Raciocínio Abdutivo usando Cálculo de Eventos e sua correspondência com Sistemas de Planejamento

Silvio do Lago Pereira

Projeto de Dissertação de Mestrado Apresentado ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
como Requisito Parcial para Qualificação no
Mestrado em Ciência da Computação

Curso: **Mestrado em Ciência da Computação**
Área de Concentração: **Inteligência Artificial**
Orientadora: **Prof. Dra. Leliane Nunes de Barros**

O aluno recebe apoio financeiro da CAPES.

— São Paulo, SP - Fevereiro de 2001 —

Lista de Figuras

Sumário

1 Introdução

A habilidade de planejar é essencial ao comportamento inteligente e sua implementação é extremamente importante em aplicações práticas como, por exemplo, robótica, manufatura, logística, planejamento de grades curriculares, planejamento de missões espaciais, planejamento de provas de teoremas, etc. De fato, planejar atividades com antecedência é uma tarefa presente no dia-a-dia das pessoas. Por exemplo, planejar uma viagem ao exterior requer a execução de um certo número de ações, em uma determinada ordem no tempo, de forma a garantir o sucesso da viagem. O correto seqüenciamento de ações é, geralmente, necessário para que objetivos específicos sejam atingidos.

Um grande número de algoritmos de planejamento foram propostos nos últimos trinta anos na área de Inteligência Artificial. Os algoritmos provados corretos possuem grandes limitações, em particular, quanto à representação de ações e eventos e, conseqüentemente, não podem ser usados para resolver problemas no mundo real. Por outro lado, os chamados planejadores práticos, capazes de resolver problemas grandes, em geral, foram construídos de maneira *ad-hoc*. Para tais sistemas, é difícil explicar porque eles funcionam ou porque o seu comportamento pode ser considerado inteligente.

Conforme *Shanahan* [34],

> "A melhor maneira de se explicar um comportamento inteligente (...) é interpretá-lo como produto de um *raciocínio correto* sobre uma *representação correta*. (...) A melhor (ferramenta), na verdade a única candidata real que temos para explicar os conceitos de representação correta e raciocínio correto, é a lógica formal. Desta forma, uma vez que ações e eventos são tão importantes em nossas representações, precisamos compreender como representá-los, bem como seus efeitos, em lógica formal." [*Shanahan*, 1997, pp. *xx-xxi*]

Assim, acredita-se que através do uso de lógica formal é possível construir planejadores corretos, fundamentados em princípios bem conhecidos, que podem ser facilmente validados, mantidos e modificados. Resta investigar se é possível construir planejadores baseados em lógica que sejam considerados "práticos" e capazes de resolver problemas do mundo real.

Green [10] foi o primeiro a implementar um sistema de planejamento dentro de uma abordagem lógica. Entretanto, embora seu sistema tenha sido muito admirado do ponto de vista teórico, na prática, ele se mostrou bastante ineficiente. Tal ineficiência é devida, sobretudo, à necessidade de se manter um grande número de axiomas para estabelecer que propriedades e relações persistem no mundo, após a execução de uma ação (*problema do quadro*). Conforme *McCarthy* e *Hayes* [23] observam, essa necessidade é inerente a qualquer formalismo para representação de ações e efeitos baseado em lógica monotônica. Em virtude disso, e devido ao fato de diversos algoritmos resolverem satisfatoriamente o problema do quadro, criou-se uma falsa idéia de que a abordagem lógica não pode ser usada para construir sistemas de planejamento eficientes [31]. Este foi um dos motivos pelos quais a abordagem algorítmica se tornou predominante na área de planejamento.

Recentemente, entretanto, *Shanahan* [35] publicou um artigo em que a abordagem lógica é resgatada. Nesse artigo, ele mostra que, usando *cálculo de eventos circunscritivo* como formalismo para raciocinar sobre ações e seus efeitos, e *programação lógica abdutiva* como técnica de prova de teoremas, é possível reproduzir a computação efetuada por um algoritmo de planejamento. De fato, *Shanahan* sugere que o planejador lógico descrito por ele equivale ao planejador UCPOP [27], cuja representação de ações incorpora aspectos que estendem as capacidades dos chamados planejadores clássicos.

1.1 Objetivos

A proposta deste trabalho é modificar o planejador abdutivo baseado em cálculo de eventos, proposto por *Shanahan* [35], de forma a torná-lo equivalente a alguns algoritmos selecionados da literatura de planejamento em Inteligência Artificial. Com isso, visamos mostrar que, usando um formalismo puramente lógico, é possível obter sistemas de planejamento cuja eficiência seja equiparável àquela observada nos sistemas produzidos na abordagem algorítmica.

1.2 Organização

Este projeto de dissertação está organizado da seguinte maneira: a seção 2 define alguns formalismos para representação de conhecimento (cálculo de situações, STRIPS, ADL e cálculo de eventos) empregados em sistemas de planejamento; a seção 3 faz uma descrição sucinta da área de planejamento e de alguns algoritmos que serão implementados usando abdução. A seção 4 introduz planejamento como uma tarefa abdutiva e mostra como ela pode ser implementada com cálculo de eventos circunscritivo; a seção 5 apresenta a metodologia que será empregada para obtenção dos resultados propostos e, finalmente, a seção 6 estabelece o cronograma de atividades.

2 Representação de conhecimento

Essa seção apresenta os formalismos para representação de conhecimento empregados nos sistemas de planejamento abordados nesse trabalho. A linguagem desses formalismos é derivada da lógica de primeira ordem [1] e adota a seguinte convenção: variáveis são representadas por letras minúsculas e são consideradas universais quando não são explicitamente quantificadas.

2.1 Cálculo de situações

O *cálculo de situações*, introduzido por *McCarthy* e *Hayes* [23], é um formalismo lógico para raciocínio sobre ações e efeitos cuja ontologia inclui *situações*, que são como instantâneos do mundo; *fluentes*, que denotam propriedades ou relacionamentos que podem mudar de uma situação para outra; e *ações*, que transformam uma situação em outra. Na linguagem do cálculo de situações, a constante S_0 denota a *situação inicial*, a função $Do(\alpha, \sigma)$ denota a *situação resultante* da execução da ação α numa determinada situação σ e o predicado $Holds(\phi, \sigma)$ estabelece que o fluente ϕ vale na situação σ.

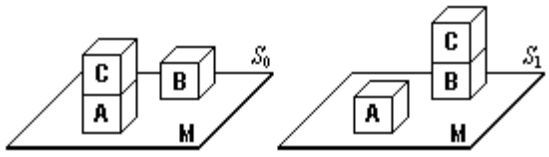

Figura 1: *A situação S_1 é resultante da execução da ação $Move(C, B)$ na situação S_0*

Considere o domínio do *mundo dos blocos*, que consiste numa série de blocos empilhados sobre uma mesa e cujo interesse é raciocinar sobre os efeitos de se mover blocos de um local para outro. O fluente $Sobre(x, y)$ pode ser usado para denotar que o bloco x está sobre o bloco y e o fluente $Livre(x)$, para denotar que não existe nenhum outro bloco sobre x. A situação $Do(Move(x, y), s)$ é aquela que resulta quando a ação $Move(x, y)$, que coloca um bloco x sobre outro y, é executada numa certa situação s.

Por exemplo, a conjunção Σ das fórmulas a seguir descreve a situação inicial para o problema conhecido como *anomalia de Sussman*, ilustrada na figura 1:

$Holds(Sobre(A, M), S_0)$
$Holds(Sobre(B, M), S_0)$
$Holds(Sobre(C, A), S_0)$
$Holds(Livre(B), S_0)$
$Holds(Livre(C), S_0)$

e a conjunção Δ dos *axiomas de efeito* a seguir, descreve a ação $Move$:

$$Holds(Sobre(x, y), Do(Move(x, y), s)) \leftarrow$$
$$Holds(Livre(x), s) \wedge Holds(Livre(y), s) \wedge x \neq y \wedge x \neq M$$
$$Holds(Livre(z), Do(Move(x, y), s)) \leftarrow$$
$$Holds(Livre(x), s) \wedge Holds(Livre(y), s) \wedge Holds(Sobre(x, z), s) \wedge x \neq y \wedge y \neq z$$

2.1.1 O problema do quadro

Quando o cálculo de situações é usado para descrever efeitos de ações, além de descrever o que muda, é preciso descrever também aquilo que permanece inalterado. Seja $S_1 := Do(Move(C, B), S_0)$ a situação, ilustrada na figura 1, que resulta quando o bloco C é movido sobre o bloco B, na situação inicial S_0. Claramente, temos $\Sigma \cup \Delta \models Holds(Sobre(C, B), S_1)$; entretanto, apesar de A estar sobre M em S_0, e de $Move(C, B)$ não alterar esse fato em S_1, não temos $\Sigma \cup \Delta \models Holds(Sobre(A, M), S_1)$.

Os axiomas de efeito conseguem descrever as mudanças resultantes da execução de uma ação, mas falham em representar aquilo que não muda. Para capturar a persistência dos fluentes que não são afetados por uma ação, precisamos ter *axiomas de quadro*. Tomando Δ' igual a Δ acrescido dos axiomas de quadro a seguir, temos $\Sigma \cup \Delta' \models Holds(Sobre(A, M), Do(Move(C, B), S_0))$, conforme esperado.

$$Holds(Sobre(v, w), Do(Move(x, y), s)) \leftarrow Holds(Sobre(v, w), s) \wedge x \neq v$$
$$Holds(Livre(z), Do(Move(x, y), s)) \leftarrow Holds(Livre(z), s) \wedge y \neq z$$

O *problema do quadro*[1] está justamente relacionado à necessidade de se manter uma enorme quantidade de axiomas de quadro para garantir a persistência dos fluentes que não são afetados por uma ação [34]. Em geral, a descrição das coisas que não mudam é muito maior do que aquela das coisas que mudam e, conseqüentemente, o número de axiomas de quadro, que descrevem apenas conhecimento de senso comum, tende a ser bem maior que o número de axiomas de efeito, que realmente descrevem o domínio.

