


In order to illustrate these concepts, suppose the world consists of a table with two labeled wood blocks on it. In the initial state both blocks are on the table.

State s_0



If the action of stacking block A on top of block B is performed, then the world will change from the state s_0 to a state s_1, as shown in the figure below.



An action a can be characterized as a set of ordered pairs of the form (cs, ns), where cs is the current state and ns is the next state to be reached, if the action a is performed in the current state cs. In this work, actions are supposed to be deterministic, meaning that if a is an action, then for every state cs there is at most one state ns such that $(cs, ns) \in a$. If action a is performed in state cs, it will always change the state of the world to ns.

In the classical planning problem, the aim is to find a sequence of actions (a plan) that, when performed sequentially, changes the state of the world from an initial state to a desired goal state. A more formal and exact definition would be as follows:

Definition A.1.1 Classical Planning Problem

A classical planning problem *is a quadruple* $< S, A, s_0, S_g >$, *where:*

1. S is the set of all possible states;

2. A is the set of actions;

3. s_0 is a special element of S called the initial state;

4. S_g is a subset of S called the set of possible goal states;

The reason why there is more than one goal state is because, in general, the goal is to reach a final state where a desired property holds. It could be any state reachable from the initial one where this property is satisfied.

A plan, or a solution to a planning problem, is a sequence of actions $\langle a_1, a_2, \ldots, a_n\rangle$ such that $\langle s_0, s_1\rangle \in a_1, \langle s_1, s_2\rangle \in a_2, \ldots, \langle s_{n-1}, s_n\rangle \in a_n$ and $s_n \in S_g$. It means that a plan is a sequence of actions that when performed, one after another, changes the state of the world from the initial one to one of the goal states.

A.2 Planning Problem Representation

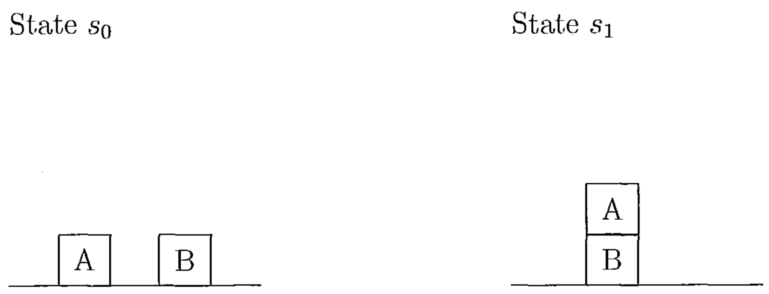
In this section, four ways of interest to represent the planning problem are presented. Three of them are classical, common approaches. The fourth one, more recent, is presented last for it's very close to one of the first three classic approaches, although it does have significant differences.

The first one is a classical first order logic representation (*Situation Calculus*). The second approach presented is a modal action logic (*Dynamic Logic*) representation. The third one is a well-known language (*STRIPS*) used to represent the planning problem. And the fourth representation is a graph-based representation (*Graphplan*).

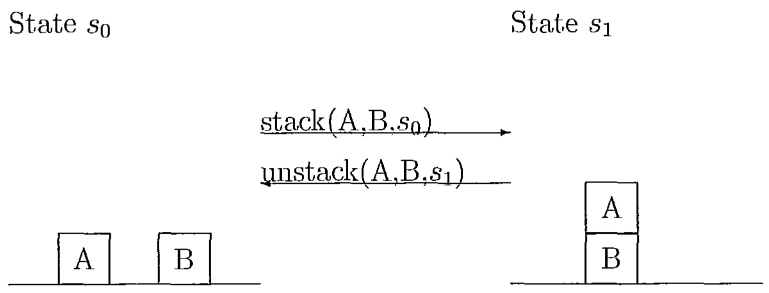
The first two methods are based on logical systems, therefore with precise semantics. The third one is not a logical system, although its syntax is very similar to a logical language. But, as section A.5 will show, its lack of semantics can sometimes lead to wrong results. On the other hand, the first two are in most cases inefficient, due to the so-called frame problem, described in sections A.3 and A.4. STRIPS can deal quite well computationally with this problem and therefore be more efficient.

Thus the interest on the fourth method: Graphplan uses several simple rules to reach even better performance than STRIPS, while holding strong semantics based on causality.

In order to illustrate each method, the Blocks World problem is used. In this example, the world is supposed to be a table with labeled wood blocks on it. All blocks have the same size, and each one can be either on the table or on top of just one other block. Each configuration of the blocks corresponds to a different state of the Blocks World.



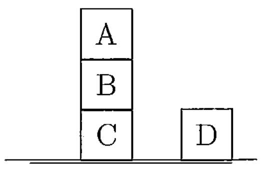
There are two actions that can be performed in this example. The first one is the action of stacking a block on top of another. And the second is the action of "unstacking"a block from top of another. The "unstacked"block is put on the table.



A.3 First Order Logic Representation: Situation Calculus

First order logic has been widely used in the representation of the planning problem. The most successful of these representations has been the one proposed by MacCarthy, which is called the *Situation Calculus*.

In Situation Calculus, *predicates* are used to represent properties about the states. In the block world example, a predicate $On(x, y, s)$ can be used to express the information that a certain block x is on top of another block y in the state s. The predicate $Clear(x, s)$ can be used to describe that there is no block on top of a block x in the state s. Finally, the predicate $Table(x, s)$ can be used to assert that a block x is on the table in the state s. Below is a figure showing a possible state of the problem, followed by a corresponding state representation in situation calculus:



$On(A, B, S1)$

$Clear(A, S1)$

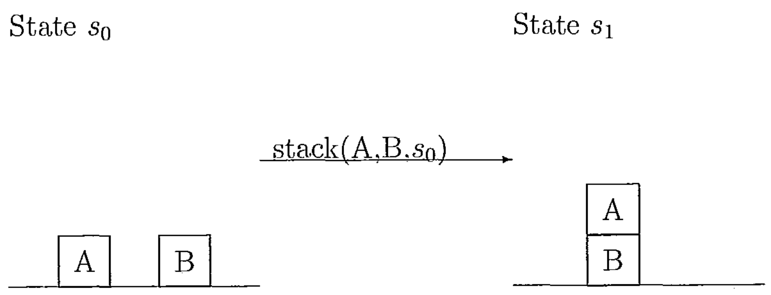
$On(B, C, S1)$

$Table(C, S1)$

$Table(D, S1)$

$Clear(D, S1)$

Actions, in Situation Calculus, are represented by functions that map states into states, i.e., $a : S \rightarrow S$, where S is the set of all states. In the block world example, the action of stacking a block x on top of a block y in a state s can be represented by a function $stack(x, y, s)$. In the same way, the action of unstacking a block x from the top of a block y in a state s, can be represented by a function $unstack(x, y, s)$. The following figure illustrates the representation of an action in Situation Calculus.



$Clear(B, s_0)$
$Clear(A, s_0)$
$Table(B, s_0)$
$Table(A, s_0)$

$Clear(A, s_1)$
$Table(B, s_1)$
$On(A, B, s_1)$

The formalization chosen to present the first order logic follows the classical style of natural deduction. The symbol $\neq$ is assumed to be defined as $\neg =$ (not equal). The axioms are presented in three groups.

The first group contains the descriptions of *properties* that are true in all states. For instance, the predicate $Free(x, s)$ is defined to assert that a block x is *Clear* and on the *Table* in the state s. This definition must hold in all states.

Ax.1 $\forall x \forall y \forall s(Table(x, s) \leftrightarrow \neg\exists y(On(x, y, s)))$

Ax.2 $\forall x \forall y \forall s(Clear(y, s) \leftrightarrow \neg\exists y(On(x, y, s)))$

Ax.3 $\forall x \forall y \forall z \forall s(On(x, y, s) \rightarrow (On(x, z, s) \leftrightarrow z = y))$

Ax.4 $\forall x \forall s(Clear(x, s) \wedge Table(x, s) \rightarrow Free(x, s))$

The axiom Ax.1 asserts that a block is on the table if and only if it is not on top of another block. Ax.2 asserts that a block is clear if and only if there is no other block on top of it. Ax.3 asserts that a block can be on top of just one block. Ax.4 asserts that a block is free only if it is clear and on the table.

The second group contains the axioms that define under what conditions (*preconditions*) one action can be performed and how it affects the state of the world.

Ax.5 $\forall x \forall y \forall s(Table(x, s) \wedge Clear(x, s) \wedge Clear(y, s) \wedge (x \neq y) \Rightarrow$

$On(x, y, stack(x, y, s)))$

The axiom Ax.5 describes the action *stack*. If a block x is on the table, blocks x and y are clear and x is different from y in the state s, then after stacking x on top of y it is true that x is on y in this new state.

Ax.6 $\forall x \forall y \forall s(On(x, y, s) \wedge Clear(x, s) \Rightarrow$
$Table(x, unstack(x, y, s)) \wedge Clear(x, unstack(x, y, s)))$

The axiom Ax.6 describes the action *unstack*. If a block x is on top of a block y and x is clear in the state s, then, after unstacking x from top of y, x is on the table and y is clear.

The third group contains the so-called *frame axioms*. It describes which properties are not affected when a certain action is performed. The axioms of the second group describe how one action changes the state of the world. But, they only say what is changed. They do not show how the properties that were not affected can pass from the current state to the next one when one action is executed. A group of frame axioms must be provided for each action.

The frame axioms of the action *stack* are:

Ax.7 $\forall x \forall y \forall u \forall s(Clear(u, s) \wedge (u \neq y) \Rightarrow Clear(u, stack(x, y, s)))$
Ax.8 $\forall x \forall y \forall u \forall s(Table(u, s) \wedge (u \neq x) \Rightarrow Table(u, stack(x, y, s)))$
Ax.9 $\forall x \forall y \forall u \forall v \forall s(On(u, v, s) \Rightarrow On(u, v, stack(x, y, s)))$

Axioms Ax.7, Ax.8 and Ax.9 assert that if a block u is clear, or on the table, or on top of a block v, and if we stack a block x on top of a block y, then u remains in the same situation that it was before, i.e., clear or on the table or on top of v, respectively.
The frame axioms of the action *unstack* are:

Ax.10 $\forall x \forall y \forall u \forall s(Clear(u, s) \Rightarrow Clear(u, unstack(x, y, s)))$
Ax.11 $\forall x \forall y \forall u \forall s(Table(u, s) \Rightarrow Table(u, unstack(x, y, s)))$

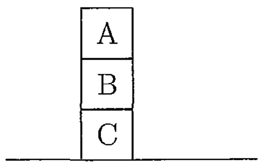
Ax.12 $\forall x \forall y \forall u \forall v \forall s (On(u, v, s) \wedge (u \neq x) \Rightarrow On(u, v, stack(x, y, s)))$

Axioms Ax.10, Ax.11 and Ax.12 assert that if a block u is clear, or on the table, or on top of a block v, and if we unstack a block x from the top of a block y, u remains in the same situation that it was before, i.e., clear or on the table or on the top of v, respectively.

The number of frame axioms is proportional to the product of the number of state predicates and the number of actions. In this case, there are three state predicates, two actions and six frame axioms.

Suppose that the Block World is in an initial state s_0, where there are three blocks on the table. Block C is on the table, block B is on top of block C, and block A is on top of block B. The goal is to reach a state where block B is free. The figure below shows the initial state.

State s_0



The description of the initial state shown above is given by the facts below:

F.1 On(A,B,s_0)

F.2 Clear(A,s_0)

F.3 On(B,C,s_0)

F.4 Table(C,s_0)

The aim is to find a proof of the formula $Free(B, s)$. This can be obtained as follows:

If we apply the axiom Ax.6, which defines the action unstack, and use the facts F1 and F2, we have:

1.$Table(A, unstack(A, B, s_0))$

1′.$Clear(B, unstack(A, B, s_0))$

If we use the frame axiom Ax.12 and the fact F3, we have:

2. $On(B, C, unstack(A, B, s_0))$

If we use the axiom Ax.6 again, 2 and 1′,we have:

3. $Table(C, unstack(B, C, unstack(A, B, s_0)))$

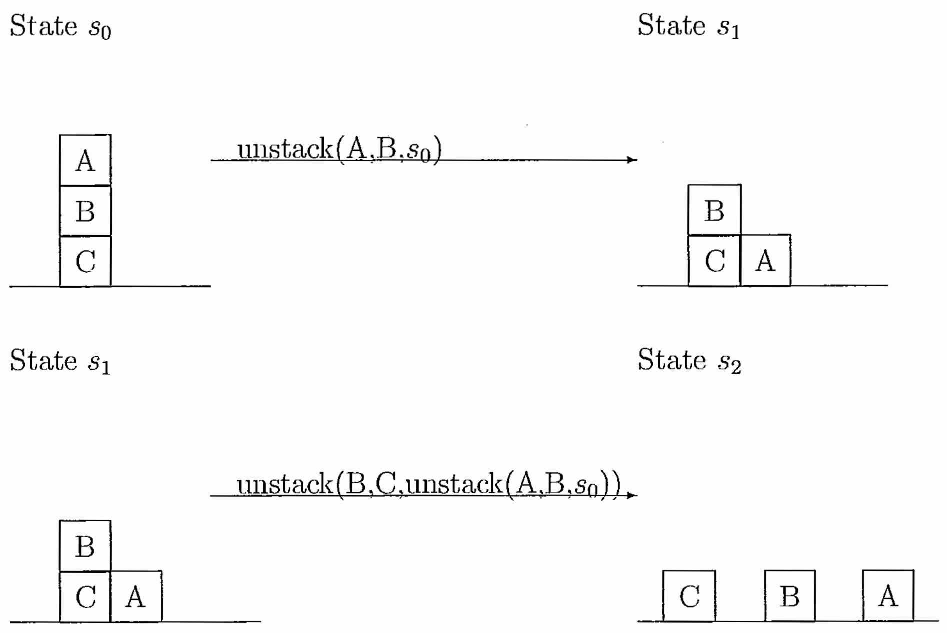
3′. $Clear(C, unstack(B, C, unstack(A, B, s_0)))$

If we use the axiom Ax.4, we have:

4. $Free(B, unstack(B, C, unstack(A, B, s_0)))$

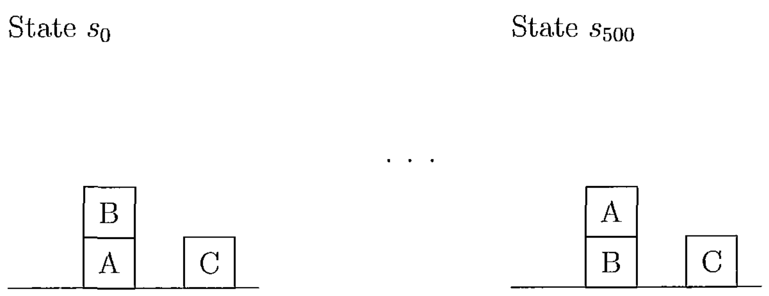
The proof above is actually an informal explanation of the proof.

The formula 4 is the goal. A plan has been constructed to reach a goal state where block B is free. The plan is the term $unstack(B, C, unstack(A, B, s_0))$. If this plan is executed in the initial state s_0 described above, the state of the world will change to a goal state, where block B is free. The following figure illustrates this situation.



Representing the planning problem in Situation Calculus has some good advantages. First, it is based on a logical system, i.e., first order logic, which has a well-known semantics. Second, there is a good range of theorem provers implemented and available, that could be used to generate plans.

On the other hand, the number of frame axioms, in general, can be quite big. All frame properties, i.e., properties that are not affected by an action, must be deduced using the frame axioms. It makes proofs very long, and consequently, the plan generation can be very inefficient. This is the so-called *frame problem*. The figure below illustrates this problem.



Suppose that the Block World is in the initial state s_0, as shown in the figure above. After five hundred changes of state, block C was not involved in any action. If any information about block C is needed in the state s_{500}, it must be obtained from the initial state s_0, using frame axioms. It can be necessary to apply some of the frame axioms hundreds of times in order to obtain simple information like 'block C is on the table in the state s_{500}', i.e., $Table(C, s_{500})$. It can be a serious problem in terms of efficiency. There are some partial solutions to this problem which can reduce the number of frame axioms, but do not eliminate the problem.

A.4 Modal Action Logic Representation

Modal Action Logic or Dynamic Logic is a modal logic, where the modal operators of necessity $\Box$ and possibility $\Diamond$ are replaced by a family of operators $[a]$ and $< a >$

respectively, where a is an action. For each action a, there are two modal operators $[a]$ and $< a >$. The semantics for this logic is a possible world semantics, where for each action there exists an accessibility relation.

It seems quite intuitive to apply dynamic logic to planning problem representation. Actions are represented by means of the modal operator $[a]$ and states by possible worlds. A formula $[a]\alpha$ holds in a state s, if α holds in all possible states which could result from the action a being performed in s.

For notational simplicity, sequences of actions $c = a_1a_2\ldots$ are also represented by means of modal operators $[c]$ and $< c >$. Analogously, a formula $[c]\alpha$ holds in a state s, if α holds in all possible states which could result from the actions in c being performed as ordered since state s.

The logical axioms are:

At.1 $\vdash \alpha \rightarrow (\beta \rightarrow \alpha)$

At.2 $\vdash (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$

At.3 $\vdash (\neg\beta \rightarrow \neg\alpha) \rightarrow ((\neg\beta \rightarrow \alpha) \rightarrow \beta)$

At.4 $\vdash (\forall x(\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \forall x\beta)$, where x is not free in α

At.5 $\vdash (\forall x\alpha(x)) \rightarrow \alpha(t)$, where t is not free for x in α

Ac.1 $\vdash [c,a]\mathbf{TRUE}$

Ac.2 $\vdash ([c,a](\alpha \rightarrow \beta)) \rightarrow ([c,a]\alpha \rightarrow [c,a]\beta)$

Ac.3 $\vdash \forall x([c,a]\alpha) \leftrightarrow [c,a]\forall x\alpha$, where x is not free in c or in a

Ac.4 $\vdash ([c,a]\neg\alpha) \rightarrow \neg([c,a]\alpha)$

Ac.5 $\vdash \exists x([c,a]\alpha) \rightarrow [c,a]\exists x\alpha$, where x is not free in c or in a

Ac.6 $\vdash ([c,a]\alpha) \wedge ([c,a]\beta) \leftrightarrow [c,a](\alpha \wedge \beta)$

Ac.7 $\vdash ([c,a]\alpha) \vee ([c,a]\beta) \rightarrow [c,a](\alpha \vee \beta)$

Definitions of other logical connectives and the quantifier $\exists$ are:

D.1 $\alpha \vee \beta \cong (\neg\alpha) \rightarrow \beta$

D.2 $\alpha \wedge \beta \cong \neg(\alpha \rightarrow \neg\beta)$

D.3 $\alpha \leftrightarrow \beta \cong (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

D.4 $\exists x\alpha \cong \neg\forall x \neg\alpha$

Rules of Inference:

$$\text{MP} : \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \alpha \rightarrow \beta}{\Gamma \vdash \beta}$$

$$\text{GEN} : \frac{\Gamma \vdash \alpha}{\Gamma \vdash \forall x\alpha}$$

$$\text{NEC} : \frac{\alpha}{[c, a]\alpha}$$

In the modal action logic representation of the planning problem, actions are represented by modal operators of the form $[a]$. Predicates are used to describe properties about the states. But, there is no state component as argument of each predicate. The modal operator of a formula indicates in which state this formula is. For instance, a formula $Table(A)$ means that the block A is on the table in the initial state, and $[unstack(A, B)]Table(A)$ means that A is on the table, in the state where A is unstacked from the top of B, i.e., after unstacking block A from the top of block B, block A will be on the table.

A Block World representation in modal action logic is presented below. As in the situation calculus version, there are three groups of axioms.

Ax.1 $\forall x\forall y(Table(x) \leftrightarrow \neg\exists y(On(x, y)))$

Ax.2 $\forall x\forall y(Clear(y) \leftrightarrow \neg\exists y(On(x, y)))$

Ax.3 $\forall x\forall y\forall z(On(x, y) \rightarrow (On(x, z) \leftrightarrow x = y))$

Ax.4 $\forall x(Clear(x) \wedge Table(x) \rightarrow Free(x))$

The axiom Ax.1 asserts that a block is on the table if and only if it is not on top of another block. Ax.2 asserts that a block is clear if and only if there is no other block on top of it. Ax.3 asserts that a block can be on top of only one block. Ax.4 asserts that a block is free if it is clear and on the table.

The second group contains the axioms which define each action, i.e., under what conditions an action can be performed and how it affects the state of the world.

Ax.5 $\forall x \forall y(Table(x) \wedge Clear(x) \wedge Clear(y) \wedge x \neq y \rightarrow$
$[stack(x,y)]On(x,y))$

The axiom Ax.5 describes the action *stack*. If a block x is on the table, blocks x and y are clear and x is different from y in the state s, then after stacking x on top of y it is true that x is on top of y in this new state.

Ax.6 $\forall x \forall y(On(x,y) \wedge Clear(x) \rightarrow [unstack(x,y)](Table(x) \wedge Clear(y)))$

The axiom Ax.6 describes the action *unstack*. If a block x is on top of a block y and x is clear in the state s, then, after unstacking x from the top of y, x is on the table and y is clear.

The third group contains the frame axioms. As in the Situation Calculus representation, they describe which properties are not affected when an action is performed.

The frame axioms for the action *stack* are:

Ax.7 $\forall x \forall y \forall u(Clear(u) \wedge u \neq y \rightarrow [stack(x,y)]Clear(u))$
Ax.8 $\forall x \forall y \forall u(Table(u) \wedge u \neq x \rightarrow [stack(x,y)]Table(u))$
Ax.9 $\forall x \forall y \forall u \forall v(On(u,v) \rightarrow [stack(x,y)]On(u,v))$

Axioms Ax.7, Ax.8 and Ax.9 assert that if a block u is clear, or on the table, or on top of a block v, and if we stack a block x on top of a block y, then u remains in the same situation that it was before, i.e., clear or on the table or on top of v, respectively.

The frame axioms for the action *unstack* are:

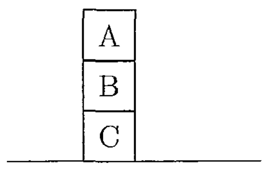
Ax.10 $\forall x \forall y \forall u(Clear(u) \rightarrow [unstack(x,y)]Clear(u))$
Ax.11 $\forall x \forall y \forall u(Table(u) \rightarrow [unstack(x,y)]Table(u))$
Ax.12 $\forall x \forall y \forall u \forall v(On(u,v) \wedge u \neq x \rightarrow [unstack(x,y)]On(u,v))$

Axioms Ax.10, Ax.11 and Ax.12 assert that if a block u is clear, or on the table, or on top of a block v, and if we unstack a block x from the top of a block y, u remains in the same situation that it was before, i.e., clear or on the table or on the top of v, respectively.

The number of frame axioms is proportional to the product of the number of state predicates and the number of actions. In this case, there are three state predicates, two actions and six frame axioms.

Suppose that the Block World is in an initial state s_0, where there are three blocks on the table. Block C is on the table, block B is on top of block C, and block A is on top of block B. The goal is to reach a state where block B is free. The figure below shows the initial state.

State s_0



The description of the initial state is given by the facts below:

F.1 On(A,B)
F.2 Clear(A)
F.3 On(B,C)
F.4 Table(C)

The aim is to find a possible world accessible from the initial one such that block B is free, i.e., the formula $Free(B)$ holds. This can be obtained as follows.

If we apply the axiom Ax.6, which defines the action *unstack*, and use the facts F1 and F2, we have the situation S2:

1.$[unstack(A, B)](Table(A) \wedge Clear(B))$

If we use the axiom Ac.6, which gives us a kind of distributivity of modalities over conjunction, we have:

2.$[unstack(A, B)]Table(A)$

3.$[unstack(A, B)]Clear(B)$

If we use the frame axiom Ax.12 and the fact F3, in the situation S2, we have:

4.$[unstack(A, B)]On(B, C)$

We can apply the rule NEC to the axiom Ax.6 and then use axioms Ac.3, Ac.2 and Ac.6.

5.$\forall x \forall y([unstack(A, B)]On(x, y) \wedge [unstack(A, B)]Clear(x)) \rightarrow$
$[unstack(A, B)][unstack(x, y)]Table(x) \wedge$
$[unstack(A, B)][unstack(x, y)]Clear(y)$

If we use 5, 4 and 2,we have the situation S3:

6.$[unstack(A, B)][unstack(B, C)]Table(B)$

7.$[unstack(A, B)][unstack(B, C)]Clear(C)$

We can apply the rule NEC twice to axiom Ax.4 and then use the axioms Ac3, Ac2 and Ac6.

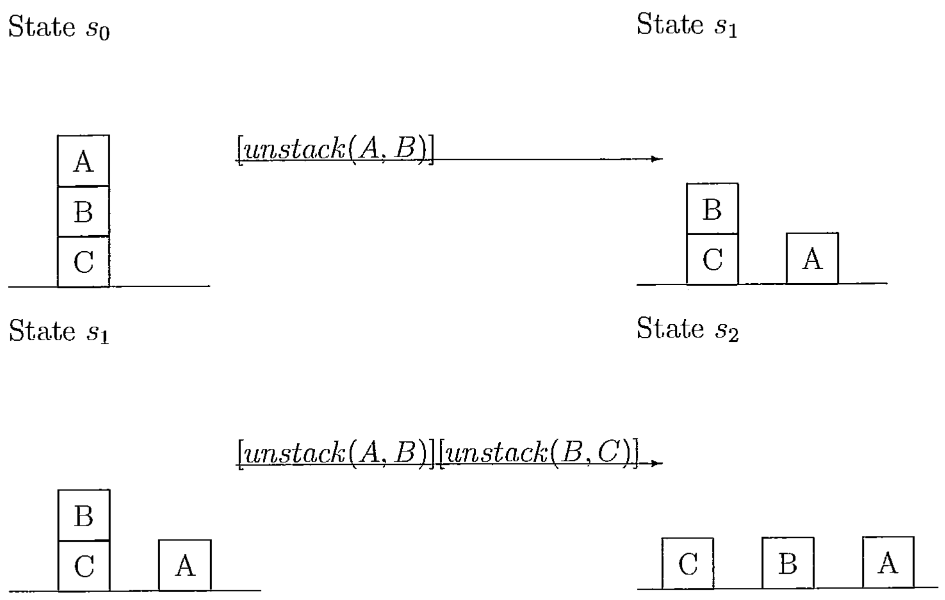
8.$\forall x[unstack(A, B)][unstack(B, C)]Clear(x) \wedge$
$[unstack(A, B)][unstack(B, C)]Table(x) \rightarrow$
$[unstack(A, B)][unstack(B, C)]Free(x)$

If we use 8, we have:

9.$[unstack(A, B)][unstack(B, C)]Free(B)$

The proof above is actually an informal explanation of the proof.

The formula 4 is the goal. A plan has been constructed to reach a goal state where block B is free. The plan is the term $[unstack(A,B)][unstack(B,C)]$. If this plan is executed in the initial state s_0 described above, the state of the world will change to a goal state, where block B is free. The following figure illustrates this situation.



In modal logics (not in modal action logic presented above), the rule of necessitation can only be applied to logical axioms. The reason for that is to make them hold in all possible worlds. In dynamic logic, the situation is different. In most applications, there are some extra-logical axioms which are intended to hold in all possible worlds. The necessitation rule can be applied over these axioms.

In the Block World example, the axioms Ax.1 to Ax.12 must be valid in all possible states, and the necessitation rule is allowed to be applied over them. But, facts F1 to F5 are only valid in the initial world, therefore the necessitation rule cannot be applied over them. In conclusion, in dynamic logic there are two groups of extra-logical axioms: those that are intended to be valid in all possible states, and others that are only valid in the initial state.

Representing the planning problem in modal action logic has some advantages.

First, it is based on a logical system, dynamic logic, which has a well-known semantics. Second, there are some theorem provers that could be used for plan generation. Third, there is not a state component as argument of each predicate, like we have in Situation Calculus. It provides more clarity and elegance in the representation.

The disadvantage is the same as that for the first order logic representation, i.e., the number of frame axioms, in general, is quite big. It can make the plan generation very inefficient.

A.5 STRIPS - Problem Solver

STRIPS is a problem solver proposed by Fikes & Nilson. *States* of the world are represented by world models, which are sets of first order logic formulas. Each *action* is defined by an operator consisting of a description of the effects of the action in the world model, and under which conditions this operator can be performed. A *planning problem* represented in STRIPS consists of an initial world model, a set of available operators and a goal statement. A resolution theorem prover is used to find a sequence of operators that transforms the initial world model into a final model, where the goal is satisfied.

The description of each operator follows the schemata below.

$< operator >$:

PRECONDITION	$< formula >$
ADD LIST	$< list - of - formulas >$
DELETE LIST	$< list - of - formulas >$

The precondition is a formula that must be satisfied before the application of the operator. The ADD LIST is a list of formulas that must be added to the current world model. The DELETE LIST is a list of formulas that are no longer valid and must be deleted.

A Block World representation, in STRIPS, is presented below.

The operators descriptions are:

$stack(x, y)$:

PRECONDITION	$Table(x) \wedge Clear(x) \wedge Clear(x) \wedge x \neq y$
ADD LIST	$On(x, y)$
DELETE LIST	$Clear(y), Table(x)$

$unstack(x, y)$:

PRECONDITION	$Clear(x) \wedge On(x, y)$
ADD LIST	$Clear(y), Table(x)$
DELETE LIST	$On(x, y)$

There is some information that must be valid in all world models. This is expressed by the following axioms:

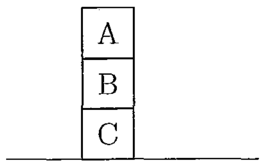
Ax1. $\forall x \forall y (Table(x) \leftrightarrow \neg \exists y (On(x, y)))$
Ax2. $\forall x \forall y (Clear(y) \leftrightarrow \neg \exists x (On(x, y)))$
Ax3. $\forall x \forall y \forall z (On(x, y) \rightarrow (On(x, z) \leftrightarrow z = y))$
Ax4. $\forall x (Clear(x) \wedge Table(x) \rightarrow Free(x))$

The axiom Ax1 asserts that a block is on the table if and only if it is not on top of another block. Ax2 asserts that a block is clear if and only if there is no other block on top of it. Ax3 asserts that a block can be on top of only one block. Ax4 asserts that a block is free if it is clear and on the table.

Suppose that the Block World is in an initial state s_0, where there are three blocks on the table. Block C is on the table, block B is on top of block C, and block

A is on top of block *B*. The goal is to reach a state where block *B* is free. The figure below shows the initial state.

State s_0



The description of the initial state shown above is given by the facts below:

F.1 On(A,B,s_0)

F.2 Clear(A,s_0)

F.3 On(B,C,s_0)

F.4 Table(C,s_0)

The goal is to find a final state where block *B* is free, i.e., formula $Free(B)$. This can be obtained as follows:

Initial state s_0:

$CLEAR(A)$	$ON(A,B)$
$ON(B,C)$	$TABLE(C)$

Applying the operator *unstack*:

State s_1:

$CLEAR(B)$	$TABLE(A)$	-
$ON(B,C)$	$CLEAR(A)$	$TABLE(C)$

Applying the operator *unstack* again:

Final state s_2:

$CLEAR(B)$	$TABLE(A)$	$CLEAR(C)$
$TABLE(B)$	$CLEAR(A)$	$TABLE(C)$

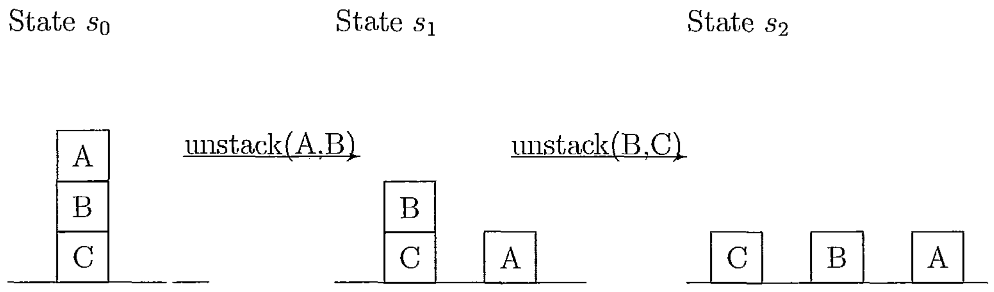
If the operator *unstack* is applied twice, the last world model is obtained. By instantiating the axiom Ax.4, we have:

$(Clear(B) \wedge Table(B)) \rightarrow Free(B)$

Applying the rules $\wedge$-I and modus ponens, the goal is reached.

$Free(B)$

The formula above is the goal. A plan has been constructed to reach a goal state where block B is free. The plan is the term $unstack(B, C, unstack(A, B))$. If this plan is executed in the initial state s_0 described above, the state of the world will change to a goal state, where block B is free. The following figure illustrates this situation.



Semantics

Since Fikes & Nilsson presented STRIPS, in 1971, very few investigations were done in the direction of providing a semantics for STRIPS. Only in 1987 Lifschitz provided a semantical investigation for it. It is very useful, in the sense that it brings out what are the limitations of this problem solver. Quoting from Lifschitz: 'it draws a clear line between *good* and *bad* uses of the language of STRIPS'. The rest of this section is dedicated to presenting Lifschitz's semantical approach to STRIPS.

Let L be a first order language. A *world model* is any set of sentences in L. An operator description is the triple (P, A, D), where P is a formula (*pre-condition*), and A and D are sets of formulas (*add* and *delete lists*).

A *STRIPS system* Σ is a tuple $< M_0, Op, OD >$, where:

M_0 - is the *initial world model*;

Op - a set of *operators symbols*;

OD - a family of *operator descriptions* $\{(P_\alpha, A_\alpha, D_\alpha) | \alpha \in Op\}$;

A *plan* is a sequence of operators $\bar{a} = (a_1, ..., a_n)$.

It is assumed that s_0 is the initial state, and that it is defined for each state s which formulas are satisfied in this state. An action is a partial function f that maps states into states. And for each operator a there is an action f_a. A function f is applicable in a state s, if f is defined in s. An *interpreted STRIPS system* is a STRIPS system with all this additional information.

A world model M of an interpreted STRIPS system Σ is satisfied in a state s, if every element of M is satisfied in s.

Lifschitz proposes and discusses three possible semantics for STRIPS. They are briefly explained below.

a) **First**:

The first step is to give a definition of an operator description. Let Σ be an interpreted STRIPS system $\Sigma = (M_0, \{P_a, A_a, D_a\}_{a \in Op})$

Definition A: An operator description (P, A, D) is *sound relative to an action* f, if for every state s such that P is satisfied in s,

(i) f is applicable in state s,

(ii) every sentence which is satisfied in s and does not belong to D is satisfied in $f(s)$,

(iii) A is satisfied in $f(s)$.

Σ is *sound* if M_0 is satisfied in the initial state s_0, and each operator description (P_a, A_a, D_a) is sound relative to f_a.

Theorem A.5.1 ***(Soundness)***

If Σ is sound, and a plan $\bar{a}$ is accepted by Σ, then the action $f_{\bar{a}}$ is applicable in the initial state s_0, and the world model $R(\bar{a})$ is satisfied in the state $f_{\bar{a}}(s_0)$.

Quoting from Lifschitz:

"There is a serious problem, however, with definition A: it eliminates all usual STRIPS systems as *unsound*."

In the description of the operator $stack(x, y)$, in the example above, the delete list contains the facts $Clear(y)$ and $Table(x)$. But these are not the only formulas that become false when action *stack* is performed. For instance, their conjunction with any other formula will be false, as well as many other formulas. In order to satisfy definition A, the delete list should be infinite.

A solution to this problem could be to allow the world models to have only atomic formulas. The operator description should change: the add list and the delete list should only have atomic formulas. This solution is too restrictive. Most of the practical problems can not be represented in STRIPS if these conditions are imposed.

b) **Second:**

In order to have a more general semantics, which satisfies more STRIPS systems, some modifications are made necessary. First, non-atomic formulas are allowed in the initial state, provided they are valid in all world models. The reason for this can be illustrated by the following example.

Suppose that in the Block World example, in the initial state, all blocks are on the table and clear. It can be expressed by the following formula:

$$\forall x Table(x) \wedge Clear(x)$$

If the operator $stack(A, B)$ is performed, this sentence will not be valid any more. But according to the *stack* operator description, it will not be deleted.

The second modification is to allow the add list to have non-atomic formulas. For the same reason, these formulas must be valid in all world models. Otherwise, in the next change of state, it will lead the STRIPS system to malfunction.

In order to enforce the modification proposed above, definition A must be rewritten to definition B as follows.

Definition B: An operator description (P, A, D) is sound relative to an action f, if for every state s such that P is satisfied in s,

(i) f is applicable in state s,

(ii) every atomic formula which is satisfied in s and does not belong to D is satisfied in $f(s)$,

(iii) A is satisfied in $f(s)$

(iv) every non-atomic formula in A is satisfied in all states of the world.

Σ is sound if

(v) M_0 is satisfied in the initial state s_0,

(vi) every non-atomic formula in M_0 is satisfied in all states of the world,

(vii) every operator description (P_a, A_a, D_a) is sound relative to f_a.