Para ver como cresce o número de axiomas de quadro, considere a adição de um novo fluente e de uma nova ação ao domínio do mundo dos blocos: $Cor(x, c)$, denotando que a cor do bloco x é c, e $Pinta(x, c)$, denotando a ação de pintar o bloco x com a cor c. Para comportar essa adição, além dos axiomas que já temos, são necessários mais dois: $Holds(Cor(x, c), Do(Pinta(x, c), s))$, estabelecendo que a cor de um bloco muda quando ele é pintado; e, supondo que inicialmente todos os blocos sejam azuis, $Holds(Cor(x, Azul), S_0))$. Sejam Δ_1 e Σ_1, respectivamente, o novo domínio e a nova situação inicial. Evidentemente, a cor de um bloco não se altera quando ele é movido; mas, sem um axioma de quadro extra, não é possível mostrar $\Sigma_1 \cup \Delta_1 \models Holds(Cor(C, Azul), Do(Move(C, B), S_0))$, i.e. que o bloco C permanece azul após ter sido movido sobre B. Precisamos estabelecer explicitamente que a cor de um bloco persiste quando um deles é movido ou quando um outro bloco é pintado. Além disso, a adição da ação $Pinta$ requer também a inclusão de novos axiomas de quadro para os fluentes $Sobre$ e $Livre$:

[1] *Frame problem.*

$Holds(Cor(x,c), Do(Move(y,z),s)) \leftarrow Holds(Cor(x,c),s)$
$Holds(Cor(x,c), Do(Pinta(y,d),s)) \leftarrow Holds(Cor(x,c),s) \wedge x \neq y$
$Holds(Sobre(x,y), Do(Pinta(z,c),s)) \leftarrow Holds(Sobre(x,y),s)$
$Holds(Livre(x), Do(Pinta(y,c),s)) \leftarrow Holds(Livre(x),s)$

Das nove fórmulas que descrevem o domínio, seis são axiomas de quadro. Em geral, como a maioria dos fluentes não são afetados por uma particular ação, sempre que um novo fluente é considerado, um novo axioma de quadro é necessário para cada uma das ações do domínio. Reciprocamente, para cada nova ação, é necessário acrescentar um novo axioma de quadro para cada um dos fluentes do domínio. Sendo assim, num domínio com m ações e n fluentes, são necessários $O(m \times n)$ axiomas de quadro.

2.1.2 Circunscrição: uma solução para o problema do quadro

O ideal seria ter apenas axiomas de efeito, supor que nenhuma mudança ocorre além daquelas resultantes desses axiomas e, então, empregar um formalismo que pudesse derivar as conseqüências esperadas. De fato, a *lei do senso comum da inércia* [34] nos permite assumir que *um fluente não muda, a menos que ocorra um evento que o afete*. Essa regra de conjectura, proposta originalmente por *McCarthy* [21], pode ser formalizada através da circunscrição.

A idéia básica da *circunscrição* [21, 22, 19] é limitar o conjunto de objetos para os quais um predicado é verdadeiro, i.e. *minimizar* sua extensão. Por exemplo, seja $\Sigma := Livre(B) \wedge Livre(C) \wedge Sobre(C,A)$. Claramente, temos $\Sigma \models Livre(B)$ e $\Sigma \models Livre(C)$; mas, devido à neutralidade da lógica clássica na ausência de informação explícita, não temos $\Sigma \models Livre(A)$, nem $\Sigma \models \neg Livre(A)$. Seja $\textsc{Circ}[\Sigma; Livre]$ a circunscrição de Σ minimizando a extensão do predicado $Livre$, i.e. tornando-o verdade apenas para aqueles objetos que Σ o força a ser. Então, temos que $\textsc{Circ}[\Sigma; Livre] \models \neg Livre(A)$. De modo geral, temos que $\textsc{Circ}[\Sigma; Livre] \models \forall x[Livre(x) \leftrightarrow (x = B \vee x = C)]$.

A solução para problema do quadro no cálculo de situações requer que os axiomas de quadro, específicos do domínio, sejam substituídos por um axioma genérico, da forma $Holds(f, Do(a,s)) \leftarrow Holds(f,s) \wedge \neg Affects(a,f,s)$, denominado *axioma de quadro universal*. Esse axioma corresponde justamente à conjectura de *McCarthy* [21], ou seja, estabelece que os fluentes persistem, a menos que sejam afetados pela execução de uma ação. Assim, uma vez especificado que ações afetam que fluentes do domínio, a circunscrição do predicado $Affects$ permite que os fluentes não afetados por uma determinada ação persistam na situação resultante de sua execução.

Formalmente, a circunscrição estende a lógica de primeira ordem com um axioma de segunda ordem que serve justamente para minimizar a extensão de um predicado. Sejam ρ_1, ρ_2 predicados n-ários e $\overline{x}$ uma tupla de n variáveis distintas, empregaremos as seguintes abreviações: $\rho_1 = \rho_2 \equiv_{def} \forall \overline{x}[\rho_1(\overline{x}) \leftrightarrow \rho_2(\overline{x})]$, $\rho_1 \leq \rho_2 \equiv_{def} \forall \overline{x}[\rho_1(\overline{x}) \rightarrow \rho_2(\overline{x})]$ e $\rho_1 < \rho_2 \equiv_{def} [\rho_1 \leq \rho_2] \wedge \neg[\rho_1 = \rho_2]$.

Definição 2.1 *Seja ϕ uma fórmula que menciona o predicado ρ. A circunscrição de ϕ minimizando ρ, escrita como* $\textsc{Circ}[\phi; \rho]$, *é a fórmula de segunda ordem $\phi \wedge \neg\exists q[\phi(q) \wedge q < \rho]$, onde $\phi(q)$ é a fórmula obtida pela substituição de toda ocorrência de ρ em ϕ por q.* □

Note que a circunscrição de ϕ minimizando ρ é a restrição ϕ mais a exigência que a extensão de ρ seja tão pequena quanto $\phi(q)$ permite que ela seja. Ademais, sempre que uma nova fórmula é conectada a ϕ, uma outra fórmula também é conectada a $\phi(q)$, que está dentro da parte quantificada existencialmente na circunscrição. Então, embora seja uma fórmula da lógica de segunda ordem clássica, a circunscrição implica em não-monotonicidade. Isso nos permite chegar a conclusões razoáveis, na ausência de informação em contrário, mas que não sejam estritamente garantidas pelos fatos estabelecidos. Uma futura adição de informação poderá invalidar tais conclusões, sem no entanto causar inconsistência.

2.2 A representação STRIPS

A representação STRIPS[2] foi proposta por *Fikes* e *Nilsson* [9] como uma alternativa ao cálculo de situações e tem sido usada amplamente como forma de representação de conhecimento nos sistemas desenvolvidos segundo a abordagem algorítmica. Sua principal vantagem é eliminar a necessidade de axiomas de quadro, permitindo um acréscimo de eficiência computacional.

Em STRIPS, um *estado* (situação) do mundo é representado por um conjunto de literais (fluentes) que estabelecem que proposições são verdadeiras nesse estado. Por exemplo, $\Sigma_{AS} := \{Sobre(A,M),\ Sobre(B,M),\ Sobre(C,A),\ Livre(B),\ Livre(C)\}$ descreve o estado inicial para o problema da anomalia de *Sussman*, ilustrada na figura 1. Como Σ_{AS} é completo, pela *hipótese do mundo fechado* [28], podemos concluir $\neg Livre(A)$. Uma *ação* é representada por um *operador*, i.e. um mapeamento especificado por um *identificador*, um conjunto de *precondições*, que determinam em que estados a ação é aplicável, e um conjunto de *efeitos*, que descrevem os resultados de sua aplicação. Por exemplo, temos a seguir a descrição de uma ação que move o bloco C de cima do bloco A para a mesa M:

OPERADOR($Move(C,A,M)$,
  PRECONDS: $\{Sobre(C,A), Livre(C)\}$,
  EFEITOS: $\{Sobre(C,M), \neg Sobre(C,A), Livre(A)\}$)

Os conjuntos PRECONDS(α) e EFEITOS(α) denotam, respectivamente, as precondições e os efeitos de uma ação α. Uma ação α é *aplicável* a um estado Σ se e só se PRECONDS(α) $\subseteq \Sigma$. Por exemplo, a ação $Move(C,A,M)$ é aplicável ao estado Σ_{AS}, pois PRECONDS($Move(C,A,M)$) $\subseteq \Sigma_{AS}$. Uma suposição implícita na representação STRIPS é que todas as mudanças causadas pela aplicação de uma ação α são explicitamente representadas pelo conjunto EFEITOS(α). Esse conjunto é particionado em dois subconjuntos disjuntos, EFEITOS$_+(\alpha)$ e EFEITOS$_-(\alpha)$, contendo átomos que representam, respectivamente, seus efeitos positivos e negativos. Se α é aplicada a um estado Σ, então o *estado resultante* de sua aplicação é representado por Σ+EFEITOS$_+(\alpha)$−EFEITOS$_-(\alpha)$. Por exemplo, o estado resultante da aplicação de $Move(C,A,M)$ a Σ_{AS} é $\{Sobre(A,M),\ Sobre(B,M), Sobre(C,M), Livre(A), Livre(B), Livre(C)\}$.

Na representação STRIPS original, precondições e efeitos podiam conter fórmulas arbitrárias da lógica de primeira ordem. Essa generalidade, entretanto, impossibilitou que uma semântica precisa fosse estabelecida para essa representação. Em decorrência disso, várias restrições foram feitas quanto às fórmulas que podem ser usadas num operador [18]. Com essas restrições, a versão mais comum passou a ser o STRIPS *proposicional*, na qual apenas literais livres de variáveis são permitidos.