The soundness theorem remains valid for definition B.

c) **Third**:

The semantics presented in b) can be slightly generalized if a set E of essential formulas is defined. The aim is to allow non-atomic formulas to appear in the delete list. Definition B could be changed to definition C as follows:

Definition C: An operator description (P, A, D) is sound relative to an action f, if for every state s such that P is satisfied in s,

(i) f is applicable in state s,

(ii) every essential formula which is satisfied in s and does not belong to D is satisfied in $f(s)$,

(iii) A is satisfied in $f(s)$

(iv) every formula in A, which is not essential, is satisfied in all states of the world.

Σ is sound if

(v) M_0 is satisfied in the initial state s_0,

(vi) every formula in M_0, which is not essential, is satisfied in all states of the world,

(vii) every operator description (P_a, A_a, D_a) is sound relative to f_a.

The soundness theorem remains valid for definition C.

If the set of essential formulas is chosen as the set of atomic formulas, then definition B and C are the same. The set E must be carefully chosen in order to avoid the delete list being infinite, as we have already described.

Representing the planning problem in STRIPS has a very interesting advantage: the frame problem can be avoided. It is not necessary to have axioms to express the properties that are not changed by an action. This is made possible because the operator description contains already the information of how the world is going to be changed, when this operator/action is performed. In general, this can improve the efficiency of the plan generation a lot. On the other hand, STRIPS's lack of semantics can lead in some cases to wrong results. We can only obtain a reasonable semantics to STRIPS, if strong constraints are imposed on it, and only in this case the soundness of the plan generation can be assured.

A.6 Planning Graph Analysis

The text in this section is based on [Blum and Furst, 1995].

The *Graphplan* is a planner presented by Blum & Furst. It introduces a new approach to planning in STRIPS-like domains. This paradigm is based on building and analyzing a compact structure called a *Planning Graph.*

A Planning Graph encodes the planning problem in a way that many useful constraints inherent in the problem become explicitly available, reducing the amount of search needed to reach a conclusion (be it finding a plan or realizing that there isn't such a plan). The Planning Graph is *not* the state-space graph. Unlike a state-space graph, where a plan is a *path* through the graph, in a Planning Graph a plan is a *flow* in the network flow sense.

The Graphplan planner uses the Planning Graph that it creates to guide its search for a plan. The search it performs combines aspects of both total-order and partial-order planners. Like total-order planners, Graphplan makes strong commitments in its search - when it considers an action, it considers it at a specific point in time. On the other hand, like partial-order planners, Graphplan generates partially ordered plans. The semantics of such a plan is that the actions in the same time step may be performed in any desired order - it is a kind of "parallel" plan.

One valuable feature of this approach is that it guarantees it will find the *shortest* partial-order plan. Another interesting feature is that this approach isn't particularly sensitive to the order of the components of the goal in a planning task, unlike traditional approaches.

A.6.1 Definitions and Notations

Planning Graph Analysis applies to STRIPS-like planning domains. These domains are characterized by a set of *operators*. Operators have preconditions, add-effects and delete-effects, all of which are conjuncts of propositions, and have parameters that can be instantiated to objects in the world. Operators do not create or destroy objects, and time may be represented discretely.

Specifically, Graphplan defines a *planning problem* through:

- A set of operators (a STRIPS-like domain);
- A set of objects;
- A set of propositions (literals) called the Initial Conditions;
- A set of Problem Goals which are propositions that are required to be true at the end of a plan.

An *action*, in the Planning Graph approach, is a fully-instantiated operator. For example, the operator `'put ?x into ?y'` may be instantiated to the specific action `'put Object1 into Container2'`. An action taken at time t adds to the world all the propositions which are among its Add-Effects and deletes all the propositions which are among its Delete-Effects. It will be convenient to think of "doing nothing" to a proposition in a time step as a special kind of action we call a *no-op* or *frame* action.

A.6.2 Valid Plans

A *valid plan* for a planning problem consists of a set of actions and specified times in which each is to be carried out. Several actions may be specified to occur at the same time step so long as they do not interfere with each other.

We say two actions *interfere* if one deletes a precondition or an add-effect of the other. In a linear plan, two actions that don't interfere could be arranged in any order with exactly the same outcome. Such actions are called *independent.*

A valid plan may perform an action at time $t > 1$ if the plan makes all its preconditions true at time t. Because no-op actions carry truth forward in time, a proposition can be defined to be true at time $t > 1$ if and only if it is an Add-Effect of some action taken at time $t - 1$.

Finally, a valid plan must make all the Problem Goals true at the final time step.

A.6.3 Planning Graphs

A Planning Graph is similar to a valid plan, but without the requirement that the actions at a given time step must not interfere. It is a type of constraint graph that encodes the planning problem.

A Planning Graph is a directed, leveled graph with two kinds of nodes and three kinds of edges. The levels alternate between *propositions levels* containing *proposition nodes* (each node labeled with some proposition) and *action levels* containing *action nodes* (each one labeled with some instantiated action).

The first level of a Planning Graph is a proposition level and consists of one node for each proposition in the Initial Conditions. The levels in a Planning Graph, from earliest to latest, are: Propositions true at time 1, possible actions at time 1, propositions possibly true at time 2, possible actions at time 2, propositions possibly true at time 3, and so on.

Edges in a Planning Graph explicitly represent relations between actions and propositions. The action nodes in action-level i are connected by "precondition-edges" to their preconditions in proposition level i, by "add-edges" to their Add-Effects in proposition level $i+1$, and by "delete-edges" to their Delete-Effects in proposition-level $i+1$.[1]

The conditions imposed on a Planning graph are much weaker than those imposed on valid plans. Actions may exist at action-level i if their preconditions exist at proposition-level i but there is no requirement of "independence". In particular, action-level i may legally contain *all* the possible actions whose preconditions all exist in proposition-level i.

A proposition may exist at proposition-level $i+1$ if it is an Add-Effect of some action in action-level i, even if it is also a Delete-Effect of some other action in action-level i. Since there are "no-op actions", every proposition that appears in proposition-level i may also appear in proposition-level $i+1$.

A.6.4 Exclusion Relations

An integral part of Planning-Graph Analysis is noticing and propagating certain *mutual exclusion* relations among nodes. Two actions at the same action level in a Planning Graph are *mutually exclusive* if no valid plan could possibly contain both.

[1]A length-two path from an action a at one level, through a proposition Q at the next level, to an action b at the following level is semantically similar to a causal link $a \xrightarrow{Q} b$ in a traditional partial-order planner.

Similarly, two propositions at the same given proposition level are mutually exclusive if no valid plan could possibly make both true at the given level.

Identifying mutual exclusion relationships can be of enormous help in reducing the search for a subgraph of a Planning Graph that might correspond to a valid plan.

The method explained here notices and records mutual exclusion relationships by propagating them through the Planning Graph using a few simple rules. These rules do not guarantee to find all mutual exclusion relationships (a task proven to be as hard as finding a legal plan), but usually find a large number of them.

Definition A.6.1 ***Mutually Exclusive Actions***

Two actions a and b are considered to be mutually exclusive if they both belong to the same action-level and one of the conditions below is true:

1. *If either of the actions deletes a precondition or Add-Effect of the other **(Interference)**.*
2. *If there is a precondition of action a and a precondition of action b that are marked as mutually exclusive of each other in the previous proposition level **(Competing Needs)**.*

Definition A.6.2 ***Mutually Exclusive Propositions***

Two propositions p and q are considered mutually exclusive if they are both in the same proposition level and each action a with an add-edge to proposition p is marked as exclusive of each action b having an add-edge to proposition q.

It is interesting to notice that a pair of propositions may be mutually exclusive at every level in a planning graph or they may start out being exclusive in early levels and then become non-exclusive at later levels.

Note that the notion of *competing needs* and the exclusivity between propositions are not just logical properties of the operators. Rather, they depend on the interplay between operators and the Initial Conditions. In many different domains, exclusion relations seem to propagate a variety of intuitively useful facts about the problem throughout the graph.

A.6.5 Algorithm

In high-level terms, the basic algorithm of Planning Graph Analysis is the following. The planner runs in stages and it starts with a Planning Graph that has a single proposition level (containing the Initial Conditions).

In stage i, the planner takes the graph from stage $i-1$, extends it one step (adding the next action level and the following proposition level) and then searches the extended Planning Graph for a valid plan of length i. This search either finds a valid plan (and halts) or else determines that the goals are not achievable by time i (in which case the planner goes on to the next stage).

This algorithm is sound and complete: cite [Blum and Furst, 1995] for the proves. We will now pay more attention to a few key details of the algorithm.

Extending Planning Graphs

All the initial conditions are placed in the first proposition level of the graph. To create an action level, the following is done: For each operator and each way of instantiating preconditions of that operator to propositions of the previous level, insert an action node *if no two of its preconditions are labeled as mutually exclusive.* Also insert all the no-op actions and insert the precondition edges. Then check the action nodes for exclusivity and create an "actions-that-I-am-exclusive-of" list for each action.

To create a proposition level, simply look at all the Add-Effects of the actions in the previous level (including no-ops) and place them in the next level as propositions, connecting them via the appropriate add and delete-edges. Mark two propositions as exclusive if all the actions that generate the first are exclusive of all the actions that generate the second.

Searching for a Plan

Given a Planning Graph, the planner searches for a valid plan using a backward-chaining strategy. It uses a level-by-level approach, in order to make best use of the mutual exclusion constraints.

Given a set of goals at time t, it attempts to find a set of actions (no-ops included) at time $t-1$ having these goals as add effects. The preconditions to these actions form a set of subgoals at time $t-1$ having the property that if *these* goals can be achieved in $t-1$ steps, then the original goals can be achieved in t steps. If the goal set at time $t-1$ turns out not to be solvable, the planner tries to find a different set of actions, continuing until it either succeeds or has proven that the original set of goals is not solvable at time t.

In order to implement this strategy, the following recursive search method is used: For each goal at time t in some arbitrary order, select some action at time $t-1$ achieving that goal that is not exclusive of any actions that have already been selected. Continue recursively with the next goal at time t. Of course, if a goal has already been achieved by some previously-selected action, we do not need to select a new action for it. If the recursive call returns failure, then it tries a different action achieving our current goal, and so forth, returning failure once all such actions have been tried. Once finished with all the goals at time t, the preconditions to the selected actions make up the new goal set at time $t-1$. We call this a "goal-set creation step". The planner then continues this procedure at time step $t-1$.

A "forward-checking" improvement to this approach is that after each action is considered, a check is made to make sure that no goal in the list has been "cut-off". In other words the planner checks to see if for some goal still ahead in the list, all the actions creating it are exclusive of actions we have currently selected. If there is some such goal, then the planner knows it needs to back up right away.

Memoization

One additional aspect of this search is that when a set of (sub)goals at some time t is determined to be not solvable, then before popping back in the recursion it *memoizes* what it has learned, storing the goal set and the time t in a hash table.

Similarly, when it creates a set of subgoals at some time t, before searching it first probes the hash table to see if the set has already been proved unsolvable. If so, it then backs up right away without searching further.

This memoization step, in addition to its use in speeding up search, is needed for our termination check as described below.

A.6.6 Terminating on Unsolvable Problems

To a first view, the Planning Graph Analysis conducts something like an iteratively-deepened search. As described so far, nothing prevents the algorithm from mindlessly running forever through an infinite number of stages.

Now a simple and efficient test will be described, a test that can be added after every unsuccessful stage so that if the problem has no solution then the planner will eventually halt and say "No Plan Exists".

Planning Graphs "Level Off"

Assume a problem has no valid plan. Observe that in the sequence of Planning Graphs generated there will eventually be a proposition level P such that all future proposition levels are exactly the same as P, i.e., they contain the same set of propositions and have the same exclusivity relations.

The reason is as follows. Because of the no-op actions, if a proposition appears in some proposition level, then it also appears in all future proposition levels. Since only a finite set of propositions can be created by STRIPS-style operators (when applied to a finite set of initial conditions), there must be some proposition level Q such that all future levels have exactly the same set of propositions as Q.

Also, again because of the no-op actions, if propositions p and q appear together in some level and are *not* marked as mutually exclusive, then they will not be marked as mutually exclusive in any future level. Thus there must be some proposition level P after Q such that all future proposition levels also have exactly the same set of mutual exclusion relations as P.

In fact, once two adjacent levels P_n, P_{n+1} are identical, then all future levels will be identical to P_n as well. At this point, we say the graph has *leveled off.*

A Quick and Easy Test

Let P_n be the first proposition level at which the graph has leveled off. If some Problem Goal does not appear in this level, or if two Problem Goals are marked as mutually exclusive in this level, then the planner can immediately say that no plan exists.

Notice that in this case, the planner is able to halt without performing any search at all. However, in some cases it may be that no plan exists but this simple test does not detect it. So we need to do something slightly more sophisticated to guarantee termination in all cases.

A Test to Guarantee Termination

As mentioned earlier, the planner memoizes, or records, goal sets that it has considered at some level and determined to be unsolvable. Let S_i^t be the collection of all such sets stored for level i after an unsuccessful stage t. In other words, after an unsuccessful stage t, the planner has determined two things:

1. Any plan of t or fewer steps must make one of the goal sets in S_i^t true at time i, and
2. None of the goal sets in S_i^t are achievable in i steps.

The modification to ensure termination is just the following:

> If the graph has leveled off at some level n and a stage t has passed in which $|S_i^{t-1}| = |S_i^t|$, then output "No Plan Exists".

[Blum and Furst, 1995] proves that this modification is sufficient.

A.6.7 Example

As an illustration, this subsection presents the graphical representation of a planning graph for the Block World composed only of two blocks: Block A and Block B.

The picture shows only the most basic of the edges - precondition edges, add-edges and del-edges - for purposes of clarity. A complete picture would include the edges for mutual-exclusion relations in proposition levels and the no-op nodes in action levels.

In this picture, the precondition edges are shown as the directed edges that link a proposition node to an action node in the next action level. The add-edges are all the directed full edges that link an action node to the proposition nodes on the next proposition level corresponding to the action's add-list. Finally, the remaining directed edges linking an action node to the proposition nodes on the next proposition level corresponding to the action's del-list are the del-edges.

$$\begin{tikzcd}
 & & On(A,B) \arrow[r] & unstack(A,B) \arrow[r, dashed] \arrow[dr] \arrow[dddr] & On(A,B) \\
Table(A) \arrow[r] & stack(A,B) \arrow[ur] \arrow[r, dashed] \arrow[ddr, dashed] & Table(A) \arrow[r] & stack(A,B) \arrow[ur] \arrow[r, dashed] \arrow[ddr, dashed] & Table(A) \\
Clear(A) \arrow[ur] \arrow[ddr] & & Clear(A) \arrow[ur] \arrow[uur] \arrow[ddr] & & Clear(A) \\
Clear(B) \arrow[uur] \arrow[dr] & & Clear(B) \arrow[uur] \arrow[dr] \arrow[ddr] & & Clear(B) \\
Table(B) \arrow[r] & stack(B,A) \arrow[dr] \arrow[r, dashed] \arrow[uur, dashed] & Table(B) \arrow[r] & stack(B,A) \arrow[dr] \arrow[r, dashed] \arrow[uur, dashed] & Table(B) \\
 & & On(B,A) \arrow[r] & unstack(B,A) \arrow[r, dashed] \arrow[ur] \arrow[uuur] & On(B,A)
\end{tikzcd}$$

Appendix B

Multi-agent planning

The planning problem described in the previous chapter is the original, classical kind. The focus of this text is an extrapolation of the classical problem: the *Multi-agent Planning Problem.* The classical problem assumes only one active agent involved in the plan-making process and handles only the interactions of this agent with the world around. Multi-agent planning, however, takes into consideration not only the interactions of more than one agent with the world, but also the interactions between two or more agents. It allows for reasoning about concepts like coalitions and disputes, depending on whether the plans of the agents aim for complementing or conflicting goals.

Before handling this more sophisticated problem, though, it's necessary to choose a suitable representation. Such a good representation must be both semantically accurate and easily readable by human eyes. It should also be compact and simple to compile - which would favor the development of correct and automatic planning tools.

STRIPS fulfills most of these requirements. Unfortunately, STRIPS fails to live up to the most important requirement: semantic accuracy. Such a shortcoming leads us to suspect it'd be less than wise to try extending STRIPS to represent multi-agent planning problems. Thus, this text explores the use of modal logics to represent such problems. Still, STRIPS was proven to be a valuable development in the historical planning paradigm, so it's only natural that it could influence any modal logic designed for multi-agent planning.

B.1 The Multi-Agent Planning Problem

This section is intended to extend the definitions and concepts introduced in section A.1 in order to present a detailed description of what we call the *Multi-Agent Planning Problem.*

The Multi-Agent Planning (or *MAP*) Problem is a generalization of the classical Planning Problem. The classical version assumes that only one agent, the planner, is active in the scope of any problem. Thus, the planner must only consider the changes it can inflict upon the world in its reasoning.

The multi-agent version, though, is intended to handle problems which involve not one, but several acting agents. Thus, a multi-agent planner must not only evaluate the changes it can inflict on the world, but it also must consider the changes the other agents can bring to the world. It must also consider the way such changes fit together and find ways to reach his goals no matter how the other agents behave.

In more concrete terms, any two agents can have different relationships, depending on the goals each of them pursues. There are 3 possibilities:

- *Collaborative*: The two agents share the same goals and may unite to bring such goals about;
- *Adversarial*: The two agents have opposing goals (or the goal of one is exactly to keep the other to reach some other goal) and must then compete;
- *Indifferent*: Neither agent is concerned about the other's goals, or they just have independent policies.

The most important aspects of the MAP problem revolves around the two first, extreme cases.

The collaborative situation is the easiest to deal with. When collaborative agents act together, they form what is called a *coalition*. A coalition can be thought of as a meta-agent, with the combined abilities of its member agents.

The adversarial case is at the core of *Game Theory*. As such, it brings a bias from the classical Game Theory: the agent must play safe - it must be protected from the worst line of events. Some of the recent work on MAP through model-checking follow the same tradition: a goal is achievable only when it can be enforced for every possible response from the rest of the agents, otherwise no plan can be generated [van der Hoek and Wooldridge, 2002].

However, it's not always acceptable that no plans can be generated, so [Bui and Jamroga, 2003] proposes a measure of "degree of satisfiability"- when not all goals can be achieved, an utility function can judge how good a plan is, depending on which goals it does fulfills. The idea is that a plan that fulfills the most important portion of the goal is better than no plan at all.

Even in problems where the agents have independent goals, the planning agent must look for a plan against the worst possible line of events. However, in this case, there are two possible approaches: independent searches, or collaborative search to the best extent.

In the literature regarding MAP, "plans" and "strategies" are sometimes used with the same meaning. In this text, we make a point in separating these notions in the following way:

- *Plan*: A plan is an ordered list of actions which leads from the initial state to another state containing the most reachable goals possible for the problem;

- *Strategy*: A strategy is a function, or a set of rules, to help select the best action to take given the properties of the current time.

There are other aspects to keep in mind when dealing with MAP:

- Whether the agents have complete knowledge of the situation;

- Whether the agents have complete knowledge of the outcomes of every action;
- Whether the outcome of every action is deterministic:
- Whether the problem is characterized by a synchronous structure (discrete time where the agents take turns);
- Whether all the goals of all the agents are public.

B.2 Knowledge in Computation Tree Logic - KCTL

Computation Tree Logic was developed to reason about the behavior of a given program during its execution. It was intended to help handling non-determinism and its consequences. Although there were several extensions and derivatives, CTL stays as an interesting and important modal logic, not only for its historical value, but also because of the efficiency of the algorithms it uses.

In order to create a correct and powerful logic to reason about *distributed knowledge*, and aiming at achieving linearly efficient algorithms, [Benevides et al., 2004] proposed a natural extension to CTL: KCTL.

The resulting extension, which will be presented with details during this section, allows for reasoning about Time and Knowledge, allowing for a flexible combination of these two concepts in the properties of any problem.

Just like other Model-Checking logics were proven to be useful in planning problems, KCTL was shown to be useful in Multi-Agent Planning problems. Thus the interest in using and extending it.[1]

B.2.1 Semantic Structures

The text of this section is based on [Benevides et al., 2004].

As mentioned before, KCTL is an extension of CTL intended to reason about knowledge in a Multi-Agent System (*MAS*). Computation in a Multi-agent System is dictated by the local programs each agent runs. Global states of the system are compositions of local states of each agent. [Lamport, 1978] presents a classic event-based model for Asynchronous *MAS*. The *Asynchronous Multi-Agent System Model* is composed by:

- a network with m agents, connected by communication channels;
- a set R of asynchronous runs (distributed computations or parallel runs of all agents involved);
- a set E of events, including internal actions and communication events;
- a set C of global states of the system; and
- a protocol P (or distributed algorithm) corresponding to a set of local programs that specifies the behavior of each agent.

We represent the program for each agent in the *MAS* as an automaton. Each automaton represents the local states (nodes) and events from E (edges) for an agent. P is the set of all automata.

Definition B.2.1 *Automaton*
Let Σ be an alphabet, an automaton is a structure $\mathfrak{A} = \langle S, \Sigma^*, E, \{p_i\}_{i \in \mathcal{I}}, \mathcal{L} \rangle$ such that S is a set of states, Σ^* is the language generated by the alphabet Σ, $E \subseteq S \times \Sigma^* \times S$ a set of edges, $\{p_i\}_{i \in \mathcal{I}}$ a set of propositions, $\mathcal{I}$ is the agent set and $\mathcal{L} : S \longrightarrow 2^{\{p_i\}_{i \in \mathcal{I}}}$ the function that assigns to each state a subset of the propositions.

When we put all the agents running together, we obtain a global automaton that is the parallel asynchronous composition of the automata of the agents.

The idea is that the states of the composed automaton are n-tuples where each of its components corresponds to a local state of each of the agents, and the edges correspond to all edges of local automata.

Definition B.2.2 *Parallel Asynchronous Composition of Automata*
Let $\mathfrak{A}_i = \langle S_i, \Sigma^*, E_i, \{p_k\}_{k\in\mathcal{I}_i}, \mathcal{L}_i\rangle$ *be an automaton for all* $i \in [1,\ldots,n]$. *Then the parallel asynchronous composition of the automatons, denoted as* $||_{1\leq i\leq n}\mathfrak{A}_i$, *is the structure* $\langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ *defined as follows:*

- $S = \Pi_{1\leq i\leq n} S_i$,
- $((s_1,\ldots,s_n), l, (s_1',\ldots,s_n')) \in E$ *iff* $\bigvee_{1\leq i\leq n} l \in \Pi_2(E_i)$ [1],
- $\mathcal{P} = \bigcup_{1\leq i\leq n} \mathcal{P}_i$
- $\mathcal{L}((s_1,\ldots,s_n)) = \bigcup_{1\leq i\leq n} \mathcal{L}_i(s_i)$

States in the composed automaton correspond to the *MAS*'s global states, and are the elements in C. The set of runs can be easily obtained from the global automaton: each run in R is a path on the global automaton's computation tree.

When making the Parallel Asynchronous Composition of a set of automata (where each automaton dictates the behavior of an agent), it is possible and reasonable to get a composed global automaton $G = \langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ where many states from S correspond to the same local state component for a particular agent. When two such states s and t with the same local state component for agent i are connected by an edge $(s, l, t) \in E$, $l \in \Pi_2(E_j)$, $j \neq i$, it means that agent i is incapable of noticing that a global state change has occurred when the global state passes from s to t. These edges denote local actions made by other agents different from i, and for this reason are imperceptible for agent i.

An equivalence relation for each agent can be defined over the states with this "indistinguishable" property.

Definition B.2.3 *Possibility Relation* $\sim_i$ *for agent i*
Let $||_{1\leq i\leq n}\mathfrak{A}_i = \langle S, \Sigma^*, E, \mathcal{P}, \mathcal{L}\rangle$ *be the parallel asynchronous composition of automata* $\mathfrak{A}_i = \langle S_i, \Sigma^*, E_i, \{p_k\}_{k\in\mathcal{I}_i}, \mathcal{L}_i\rangle$ *for all* $i \in [1,\ldots,n]$. *The possibility relation*

[1]Given a label $l \in \Sigma^*$, a set of states S and a set of edges $E \in S \times \Sigma^* \times S$, $l \in \Pi_2(E)$ if and only if there exists $s, s' \in S$ such that $(s, l, s') \in E$.

$\sim_i \in S \times S$ for each agent i is the smaller equivalence relation containing all pairs (s,t) such that $s,t \in S$ and there is an edge $(s,l,t) \in E$, $l \in \Pi_2(E_j)$ and $j \neq i$.

Intuitively, two states are related by $\sim_i$ if agent i can't distinguish the two states. That is, i does not see any difference between the two states.

B.2.2 Semantics & Syntax

This section is also based on [Benevides et al., 2004].

As in *CTL*, *KCTL* formulas reason about properties of computation trees. The tree is formed just by unwinding the global automaton that represents the *MAS* from it's initial state. The computational tree illustrates all possible runs in R [Clark et al., 1999].

However, the classical Kripke model doesn't give us support to proper handling of distributed knowledge.

The way chosen by [Benevides et al., 2004] to deal with knowledge and uncertainty in Kripke Models is by means of "indistinguishable states". We also label propositions with its corresponding agent identification, assigning propositions to one agent information set and reinforcing the notion that each agent has its own separate knowledge base.

Thus, a classic Kripke model isn't enough to map all the information we need. The structure [Benevides et al., 2004] uses to model a MAS and supply the semantics of KCTL is an enhanced version of the classic Kripke Model, which includes the possibility relations $\sim_k$. For each agent k, there is an equivalence relation $\sim_k$ over S. Two states s and t of S are related by $\sim_k$ if and only if agent k can not tell them apart. This means that the information that agent k has in both s and t states is the same.

In the KCTL semantics, a MAS is described by a Knowledge-Extended (or K-Extended) Kripke Model as defined below:

Definition B.2.4 *(K-extended Kripke Model)*

A structure $\mathfrak{M} = \langle S, S_0, R, \mathcal{E}, \mathcal{P}, \mathcal{L}\rangle$ *is said to be a K-extended Kripke Model if its components are as follow:*

- S *is the set of possible states of the system (also called the set of* global states*);*
- S_0 *is a singleton set containing the unique initial state of the system (also known as the* initial global state*);*
- R *is the set of directed edges (transitions) linking the states of the system (also known as the set of* global transitions*) - a transition from state* x *to state* y *is written* xRy*;*
- $\mathcal{E} = \{\sim_k\}_{1\leq k\leq j}$ *is the set of "indistinguishability" relations of the system, where* j *is the number of agents of the system and each* $\sim_k$ *is an equivalence relation of the form "*$X \sim_k Y$*" which exists if and only if* X *and* Y *are two indistinguishable global states for agent* k*;* [2]
- $\mathcal{P} = \{\mathcal{P}_k\}_{1\leq k\leq j}$ *is the set of the sets of propositions of the multi-agent system, where* j *is the number of agents of the system and each* $\{\mathcal{P}_k\}$ *is the set of propositions corresponding to agent* k*;*
- $\mathcal{L} : S \rightarrow 2^{\bigcup_{k=1}^{j}\mathcal{P}_k}$ *is a function which maps each global state* X *to a set formed only by the propositions which hold in this state* X*.*

The set of KCTL formulas regarding the MAS described by some K-Extended Kripke model follows the syntax defined below:

Definition B.2.5 *(Syntax of* KCTL *formulae)*
Let $j \in I\!N$ *and* $\{\mathcal{P}_k\}_{1\leq k\leq j}$ *be the set of disjoint sets of propositions. The language of* KCTL *formulae is defined as follows:*

$ForCTL(j, \{\mathcal{P}_k\}_{1\leq k\leq j})$ *is the smallest set* For *of formulae such that:*

- $p \in For$ *if and only if there exists* k *such that* $1 \leq k \leq j$ *and* $p \in \mathcal{P}_k$,

- $\{\neg\phi_1, \phi_1 \vee \phi_2, \exists \mathsf{X}\phi_1, \exists \mathsf{G}\phi, \exists[\phi_1 \mathsf{U} \phi_2]\} \subseteq For$ *iff* $\{\phi_1, \phi_2\} \subseteq For$.
- $\mathcal{K}_i(\phi) \in For$ *iff* $1 \leq k \leq j$ *and* $\phi \in For$,

Definition B.2.6 *(Set of runs of a K-Extended Kripke model)*
Let $\mathfrak{M} = \langle S, S_0, R, \mathcal{E}, \mathcal{P}, \mathcal{L}\rangle$ be a K-Extended Kripke model. Then, the runs of $\mathfrak{M}$, denoted by $\mathcal{R}^{\infty}_{\mathfrak{M}}$ are characterized as follows:

$$\mathcal{R}^{\infty}_{\mathfrak{M}} = \{\sigma \in seq^{\infty}(S) \mid \pi_1(\sigma) \in S_0 \ \wedge (\forall i \in I\!N : \pi_i(\sigma)\ R\ \pi_{i+1}(\sigma))\}.^2$$

We will use $\mathcal{R}_{\mathfrak{M}}$ to denote the infinite set of finite prefixes of the sequences of $\mathcal{R}^{\infty}_{\mathfrak{M}}$.

Notation B.2.1 *(Prefix of a run)*
Let $\mathfrak{M} = \langle S, S_0, R, \mathcal{E}, \mathcal{P}, \mathcal{L}\rangle$ be a K-Extended Kripke model, $T \subseteq \mathcal{R}^{\infty}_{\mathfrak{M}}$, $\sigma \in \mathcal{R}^{\infty}_{\mathfrak{M}}$ and $i \in I\!N$, ${}_i\sigma$ will denote the prefix of length i of σ, defined as ${}_i\sigma = \sigma' \in seq(S) \mid Length(\sigma') = i \wedge (\forall j \in I\!N : 1 \leq j \leq i \implies \pi_j(\sigma') = \pi_j(\sigma))$.

The intuition behind the next definition is that of a run where, given a binary relation over the set of states, successions of states that are related via that relation are compressed keeping only one of them, for instance, the first one. This definition somehow establishes a notion of indistinguishability of states in a run.

Definition B.2.7 *($\sim$-quotient of a run)*
Let S be a non-empty set, and $\sim \subseteq S \times S$ a binary relation on S, we define the $\sim$-quotient of a sequence, of elements of S, σ as $\sigma|^{\sim} = \sigma' \in seq^{\infty}(S) \mid \pi_1(\sigma) = \pi_1(\sigma') \wedge (\forall i \in I\!N : (\exists j, k \in I\!N : j < k \wedge \pi_i(\sigma') = \pi_j(\sigma) \wedge \pi_{i+1}(\sigma') = \pi_k(\sigma) \wedge (\forall r \in I\!N : j \leq r < k \implies \pi_r(\sigma) \sim \pi_j(\sigma))))$.

The satisfiability relation for a *KCTL* formula will be slightly modified to consider the previous definition to give meaning to the operator "K".

[2]We use $seq^{\infty}(S)$ to denote the set of infinite sequences of elements taken from the set S, and π_i as the projection of the i^{th} element of a sequence.

Definition B.2.8 *(Satisfiability relation for* KCTL *formulae)*
Let $\{\mathcal{P}_k\}_{1\leq k\leq j}$ *be a set of disjoint sets of propositions and* $\mathfrak{M} = \langle S, S_0, R, \{\sim_k\}_{1\leq k\leq j}, \bigcup_{k=1}^{j}\mathcal{P}_k, \mathcal{L}\rangle$ *be a K-extended Kripke Model, the satisfiability relation is defined in exactly the same way that it was done for* CTL *formulae, except for the new operator that is interpreted as follows:*

$$
\begin{array}{lll}
\mathfrak{M}, \langle\sigma, i\rangle \models p & \textit{iff} & p \in \mathcal{L}(\pi_i(\sigma)) \\
\mathfrak{M}, \langle\sigma, i\rangle \models \neg\phi & \textit{iff} & \mathfrak{M}, \langle\sigma, i\rangle \not\models \phi \\
\mathfrak{M}, \langle\sigma, i\rangle \models \phi \vee \psi & \textit{iff} & \mathfrak{M}, \langle\sigma, i\rangle \models \phi \textit{ or } \mathfrak{M}, \langle\sigma, i\rangle \models \psi \\
\mathfrak{M}, \langle\sigma, i\rangle \models \exists \mathsf{X}\phi & \textit{iff} & \exists\sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \mathfrak{M}, \langle\sigma', i+1\rangle \models \phi \\
\mathfrak{M}, \langle\sigma, i\rangle \models \exists \mathsf{G}\phi & \textit{iff} & \exists\sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \forall j : i \leq j \implies \mathfrak{M}, \langle\sigma', j\rangle \models \phi \\
\mathfrak{M}, \langle\sigma, i\rangle \models \exists[\phi \mathsf{U} \psi] & \textit{iff} & \exists\sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : {}_i\sigma' = {}_i\sigma \wedge \\
 & & \quad (\exists j \in I\!N : i < j \wedge \mathfrak{M}, \langle\sigma', j\rangle \models \psi \wedge \\
 & & \quad\quad (\forall k \in I\!N : i \leq k < j \implies \\
 & & \quad\quad\quad \mathfrak{M}, \langle\sigma', k\rangle \models \phi)) \\
\mathfrak{M}, \langle\sigma, i\rangle \models \mathcal{K}_k(\phi) & \textit{iff} & \forall\sigma' \in \mathcal{R}^{\infty}_{\mathfrak{M}} : \forall j \in I\!N : {}_j\sigma'|^{\sim_k} = {}_i\sigma|^{\sim_k} \implies \\
 & & \quad \mathfrak{M}, \langle\sigma', j\rangle \models \phi
\end{array}
$$

Appendix C

The MAPKAT Semantic Framework

As mentioned, this text is centered on the idea of using and evolving the KCTL language not only as a model-checking tool, but also as a multi-agent planning tool.

There are several reasons for such a course of action. They include our choice on approaching the multi-agent planning problem as a model-checking problem, a choice which, in its turn, is motivated by the observations explained in the introduction of section B.

Section B.2 presented the language we chose as a starting point, KCTL. The current section will now propose a new semantic and logical framework for a new kind of model. This framework is designed to support and develop KCTL's semantics on Multi-Agent Planning Problems and for Model-Checking multi-agent systems about time and knowledge.

The MAPKAT framework was designed to overcome a limitation of the original KCTL: The original logical structure was based on automata. Thus, the model-checking process can not detect whether it has already passed through a given state.

This shortcoming happens because KCTL's approach doesn't have *perfect recall.*

[van der Meyden, 1994] defines “perfect recall” as the property of a distributed system keeping, at all times, a record in each processor state of all the previous states of this processor. Using the proper terms used in this text, a multi-agent system is said to have perfect recall if and only if each local state contains a complete record of all the previous states that agent has assumed.

The MAPKAT enforces the perfect recall property. The proof is the very nature of the model-checking process itself: it returns not only ”true”or ”false”, it also produces a set of alternative strategies that bring the model to a state where a query is successful. Each strategy itself is a sequence of actions starting from some initial state. Thus MAPKAT brings the model to remember each and every step that must be taken and, thus, can recognize a given state if it has already been reached.

C.1 Basic Concepts

The original KCTL uses an expanded version of the classical Kripke model to include the relations needed for its knowledge modalities. This approach had several advantages, including the fact of being based on the analysis of automata to accelerate its functions. MAPKAT uses a hybrid structure that mix the semantic concepts of KCTL and those of the planning graphs. As such, we will need to define the tools we will need to build such a hybrid structure in a non-ambiguous way.

We'll start by the definition of the *agents* themselves. An agent is defined by its set of propositions - also called *local propositions* because they are used to map local, specific facts or properties on the agent's own state -, its initial state - a set of the local propositions that map the conditions under which the agent start acting - and by the set of actions it can execute. An agent's state is defined by the propositions it contains, and it changes as the time passes because of the actions the agent executes, which in turn are recorded as a sequence. In a system with n agents, the state of a single agent i, where $1 \leq i \leq n$, is called a *local state*, or the *local state of agent i*.

An *action* is a specific application of some operator to some given state. This operator is defined by it's preconditions and its effects. If an operator is applied to a state, the action changes the current state into a new state. The preconditions are a set of propositions that must belong to any state of the agent to which the operator is to be applied. The effects may involve the inclusion of new propositions to the state (such effects are called *add-effects*) and they may involve the exclusion of old propositions (these effects are named *del-effects*).

We define a special kind of operator, with the form "Δ_p". There is one such operator for each proposition p, which is the only precondition of the operator and also its only effect. An operator "Δ_p" can be read as "do nothing about proposition p". This kind of operator forms the "null action family".

The *multi-agent system* ("MAS") itself is defined by its agent set. In contrast with the local states of an agent, the multi-agent system has *global states.* A global state is a parallelization of the current local states of all the n agents of the system and the environment. It is composed of n local states, one local state for each agent. The *initial state of a multi-agent system* is the global state formed by the union of the initial local state of each agent. A proposition is said to belong to a certain global state if and only if that proposition belongs to one of the states that form the global state. It is assumed that no preposition belongs to the preposition set of more than one agent.

Analogously, a system doesn't evolve by single actions, but by *moves.* A move is a parallelization of sets of actions, one set for each agent. The set of actions for an agent i belonging to a certain move is regarded as the *component i of the move.* An action belonging to a component of a move is said to also belong to the move. A move has preconditions, which is a set containing the preconditions of all the actions belonging to the move. In the same manner, the add-effects of a move is the union of the add-effects of all the actions belonging to the move and the del-effects of a move is the union of the del-effects of all the actions belonging to the move.