2.2.1 Estendendo STRIPS para ADL

ADL[3] é uma extensão de STRIPS, proposta por *Pednault* [25], que permite representar efeitos condicionais e quantificação universal, entre outras coisas. Em ADL, ações são representadas por esquemas, i.e. operadores onde os objetos são identificados por variáveis. Por exemplo, o esquema $Move(x,y,z)$, definido a seguir, descreve uma ação genérica que move um bloco x de cima de y para cima de outro bloco z. Num domínio com n blocos, esse esquema equivale a $O(n^3)$ ações completamente instanciadas e, portanto, proporciona uma descrição bem mais compacta.

OPERADOR($Move(x,y,z)$,
  PRECONDS: $\{Sobre(x,y),\ Livre(x),\ Livre(z),\ x \neq y,\ x \neq z,\ y \neq z,\ x \neq M,\ z \neq M\}$,
  EFEITOS: $\{Sobre(x,z),\ \neg Sobre(x,y),\ Livre(y),\ \neg Livre(z)\}$)

[2] *STanford Research Institute Problem Solver.*
[3] *Action Description Language.*

Uma restrição incômoda no operador $Move(x, y, z)$ é que, devido à precondição $z \neq M$, ele não pode mover um bloco para cima da mesa. Para evitar que um outro operador, específico para essa finalidade, tenha que ser definido, podemos empregar a idéia de *efeito condicional*. Operadores com efeitos condicionais produzem diferentes resultados, dependendo do estado do mundo no momento em que são aplicados. A idéia é implementada através de uma cláusula especial $\phi \rightsquigarrow \psi$ que é adicionada entre os efeitos de uma ação. Note que $\phi \rightsquigarrow \psi$ não é o mesmo que $\phi \rightarrow \psi$, já que $\phi \rightarrow \psi$ equivale a ter $\neg\phi \vee \psi$ num *mesmo* estado do mundo (não-determinismo). Na verdade, $(\phi \rightsquigarrow \psi) \in$ Efeitos(α) equivale à fórmula $Holds(\phi, \sigma) \rightarrow Holds(\psi, Do(\alpha, \sigma))$ do cálculo de situações e denota que ψ é um efeito de α apenas quando ϕ é satisfeita na situação σ em que essa ação é executada.

Operador$(Move(x, y, z)$,
Precons: $\{Sobre(x, y),\ Livre(x),\ Livre(z)\}$,
Efeitos: $\{Sobre(x, z),\ \neg Sobre(x, y),\ Livre(y),\ z \neq M \rightsquigarrow \neg Livre(z)\})$

Em Adl, tanto precondições quanto efeitos podem ser quantificados universalmente. Por exemplo, a precondição $\forall v(Bloco(v) \rightarrow \neg Sobre(v, x))$ evitaria a necessidade de se ter o literal $Livre(x)$. Efeitos condicionais universalmente quantificados permitem, por exemplo, especificar uma operação $Move$ de tal forma que, se o objeto movido é uma pasta, então todos os objetos dentro dela também o são.

Operador$(Move(x, y, z)$,
Precons: $\{Pasta(x),\ Em(x, y),\ y \neq z\}$,
Efeitos: $\{Em(x, z),\ \neg Em(x, y),\ \forall v(Objeto(v) \rightarrow (Dentro(v, x) \rightsquigarrow (Em(v, z) \wedge \neg Em(v, y))))\})$

O uso de quantificação universal em Adl é baseado em duas suposições: (1) o mundo é modelado como um universo *finito* e *estático* de objetos e (2) cada objeto tem um *tipo declarado* na descrição do estado inicial. Isso permite que fórmulas universalmente quantificadas sejam satisfeitas por simples enumeração. Por exemplo, se o estado inicial declara $\{Pasta(P), Artigo(A), Artigo(B), Artigo(C)\}$, então a sentença $\forall v(Artigo(v) \rightarrow Dentro(v, P))$ é expandida em $Dentro(A, P) \wedge Dentro(B, P) \wedge Dentro(C, P)$.

Note que as extensões propostas em Adl não recuperam completamente a expressividade da lógica de primeira ordem, já que a sintaxe e a semântica dessas fórmulas ainda são estritamente limitadas. Na verdade, a vantagem dessas extensões está no ganho de eficiência computacional que elas proporcionam com a abordagem algorítmica de planejamento [38].

2.3 Cálculo de eventos

O *cálculo de eventos*, introduzido por *Kowalski* e *Sergot* [17], é um formalismo lógico para raciocínio sobre ações e efeitos cujas primitivas ontológicas são *eventos* ou *ações* que iniciam e terminam intervalos de *tempo* durante os quais determinados *fluentes* valem. O cálculo de eventos suporta os predicados listados a seguir e sua axiomatização, denotada por Ec, é dada na figura 2.

- $Inititiates(\alpha, \phi, \tau)$: o fluente ϕ começa a valer após a execução da ação α, no instante τ.
- $Terminates(\alpha, \phi, \tau)$: o fluente ϕ deixa de valer após a execução da ação α, no instante τ.
- $Releases(\alpha, \phi, \tau)$: o fluente ϕ tem valor indefinido após a execução da ação α, no instante τ.
- $Initially_P(\phi)$: o fluente ϕ vale no instante 0.
- $Initially_N(\phi)$: o fluente ϕ não vale no instante 0.
- $Happens(a, \tau_1, \tau_2)$: a ação α inicia-se no instante τ_1 e termina em τ_2.
- $HoldsAt(\phi, \tau)$: o fluente ϕ vale no instante τ.
- $Clipped(\tau_1, \phi, \tau_2)$: o fluente ϕ deixa de valer entre os instantes τ_1 e τ_2.
- $Declipped(\tau_1, \phi, \tau_2)$: o fluente ϕ começa a valer entre os instantes τ_1 e τ_2.

(EC1) $HoldsAt(f,t) \leftarrow Initially_P(f) \ \wedge \ \neg Clipped(0,f,t)$
(EC2) $HoldsAt(f,t) \leftarrow Happens(a,t_1,t_2) \ \wedge \ Initiates(a,f,t_1) \ \wedge \ t_2 < t \ \wedge \ \neg Clipped(t_1,f,t)$
(EC3) $\neg HoldsAt(f,t) \leftarrow Initially_N(f) \ \wedge \ \neg Declipped(0,f,t)$
(EC4) $\neg HoldsAt(f,t) \leftarrow Happens(a,t_1,t_2) \wedge Terminates(a,f,t_1) \wedge t_2 < t \wedge \neg Declipped(t_1,f,t)$
(EC5) $Clipped(t_1,f,t_2) \leftrightarrow \exists a,t_3,t_4[Happens(a,t_3,t_4) \ \wedge \ t_1 < t_3 \ \wedge \ t_4 < t_2 \wedge (Terminates(a,f,t_3) \vee Releases(a,f,t_3))]$
(EC6) $Declipped(t_1,f,t_2) \leftrightarrow \exists a,t_3,t_4[Happens(a,t_3,t_4) \ \wedge \ t_1 < t_3 \ \wedge \ t_4 < t_2 \wedge (Initiates(a,f,t_3) \vee Releases(a,f,t_3))]$
(EC7) $Happens(a,t_1,t_2) \rightarrow t_1 \leq t_2$

Figura 2: *Axiomatização do cálculo de eventos*

A modelagem de operadores STRIPS é direta no cálculo de eventos. Por exemplo, o operador

OPERADOR($Empurra(x,y)$,
PRECONDS: $\{Em(x),\ x \neq y\}$,
EFEITOS: $\{Em(y),\ \neg Em(x)\}$)

é representado no cálculo de eventos por:

$Initiates(Empurra(x,y),Em(y),t) \leftarrow Holds(Em(x),t) \wedge x \neq y$
$Terminates(Empurra(x,y),Em(x),t) \leftarrow Holds(Em(x),t) \wedge x \neq y$

É interessante notar que, ao contrário do cálculo de situações, o cálculo de eventos não requer o uso de axiomas de quadro. A persistência embutida na axiomatização do cálculo de eventos (circunscritivo) é baseada em quatro suposições: (1) nenhum evento ocorre além daqueles que são conhecidos, (2) nenhum evento afeta um dado fluente além daqueles que são conhecidos, (3) os fluentes persistem até a ocorrência de algum evento que os afete e (4) todo fluente é efeito de algum evento conhecido.

3 Planejamento clássico: uma abordagem algorítmica

Essa seção apresenta uma descrição sucinta da área de planejamento clássico e de alguns algoritmos, com características de planejadores práticos, que serão implementados com abdução.

Dada uma descrição das ações que um agente pode executar, do estado inicial do mundo e dos objetivos desse agente, o planejamento consiste em determinar uma seqüência de ações que, quando executadas num mundo qualquer satisfazendo a descrição do estado inicial, atingem um estado meta onde os objetivos especificados são satisfeitos [9]. Tais descrições devem ser feitas empregando-se um formalismo de representação de conhecimento. No planejamento clássico, esse formalismo é, geralmente, baseado no estilo de representação STRIPS. Além disso, o planejamento clássico admite as seguintes suposições: *tempo atômico*, i.e. a execução de uma ação é indivisível e ininterrupta e a execução simultânea de duas ou mais ações é impossível; *efeitos determinísticos*, i.e. o efeito de uma ação é uma função do estado corrente do mundo no momento em que ela é executada; *onisciência*, i.e. o agente tem conhecimento completo do estado inicial e dos efeitos de suas próprias ações; e *causa de mudança única*, i.e. o mundo muda apenas quando o agente age e existe apenas um agente no mundo [38].

3.1 Busca no espaço de estados

Um *espaço de estados* consiste de um conjunto finito de estados S, um conjunto finito de ações A e uma função de transição f que descreve como as ações mapeiam um estado em outro. Um espaço de

estados juntamente com um estado inicial s_0 e um conjunto G de estados meta é denominado *modelo de estados*.