A move can be applied to a certain global state if and only if that global state contains all the propositions which form the preconditions of the move. If a move is applied to a global state, that global state is called the *origin of the move* or the *origin global state.* The resulting global state, called the *destination of the move* or the *destination global state*, is formed by the parallel application of the actions of the move to the corresponding component of the origin state of the move.

Time is defined on terms of "time steps". A *time step* is a function that maps a natural number to the global states that the MAS can reach at that time step. Whenever a move is applied to a global state, the MAS advances one time step. The value of the time step has no real-life meaning and is only meant to give an order to the moves and the global states of the MAS.

Instead of automata, like those used by KCTL, MAPKAT uses a structure similar to the *planning graphs* introduced by *Graphplan* as described in section A.6. The graph used by MAPKAT, just like the one used by Graphplan, is formed by two kinds of nodes: *proposition nodes* and *action nodes.* Just like Graphplan, there are ordered levels in the graph, where levels formed only by proposition nodes alternate with levels composed only of action nodes. Still following the original planning graphs, the first and last levels are always formed by proposition nodes only. Also, any two pair of elements in a proposition or action level can be linked by a special kind of semantic edges: *mutual-exclusion edges.* These special edges always link only pairs in the same level and mean that those two elements cannot co-exist.

Unlike the original planning graphs, though, MAPKAT splits proposition nodes into two kinds: the *positive nodes* and the *negative nodes.* A positive node for a proposition p encodes "proposition p is true", while a negative node for the same proposition p explicitly encodes "proposition p is not true". A positive node for p is labelled simply p, while a negative node for p is labelled as $neg(p)$.

Next, we will follow with the *formal* definitions of each element described before and of any other element needed to make the system described so far to function properly.

C.2 Formal Conceptual Definitions

This section presents the formal definitions of the semantic concepts related to a model of the MAPKAT framework. One of MAPKAT's roots is in Model-Checking. The other is in the approach to multi-agent planning using planning graphs. Since the first uses propositions, and the former can easily translate STRIPS-like domains to propositional domains, as seen in [Blum and Furst, 1995], the MAPKAT model is based on *propositions.*

Definition C.2.1 Operator.

Let $\mathbb{P}$ *be a set of possible propositions and let there be a set* $\mathbb{F} \subseteq \mathbb{P}$. *An* operator described by $\mathbb{F}$*is a structure* $\mathcal{O} = \langle \mathbb{C}, \mathbb{A}, \mathbb{D} \rangle$ *such as:*

- $\mathbb{C} \subseteq \mathbb{F}$ *is the operator's* set of preconditions, *denoted as precond*($\mathcal{O}$)*;*
- $\mathbb{A} \subseteq \mathbb{F}$ *is the operator's* Add-List, *denoted as addlist*($\mathcal{O}$)*;*
- $\mathbb{D} \subseteq \mathbb{F}$ *is the operator's* Del-List, *denoted as dellist*($\mathcal{O}$).

■

There is one special family of operator called the "null operator family". There is one such null operator for each proposition p and it has the form "Δ_p". Every Δ_p is such as precond(Δ_p) =addlist(Δ_p) $= \{p\}$ and dellist(Δ_p) $= \emptyset$. Its semantics mean "do nothing about p".

Definition C.2.2 Agent.

Let $\mathbb{P}$ *be a set of possible propositions. An agent is a structure* $\langle \mathbb{F}, \mathbb{O}, S_0 \rangle$ *such as:*

- $\mathbb{F} \subseteq \mathbb{P}$ *represents the agent's own* set of facts *or set of* possible propositions*;*
- $\mathbb{O}$ *is a set of operators described by* $\mathbb{F}$ *such as* $\Delta_p \in \mathbb{O}$ *for each* $p \in \mathbb{F}$*;*

- $S_0 \subseteq \mathbb{F}$ *represents the agent's own* initial state.

■

An agent's operators may be described by a superset of the agent's own facts, even if it will allow for many operators involving issues that the agent's not even aware of. The reason is that we are building the model of a Multi-Agent System and, as such, we must allow communication. However, the *preconditions* of one agent's operator *must* contain only elements of this agent's set of propositions.

Definition C.2.3 Local State.

Let $\mathcal{X}$ be an agent, and let $\mathbb{F}^{\mathcal{X}}$ be $\mathcal{X}$'s set of facts. Then a Local State of agent $\mathcal{X}$ is a set $S \subseteq \mathbb{F}^{\mathcal{X}}$. ■

The set of possible local states of a given agent $\mathcal{X}$ will be denoted by $\mathbb{S}^{\mathcal{X}}$. Also, it comes directly from the definitions that $\mathbb{S}^{\mathcal{X}} \subseteq 2^{\mathbb{F}^{\mathcal{X}}}$.

Definition C.2.4 Applicable Operator.

Let $\mathbb{P}$ be a set of possible propositions. Let us consider some set $\mathbb{Z} \subset \mathbb{P}$ and some operator $\mathcal{O}$ described by $\mathbb{F} \subseteq \mathbb{P}$. $\mathcal{O}$ is said to be applicable to a set $\mathbb{Z}$ *if and only if $\mathbb{Z} \supseteq precond(\mathcal{O})$.* ■

Please note that $\mathbb{Z}$ is a set of propositions, just like any agent $\mathcal{X}$'s local state was defined. However, since we are building a Multi-Agent System, we can't restrict our view by considering only operators applicable to a local state of some agent.

Definition C.2.5 Multi-Agent System

Let $\mathbb{P} \subseteq \mathcal{L}$ be a set of possible propositions. A Multi-Agent System (or MAS*) is a structure $\mathfrak{M} = \langle \mathbb{X}, \mathbb{N}, \mathbb{Q} \rangle$ such as:*

- $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *is a set of agents;*
- $\mathbb{N} = \bigcup_{i=1}^{n} \mathbb{O}^i \Big| \mathbb{O}^i$ *is the set of operators of agent* $X^i \in \mathbb{X}$

- $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{F}^i \Big| \mathbb{F}^i$ *is the set of facts of agent* $X^i \in \mathbb{X}$

$\mathfrak{M}$ *is a* well-formed Multi-Agent System *if, and only if, every operator* $\mathcal{O} \in \mathbb{N}$ *is described by* $\mathbb{Q}$.

$\mathbb{Q}$ *must be an union of disjunct sets; that is, for every* $P \in \mathbb{Q}$ *there is only one agent* X^i *whose set of facts* $\mathbb{F}^i$ *is such as* $P \in \mathbb{F}^i$.■

As an illustration, consider the example agents described in figures 1 and 2. Together, they form a Multi-Agent System which is partially described in figure 3.

This MAS encodes a simple planning problem: How can Mr. Walker, who lives in a luxurious building, get help in case of problems, like his telephone line suddenly going mute?

Following this toy problem, something can be done about it by at least asking the receptionists at the front door for some help and some information on who to contact about it — assuming, of course, that the building offers maintenance services for its dwellers.

Yet, the receptionists are mere human and may be distracted — in this simple illustration, talking on the phone should be distraction enough.

Definition C.2.6 Global State and Global Truth

Let $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ *be the set of agents of a well-formed Multi-Agent System* $\mathfrak{M}$*(as in Definition C.2.5). A* global state *is a* parallel composition $\mathcal{G} = \langle S^1 \parallel S^2 \parallel \ldots \parallel S^n \rangle$ *where, for each* $1 \leq i \leq n$, S^i *is a local state of agent* X^i.

The Global State $\mathcal{G}$ *represents a state of the whole Multi-Agent System and it defines a set* $\mathcal{G}^p$ *of propositions such as* $\mathcal{G}^p = \bigcup_{i=1}^{n} S^i$. *The set* $\mathcal{G}^p$ *is called the* Global Truth of Global State $\mathcal{G}$.

The Global Initial State *of* $\mathfrak{M}$ *is the parallel composition* $\mathcal{G}_0 = \langle S_0^1 \parallel S_0^2 \parallel \cdots \parallel S_0^n \rangle$ *where, for each* $1 \leq i \leq n$, S_0^i *is the initial local state of agent* X^i. ■

When we state that an operator is applicable to a global state, we imply that the operator is applicable to the global truth defined by this global state.

Figure 1 A simple Server with "distractions"

```
Agent: Receptionist {
    Propositions Set: { Receptionist:Free; Receptionist:Distracted; Receptionist:Busy; Receptionist:Finished }
  Initial State: { Receptionist:Free }
  Operators Set: {
    Operator: Receptionist-Talk {
      Preconditions: { Receptionist:Free }
      Add-List: { Receptionist:Distracted }
      Del-List: { Receptionist:Free }
    Operator: Receptionist-Return {
      Preconditions: { Receptionist:Talking }
      Add-List: { Receptionist:Free }
      Del-List: { Receptionist:Talking }
    Operator: Receptionist-Attend {
      Preconditions: { Receptionist:Free; Walker:AtFront; Walker:NeedHelp }
      Add-List: { Receptionist:Busy; Walker:GotAttention }
      Del-List: { Receptionist:Free; Walker:NeedHelp }
    Operator: Receptionist-GiveHelp {
      Preconditions: { Walker:GotAttention }
      Add-List: { Receptionist:Finished; Walker:GotHelp }
      Del-List: { Receptionist:Busy }
    Operator: Receptionist-Reset {
      Preconditions: { Receptionist:Finished; Walker:AtHome }
      Add-List: { Receptionist:Free }
      Del-List: { Receptionist:Finished; Walker:GotAttention }
  }
}
```

Figure 2 Example of a client for Receptionist 1

```
Agent: Walker {
    Propositions Set: { Walker:AtHome; Walker:AtFront; Walker:NeedHelp;
Walker:GotHelp; Walker:GotAttention }
  Initial State: { Walker:NeedHelp; Walker:AtHome }
  Operators Set: {
    Operator: Walker-WalkToFront {
      Preconditions: { Walker:AtHome }
      Add-List: { Walker:AtFront }
      Del-List: { Walker:AtHome }
    Operator: Walker-ReturnHome {
      Preconditions: { Walker:GotHelp}
      Add-List: { Walker:AtHome }
      Del-List: { Walker:AtFront }
  }
}
```

Figure 3 Part of the description of the resulting Multi-Agent System

$\mathbb{X} = \left\{ \textit{Receptionist}; \textit{Walker} \right\}$

$\mathbb{Q} = \Big\{$ *Receptionist:Free*; *Receptionist:Talking*; *Receptionist:Busy*; *Receptionist:Finished*; *Walker:AtHome*; *Walker:AtFront*; *Walker:NeedHelp*; *Walker:GotHelp*; *Walker:GotAttention* $\Big\}$

An example of a global state and its corresponding global truth is given in Figure 4.

Figure 4 An example of global states.

$G_0 = \langle$ {Receptionist:Free} $\|$ {Walker:NeedHelp; Walker:AtHome} $\rangle$

Global Initial Truth:

$G_0^p = \Big\{$ Receptionist:Free; Walker:NeedHelp; Walker:AtHome $\Big\}$

Definition C.2.7 Action

Let $\mathbb{G}$ *be the set of all possible global states of a well-formed MAS* $\mathfrak{M}$ *and let* $\mathbb{N}$ *be the set of operators of* $\mathfrak{M}$.

An Action *is an instance of some function* $g : \mathbb{G}\times\mathbb{N} \rightarrow \mathbb{G}$ *that applies an operator* $\mathcal{O} \in \mathbb{N}$ *to a global state* $\mathcal{G} \in \mathbb{G}$ *— that is, it checks whether* $\mathcal{O}$ *is applicable to the global state* $\mathcal{G}$ *and produces a new global state* $g(\mathcal{G}, \mathcal{O})$ *by changing the elements of* $\mathcal{G}$ *according to the contents of* $\mathcal{O}$*'s Add-List and Del-List:*

$$\begin{aligned}
g(\langle S_a^1 \parallel S_a^2 \parallel \dots \parallel S_a^n\rangle, \mathcal{O}) &= \langle S_b^1 \parallel S_b^2 \parallel \dots \parallel S_b^n\rangle \\
\mathbb{F}^i &= \textit{Set of facts of agent } X^i. \\
S_c^i &= S_a^i \cup (\textit{Addlist}(\mathcal{O}) \cap \mathbb{F}^i) \\
S_b^i &= S_c^i \setminus (\textit{Dellist}(\mathcal{O}) \cap \mathbb{F}^i)
\end{aligned}$$

If our MAS were to evolve only through individual actions, it would be very limited. Using an approach inspired by [Blum and Furst, 1995], we will work with *moves*.

Definition C.2.8 Moves

Let $\mathbb{X} = \{X^1, X^2, \dots, X^n\}$ *be the set of agents of a well-formed MAS* $\mathfrak{M}$ *and let* G *be a global state of* $\mathfrak{M}$. *Let* $\mathbb{A}_G^1, \mathbb{A}_G^2, \dots, \mathbb{A}_G^n$ *be sets of operators such as each* $\mathbb{A}_G^i$ *denotes the set of all operators of agent* $X^i \in \mathbb{X}$ *applicable to the global state* G, *for all* $1 \leq i \leq n$.

A Move *applicable to a global state G of a MAS $\mathfrak{M}$, or simply* move, *is the parallel composition $m = \langle A_G^1 \parallel A_G^2 \parallel \ldots \parallel A_G^n \rangle$ where, for each positive integer $j \leq n$, $A_G^j \subseteq \mathbb{A}_G^j$.* ■

A *move* is, intuitively, the collection of all the actions executed during a given *time step*, or turn. An example is given in Figure 5.

Figure 5 Examples of moves

Two valid moves applicable to the global state given in Figure 4:

$m = \langle$ {Receptionist-Talk} $\parallel$ {Walker-WalkToFront} $\rangle$

$m' = \langle\{\Delta_{\text{Receptionist:Free}}\} \parallel$ {Walker-WalkToFront} $\rangle$

There are a few conventions we follow for notational simplicity:

- If $m = \langle A_G^1 \parallel A_G^2 \parallel \ldots \parallel A_G^n \rangle$ then we define the aliases m^i such as $\forall i \Big(\big(1 \leq i \leq n \big) \wedge \big(A_G^i = m^i \big) \Big)$;
- For any $m = \langle m^1 \parallel m^2 \parallel \ldots \parallel m^n \rangle$, we say $\mathcal{O} \in m$ if and only if $\exists i (\mathcal{O} \in m^i)$;
- For any $m = \langle m^1 \parallel m^2 \parallel \ldots \parallel m^n \rangle$, if there is some $1 \leq i \leq n$ such as m^i is composed only by operators of the Δ_p form then we call m^i a *null component*;
- For any $m = \langle m^1 \parallel m^2 \parallel \ldots \parallel m^n \rangle$, if m^i is a null component for all $1 \leq i \leq n$, then we call m a *null move*;

Definition C.2.9 Production

Let $\mathbb{G}$ be the set of all possible global states of a well-formed MAS $\mathfrak{M}$ and let $\mathbb{M}$ be the set of all possible moves of $\mathfrak{M}$. A Product *is an instance of some function $h : \mathbb{G} \times \mathbb{M} \rightarrow \mathbb{G}$ that applies a move $m \in \mathbb{M}$ to a global state $\mathcal{G} \in \mathbb{G}$ — that is, it checks whether each operator $\mathcal{O} \in m$ is applicable to the global state $\mathcal{G}$ — and produces a new global state $h(\mathcal{G}, m)$ by changing the elements of $\mathcal{G}$ according to the contents of the Add-List and Del-List of each operator $\mathcal{O} \in m$:*

$$\begin{aligned}
h(\mathcal{G}, m) &= h(\mathcal{G}, \langle m_a^1 \parallel m_a^2 \parallel \ldots \parallel m_a^n \rangle) \\
h(\mathcal{G}, \langle m_a^1 \parallel m_a^2 \parallel \ldots \parallel m_a^n \rangle) &= g(\mathcal{G}, j(\textstyle\bigcup_{i=1}^{n} m_a^i)) \\
j(\{\mathcal{O}^1, \mathcal{O}^2 \ldots, \mathcal{O}^z\}) &= \textit{an operator } \mathfrak{O} \textit{ such as:} \\
\textit{Preconds } (\mathfrak{O}) &= \bigcup_{j=1}^{z} \textit{Preconds } (\mathcal{O}^j) \\
\textit{Addlist } (\mathfrak{O}) &= \bigcup_{j=1}^{z} \textit{Addlist } (\mathcal{O}^j) \\
\textit{Dellist } (\mathfrak{O}) &= \bigcup_{j=1}^{z} \textit{Dellist } (\mathcal{O}^j)
\end{aligned}$$

The function $j : 2^{\mathbb{N}} \rightarrow \mathbb{N}$, *where* $\mathbb{N}$ *is* $\mathfrak{M}$*'s set of operators, returns a single special operator* $\mathfrak{O}$ *which sums up the effects of the whole move* m *applied to* $\mathcal{G}$*.*

The function h *is assumed to follow the property that, if* m_Δ *is a null move, then* $\forall \mathcal{G} \big(h(\mathcal{G}, m_\Delta) = \mathcal{G} \big)$.■

A global state G is said to be *produced* by a move m if m is a valid move and there is some G' such as $h(G', m) = G$.

Definition C.2.10 Valid Move

A move $m = \langle m^1 \parallel m^2 \parallel \ldots \parallel m^n \rangle$ *is valid if and only if the set* $m_a = \bigcup_{i=1}^{n} m^i$ *doesn't contain any pair of* mutually exclusive *operators — see definition C.2.13.*■

This brings about an important concept mentioned in [Blum and Furst, 1995]: Mutual Exclusion. In essence, two mutually exclusive elements can't happen at the same time — or coexist — because they are conflicting. Mr Walker can't be in two different places at the same time, thus *Walker:AtHome* and *Walker:AtFront* are mutually exclusive. This concept can be applied to both propositions and actions.

Mutual exclusion of two actions imply that these two actions must be *explicitly ordered.* A Move, as defined above, is a set of possible actions that each agent can execute at a given global state. The actions of any move can be executed *in parallel,* or in any order. Therefore, these actions must be *independent,* that is, one must not *interfere* with another. The formal definition of this concept is presented in Definition C.2.13 because it uses concepts yet not introduced in this text.

Definition C.2.11 Runs

Let $\mathfrak{M}$ be a well-formed MAS. A run of $\mathfrak{M}$ is a sequence $R = m_1m_2m_3\ldots$ of moves of $\mathfrak{M}$ ordered in time and starting at some global state G of $\mathfrak{M}$. G is then called the origin.

It may be finite *or* infinite. *A finite run $R_k = m_1m_2\ldots m_k$ of length k is also called a* k-run.

A run is called valid if and only if the first move is applicable to the origin and each move after the first is applicable to the product of the previous move.

The product of the j-th move *of R is denoted by $\pi_j(R)$ and the* origin *is denoted by $\pi_0(R)$. The* product of a k-run *R_k is $\pi_k(R_k)$.*

The part of a run R that is formed by an agent X^i's operators is called the local run of X^i *and is denoted by R^i.* ■

Figure 6 Example of a run and a local run.

For G_0 and m (from figures 4 and 5, respectively):

$R = \langle$ {Receptionist-Talk} || {Walker-WalkToFront} $\rangle$; $\langle$ { Receptionist-Return } || $\{\Delta_{\text{Walker:AtFront}}\}\rangle$ is a 2-run and

$R_{\text{Walker}} = \langle$ {Walker-WalkToFront} $\rangle$; $\langle\{\Delta_{\text{Walker:AtFront}}\}\rangle$ is a local 2-run for agent Walker.

Definition C.2.12 Proposition Levels and Action Levels

A proposition level PL_t, *for a well-formed MAS $\mathfrak{M}$ and some $t \geq 0$, is the set of all propositions that can be true at time step t. The action level* AL_t, *for some $t \geq 1$, is the set of all actions that can be considered for execution at time step t or, in other words, applied to proposition level $t-1$.*

PL_0 *is always defined as the initial state of $\mathfrak{M}$.* PL_t, *for every $t \geq 1$, is obtained by unifying* PL_{t-1} *with the Add-List of each operator in action level* AL_t.■

We say $G \subseteq PL_i$ for some global state G and some proposition level PL_i as a notational simplification for $G^P \subseteq PL_i : G^P$ is G's global truth. Consider two global states: G and H. Suppose that they have the same propositions. We only may say $G = H$ if, and only if, they belong to the same proposition level — and in this case they are the same global state. Otherwise, $G \neq H$ because they represent sets of propositions valid in *different time steps.*

Definition C.2.13 Mutual Exclusion

- Among Operators

  Let O_1 and O_2 denote two different operators of the action-level AL_k of some well-formed MAS $\mathfrak{M}$. Then O_1 and O_2 are called mutually exclusive in k *if:*

  - *For $k \geq 1$, the Del-List of either O_1 or O_2 contains a proposition which also belongs to the Preconditions or the Add-List of the other; or*
  - *For $k \geq 2$, there is a precondition of O_1 and a precondition of O_2 that are mutually exclusive in the proposition level PL_{k-1}.*

- Among Propositions

  Let $p, q \in \mathbb{Q}$ be two propositions of some well-formed MAS $\mathfrak{M}$ in the same proposition level PL_k. They're called mutually exclusive *if every operator $O_1 \in AL_k$ with p in its Add-List is mutually exclusive of every operator $O_2 \in AL_k$ with q in its Add-List.*

- Among Local States

  Let $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ be the set of agents of a well-formed MAS $\mathfrak{M}$ and let PL_k be the proposition level of $\mathfrak{M}$ for some time step k. For all $1 \leq i \leq n$, let $\mathbb{F}^i$ be the set of facts of agent X^i and let the set $\mathbb{S}^i_k = 2^{\mathbb{F}^i \cap PL_k}$ contain all possible local states of agent X^i in the proposition level PL_k. Let the set $\Phi_k = \bigcup_{i=1}^{n} \mathbb{S}^i_k$ contain all possible local states of all agents for the proposition *level k. Any agent X^i can only achieve a given local state s in time k if $s \in \Psi_k$. Two local states $S, Z \in \Phi_k$ are called* mutually exclusive of each other in proposition level k *if:*

- *S and Z are different local states of the same agent; or*
- *S contains a proposition that is mutually exclusive of some proposition in Z in proposition level k.*

- Among Moves

  Let m and n be two different moves belonging to the same action level k of a well-formed MAS $\mathfrak{M}$. Then m and n are called mutually exclusive of each other in action level k *if and only if there is at least one operator $p \in m$ such as p is mutually exclusive of at least one operator $q \in n$.*

- Among Global States

  Let PL_k be the proposition level of a well-formed MAS $\mathfrak{M}$ for some time step k and let $\mathbb{X}$ be the set of agents of $\mathfrak{M}$. Two global states $G = \langle\ S^1\ \ \|\ \ \ldots\ \ \|\ \ S^{|\mathbb{X}|}\ \rangle$ and $H = \langle\ Z^1\ \ \|\ \ \ldots\ \ \|\ \ Z^{|\mathbb{X}|}\ \rangle$ are called mutually exclusive in proposition level k *if $G, H \subseteq PL_k$ and if there are integers i and j such as S^i and Z^j are mutually exclusive.*

- Among Runs

  Let R_1 and R_2 be two different runs of a well-formed MAS $\mathfrak{M}$. Then R_1 and R_2 are called mutually exclusive *if there is an action level k such as the move m of R_1 mutually exclusive to the move n of R_2 in k.*

■

Definition C.2.14 Possibility Relations

Let $\mathfrak{M}$ be a well-formed MAS, let $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ be the set of agents of $\mathfrak{M}$ and let $\mathbb{G}$ be the set of all possible global states of $\mathfrak{M}$.

Let R_a be the valid run of $\mathfrak{M}$ that produces G_a and let R_b be the valid run of $\mathfrak{M}$ that produces G_b. Then $G_a \sim_i G_b$ if and only if the local run R_a^i is equal to the local run R_b^i. ■

The *possibility relation* $\sim_i$ between two global states $G_a, G_b \in \mathbb{G}$ means that agent X^i can't distinguish G_a and G_b.

There are a few observations we can derive directly from Definition C.2.14:

- Since $G_a \sim_i G_b \longleftrightarrow R_a^i = R_b^i$, then G_a and G_b must belong to the same Proposition Level. Otherwise, the length of R_a^i would be different from the length of R_b^i and thus $R_a^i \neq R_b^i$, which contradicts Definition C.2.14;

- For the same reason, it is clear that we assume that the local state of X^i is the same in both G_a and G_b;

- If $G_a \sim_i G_b$ for all i, $1 \leq i \leq n$, then $G_a = G_b$. Conversely, if $G_a \sim_i G_b$ and $G_a \neq G_b$, then there is at least one agent X^j such as $R_a^j \neq R_b^j$.

- The only constraint for the $\sim_i$ relation involves agent X^i's own local run: it must stay the same. The local runs of each other agent can be any possible local run for that agent, as long as none of them are mutually exclusive of X^i's local run;

- When we consider whether $G_a \sim_i G_b$ at a given proposition level k, we are actually considering whether there are valid runs R_a and R_b, both featuring the same local run for X^i but resulting in different global states;

- Since validity is tied to mutual exclusion as much as runs are tied to global states, the only global states that can't be reached by valid runs featuring the desired local run for X^i are those featuring local states of other agents mutually exclusive to the desired local state for X^i.

C.3 The Model Definition and Verification

MAPKAT has its roots in two different disciplines of the Artificial Intelligence scientific literature: Model Checking and Planning. As such, one expects to combine the best aspects of both disciplines: the semantic strength and tractability of the first and the efficiency of the second.

The result is a logic of flexible expressiveness and semantic complexity, which involves the use of an unusual model built step by step. Thus, it's necessary to specify what is expected from each step. There are two major steps to be considered: model definition and model verification. The first step must handle the specifications of a given system for the construction of a correct model. The second step involves the model-checking of properties, which include the operations of automatic planning.

The *Model Definition* step involves the configuration of one whole Multi-Agent System by declaring the key aspects of the system: the agents themselves. The agents carry the possible facts of the system and also the operators that will change these facts as the system evolves. Thus, the core of each agent, and of the resulting MAS, is composed by the propositions and the operators defined in this step.

The second step, *Model Verification*, is the very reason of existence for this model. It's where the MAPKAT logic works to reason about the properties of the model and to produce plans for each agent. In fact, when one mentions the *MAPKAT Language*, one is referring to the language used in this step.

C.3.1 Agent definition language

As stated before, both System Definition and System Verification are equally important. Without a well-formed definition, any verification risks to be inconsistent. On the other hand, there is no use for a detailed definition without the verification itself. Thus, while the MAPKAT Language itself is about verification, this text will also state the specifications of a language for system definition.

Syntax

MAPKAT is a propositional logic: the agents must be declared through the use of *propositions.* The sets of propositions mentioned in this section are assumed to be in alphabetic order of the proposition's name, one proposition separated of another by a ";".

An agent is described by a structure of the form show in Figure 7.

Figure 7 Agent Definition

```
Agent: Agent Name {
  Propositions Set: {...}
  Initial State: {...}
  Operators Set: {...}
}
```

Within this structure, *Propositions*, *Initial State* and *Operators* are the *parameters* of the agent and form the information needed to make this agent functional, while *Agent Name* is a sequence of characters to uniquely identify this agent. No two agents may have the same name for obvious reasons.

The parameters have the following specifications:

- The *Propositions Set* parameter is a non-empty set of propositions — it is not allowed for one agent to have a proposition in this set with the same name as some element of another agent's propositions set;
- The *Initial State* is also a set of propositions, but it must be a subset of the propositions parameter and must be interpreted as a *conjunction* of its elements;
- The *Operators Set* parameter is a non-empty set where each element is a structure called *Operator* that describes an instance of the concept with the same name.

Figure 8 Operator Definition

```
Operator: Operator Name {
  Preconditions: {...}
  Add-List: {...}
  Del-List: {...}
}
```

An Operator of some agent is a structure with the form shown in Figure 8.

Analogously to the agent structure, *Preconditions*, *Add-List* and *Del-List* are the *parameters* of the operator. The *Operator Name* is the word used to uniquely identify this operator *within the agent's own set of operators.* No two operators may have the same name in one agent's operators set, but one agent may have an operator in its operators set with the same name as any operator of any other agent's operators set.

Each parameter is a set of propositions. It is possible to include in a parameter of some agent's operator a proposition that doesn't belong to this agent. However, it is necessary to clarify to which agent the proposition belongs by adding a prefix of the form " *agent_name:* ".

A very simple example of a complete agent definition is shown in Figure 9.

When describing a Multi-Agent System, one needs only state which are the component agents, one after another. A more complete but still simple example is shown in Figure 10.

Semantics

The semantics of the language introduced in the previous section is quite straightforward.

A proposition symbolizes a *possible fact.* One shouldn't assume that a proposition is true or false by itself. It'd be like stating that a fact is true or false by definition, a value which would never change and thus would not be mapped into a

Figure 9 A very simple example

```
Agent: *Walker* {
  propositions Set: { Here; There }
  Initial State: { Here }
  Operators Set: {
    Operator: Walk {
    Preconditions: { Here }
    Add-List: { There }
    Del-List: { Here }
  }
}
```

proposition, or even needed to begin with.

An agent structure describes *some entity able to reason and/or act* — for example, an intelligent program, a reactive system or a human. One agent's set of proposition is assumed to declare the propositions that symbolize each and every possible property of the agent. One agent's set of operators is assumed to specify the operators that map each and every possible action this agent can take. One agent's initial state is assumed to include all the propositions that describe the unique configuration with which the agent always begins.

An operator in some agent's set of operators interprets a *possible action* of the entity described by this agent in terms of changes in the agent's state and the special conditions under which such changes may happen.

The operator's *Preconditions* is assumed to include only the propositions necessary and sufficient to map such special conditions. The changes are mapped as parameters for operations on propositions, including the propositions that will be true right after the operator is applied (the elements in the *Add-List*) and the propositions that won't be true anymore after the operator is applied (the elements in the *Del-List*).

Figure 10 Another simple example

```
Agent: Walker {
  propositions Set: { Here; There }
  Initial State: { Here }
  Operators Set: {
    Operator: Walk {
    Preconditions: { Here }
    Add-List: { There }
    Del-List. { Here }
  }
Agent: Receptionist {
  propositions Set: { Free; Busy }
  Initial State: { Free }
  Operators Set: {
    Operator: Attend {
    Preconditions: { Free; Walker:There }
    Add-List: { Busy }
    Del-List: { Free }
  }
}
```

There are no requirements *a priori* on the validity of any element of the *Del-List* on the state to which the action is applied. Analogously, there are no requirements on the validity of any element of the *Add-List* on the state to which the action is applied. If such requirements are needed, they must be mapped in the preconditions of the operator.

It is necessary that the set of operators of the agents are disjoint. To guarantee such property, the operators of each agent are renamed by adding a prefix of the form "*agent_name-*" to the operator's old name, creating the new name "*agent_name-operator_old_name*".

We assume that each agent's proposition set is disjoint of the proposition set of any other agent's set. But similar agents will have similar properties and, thus, similar propositions. Although similar, these propositions are not the same: any proposition in agent X^1's proposition set reflects only one property of X^1 and is unrelated to any properties of agent X^2 or X^3, for example.

To guarantee that the propositions sets will be disjoint, it is necessary to relabel each element of each agent's propositions set by adding a prefix of the form "*agent_name:*".

For clarity, this relabelling should be applied on each element of each parameter of each operator of each agent, in order to allow only propositions of the form *agent_name:old_proposition_name* in the resulting MAS.

Figure 11 shows the effects of such changes applied to the MAS described in Figure 10:

C.3.2 Verification & Planning language

While the System Definition language of the previous section is a straightforward product of the classical planning literature, the System verification language reflects the influence of model-checking literature. In fact, the language presented here is the same from KCTL logic.

Figure 11 Another simple example, revisited

```
Agent: Walker {
  propositions Set: { Walker:Here; Walker:There }
  Initial State: { Walker:Here }
  Operators Set: {
    Operator: Walker-Walk {
    Preconditions: { Walker:Here }
    Add-List: { Walker:There }
    Del-List: { Walker:Here }
  }
Agent: Receptionist {
  propositions Set: { Receptionist:Free; Receptionist:Busy }
  Initial State: { Receptionist:Free }
  Operators Set: {
    Operator: Receptionist-Attend {
    Preconditions: { Receptionist:Free; Walker:There }
    Add-List: { Receptionist:Busy }
    Del-List: { Receptionist:Free }
  }
}
```

Syntax

During this section there will be references to a multi-agent system $\mathfrak{M}$ with $\mathbb{X} = \{X^1, \ldots, X^n\}$ as its set of agents. Also, it will use the set $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{P}^i$, where $\mathbb{P}^i$ is the set of propositions of agent X^i.

Definition C.3.1 The KCTL Language, revisited

The set of KCTL formulae for the MAS $\mathfrak{M}$, denoted by ForMAPKAT($\mathfrak{M}$), *is the smallest set* For *such as:*

1. $p \in \mathbb{Q} \leftrightarrow p \in \text{For}$;
2. $\Big\{\neg\phi_1, \phi_1 \vee \phi_2\Big\} \subseteq \text{For} \leftrightarrow \Big\{\phi_1, \phi_2\Big\} \subseteq \text{For}$;
3. $\Big\{\exists \bigcirc \phi_1, \exists\Box\phi_1, \exists\big[\phi_1 \mathsf{U} \phi_2\big]\Big\} \subseteq \text{For} \leftrightarrow \Big\{\phi_1, \phi_2\Big\} \subseteq \text{For}$;
4. $\mathcal{K}_i\phi_1 \in \text{For} \leftrightarrow \Big((\phi_1 \in \text{For}) \wedge (1 \leq i \leq n)\Big)$;

■

The operators defined in the second form compose the class of *boolean operators*. The third form defines the class of *temporal operators*. Finally, the last class is that of the *epistemic operator*.

Besides the previous definition, there are a number of optional but useful "aliases". For simplicity, the alias of some operator is said to belong to the same class of the original operator. Such aliases include:

- $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \neg\phi_2)$;
- $\phi_1 \longrightarrow \phi_2 = \phi_2 \vee \neg\phi_1$;
- $\phi_1 \longleftrightarrow \phi_2 = \big(\phi_1 \longrightarrow \phi_2\big) \wedge \big(\phi_2 \longrightarrow \phi_1\big)$;
- $\exists\Diamond\phi = \exists[\mathit{true}\mathsf{U}\phi]$;
- $\forall \bigcirc \phi = \neg\exists \bigcirc \neg\phi$;

- $\forall\Box\phi = \neg\exists\Diamond\neg\phi$;
- $\forall\Diamond\phi = \neg\exists\Box\neg\phi$;
- $\forall[\phi_1 \mathsf{U} \phi_2] = \neg\exists[\neg\phi_2 \mathsf{U}(\neg\phi_1 \wedge \neg\phi_2)]$
- $\mathcal{K}_i\phi = \neg\mathcal{B}_i\neg\phi$

A formula φ is called a *formula of KCTL over* $\mathfrak{M}$ if and only if $\varphi \in \mathit{ForMAPKAT}(\mathfrak{M})$.

Semantics

The semantics of a formula $\phi \in \mathit{ForMAPKAT}(\mathfrak{M})$ is naturally given in terms of the model built for $\mathfrak{M}$. Let $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$ denote the set of agents of $\mathfrak{M}$ and let $G = \langle S^1 \parallel S^2 \parallel \ldots \parallel S^n \rangle$ be some global state of $\mathfrak{M}$.

A formula φ of KCTL over $\mathfrak{M}$ is always tested for satisfiability in relation to some global state G of $\mathfrak{M}$. We write $\mathfrak{M}, G \vDash \varphi$ for some KCTL formula φ if and only if φ is true in state G of the system $\mathfrak{M}$.

There are a few valid variants to the syntax of the *satisfaction relation* $\vDash$, as follows:

- $G \vDash \varphi$, for any state G of the system $\mathfrak{M}$, is the same as $\mathfrak{M}, G \vDash \varphi$;
- $\mathfrak{M} \vDash \varphi$ is the same as $\mathfrak{M}, G_0 \vDash \varphi$, where G_0 is the initial state of the system $\mathfrak{M}$;
- $\vDash \varphi$ means that φ is an axiom, a formula valid at any state of any multi-agent system.

Let $\mathbb{Q} = \bigcup_{i=1}^{n} \mathbb{P}^i$ — where $\mathbb{P}^i$ is the set of propositions of agent X^i — be the set of propositions of $\mathfrak{M}$ and $\mathbb{N} = \bigcup_{i=1}^{n} \mathbb{O}^i$ — where $\mathbb{O}^i$ is the set of operators of agent X^i — denote the set of operators of $\mathfrak{M}$. The semantics of KCTL formulae over a MAPKAT model $\mathfrak{M}$ is defined below:

1. $\mathfrak{M}, G \models \top$, for all MAS $\mathfrak{M}$ and all global state G;

2. $\mathfrak{M}, G \models p$ if and only if $p \in G^P$, where G^P is $G's$ global truth;

3. $\mathfrak{M}, G \models \neg\, \varphi$ if and only if $\mathfrak{M}, G \not\models \varphi$;

4. $\mathfrak{M}, G \models \varphi_1 \vee \varphi_2$ if and only if $\mathfrak{M}, G \models \varphi_1$ or $\mathfrak{M}, G \models \varphi_2$;

5. $\mathfrak{M}, G \models \exists \bigcirc \varphi$ if and only if there is a move m applicable to G such as $\mathfrak{M}, g(G, m) \models \varphi$;

6. $\mathfrak{M}, G \models \exists \Box \varphi$ if and only if there is an infinite run $R = m_1 m_2 m_3 \ldots$ starting at state G such as $\mathfrak{M}, \pi_k(R) \models \varphi$, for each integer $k \geq 1$;

7. $\mathfrak{M}, G \models \exists [\varphi_1 \mathsf{U} \varphi_2]$ if and only if there is a run of finite length $h \geq 1$, $R^h = m_1 m_2 \ldots m_h$, such as $\mathfrak{M}, \pi_h(R^h) \models \varphi_2$ and that, for each positive integer $1 \leq k < h$, $\mathfrak{M}, \pi_k(R^h) \models \varphi_1$;

8. $\mathfrak{M}, G \models \mathcal{K}_i \varphi$ if and only if $\mathfrak{M}, G' \models \varphi$ for every global state G' such as, if $\pi_k(R_k) = G$ for some $k \geq 1$ and R^i_k is the local k-run in R_k for X^i, then $G' = \pi_k(R'_k)$ for some k-run R'_k which also contains the local k-run R^i_k.