Definição 3.1 *Um modelo de estados é uma tupla $\langle S, s_0, G, A, f\rangle$, onde S é um conjunto finito não-vazio de estados s, $s_0 \in S$ é o estado inicial, $G \subseteq S$ é um conjunto não-vazio de estados metas, $A(s) \subseteq A$ denota as ações aplicáveis em cada estado $s \in S$, e $f(a, s)$ denota a função de transição de estados para todo $s \in S$ e $a \in A(s)$.*

O *espaço de estados* de um mundo pode ser modelado por um grafo cujos nós representam os estados desse mundo e cujas arestas representam as ações que transformam um estado em outro [14]. Um *problema* de planejamento é definido por um modelo de estados. Uma *solução*, ou *plano*, para um tal problema consiste numa seqüência de ações que definem nesse grafo um caminho que leva do estado inicial a um estado meta.

Uma vantagem em modelar planejamento como busca é que podemos aplicar diversos algoritmos de *força bruta* ou *heurísticos* bem conhecidos [15, 16, 30]. A busca no grafo que representa o espaço de estados para um problema de planejamento pode ser feita de duas maneiras distintas: *progressiva*, i.e. a partir do nó que representa o estado inicial, tentamos encontrar um nó representando um estado meta; ou *regressiva*, i.e. a partir de um nó que representa um estado meta, tentamos encontrar o nó que representa o estado inicial.

3.1.1 Planejamento progressivo

O algoritmo não-determinístico PROG, apresentado na figura 3, realiza uma busca progressiva no espaço de estados. Ele é chamado inicialmente com o estado corrente Σ igual ao estado inicial do problema e com Π sendo uma seqüência de ações nula. No início de cada iteração, o algoritmo (1) verifica se o estado corrente é um estado meta; caso seja, o plano Π é devolvido como solução do problema. Senão, (2) o algoritmo determina todas as ações aplicáveis ao estado corrente (seção 2.2), (3) *escolhe* uma delas que seja apropriada, anexa-a ao final do plano e continua a busca no estado resultante de sua aplicação (seção 2.2). Caso uma tal escolha não seja possível, (4) o algoritmo devolve FALHA.

Algoritmo PROG$(\Delta, \Sigma, \Gamma, \Pi)$
Entrada: A descrição das ações Δ.
A descrição do estado corrente Σ.
A descrição de um estado meta Γ.
Um plano parcialmente especificado Π.
Saída: FALHA ou um plano completo Π.
1. Se $\Sigma \supseteq \Gamma$ então devolva Π.
2. Seja $\mathcal{A} := \{\alpha \mid \alpha \in \Delta \wedge$ PRECONDS$(\alpha) \subseteq \Sigma\}$.
3. Escolha $\alpha \in \mathcal{A}$.
3.1. Seja $\Pi' :=$ PROG$(\Delta, \Sigma +$EFEITOS$_+(\alpha) -$EFEITOS$_-(\alpha), \Gamma, \Pi \circ \alpha)$.
3.2. Se $\Pi' \neq$FALHA então devolva Π'.
3.3. Retroceda.
4. Devolva FALHA.

Figura 3: *Busca progressiva no espaço de estados*

3.1.2 Planejamento regressivo

O algoritmo não-determinístico REGR, apresentado na figura 4, realiza uma busca regressiva no espaço de estados. O passo mais importante nesse algoritmo é aquele que *regride* a um estado imediatamente

anterior àquele corrente, antes que uma determinada ação seja executada, a fim de garantir que o estado corrente seja consistente [37, 24]. REGR é chamado inicialmente com o estado corrente Σ igual a um estado meta e com Π sendo uma seqüência de ações nula. No início de cada iteração, o algoritmo (1) verifica se o estado corrente é o estado inicial; caso seja, o plano Π é devolvido como solução do problema. Senão, (2) o algoritmo determina todas as ações cujos efeitos suportem pelo menos um literal do estado corrente e cujos efeitos negativos não entrem em conflito com esse estado, (3) *escolhe* uma delas que seja apropriada, anexa-a ao início do plano e continua a busca no estado prévio, $\Gamma + \text{PRECONDS}(\alpha) + \text{EFEITOS}_-(\alpha) - \text{EFEITOS}_+(\alpha)$, resultante da regressão do estado corrente através da ação escolhida. Caso uma tal escolha não seja possível, (4) o algoritmo devolve FALHA.

Algoritmo REGR$(\Delta, \Sigma, \Gamma, \Pi)$
Entrada: A descrição das ações Δ.
A descrição do estado inicial Σ.
A descrição do estado corrente Γ.
Um plano parcialmente especificado Π.
Saída: FALHA ou um plano completo Π.
1. Se $\Sigma \subseteq \Gamma$ então devolva Π.
2. Seja $\mathcal{A} := \{\alpha \mid \alpha \in \Delta \wedge \text{EFEITOS}(\alpha) \cap \Gamma \neq \emptyset \wedge \text{EFEITOS}_-(\alpha) \cap \Gamma = \emptyset\}$.
3. Escolha $\alpha \in \mathcal{A}$.
3.1. Seja $\Pi' := \text{REGR}(\Delta, \Sigma, \Gamma + \text{PRECONDS}(\alpha) + \text{EFEITOS}_-(\alpha) - \text{EFEITOS}_+(\alpha), \alpha \circ \Pi)$.
3.2. Se $\Pi' \neq$FALHA então devolva Π'.
3.3. Retroceda.
4. Devolva FALHA.

Figura 4: *Busca regressiva no espaço de estados*

Note que a condição em (2) garante que a regressão através de qualquer $\alpha \in \mathcal{A}$ não torna o estado corrente inconsistente. Por exemplo, considere o estado corrente $\Gamma = \{Livre(B), Livre(C)\}$ e a ação $Move(A, B, C)$. A condição $\text{EFEITOS}(Move(A, B, C)) \cap \Gamma = \{Livre(B)\} \neq \emptyset$ é satisfeita, mas $\text{EFEITOS}_-(Move(A, B, C)) \cap \Gamma = \{Livre(C)\} = \emptyset$, não. Portanto, tal ação não deve fazer parte de $\mathcal{A}$. Isso é desejado; pois, qualquer que seja o estado prévio, a aplicação da ação $Move(A, B, C)$ resultará num estado Γ, onde o bloco C está livre. Mas isso é uma inconsistência, já que essa ação move o bloco A para cima do bloco C.

3.1.3 Comparação entre busca progressiva e regressiva

Na prática, o não-determinismo da primitiva *escolha* é implementado como busca. Sendo assim, o fator de ramificação[4] da árvore de busca tem um papel fundamental na eficiência dos algoritmos PROG e REGR. Se o fator de ramificação é b, a complexidade desses algoritmos é $O(b^n)$, onde n é o número de passos no plano [31]. Se admitirmos a suposição de que apenas uma pequena parte dos literais usados na descrição do problema de planejamento são empregados na descrição de um particular estado, então planejamento regressivo apresenta um fator de ramificação bem inferior àquele do planejamento progressivo e, portanto, é muito mais eficiente. Note que muitas ações podem ser executadas no estado inicial, mas apenas algumas delas são relevantes para atingir um dado estado meta. Como o planejamento progressivo considera todas as ações cujas precondições sejam satisfeitas no estado inicial, ele não se beneficia da orientação proporcionada pela descrição dos objetivos do planejamento.

[4] O número médio de ações aplicáveis a cada estado.

3.2 Busca no espaço de planos

Em vez de busca no espaço de estados, podemos fazer busca no espaço de planos [32]. O *espaço de planos* pode ser modelado por um grafo cujos nós representam planos parcialmente especificados e cujas arestas denotam operações de refinamento do plano, tal como adição de uma nova ação. Começamos com um plano simples e então consideramos meios de transformá-lo num plano completo que solucione o problema de planejamento. Nessa abordagem, a solução não é representada por um caminho no grafo, mas sim pelo nó que representa um plano completo.

A maior motivação para essa mudança de paradigma é que podemos evitar o retrocesso desnecessário devido ao tratamento de metas numa ordem diferente daquela de execução; já que a busca no espaço de estados compromete a ordem de planejamento com a ordem de execução do plano. Note que, de qualquer forma, todas as metas deverão ser atingidas, não importando a ordem em que sejam tratadas. O planejamento como busca no espaço de planos permite que o agente raciocine explicitamente sobre a construção de planos e a interação de ações.

3.2.1 Planejamento de ordem total

Quando plano é definido como uma seqüência totalmente ordenada de ações, o planejamento de ordem total como busca no espaço de planos é isomorfo ao planejamento regressivo como busca no espaço de estados. Como a cada chamada recursiva REGR recebe como argumento uma seqüência de ações totalmente ordenada (Π), é como se ele estivesse fazendo busca no espaço de planos e a operação de refinamento fosse simplesmente adicionar uma nova ação no início dessa seqüência.

A vantagem em considerar busca no espaço de planos é que isso possibilita pensar em outras operações de refinamento, que podem tornar o processo mais eficiente. Por exemplo, podemos inserir ações em posições arbitrárias do plano.

3.2.2 Planejamento de ordem parcial

No planejamento de ordem parcial, planos são representados por triplas $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$, onde $\mathcal{A}$ é um conjunto de *ações*, $\mathcal{O}$ é um conjunto de restrições de *ordenação temporal* e $\mathcal{L}$ é um conjunto de *vínculos causais*[5]. As restrições de ordenação temporal impõem uma ordem parcial sobre as ações do plano e os vínculos causais estabelecem o propósito de cada uma delas.

Um *vínculo causal* é uma estrutura da forma $\alpha_p \xrightarrow{\phi} \alpha_c$, onde α_p é uma *ação produtora* cujo efeito ϕ é uma precondição da *ação consumidora* α_c [36, 20]. Dizemos que o vínculo $\alpha_p \xrightarrow{\phi} \alpha_c$ *suporta* a precondição ϕ. Vínculos causais são usados para determinar quando uma ação interfere com (ameaça) decisões passadas.