There are not many differences between the semantic given above and the one presented in section B.2. In fact, the only difference is that the semantics presented above are all given in terms of a MAPKAT model. That was the intention, since MAPKAT is supposed to support *at least* the same semantics that the K-Extended Kripke models allows for the original version of KCTL.

Appendix D

Model-checking MAPKAT models

The previous chapter brought an extensive explanation on the MAPKAT framework developed to enrich the semantics of KCTL. Using a structure which mixes concepts from the K-Extended Kripke models and the planning graphs, it manages to describe a very dynamic system.

The current chapter shows how the semantic MAPKAT framework can be used to perform model-checking of multi-agent systems in the most general case. The model-checking itself is carried out by the algorithms presented in section D.2, which takes a MAPKAT model - built as described in section D.1 - and a KCTL formula as parameters.

D.1 Building the Model of a Multi-Agent System

Section C.2 presented and defined each and every concept involved in the semantic framework of MAPKAT. However, the following explanation makes explicit the ideal method of setting up a MAPKAT model in order to be model-checked by the algorithms given in this chapter.

In order to build a model for MAPKAT, the following parameters are needed:

- A set of agents;

- An *objective*, which may be one of the following:
  - a set of *goals*
  - a *time limit*
  - an extensive, complete model.

The objective is used as a halting criterion for the model construction. It'll be further explained in page 106.

Given a set of agents $\{X^1, X^2, \ldots, X^n\}$, a Multi-Agent System $\mathfrak{M} = \langle \mathbb{X}, \mathbb{N}, \mathbb{Q} \rangle$ is directly derived as the definition states:

1. Making $\mathbb{X} = \{X^1, X^2, \ldots, X^n\}$;
2. Making $\mathbb{N} = \bigcup_{i=1}^{n} \{\mathbb{O}^i \mid \mathbb{O}^i \text{ is agent } X^i\text{'s set of operators }\}$;
3. Making $\mathbb{Q} = \bigcup_{i=1}^{n} \{\mathbb{P}^i \mid \mathbb{P}^i \text{ is agent } X^i\text{'s set of facts }\}$;

Then a check is made to verify whether each element $\mathcal{O} \in \mathbb{N}$ is described by $\mathbb{Q}$. If, and only if, the check verifies so, then the MAS $\mathfrak{M}$ is recognized as well-formed and the construction of the model may proceed.

As stated before, the model is a directed, levelled graph. This graph is composed by several kinds of edges and two kinds of nodes: *proposition nodes*, which can be either *positive* or *negative*, and *action nodes*.

Each level of the graph is composed by only one kind of node. Levels composed by proposition nodes are called *proposition levels* and levels composed by action nodes are called *action levels*. Do not mistake the proposition and action levels of the model with the semantic concepts with the same name presented in definition C.2.12. The node levels of the model are mappings which carry similar meaning, but while the semantical constructs are sets of *propositions* or *operators*, the model levels are sets of *labelled nodes*.

The first and the last levels are always proposition levels. The immediate level after a proposition level, unless it is the last proposition level, is always an action level and the immediate level after an action level is always a proposition level.

It should be noted that, just like in the original planning graphs, the action nodes represent *fully instantiated* operators: they already have their parameters chosen and their outcome is a set of defined propositions.

The first proposition level is obtained from $G_0^{\mathcal{P}}$, which denotes the global truth of the initial global state of $\mathfrak{M}$: this level has one positive proposition node for each proposition $q \in G_0^{\mathcal{P}}$ and each node is labelled with the correspondent proposition. This level also includes one negative proposition node for each proposition $q \in (\mathbb{N} \setminus G_0^{\mathcal{P}})$.

All other levels are obtained by *expansion*. Expansion consists of an operation performed on a given proposition level — the *origin* — composed of these steps:

1. Form an action level composed by one action node for each operator applicable to the origin that doesn't have a pair of preconditions mutually exclusive in the origin — analogously, each action node is labelled with the corresponding operator.;

2. Link with special edges each action node to the proposition nodes of the origin labelled by the preconditions of its correspondent operator — these special edges are called *precondition edges*;

3. Scan the new action level to detect relations of mutual exclusion — the pairs of mutually exclusive action nodes detected are linked by *exclusion edges*;

4. Build a proposition level — called the *destination* — by following this procedure:

    - Copy each proposition node in the origin level to the (initially empty) destination level;

- For each proposition in the Add-list of each operator labelling a node in the newly-created level, add to the destination level one positive proposition node labelled with said proposition unless there already is a positive proposition node labelled with that proposition in the origin level;
- For each proposition p in the Del-list of each operator labelling a node in the newly-created level, add to the destination level one negative proposition node labelled with $neg(p)$ unless there already is a negative proposition node labelled with that proposition in the origin level;

5. Scan the destination to detect relations of mutual exclusion — the pairs of mutually exclusive proposition nodes detected also are linked by *exclusion edges*;

6. Link each action node to the positive proposition nodes of the destination labelled by the elements of the Add-List of its correspondent operator with *Add-Effect edges*;

7. Link each action node to the positive proposition nodes of the destination labelled by the elements of the Del-List of its correspondent operator with *Del-Effect edges*;

8. Link each action to any negative proposition nodes present in the destination labelled by the elements of the Del-List of its corresponding operator with *Add-Effect edges*;

9. Link each action to any negative proposition nodes present in the destination labelled by the elements of the Add-List with *Del-Effect edges.*

The rules to detect mutual exclusion between proposition or action nodes are the same as those in page 89, plus the rule that, for every proposition p, if the proposition level features both the positive and negative nodes for p, then they are mutually exclusive.

The *mutual exclusion edges* may violate the characterization of the model as a levelled graph, but this is a characteristic inherited from the planning graphs approach.

The model for MAPKAT is built by expanding first the initial proposition level and then each proposition level produced by a previous expansion. So far, it is done much like building a planning graph (see section A.6.3). The first, most obvious difference is the division of proposition nodes between positive and negative ones.

Another important difference appears at the end of the construction of the model. The process is concluded when the *objective* is fulfilled. The fulfillment of the objective is purposefully flexible. It covers any criteria one could have to stop the construction of the model in order to avoid making it unnecessarily large. Such valid objectives include:

1. Graph "levels off" (see section A.6.6) — useful for complete analysis of the problem and model-checking;

2. The last expansion produces a proposition level which includes a desired set of proposition nodes (or "goals") such as no two of these nodes are mutually exclusive — useful if only seeking a fast plan;

3. There is a maximum number of expansions allowed — useful to map time limit constraints.

The model, once finished, is closely related to a planning graph. It is not accidental. The planning graph was proven to be an efficient tool to map the complex interaction of dynamic facts. The use of a model similar to a planning graph allows MAPKAT to be efficient without sacrificing any expressive power.

This model is even closer to the application of planning graphs to multi-agent planning problems depicted in [Bui and Jamroga, 2003]. However, this model doesn't require for the agents to have a synchronous behavior. In fact, they may act at the same time, or not act at all.

D.2 Formulae Verification Algorithms

The verification algorithm calculates the runs which bring the MAS to satisfy some MAPKAT formula φ. It does so by calculating the *preconditions* of φ and checking whether the initial state satisfies these preconditions. The whole computation makes use of the levelled graph structure described in section D.1. For the purpose of these algorithms, we assume that the levelled graph is part of the MAS description $\mathfrak{M}$.

The preconditions of some formula φ are the preconditions of at least one run r within the model that brings about the property encoded in φ. The process of calculating the mentioned preconditions is the same as calculating r and, thus, the tuple of r and its preconditions is called a *solution.*

A solution is calculated in a backward-chaining fashion, detecting what is needed to satisfy the atomic (or "most inward") portions of the formula and then, step-by-step, it calculates what is needed to satisfy the whole formula.

Each step corresponds to the treatment of one logical operator and it is always based on some Preposition Level (PL) of the MAS. The data returned by each step of the verification algorithms is always a set of tuples (C, E, r) such as:

- C and E are sets of prepositions and r is a run;
- C is the set of preconditions for the run r and is contained in some valid global state of the PL of the current step;
- $\mathfrak{M}, G \models \varphi$, where G is the global state obtained by applying r to some global state of the PL of the current step;
- E is a set of evidences required to be collected for model-checking knowledge of some agent.

In other words, each step of the process calculates the set of *possible solutions* for an operator or preposition of the formula. Each solution is to be viewed as a *possibility* and such a possibility is an ordered pair composed of a set of *necessary prepositions* (the preconditions) and a *necessary run.*

The first step is based on the initial state of the system given by the problem configuration. That is the first preposition level, or PL_0 for short. Once the set of possible solutions of all of φ are calculated, we return only the solutions whose sets of necessary prepositions are a subset of PL_0.

Boolean operators are evaluated on the current preposition level. Their possible solutions involve *only* the prepositions for that preposition level and they don't manipulate runs in any way.

The model-checking of time-modal operators involves two preposition levels, the current one as the *origin* and a later one as the *destination*. They always return possibilities composed of a run, which leads to a future state that satisfies some formula, and the prepositions that form the precondition of the said run. These prepositions are always required to form a subset of the global truth of a valid global state on the origin preposition level.

The Knowledge operator is a bit complicated. The original semantics in section C.3.1 involves iterations through runs, which can be too slow to compute. Instead, we follow the following criteria.

While model-checking a formula φ, we need to find certain prepositions required for the verification of φ. Some of these prepositions may be known by an agent i, while the others aren't. By definition, every agent knows the prepositions which form its own local state. The problem is identifying the prepositions of the local states of other agents.

We assume that each agent knows the preconditions, the add-effects and the del-effects of every operator. We also assume that every agent knows (or "can see") when some action is executed by some agent. Thus the identification of the prepositions of other agents depends on the actions these agents perform: in fact, some agent i cannot be sure of anything about the state of any other agent, except whatever facts the actions of the other agent make evident. Generally, an action makes evident any preposition belonging to either its Add-list or to its Preconditions (except for those prepositions that may appear in its Del-list - On the contrary, the negation of the elements in its Del-list is evident).

To model-check knowledge, the algorithm uses the *evidence*. This parameter indicates the prepositions which must be made evident by a run to be calculated in the next step of model checking. Notice that it doesn't use or trigger the planning algorithm, it only signals a special kind of target. If by the end of the model checking no step has triggered the planning routine, the model-check returns a failure.

This computation of the knowledge operator isn't proven to fulfill the exact semantics of the knowledge operator presented in section C.3.1, but it is trivial that this approach computes a correct subset of the original semantics.

D.2.1 Root Algorithms

```
Procedure IsValid($\mathfrak{M}, \varphi$)
Data: A MAS $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: In case of successful model-checking the value TRUE is returned
        followed by a list of runs $r$, if applicable, such as each $r$ brings $\mathfrak{M}$ to
        a global state $G$ where $\mathfrak{M}, G \models \varphi$. Otherwise, it returns FALSE.
begin
    $P \leftarrow$ CheckValid($\mathfrak{M}, 0, \varphi$);
    $R \leftarrow \emptyset$;
    for each tuple $t$ =($C, E, r$), $t \in P$ do
        if $E = \emptyset$ then
            $R \leftarrow R \cup \{r\}$;
        else
            $P \leftarrow P \setminus t$;
    if $P = \emptyset$ then
        return (False,<null>)
    else
        return (True,R);
end
```

Procedure `CheckValid`($\mathfrak{M}, PL, \varphi$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    Let $x$ be an empty set;
    switch $\varphi$ is of the form... do
        case $(\theta)$
            $x \leftarrow$CheckValid($\mathfrak{M}, PL, \theta$);
        case $\neg\theta$
            $x \leftarrow$CheckValidBoolean($\mathfrak{M}, PL, \varphi$);
        case $\theta_1 \wedge \theta_2$
            $x \leftarrow$CheckValidBoolean($\mathfrak{M}, PL, \varphi$);
        case $\theta_1 \vee \theta_2$
            $x \leftarrow$CheckValidBoolean($\mathfrak{M}, PL, \varphi$);
        case $\theta_1 \rightarrow \theta_2$
            $x \leftarrow$CheckValidBoolean($\mathfrak{M}, PL, \varphi$);
        case $\exists\theta$
            $x \leftarrow$CheckValidTime($\mathfrak{M}, PL, \varphi$);
        case $\forall\theta$
            $x \leftarrow$CheckValidTime($\mathfrak{M}, PL, \varphi$);
        case $\mathcal{K}_i\theta$
            $x \leftarrow$CheckValidKnowledge($\mathfrak{M}, PL, i, \theta$);
        otherwise
            {$\varphi$ is actually a preposition.} $x \leftarrow \{(\{\varphi\}, \emptyset, \emptyset)\}$
    return $x$;
end
```

Procedure `CheckValidBoolean`($\mathfrak{M}, PL, \varphi$)

Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a boolean formula φ.

Result: A set of tuples (C, E, r) as explained before.

```
begin
    Let $x$ be an empty set;
    switch $\varphi$ is of the form... do
        case $\neg\theta$
            $x \leftarrow$CheckNeg($\mathfrak{M}, PL, \theta$);
        case $\theta_1 \wedge \theta_2$
            $x \leftarrow$CheckAnd($\mathfrak{M}, PL, \theta_1, \theta_2$);
        case $\theta_1 \vee \theta_2$
            $x \leftarrow$CheckOr($\mathfrak{M}, PL, \theta_1, \theta_2$);
        case $\theta_1 \rightarrow \theta_2$
            $x \leftarrow$CheckImp($\mathfrak{M}, PL, \theta_1, \theta_2$);
    return $x$;
end
```

Procedure `CheckValidTime(`$\mathfrak{M}, PL, \varphi$`)`

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a time-modal MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    Let $x$ be an empty set;
    switch $\varphi$ is of the form... do
        case $\exists \bigcirc \theta$
            $x \leftarrow$ CheckExistsNext($\mathfrak{M}, PL, \theta$);
        case $\forall \bigcirc \theta$
            $x \leftarrow$ CheckForAllNext($\mathfrak{M}, PL, \theta$);
        case $\exists \Diamond \theta$
            $x \leftarrow$ CheckExistsEventual($\mathfrak{M}, PL, \theta$);
        case $\forall \Diamond \theta$
            $x \leftarrow$ CheckForAllEventual($\mathfrak{M}, PL, \theta$);
        case $\exists \Box \theta$
            $x \leftarrow$ CheckExistsConstant($\mathfrak{M}, PL, \theta$);
        case $\forall \Box \theta$
            $x \leftarrow$ CheckForAllConstant($\mathfrak{M}, PL, \theta$);
        case $\exists [\theta_1 \mathsf{U} \theta_2]$
            $x \leftarrow$ CheckExistsUntil($\mathfrak{M}, PL, \theta_1, \theta_2$);
        case $\forall [\theta_1 \mathsf{U} \theta_2]$
            $x \leftarrow$ CheckForAllUntil($\mathfrak{M}, PL, \theta_1, \theta_2$);
    return $x$;
end
```

D.2.2 Boolean Algorithms

```
Procedure CheckNeg($\mathfrak{M}, PL, \varphi$)
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $P \leftarrow$ CheckValid($\mathfrak{M}, PL, \varphi$);
    $R \leftarrow \emptyset$;
    $P \leftarrow \emptyset$;
    $C' \leftarrow \emptyset$;
    for each $(C, E, r) \in P$ do
        for each $p \in C$ do
            if $p$ is of form "not($q$)" then
                $C' \leftarrow C' \cup \{q\}$;
            else
                $C' \leftarrow C' \cup \{\text{not}(p)\}$;
        $P' \leftarrow$ Junction($P', C'$);
    for each $D \in P'$ do  $R \leftarrow R' \cup \{(D, \emptyset, \text{<null>})\}$;
    return $R$;
end
```

Procedure CheckOr($\mathfrak{M}, PL, \varphi_1, \varphi_2$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and two MAPKAT formulae: $\varphi_1$ and $\varphi_2$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $P_1 \leftarrow CheckValid(\mathfrak{M}, PL, \varphi_1)$;
    $P_2 \leftarrow CheckValid(\mathfrak{M}, PL, \varphi_2)$;
    $P \leftarrow \{p \mid p \in (P_1 \cup P_2)\}$;
    for each $(C, E, r) \in P$ do
        $D \leftarrow \{p \mid p \in C\}$;
        $F \leftarrow \{q \mid q \in E\}$;
        $s \leftarrow$<null> ;
        $R \leftarrow R \cup \{(D, F, s)\}$;
    return $R$;
end
```

Procedure CheckAnd($\mathfrak{M}, PL, \varphi_1, \varphi_2$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and two MAPKAT formulae: $\varphi_1$ and $\varphi_2$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $P_1 \leftarrow CheckValid(\mathfrak{M}, PL, \varphi_1)$;
    $P_2 \leftarrow CheckValid(\mathfrak{M}, PL, \varphi_2)$;
    $R \leftarrow \emptyset$;
    for each $(C_x, E_x, r_x) \in P_1$ do
        for each $(C_y, E_y, r_y) \in P_2$ do
            $C_z \leftarrow \{p \mid p \in (C_x \cup C_y)\}$;
            $E_z \leftarrow \{q \mid q \in (E_x \cup E_y)\}$;
            $r_z \leftarrow$<null> ;
            if $PLValid(\mathfrak{M}, C_z \cup E_z, PL)$ then
                $R \leftarrow R \cup \{(C_z, E_z, r_z)\}$;
    return $R$;
end
```

Procedure CheckImp($\mathfrak{M}, PL, \varphi_1, \varphi_2$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and two MAPKAT formulae: $\varphi_1$ and $\varphi_2$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    return $CheckValid(\mathfrak{M}, PL, \varphi 2 \vee \neg\varphi_1)$;
end
```

D.2.3 Time-Modal Algorithms

Procedure `CheckExistsNext`($\mathfrak{M}, PL, \varphi$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $P \leftarrow \mathit{CheckValid}(\mathfrak{M}, PL + 1, \varphi)$;
    $R \leftarrow \emptyset$;
    for each $(C, E, s) \in P$ do
        if PLValid($\mathfrak{M}, C \cup E, PL + 1$) then
            $P' \leftarrow \mathit{CheckPlan}(\mathfrak{M}, PL, PL + 1, C, E)$;
            for each $(C', E', s') \in P'$ do
                if $\mathit{PLValid}(\mathfrak{M}, C', PL)$ then
                    $r \leftarrow s' + s$;
                    $R \leftarrow R \cup \{(C', E', r)\}$;
    return $R$;
end
```

Procedure `CheckForAllNext`($\mathfrak{M}, PL, \varphi$)

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $R \leftarrow$ CheckValid($\mathfrak{M}, PL, \neg\exists \bigcirc \neg\varphi$);
    return $R$;
end
```

Procedure `CheckExistsEventual`($\mathfrak{M}, PL, \varphi$)

Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula φ.

Result: A set of tuples (C, E, r) as explained before.

```
begin
    $PT \leftarrow PL$;
    $Ready \leftarrow No$;
    $R \leftarrow \emptyset$;
    while $Ready \neq Yes$ and $PT \leq LastPL(\mathfrak{M})$ do
        $PT \leftarrow PT + 1$;
        $P \leftarrow CheckValid(\mathfrak{M}, PT, \varphi)$;
        for each $(C, E, s) \in P$ do
            if $PLValid(\mathfrak{M}, C \cup E, PT)$ then
                $P' \leftarrow CheckPlan(\mathfrak{M}, PL, PT, C, E)$;
                for each $(C', E', s') \in P'$ do
                    if $PLValid(\mathfrak{M}, C', PL)$ then
                        $r \leftarrow s' + s$;
                        $R \leftarrow R \cup \{(C', E', r)\}$;
                        $Ready \leftarrow Yes$;
    return $R$
end
```

Procedure `CheckExistsConstant`($\mathfrak{M}, PL, \varphi$)

Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula φ.

Result: A set of tuples (C, E, r) as explained before.

```
begin
    $PT \leftarrow PL$;
    $Ready \leftarrow No$;
    $R \leftarrow \emptyset$;
    while $Ready \neq Yes$ and $PT \leq LastPL(\mathfrak{M})$ do
        $PT \leftarrow PT + 1$;
        $P \leftarrow CheckValid(\mathfrak{M}, PT, \varphi)$;
        for each $(C, E, s) \in P$ do
            if $PLValid(\mathfrak{M}, C \cup E, PT)$ then
                $P' \leftarrow CheckPlanFix(\mathfrak{M}, PL, PT, C, E, \varphi)$;
                for each $(C', E',' s) \in P'$ do
                    if $PLValid(\mathfrak{M}, C, PL)$ then
                        $r \leftarrow s' + s$;
                        $R \leftarrow R \cup \{(C', E', r)\}$;
                        $Ready \leftarrow Yes$;
    return $R$
end
```

Procedure `CheckExistsUntil`$(\mathfrak{M}, PL, \varphi_1, \varphi_2)$

```
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and two MAPKAT formulae: $\varphi_1$ and $\varphi_2$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $PT \leftarrow PL$;
    $Ready \leftarrow No$;
    $R \leftarrow \emptyset$;
    while $Ready \neq Yes$ and $PT \leq LastPL(\mathfrak{M})$ do
        $PT \leftarrow PT + 1$;
        $P \leftarrow \text{CheckValid}(\mathfrak{M}, PT, \varphi_2)$;
        for each $(C, E, s) \in P$ do
            if $PLValid(\mathfrak{M}, C \cup E, PT)$ then
                $P' \leftarrow \text{CheckPlanFix}(\mathfrak{M}, PL, PT, C, E, \varphi_1)$;
                for each $(C', E', s') \in P'$ do
                    if $PLValid(\mathfrak{M}, C, PL)$ then
                        $r \leftarrow s' + s$;
                        $R \leftarrow R \cup \{(C', E', r)\}$;
                        $Ready \leftarrow Yes$;
    return $R$
end
```

```
Procedure CheckForAllEventual($\mathfrak{M}, PL, \varphi$)
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $R \leftarrow$ CheckValid($\mathfrak{M}, PL, \neg\exists\Box\neg\varphi$);
    return $R$;
end
```

```
Procedure CheckForAllConstant($\mathfrak{M}, PL, \varphi$)
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $R \leftarrow$ CheckValid($\mathfrak{M}, PL, \neg\exists\Diamond\neg\varphi$);
    return $R$;
end
```

```
Procedure CheckForAllUntil($\mathfrak{M}, PL, \varphi_1, \varphi_2$)
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$ and two MAPKAT formulae: $\varphi_1$ and $\varphi_2$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $R \leftarrow$ CheckValid($\mathfrak{M}, PL, \neg\exists[\neg\varphi_2 \mathsf{U}(\neg\varphi_1 \wedge \neg\varphi_2)] \wedge \neg\exists\Box\neg\varphi_2$);
    return $R$;
end
```

D.2.4 Knowledge-Modal Algorithms

```
Procedure CheckValidKnowledge($\mathfrak{M}, PL, i, \varphi$)
Data: A MAS $\mathfrak{M}$, some PL of $\mathfrak{M}$, an agent index $i$ and a MAPKAT formula $\varphi$.
Result: A set of tuples $(C, E, r)$ as explained before.
begin
    $P \leftarrow$ CheckValid($\mathfrak{M}, PL, \varphi$);
    $R \leftarrow \emptyset$;
    $S \leftarrow$ LocalPrepositions($\mathfrak{M}, i$);
    for each tuple $B \in P, B = (C, E, r)$ do
        $X_1 \leftarrow \{p \mid \forall p \in (S \cap C)\}$;
        $X_2 \leftarrow \{p \mid \forall p \in (C \setminus X_1)\}$;
        $X_2 \leftarrow E \cup X_2$;
        $R \leftarrow R \cup \{(X_1, X_2, r)\}$;
    return $R$;
end
```

D.2.5 Utilitary Algorithms

This subsection details all the algorithms that support the correct functioning of the formula verification algorithms presented in section D.2. While the verification algorithms deal explicitly with the logic of the operators they encode, the utilitary algorithms described in this section are the "tools" through which those operators can be encoded.

```
Procedure PLValid($\mathfrak{M}, PL, P$)
Data: A MAS $\mathfrak{M}$, the number $PL$ of some preposition level in $\mathfrak{M}$ and a set
      of prepositions $P$.
Result: TRUE, if the elements in $P$ belong to the global truth of some valid
        global state in preposition level $PL$ of $\mathfrak{M}$; FALSE, otherwise.
begin
    Let $X$ be the set of mutual exclusion edges among the elements of the
    preposition level $PL$ of $\mathfrak{M}$;
    foreach $(p, q) \in X$ do
        if $p \in X$ and $q \in X$ then
            return FALSE;
    return TRUE;
end
```

Above is the description of the main procedure used to verify the correctness of any solution found by the model-checking process. It uses two principles: First, each preposition level of any Multi-Agent System model is the minimal superset of the sets of Global Truths corresponding to each and every possible Global State in a given time step. Second, any subset of a preposition level is a subset of the global truth of at least one global state as long as it doesn't contain any pair of mutually-exclusive prepostions.

```
Procedure CheckPlan($\mathfrak{M}$, $PL_I$, $PL_F$, $Goals$, $Evidence$)
Data: A MAS $\mathfrak{M}$, The PL $PL_I$ of $\mathfrak{M}$ where the plan should begin, the PL
      $PL_F$ where the plan should end, the set of $Goals$ and the set of
      required $Evidences$.
Result: The set of tuples (C,E,r) which describe the possible plans found.
begin
    if not PLValid($\mathfrak{M}$, $PL_F$, $Goals$) then
        return $\emptyset$;
    else
        $G \leftarrow \{Goals \cup Evidence\}$;
        $R \leftarrow$ RecursivePlanStep($\mathfrak{M}, PL_I, PL_F, G$);
        foreach $(C, E, r) \in R$ do
            if $(C \cap Evidence) = \emptyset$ then
                $R \leftarrow R \setminus \{(C, E, r)\}$;
        return $R$;
end
```

```
Procedure RecursivePlanStep($\mathfrak{M}$, $PL_I$, $PL_F$, $Goals$)
Data: A MAS $\mathfrak{M}$, The PL $PL_I$ of $\mathfrak{M}$ where the subplan should begin, the
      PL $PL_F$ where the subplan should end and the set of $Goals$.
Result: The set of tuples (C,E,r) which describe the possible subplans found.
begin
    if $PL_I < (PL_F - 1)$ then
        $P \leftarrow$ RecursivePlanStep($\mathfrak{M}, PL_I + 1, PL_F$,Goals);
    else
        $P \leftarrow \{$(Goals, $\emptyset$, <null>)$\}$;
    $R \leftarrow \emptyset$;
    foreach $(C, E, r) \in P$ do
        $L \leftarrow \emptyset$;
        foreach $p \in C$ do
            $L_p \leftarrow$Causes($\mathfrak{M}, AL_I + 1, p$);
            $L \leftarrow$Junction($L, L_p$);
        foreach $m \in L$ do
            $X \leftarrow \emptyset$;
            foreach $a \in m$ do
                $X \leftarrow X \cup$ Preconds($\mathfrak{M}, a$);
            if PLValid($\mathfrak{M}, PL_I, X$) then
                $R \leftarrow R \cup \{(X, E,$AppendRunToMove$(r, m))\}$;
    return $R$;
end
```

The previous two procedures compose a translation of the original Graphplan planning algorithm for the MAPKAT model. It was adaptated at the initial call so it can handle the use of Evidence Sets properly. The next two procedures form an useful extension: a planning algorithm that ensures that a given formula φ is true at each and every point of any returned plan.

```
Procedure CheckPlanFix($\mathfrak{M}$, $PO$, $PF$, $Goals$, $\varphi$)
Data: A MAS $\mathfrak{M}$, The PL $PO$ of $\mathfrak{M}$ where the plan should begin, the PL
      $PF$ where the plan should end, the set of $Goals$ and a MAPKAT
      formula $\varphi$.
Result: The set of tuples (C,r) which describe the possible plans found.
begin
    begin
        if not PLValid($\mathfrak{M}$, $PL_F$, $Goals$) then
            return $\emptyset$;
        else
            $G \leftarrow \{$ Goals $\cup$ Evidence$\}$;
            $R \leftarrow$RecursivePlanFixStep($\mathfrak{M}, PL_I, PL_F, G, \varphi$);
            foreach $(C, E, r) \in R$ do
                if $(C \cap$ Evidence$) = \emptyset$ then
                    $R \leftarrow R \setminus \{(C, E, r)\}$;
            return $R$;
    end
end
```

```
Procedure RecursivePlanFixStep($\mathfrak{M}$, $PL_I$, $PL_F$, $Goals$, $\varphi$)
Data: A MAS $\mathfrak{M}$, The PL $PL_I$ of $\mathfrak{M}$ where the subplan should begin, the
      PL $PL_F$ where the subplan should end, the set of $Goals$ and a
      MAPKAT formula $\varphi$.
Result: The set of tuples (C,E,r) which describe the possible subplans found.
begin
    if $PL_I < (PL_F - 1)$ then
        $P \leftarrow$ RecursivePlanFixStep($\mathfrak{M}, PL_I + 1, PL_F$,Goals, $/varphi$);
    else
        $P \leftarrow \{(\text{Goals}, \emptyset, \text{<null>})\}$;
    $Q \leftarrow$ CheckValid($\mathfrak{M}, PL_I, \varphi$);
    $R \leftarrow \emptyset$;
    $S \leftarrow$ CombineSolutions($\mathfrak{M}, PL_I, P, Q$);
    foreach $(C, E, r) \in P$ do
        $L \leftarrow \emptyset$;
        foreach $p \in P$ do
            $L_p \leftarrow$Causes($\mathfrak{M}, AL_I + 1, p$);
            $L \leftarrow$Junction($L, L_p$);
        foreach $m \in L$ do
            $X \leftarrow \emptyset$;
            foreach $a \in m$ do
                $X \leftarrow X \cup$ Preconds($\mathfrak{M}, a$);
            if PLValid($\mathfrak{M}, PL_I, X$) then
                $R \leftarrow R \cup \{(X, E, \text{AppendRunToMove}(r, m))\}$;
    return $R$;
end
```

Procedure `Causes`$(\mathfrak{M}, AL, p)$

```
Data: A MAS $\mathfrak{M}$, a chosen action level $AL$ of $\mathfrak{M}$ and a single preposition $p$.
Result: The subset of $AL$ formed by the actions with $p$ in their Add-lists.
begin
    $R \leftarrow \emptyset$;
    $T \leftarrow \{a \mid a$ appears on action level number $AL$ of $\mathfrak{M}\}$;
    foreach action $a \in T$ do
        if $p \in Addlist(\mathfrak{M}, a)$ then
            $R \leftarrow R \cup \{\{a\}\}$;
    return $R$;
end
```

Procedure `Junction`(L_1, L_2)

```
Data: Two sets of sets: $L_1$ and $L_2$.
Result: The set of unions $X \cup Y$ for all $X \in L_1$ and $Y \in L_2$.
begin
    $R \leftarrow \emptyset$;
    foreach $X \in L_1$ do
        foreach $Y \in L_2$ do
            $R \leftarrow R \cup \{X \cup Y\}$;
    return $R$;
end
```

Procedure `CombineSolutions`($\mathfrak{M}, PL, P, Q$)

```
Data: A MAS $\mathfrak{M}$, a preposition level number $PL$ for some preposition level
      of $\mathfrak{M}$ and two solution sets $P$ and $Q$ which represent the respective
      solution sets for two unknown formulae $\varphi_P$ and $\varphi_Q$.
Result: The set of solutions for the unknown formula $\varphi_P \wedge \varphi_Q$ for the
        preposition level $PL$ of $\mathfrak{M}$.
begin
    $R \leftarrow \emptyset$;
    foreach $(C_P, E_P, r_P) \in P$ do
        foreach $(C_Q, E_Q, r_Q) \in Q$ do
            if PLValid($\mathfrak{M}, PL, C_P \cup C_Q \cup E_P \cup E_Q$) then
                $r' \leftarrow$ CombineRuns($\mathfrak{M}, PL, r_P, r_Q$);
                if $r' \neq$ FAILURE then
                    $R \leftarrow R \cup \{(C_P \cup C_Q, E_P \cup E_Q, r')\}$;
    return $R$;
end
```

```
Procedure CombineRuns($r_1$, $r_2$)
Data: A MAS $\mathfrak{M}$, a preposition level number $PL$ for some preposition level
      of $\mathfrak{M}$ and two runs, supposedly starting at preposition level $PL$: $r_1$
      and $r_2$.
Result: The run resulting from executing $r_1$ and $r_2$ in parallel. If they can't
        run in parallel, the algorithm returns FALSE. Notice that FALSE is
        different of <null>.
begin
    Let $r^l$ be the longest run among $r_1$ and $r_2$;
    Let $r^s$ be the shortest run among $r_1$ and $r_2$;
    Let $l^l$ be the length of $r^l$;
    Let $l^s$ be the length of $r^s$;
    $r \leftarrow$<null>;
    $safe \leftarrow$ TRUE;
    for $i$ from 1 to $l^s$ do
        Let $m^l$ be the $i - th$ move of $r^l$;
        Let $m^s$ be the $i - th$ move of $r^s$;
        if $Exclusive(\mathfrak{M}, PL + i, m^l, m^s)$ then return FAILURE;
        $m \leftarrow \{a \mid a \in (m^l \cup m^s)\}$;
        $r \leftarrow r + m$;
    for $i$ from $l^s + 1$ to $l^l$ do
        Let $m^l$ be the $i - th$ move of $r^l$;
        $m \leftarrow \{a \mid a \in m^l\}$;
        $r \leftarrow r + m$;
    return $r$;
end
```

```
Procedure AppendMoveToRun($r, m$)
  Data: A run $r$ and a move $m$.
  Result: The run $r' = rm$.
  begin
    return $r$+$m$;
  end
```

```
Procedure AppendRunToMove($r, m$)
  Data: A move $m$ and a run $r$ .
  Result: The run $r' = mr$.
  begin
    return $m$+$r$;
  end
```

```
Procedure Preconds($\mathfrak{M}, a$)
  Data: A MAS $\mathfrak{M}$ and an operator $a$.
  Result: The set of prepositions which forms the list of preconditions of $a$.
  begin
    Check the definition of $a$ in $\mathfrak{M}$ and copy the preconditions of $a$ to some
    set $C$;
    return $C$;
  end
```

```
Procedure Addlist($\mathfrak{M}, a$)
  Data: A MAS $\mathfrak{M}$ and an operator $a$.
  Result: The set of prepositions which forms the add-list of $a$.
  begin
    Check the definition of $a$ in $\mathfrak{M}$ and copy the add-list of $a$ to some set $C$;
    return $C$;
  end
```

```
Procedure Dellist($\mathfrak{M}, a$)
  Data: A MAS $\mathfrak{M}$ and an operator $a$.
  Result: The set of prepositions which forms the del-list of $a$.
  begin
      Check the definition of $a$ in $\mathfrak{M}$ and copy the del-list of $a$ to some set $C$;
      return $C$;
  end
```

Bibliography

[Ben-Ari et al., 1981] Ben-Ari, M., Manna, Z., and Pnueli, A. (1981). The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium On Principles Of Programming Languages*, pages 164–176, Williamsburg, Virginia.

[Benevides et al., 2004] Benevides, M. R. F., Delgado, C. A. D., Pombo, C. G. L., Lopes, L. R. M., and Ribeiro, R. F. (2004). Model checking knowledge in multi-agent systems. Technical Report ES-653/04, PESC, Rio de Janeiro, RJ.

[Blum and Furst, 1995] Blum, A. and Furst, M. (1995). Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642.

[Bui and Jamroga, 2003] Bui, T. D. and Jamroga, W. (2003). Multi-agent planning with planning graph. In *Eunite 2003*, pages 558–565.

[Clark et al., 1999] Clark, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking.* MIT Press.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565.

[Russel and Norvig, 1995] Russel, S. J. and Norvig, P. (1995). *Artificial Inteligence, a Modern Approach.* Prentice-Hall, first edition.

[van der Hoek and Wooldridge, 2002] van der Hoek, W. and Wooldridge, M. (2002). Tractable multiagent planning for epistemic goals. In *Proceedings of the First International Conference on Autonomous Agents and Multiagent Systems.*

[van der Meyden, 1994] van der Meyden, R. (1994). Axioms for knowledge and time in distributed systems with perfect recall. In *Logic in Computer Science*, pages 448–457.