Definição 3.2 *Sejam $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ um plano, $\lambda = \alpha_p \xrightarrow{\phi} \alpha_c \in \mathcal{L}$ um vínculo causal e $\alpha_t \in \mathcal{A}$ uma ação. Dizemos que α_t ameaça λ se $\mathcal{O} \cup \{\alpha_p \prec \alpha_t \prec \alpha_c\}$ é consistente e α_t tem $\neg\phi$ como efeito.* □

Para evitar ameaças, o algoritmo deve proteger os vínculos causais adicionando restrições de ordenação: se uma ação α_t ameaça um vínculo $\alpha_p \xrightarrow{\phi} \alpha_c$, então ela deve ser *antecipada*[6], i.e. executada antes de α_p, ou *postergada*[7], i.e. executada após α_c.

[5] *Causal links.*
[6] *Demotion.*
[7] *Promotion.*

Para permitir que tanto planos parcialmente especificados quanto planos completos sejam representados da mesma maneira, utilizamos a idéia de *plano vazio*. O plano vazio tem duas ações, $\mathcal{A} = \{A_0, A_\infty\}$, uma restrição de ordenação temporal, $\mathcal{O} = \{A_0 \prec A_\infty\}$ e nenhum vínculo causal, $\mathcal{L} = \{\}$. A ação A_0 não tem precondições e seus efeitos especificam que proposições valem no estado inicial. A ação A_∞ não tem efeitos e suas precondições determinam que proposições devem valer no estado meta. A ação A_0 é a primeira, A_∞ é a última, e todas as demais ações do plano são precedidas por A_0 e seguidas por A_∞.

POP[8] é um algoritmo de planejamento regressivo de ordem parcial que realiza busca no espaço de planos [3, 20, 38]. O planejamento consiste em escolher ações e adicioná-las ao plano, até que toda precondição tenha sido suportada por um vínculo causal e toda ameaça tenha sido evitada por meio da adição de restrições de ordenação temporal.

No algoritmo POP, apresentado na figura 5, Γ é um conjunto de pares da forma $\langle \phi, \alpha_c \rangle$, onde ϕ é uma precondição para a ação α_c. Por definição, se $\langle \phi, \alpha_c \rangle \in \Gamma$, então $\alpha_c \in \mathcal{A}$ e $\phi \in$PRECONDS(α_c). Inicialmente, o conjunto Γ contém todos, e só, os pares gerados a partir das precondições da ação A_∞ e o plano $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ é vazio. Em cada iteração, (1) o algoritmo verifica se o conjunto Γ está vazio e, caso esteja, devolve o plano $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$ como solução. Em seguida, (2) o algoritmo seleciona uma das metas em Γ. Note que essa seleção não deve causar retrocesso, já que a ordem em que as metas são tratadas pelo algoritmo é irrelevante para a ordem de execução. Se houver um plano que resolva o problema, esse plano será encontrado independentemente da ordem em que as metas são selecionadas para serem resolvidas. Uma vez selecionada um meta, (3) uma ação que suporte essa meta é escolhida não-deterministicamente. Essa ação pode ser uma já existente no plano, i.e. em $\mathcal{A}$, ou uma nova instância de uma das ações descritas em Δ. Então, (3.1 e 3.2) o plano é alterado para comportar essa ação escolhida, (3.3 e 3.4) o conjunto Γ é atualizado, e (3.5) as ameaças são resolvidas. Caso nenhuma ação apropriada possa ser escolhida, (4) o algoritmo devolve FALHA.

Algoritmo POP$(\Delta, \Gamma, \langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle)$
Entrada: A descrição das ações Δ.
Um conjunto de precondições (metas) ainda não satisfeitas Γ.
Um plano parcialmente especificado e parcialmente ordenado $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$.
Saída: FALHA ou um plano completo parcialmente ordenado $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$.
1. [*termina*] Se $\Gamma = \emptyset$ então devolva $\langle \mathcal{A}, \mathcal{O}, \mathcal{L} \rangle$.
2. [*seleciona meta*] Selecione $\langle \phi, \alpha_c \rangle \in \Gamma$.
3. [*escolhe ação*] Escolha $\alpha_p \in \{\alpha \mid (\alpha \in \mathcal{A} \vee \alpha \in \Delta) \wedge \phi \in$EFEITOS$(\alpha) \wedge \mathcal{O} \cup \{\alpha_p \prec \alpha_c\}$ é consistente$\}$.
3.1. [*atualiza plano*] Seja $\mathcal{L}' := \mathcal{L} \cup \{\alpha_p \xrightarrow{\phi} \alpha_c\}$ e $\mathcal{O}' := \mathcal{O} \cup \{\alpha_p \prec \alpha_c\}$.
3.2. Se $\alpha_p \notin \mathcal{A}$, então $\mathcal{A}' := \mathcal{A} \cup \{\alpha_p\}$ e $\mathcal{O}' := \mathcal{O}' \cup \{A_0 \prec \alpha_p \prec A_\infty\}$; senão, $\mathcal{A}' := \mathcal{A}$.
3.3. [*atualiza metas*] Seja $\Gamma' := \Gamma - \{\langle \phi, \alpha_c \rangle\}$.
3.4. Se $\alpha_p \notin \mathcal{A}$, então para cada $\phi_i \in$PRECONDS(α_p) adicione $\langle \phi_i, \alpha_p \rangle$ a Γ'.
3.5. [*protege vínculos*] Para cada $\alpha_t \in \mathcal{A}$ que ameaça um vínculo $\alpha_i \xrightarrow{\phi} \alpha_j \in \mathcal{L}$, escolha:
3.5.1. [*antecipa*] adicione $\alpha_t \prec \alpha_i$ a $\mathcal{O}'$, ou
3.5.2. [*posterga*] adicione $\alpha_j \prec \alpha_t$ a $\mathcal{O}'$.
3.5.3. Se nenhuma restrição é consistente, devolva FALHA.
3.6. [*chama recursivamente*] Devolva POP$(\Delta, \Gamma', \langle \mathcal{A}', \mathcal{O}', \mathcal{L}' \rangle)$.
3.7. Retroceda.
4. Devolva FALHA.

Figura 5: *Planejamento de ordem parcial*

[8] *Partial Order Planning.*

3.2.3 Comparação entre ordenação total e parcial

Em geral o tempo gasto por uma algoritmo de busca é $O(cb^n)$, onde n representa o número de escolhas não-determinísticas que são feitas antes que uma solução seja encontrada, b representa o fator de ramificação, i.e. quantas alternativas são consideradas, em média, em cada ponto de escolha, e c representa o tempo necessário para processar um dado nó[9] [31]. Desses parâmetros, o mais significativo é o fator de ramificação b e, para POP, ele é bem menor que para REGR. Isso acontece porque o POP não retrocede na escolha de metas e, ao expandir um nó, considera apenas as ações que satisfazem precondições da meta escolhida; diferentemente do REGR, que, além de retroceder na escolha de metas, considera todas as ações que satisfaçam qualquer precondição de qualquer meta ainda não atingida.

3.3 Representação de conhecimento e paradigmas de busca

No cálculo de situações, a aplicação de uma ação α a uma determinada situação σ resulta numa nova situação $\sigma' := Do(\alpha, \sigma)$. É fácil perceber a relação existente entre a função Do e a função de transição no espaço de estados do problema de planejamento. Na verdade, um plano $\langle \alpha_1, \ldots, \alpha_n \rangle$ pode ser representado pela situação $Do(\alpha_n, Do(\ldots, Do(\alpha_1, S_0)))$ que ele atinge, quando executado a partir da situação inicial S_0. A linearidade imposta pela função Do, e o fato de que cada nova extensão do plano corresponde a um novo estado completamente definido pelas ações já consideradas, faz com que os planejadores que empregam o cálculo de situações como formalismo para representação de conhecimento acabem se conformando ao paradigma de planejamento de ordem total como busca no espaço de estados. Por outro lado, como será visto, usando cálculo de eventos abdutivo temos, naturalmente, planejamento de ordem parcial como busca no espaço de planos.

3.4 Algoritmos de planejamento

A seguir, temos a descrição de alguns algoritmos de planejamento de ordem parcial que serão implementados com abdução.

3.4.1 O algoritmo UCPOP

UCPOP[10] desenvolvido por *Weld* [27], é um planejador de ordem parcial que suporta o estilo de representação ADL [25]. Para isso, UCPOP estende a representação de plano com um conjunto de co e não-codesignações $\mathcal{B}$, necessário para restringir os valores das variáveis, usa expansão para manipular fórmulas universalmente quantificadas e adota uma nova definição de ameaça, capaz de tratar corretamente os efeitos condicionais.

Definição 3.3 *Sejam $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$ um plano, $\lambda = \alpha_p \xrightarrow{\phi} \alpha_c \in \mathcal{L}$ um vínculo causal e $\alpha_t \in \mathcal{A}$ uma ação. Dizemos que α_t ameaça λ se $\mathcal{O} \cup \{\alpha_p \prec \alpha_t \prec \alpha_c\}$ é consistente e α_t tem ψ como efeito, existe um unificador $\theta = \mu(\phi, \neg\psi, \mathcal{B})$ e, para todo $u/v \in \theta$, u ou v é uma variável universal.* □

O problema com efeitos condicionais é que eles tanto podem representar uma ameaça a um dos vínculos no plano, quanto podem servir para suportar alguma precondição selecionada. Se uma ação α tem um efeito condicional da forma $\phi \rightsquigarrow \psi$ e desejamos que o efeito ψ suporte uma precondição, então

[9] O parâmetro c é na verdade uma função do nó. Mas, como o fator b^n domina o tempo, isso é irrelevante.

[10] *Universal quantification and Conditional effects Partial Order Planner.*

basta garantir ϕ no estado em que α é aplicada; caso contrário, se ψ representa uma ameaça, então $\neg\phi$ é que deve ser garantido. Essa forma de tratar ameaças é denominada *confrontação*[11].

Assim como no caso de POP, para UCPOP, Γ é inicialmente um conjunto de pares $\langle \phi_i, A_\infty \rangle$, gerados a partir das condições ϕ_i estabelecidas no estado meta.

Algoritmo UCPOP($\Delta, \Gamma, \langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$)
Entrada: A descrição dos operadores Δ.
Um conjunto de precondições (metas) ainda não satisfeitas Γ.
Um plano parcialmente especificado e parcialmente ordenado $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$.
Saída: FALHA ou um plano completo parcialmente ordenado $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$.
1. [*termina*] Se $\Gamma = \emptyset$, devolva $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{B} \rangle$.
2. [*reduz meta*] Remova uma meta $\langle \phi, \alpha_c \rangle$ de Γ.
2.1. Se ϕ é uma sentença quantificada, reinsira sua base universal $\langle \Upsilon(\phi), \alpha_c \rangle$ em Γ e volte ao passo 2.
2.2. Se ϕ é uma conjunção de ϕ_i's, reinsira cada $\langle \phi_i, \alpha_c \rangle$ em Γ e volte ao passo 2.
2.3. Se ϕ é uma disjunção de ϕ_i's, escolha um disjunto ϕ_i, reinsira $\langle \phi_i, \alpha_c \rangle$ em Γ e volte ao passo 2.
2.4. Se ϕ é um literal e existe um vínculo causal $\alpha_p \xrightarrow{\neg\phi} \alpha_c \in \mathcal{L}$, devolva FALHA.
3. [*escolhe ação*] Escolha $\alpha_p \in \{\alpha \mid (\alpha \in \mathcal{A} \vee \alpha \in \Delta) \wedge \psi \in \text{EFEITOS}(\alpha) \wedge \mu(\phi, \psi, \mathcal{B}) \neq \text{FALHA} \wedge \mathcal{O} \cup \{A_p \prec A_c\}$ é consistente$\}$ e considere $\theta = \mu(\phi, \psi, \mathcal{B})$, sendo $\psi \in \text{EFEITOS}(\alpha_p)$.
3.1. [*atualiza plano e metas*] Sejam $\Gamma' := \Gamma$, $\mathcal{A}' := \mathcal{A}$, $\mathcal{O}' := \mathcal{O} \cup \{\alpha_p \prec \alpha_c\}$, $\mathcal{L}' := \mathcal{L} \cup \{\ \alpha_p \xrightarrow{\phi} \alpha_c\ \}$, e $\mathcal{B}' := \mathcal{B} \cup \{(u = v) \mid (u/v) \in \theta \wedge$ *nem u nem v é universalmente quantificada*$\}$.
3.2. Se $\alpha_p \notin \mathcal{A}$, adicione $\langle \text{PRECONDS}(\alpha_p)\theta, \alpha_p \rangle$ a Γ', α_p a $\mathcal{A}'$, $\{A_0 \prec \alpha_p \prec A_\infty\}$ a $\mathcal{O}$, e restrições de co e não-codesignações de α_p a $\mathcal{B}'$. Se o efeito é condicional, e ainda não foi usado para estabelecer um vínculo em $\mathcal{L}$, adicione a Γ uma variante de seu antecedente com relação a θ.
3.3. [*protege vínculos*] Para cada vínculo $\lambda = \alpha_i \xrightarrow{\psi} \alpha_j \in \mathcal{L}$ e cada ação $\alpha_t \in \mathcal{A}$ que ameaça λ, escolha:
3.3.1. [*posterga*] Se consistente, então faça $\mathcal{O}' := \mathcal{O}' \cup \{\alpha_j < \alpha_t\}$.
3.3.2. [*antecipa*] Se consistente, então faça $\mathcal{O}' := \mathcal{O}' \cup \{\alpha_t < \alpha_i\}$.
3.3.3. [*confronta*] Se o efeito de α_t que ameaça λ é da forma $\sigma \rightsquigarrow \rho$, adicione $\langle \neg\sigma\mu(\phi, \neg\rho, \mathcal{B}), \alpha_t \rangle$ a Γ'.
3.4. [*chama recursivamente*] Se $\mathcal{B}$ é consistente, devolva UCPOP($\Delta, \Gamma', \langle \mathcal{A}', \mathcal{O}', \mathcal{L}', \mathcal{B}' \rangle$).
3.5. Retroceda.
4. Devolva FALHA.

Figura 6: *Planejamento com efeitos condicionais e quantificação*

3.4.2 O algoritmo SNLP

SNLP[12], desenvolvido por *MacAllester* [20], é bastante semelhante a UCPOP. A principal diferença entre eles está no conceito de *sistematicidade* de busca, i.e. que um mesmo plano parcialmente especificado não pode ser visitado mais de uma vez no espaço de planos. Para garantir essa sistematicidade, SNLP adota a idéia de *ameaças positivas*: dado um vínculo causal $\alpha_p \xrightarrow{\phi} \alpha_c$, SNLP o protege não só de ações que negam ϕ, mas também daquelas que produzem ϕ como efeito. Permitindo um único suporte para cada precondição, SNLP garante que os conjuntos de soluções em cada ramo da árvore de busca sejam disjuntos e, portanto, a sistematicidade da busca.

3.4.3 O algoritmo HTN

HTN[13] [7], ou *planejamento hierárquico*, é um paradigma que contribui para a redução do espaço de busca, baseando-se na idéia de decomposição de tarefas em outras tarefas mais simples. Na terminologia

[11] *Confrontation.*
[12] *Systematic Non-Linear Planner.*
[13] *Hierarchical task network.*

HTN, uma *tarefa meta* especifica uma propriedade que deve ser verdadeira numa determinada situação do mundo; uma *tarefa primitiva* é uma tarefa que pode ser diretamente realizada, executando-se uma ação correspondente; e uma *tarefa composta* denota mudanças desejadas que envolvem várias tarefas metas e primitivas. Uma *rede de tarefas* é uma coleção de tarefas que devem ser realizadas, juntamente com restrições na ordem em que são executadas, na forma como as variáveis são instanciadas e que literais devem ser verdadeiros antes ou depois que cada tarefa é executada. Tarefas não-primitivas, i.e. tarefas metas ou compostas, não podem ser executadas diretamente, já que podem envolver a realização de várias outras tarefas. Tarefas primitivas são representadas por operadores no estilo STRIPS e tarefas compostas são representadas por *métodos*, i.e. pares (τ, ρ) que estabelecem que uma maneira de realizar a tarefa composta τ é realizar as tarefas na rede ρ, sem violar suas restrições.

Ao contrário do planejamento baseado em estados, o planejamento hierárquico enfatiza as tarefas que são planejadas e as interações existentes entre elas. Dada uma rede de tarefas inicial ρ, representando o problema a ser resolvido, um conjunto de operadores $\mathcal{O}$, especificando as precondições e os efeitos de cada ação primitiva, e um conjunto de métodos $\mathcal{M}$, estabelecendo como realizar tarefas compostas, um planejador HTN procede decompondo tarefas em tarefas mais simples e resolvendo os conflitos que surgem entre elas. Iniciando com a rede de tarefas ρ, ele executa, repetidamente, os seguintes passos: encontra uma tarefa composta $t \in \rho$ e um método $(t', \rho') \in \mathcal{M}$, tais que t unifica-se com t', e então modifica ρ, expandindo t, i.e. substituindo t pelas tarefas em ρ' e incorporando as restrições de ρ' em ρ. Uma vez que todas as tarefas compostas em ρ tenham sido expandidas, um plano para o problema proposto é uma instância ordenada de ρ que satisfaça todas as restrições impostas.

Metas de obtenção *versus* **metas de realização:** A principal diferença entre planejamento hierárquico e planejamento baseado em estados está no objetivo do planejamento. Planejadores baseados em estados visam *metas de obtenção*, i.e. condições que devem ser satisfeitas num determinado estado do mundo. Qualquer plano que torne essas condições verdadeiras no estado final é considerado uma solução válida, não importando que ações ou estados intermediários ele contenha. Planejadores HTN, entretanto, visam *metas de realização*, i.e. elaborações mais complexas denotadas por redes de tarefas. Por exemplo, *"construir uma casa"* é uma meta de realização, enquanto *"ter uma casa"* é uma meta de obtenção; note que *"comprar uma casa"* satisfaz essa última, mas não a primeira. Note também que metas de realização proporcionam um incremento na expressividade da representação. Por exemplo, *"ir a Nova York e voltar"* não pode ser expressa diretamente como uma meta de obtenção, já que os estados inicial e final são o mesmo [7]. Além disso, metas de realização reduzem o espaço de busca na medida em que a especificação de métodos restringe a atenção do planejador a expansões desejáveis de uma particular tarefa, em vez de permitir a busca entre todas as seqüências de ações possíveis.

Estendendo a representação Para incorporar a capacidade de decomposição hierárquica, é necessário introduzir duas alterações na descrição de um domínio. Primeiro, o conjunto de operadores deve ser particionado em primitivos e compostos, de modo que o planejador tenha como saber quando uma tarefa é primitiva ou não. Segundo, para cada operador composto, é preciso especificar um conjunto de métodos de decomposição. O exemplo a seguir é baseado em *Russel* e *Norvig* [31].

MÉTODO($Constrói(Casa)$,
  PLANO: $\langle \mathcal{A} = \{\alpha_1 : Constrói(Fundação), \alpha_2 : Constrói(Estrutura), \alpha_3 : Constrói(Telhado),$
  $\alpha_4 : Constrói(Paredes), \alpha_5 : Constrói(Acabamento)\}$
  $\mathcal{O} = \{\alpha_1 \prec \alpha_2 \prec \alpha_3 \prec \alpha_5, \alpha_2 \prec \alpha_4 \prec \alpha_5\}$
  $\mathcal{L} = \{ \alpha_1 \overset{Fundação}{\longrightarrow} \alpha_2, \alpha_2 \overset{Estrutura}{\longrightarrow} \alpha_3, \alpha_2 \overset{Estrutura}{\longrightarrow} \alpha_4, \alpha_3 \overset{Telhado}{\longrightarrow} \alpha_5, \alpha_4 \overset{Paredes}{\longrightarrow} \alpha_5\}$
  $\mathcal{B} = \{\}\rangle$)

Dizemos que um método *implementa corretamente* um operador se ele contém um plano completo e consistente para o problema de atingir os efeitos do operador, dadas as suas precondições. Essa implementação correta permite que um plano continue consistente quando uma tarefa composta é substituída pela rede de tarefas correspondente, indicada no método de decomposição empregado [31].

4 Planejamento abdutivo: uma abordagem lógica

Eshghi [8] foi o primeiro a mostrar que planejamento pode ser visto como uma tarefa abdutiva. Essa seção mostra que planejamento de ordem parcial pode ser, naturalmente, tratado através de abdução e cálculo de eventos circunscritivo.

4.1 Abdução

O filósofo *Pierce* [26] introduziu a noção de *abdução* como a forma de raciocínio em que uma hipótese é adotada como possível explicação para um fato observado, de acordo com leis conhecidas. Como ele mesmo observa, entretanto, a abdução é um tipo de inferência fraca, já que apenas garante que uma explicação seja *plausível*. Por exemplo, considere $\Delta := \{Verão \rightarrow Chuva, Umidade \wedge Calor \rightarrow Chuva, Chuva \rightarrow Enchente\}$. Se observamos $\Gamma_0 := \{Enchente\}$, e queremos saber o porquê, $\Sigma := \{Verão\}$ é uma explicação plausível, i.e. o conjunto de hipóteses Σ, juntamente com o conhecimento expresso em Δ, nos permite concluir Γ_0. Evidentemente, podemos ter múltiplas explicações; nesse exemplo, $\Sigma := \{Umidade, Calor\}$ seria uma outra explicação plausível para Γ_0.

Definição 4.1 *Sejam Δ um conjunto de sentenças que descrevem um domínio e Γ_0 uma sentença que descreve uma observação. A abdução consiste em encontrar um conjunto de sentenças Σ, denominado explicação abdutiva de Γ_0, tal que $\Delta \cup \Sigma$ é consistente e $\Delta \cup \Sigma \models \Gamma_0$.* □

Note que, por definição, a abdução é uma forma de raciocínio não-monotônico, pois explicações consistentes com um determinado estado de conhecimento podem se tornar inconsistentes face a novas informações [11]. Por exemplo, se a fórmula $\neg Verão$ for adicionada a Δ, $Verão$ deixa de ser uma explicação abdutiva plausível para $Enchente$.

Segundo *Cox* e *Pietrzykowski* [6], uma explicação abdutiva Σ deve ser *básica*, i.e. não pode ser dada em termos de efeitos, mas apenas de causas; *minimal*, i.e. não deve existir $\Sigma^* \subset \Sigma$ tal que $\Delta \cup \Sigma^* \models \Gamma$; e *compacta*, i.e. deve postular o menor número possível de causas. Por exemplo, $\Sigma_1 := \{Chuva\}$ não é uma explicação básica para $\Gamma_0 = \{Enchente\}$; $\Sigma_2 := \{Umidade, Calor, Verão\}$ é básica, mas não é minimal; $\Sigma_3 := \{Umidade, Calor\}$ é básica e minimal, mas não é compacta; e $\Sigma_4 := \{Verão\}$ é básica, minimal e compacta. Restringindo os fatos que podem ser abduzidos, garantimos que somente explicações básicas sejam computadas [11]; e utilizando busca em profundidade iterativa com preferência para ramos que não postulem novos eventos, tornamos o método completo e aumentamos as chances de encontrar explicações minimais e compactas [33].

4.1.1 O mecanismo abdutivo em programação lógica

Abdução é um princípio de inferência que estende a dedução, proporcionando raciocínio hipotético. Dado um programa lógico Δ e uma meta Γ_0, o objetivo da abdução é encontrar um conjunto de fatos Σ tal que $\Delta \cup \Sigma \models \Gamma_0$. A maneira mais simples de estender o procedimento de resolução [29] para gerar esse conjunto é permitir que os fatos em Σ sejam abduzidos, i.e. sejam tratados como hipóteses. Isso,

evidentemente, não é desejado para qualquer predicado[14]; então, um conjunto predefinido de predicados abdutíveis é estabelecido e os fatos abduzidos são restringidos a pertencer a esse conjunto.

Dado um conjunto de cláusulas definidas Δ e uma cláusula objetivo Γ_0, uma SLD-refutação [1] de Γ_0 a partir de Δ é uma seqüência de cláusulas objetivos $\Gamma_0, \ldots, \Gamma_n$, onde Γ_n é a cláusula vazia e cada cláusula Γ_{i+1} é derivada de Γ_i resolvendo-se um de seus literais com a cabeça de uma das cláusulas em Δ. Suponha que haja algum Γ_i cujo literal selecionado λ não resolve com nenhuma das cláusulas em Δ. Então, as seqüências iniciando com $\Gamma_0, \ldots, \Gamma_i$ não levarão à cláusula vazia. Mas se estamos procurando por um conjunto Σ de literais tal que $\Delta \cup \Sigma \models \Gamma_0$, então, claramente, incluindo em Σ um literal que resolva com λ, podemos continuar a busca com Γ_{i+1} igual a Γ_i, exceto pelo literal λ, que é removido.

Definição 4.2 *Sejam Δ um conjunto de cláusulas definidas, Γ_0 uma cláusula objetivo e Σ_n um conjunto de literais abdutíveis tal que $\Delta \cup \Sigma_n \models \Gamma_0$. Uma* SLDA*-refutação para Γ_0 a partir de Δ é uma seqüência da forma $\langle \Gamma_0, \Sigma_0 \rangle, \ldots, \langle \Gamma_n, \Sigma_n \rangle$, onde cada Γ_i é um cláusula objetivo, cada Σ_i é um conjunto de literais abdutíveis, Γ_n é a cláusula vazia, Σ_0 é o conjunto vazio e cada $\langle \Gamma_{i+1}, \Sigma_{i+1} \rangle$ é obtido de $\langle \Gamma_i, \Sigma_i \rangle$ da seguinte maneira. Seja λ o literal selecionado de Γ_i: se λ pode ser resolvido com uma das cláusulas em Δ, então a resolução é efetuada e obtemos Σ_{i+1} igual a Σ_i; caso contrário, se λ é abdutível, então Γ_{i+1} é $\Gamma_i - \lambda$ e Σ_{i+1} é $\Sigma_i \cup \{\lambda'\}$, onde λ' é λ com todas suas variáveis substituídas por constantes de Skolem. O conjunto cumulativo de literais Σ_n é denominado resíduo.* □

Uma importante questão é o tratamento de variáveis não-instanciadas num átomo a ser abduzido. Se tal átomo é tratado como uma cláusula que pode ser adicionada ao programa, as variáveis que antes eram existenciais tornam-se universais, o que é uma hipótese demasiadamente forte. A fim de preservar a corretude, ou *skolemizamos* essas variáveis, o que requer um algoritmo de unificação estendido, ou, então, mantemos o resíduo separado das cláusulas do programa.

4.1.2 Abdução e negação por falha

Para aumentar a expressividade de um programa lógico, é necessário introduzir a negação. Uma forma de tratar a negação no procedimento de resolução é empregar o mecanismo introduzido por *Clark* [5], conhecido como *negação por falha*. Esse mecanismo supõe a *hipótese do mundo fechado* [28] e, portanto, o *completamento* do programa. A idéia básica é a seguinte: a negação de um átomo ϕ pode ser inferida a partir de um programa lógico Δ se não existe uma SLD-refutação para $\leftarrow \phi$ a partir de Δ. Quando esse mecanismo é aplicado à SLD-resolução, o procedimento resultante é chamado SLDNF-*resolução* [1]. Note que a negação por falha é uma operação não-monotônica, já que a inclusão de novas cláusulas em Δ pode eliminar conclusões negativas anteriormente provadas.

Em princípio, não parece difícil juntar negação e abdução: basta adicionar o passo de abdução à SLDNF-resolução para obtermos SLDNFA-*resolução* [11]. Entretanto, como a validade de um literal negativo é demonstrada pela falha de uma *prova aninhada* do seu literal complementar e apenas as conseqüências lógicas do programa mais o resíduo já computado devem ser consideradas nessa prova, é preciso que a adição de novas hipóteses (abdução) em provas aninhadas seja desabilitada. Um outro problema importante que surge da junção de negação e abdução é que fatos abduzidos podem permitir a expansão da árvore de prova, onde anteriormente não havia possibilidade, e tornar negação por falha incorreta [11]. Suponha que o literal selecionado da cláusula objetivo seja $not(\lambda)$. O método usual de negação por falha é adotado e $not(\lambda)$ é assumido como verdadeiro se λ não pode ser provado a partir do programa mais o resíduo corrente. Porém, mais tarde, adições ao resíduo podem tornar λ provável, o que deve ser evitado. Então, para obter-se um procedimento correto, é necessário coletar negações

[14] Por exemplo, não queremos explicações abdutivas triviais do tipo $\Delta \cup \Gamma \models \Gamma$.

já provadas e checá-las sempre que uma nova hipótese for adicionada ao resíduo. Assim como no caso de literais abdutíveis, todas as variáveis em suposições negativas registradas devem ser *skolemizadas*.

4.1.3 Planejamento como uma tarefa abdutiva

Planos podem ser vistos como explicações de como um estado meta é atingido [8] e, portanto, a abdução é uma regra de inferência bastante adequada para resolver problemas de planejamento. Sejam Δ um conjunto de fórmulas descrevendo as ações do domínio, Σ um conjunto de fórmulas descrevendo um estado inicial e Γ um conjunto de fórmulas descrevendo um estado meta. O planejamento abdutivo consiste em encontrar um conjunto de fórmulas Π, descrevendo ações e restrições de ordenação, tal que $\Delta \cup \Sigma \cup \Pi \models \Gamma$. Como veremos, o conjunto Π representa um plano parcialmente ordenado capaz de atingir o estado meta Γ, a partir do estado inicial Σ.

4.2 Planejamento abdutivo com cálculo de eventos

Uma caracterização precisa de planejamento abdutivo, em termos de cálculo de eventos circunscritivo, é apresentada por *Shanahan* [35], conforme segue.

Definição 4.3 *Uma descrição de domínio é uma conjunção finita de fórmulas da forma*

$Initiates(\alpha, \phi, \tau) \leftarrow \beta$ *ou*
$Terminates(\alpha, \phi, \tau) \leftarrow \beta$ *ou*
$Releases(\alpha, \phi, \tau) \leftarrow \beta$,

onde β *é da forma* $(\neg)HoldsAt(\phi_1, \tau) \wedge \ldots \wedge (\neg)HoldsAt(\phi_n, \tau)$, α *é uma ação livre de variáveis,* ϕ, $\phi_1, \ldots, \phi_n$ *são fluentes livres de variáveis e* τ *é um instante de tempo.* □

Definição 4.4 *Uma situação inicial é uma conjunção finita de fórmulas da forma* $Initially_N(\phi)$ *ou* $Initially_P(\phi)$, *onde* ϕ *é um fluente livre de variáveis e cada fluente ocorre no máximo uma vez.* □

Definição 4.5 *Uma meta é uma conjunção finita de fórmulas da forma* $(\neg)HoldsAt(\phi, \tau)$ *onde* ϕ *é um fluente livre de variáveis e* τ *é um instante de tempo constante.* □

Definição 4.6 *Uma narrativa é uma conjunção finita de fórmulas da forma* $Happens(\alpha, \tau)$ *ou* $t_1 < t_2$, *onde* α *é uma ação livre de variáveis e* τ, τ_1, τ_2 *são instantes de tempo constantes.* □

Definição 4.7 *Sejam* Γ *uma meta,* Δ *uma descrição de domínio,* Σ *uma situação inicial e* Ω *a conjunção dos axiomas de unicidade de nomes para ações e fluentes mencionados em* Δ. *Um plano para* Γ *é uma narrativa* Π *tal que*

(1) $\textsc{Circ}[\Delta; Initiates, Terminates, Releases] \wedge \textsc{Circ}[\Sigma \wedge \Pi; Happens] \wedge EC \wedge \Omega$ *é consistente,*
(2) $\textsc{Circ}[\Delta; Initiates, Terminates, Releases] \wedge \textsc{Circ}[\Sigma \wedge \Pi; Happens] \wedge EC \wedge \Omega \models \Gamma$. □

Circunscrevendo *Initiates*, *Terminates* e *Releases* assumimos que as ações não têm efeitos inesperados e circunscrevendo *Happens*, que não há ocorrências de eventos inesperados. Para garantir a condição de consistência em (1), basta garantir que a descrição do domínio seja *livre de conflitos*.

Definição 4.8 *Uma descrição de domínio é livre de conflito se, para todo par de fórmulas em* Δ *da forma* $Initiates(\alpha, \phi, \tau) \leftarrow \beta_1$ *e* $Terminates(\alpha, \phi, \tau) \leftarrow \beta_2$ *temos* $\models \neg[\beta_1 \wedge \beta_2]$ □

Proposição 4.9 *Sejam Δ uma descrição de domínio livre de conflitos, Σ uma situação inicial, Π uma narrativa totalmente ordenada, e Ω a conjunção dos axiomas de unicidade de nomes para ações e fluentes mencionados em Δ. A fórmula* $\text{CIRC}[\Delta; Initiates, Terminates, Releases] \wedge \text{CIRC}[\Sigma \wedge \Pi; Happens] \wedge EC \wedge \Omega$ *é consistente.*

4.3 Meta-interpretador abdutivo para cálculo de eventos

Um provador de teoremas abdutivo, especializado para o cálculo de eventos, é necessário para transformar essa especificação lógica de planejamento de ordem parcial numa implementação prática. Esse provador, baseado em resolução, pode ser codificado como um meta-interpretador em PROLOG[15].

Por exemplo, na figura 7, temos um meta-interpretador trivial que simula a própria estratégia de execução do PROLOG. Em (1), ele demonstra a cláusula vazia []; em (2), realiza um passo de resolução, onde o primeiro literal de $[G|Gs1]$ é resolvido com a cabeça de uma cláusula $axiom(G, Gs2)$ do programa-objeto; finalmente, em (3), ele implementa a negação por falha finita, onde a falha da prova aninhada $demo(G)$ estabelece a validade do literal $not(G)$. Note que uma fórmula $axiom(\lambda, [\lambda_1, \ldots, \lambda_n])$ representa uma cláusula da forma λ :- $\lambda_1, \ldots, \lambda_n$.

```
(1) demo([]).
(2) demo([G|Gs1]) :-
       axiom(G,Gs2),
       append(Gs2,Gs1,Gs3),
       demo(Gs3).
(3) demo([not(G)|Gs]) :-
       not demo(G),
       demo(Gs).
```

Figura 7: *Meta-interpretador trivial*

A figura 8 mostra um outro meta-interpretador que implementa o passo abdutivo, sem negação por falha. A fórmula $abdemo(Gs, R_{in}, R_{out})$ vale se Gs segue da conjunção do resíduo R_{out} com o programa-objeto. Na chamada inicial a *abdemo*, o resíduo R_{in} deve ser instanciado a [ ]. A novidade nessa versão está no passo abdutivo realizado em (2). Note que se o literal selecionado G é declarado no programa-objeto como *abducible*, então ele é simplesmente adicionado ao resíduo, como hipótese, e o procedimento continua normalmente.

```
(1) abdemo([],R,R).
(2) abdemo([G|Gs],R1,R2) :-
       abducible(G),
       abdemo(Gs,[G|R1],R2).
(3) abdemo([G|Gs1],R1,R2) :-
       axiom(G,Gs2),
       append(Gs2,Gs1,Gs3),
       abdemo(Gs3,R1,R2).
```

Figura 8: *Meta-interpretador abdutivo*

[15] Em PROLOG variáveis iniciam com maiúsculas e constantes com minúsculas.

4.3.1 Estendendo abdução com negação por falha

As coisas ficam mais difíceis quando juntamos negação por falha e abdução. Quando novas hipóteses são adicionadas ao resíduo, literais negativos previamente provados podem deixar de valer. Então, esses literais devem ser registrados e checados a cada vez que o resíduo é modificado. Na versão de *abdemo* que implementa negação por falha, apresentada na figura 9, o último argumento é justamente uma lista de literais negativos, que são coletados para subseqüente checagem. Resumidamente, essa versão faz o seguinte: em (1), a cláusula vazia é provada trivialmente; em (2), se o literal selecionado G é abdutível e consistente com a lista de negações já provadas, ele é adicionado ao resíduo; em (3), temos um passo de resolução; e, em (4), se o literal negativo pode ser demonstrado por falha, ele é adicionado à lista de negações. Uma lista de negações $N = [[\gamma_{1,1}, \ldots, \gamma_{1,n_1}], \ldots, [\gamma_{m,1}, \ldots, \gamma_{m,n_m}]]$ representa a conjunção $\neg(\gamma_{1,1} \wedge \ldots \wedge \gamma_{1,n_1}) \wedge \ldots \wedge \neg(\gamma_{m,1} \wedge \ldots \wedge \gamma_{m,n_m})$. A fórmula $ademo\_nafs(N, R_{in}, R_{out})$ vale se essa conjunção é provável a partir do completamento da conjunção de R_{out} com o programa-objeto. Em (5) e (6), o predicado $abdemo\_nafs(N, R_{in}, R_{out})$ prova por falha cada conjunto de N aplicando o predicado *abdemo_naf*, que é definido em termos do predicado $findall$ do PROLOG. A justificativa para as cláusulas (7) e (8) é a seguinte: para mostrar que $\neg(\gamma_1 \wedge \ldots \wedge \gamma_n)$ vale, é preciso mostrar que, para toda cláusula do programa-objeto λ :- $\lambda_1, \ldots, \lambda_n$ que resolve com γ_1, $\neg(\lambda_1 \wedge \ldots \wedge \lambda_n \wedge \gamma_2 \wedge \ldots \wedge \gamma_n)$ também vale. Se nenhuma cláusula resolve com γ_1, então, sob a semântica do completamento, $\neg\gamma_1$ vale e, conseqüentemente, temos $\neg(\gamma_1 \wedge \ldots \wedge \gamma_n)$.

```
( 1) abdemo([],R,R,N,N).
( 2) abdemo([G|Gs],R1,R3,N1,N2) :-
        abducible(G),
        abdemo_nafs(N1,[G|R1],R2),
        abdemo(Gs,R2,R3,N1,N2).
( 3) abdemo([G|Gs1],R1,R2,N1,N2) :-
        axiom(G,Gs2),
        append(Gs2,Gs1,Gs3),
        abdemo(Gs3,R1,R2,N1,N2).
( 4) abdemo([not(G)|Gs],R1,R2,N1,N2) :-
        abdemo_naf([G],R1,R2),
        abdemo(Gs,R2,R3,[[G]|N1],N2).

( 5) abdemo_nafs([],R,R).
( 6) abdemo_nafs([N|Ns],R1,R3) :-
        abdemo_naf(N,R1,R2),
        abdemo_nafs(Ns,R2,R3).

( 7) abdemo_naf([G|Gs1],R,R) :-
        not resolve(G,R,Gs2).
( 8) abdemo_naf([G1|Gs1],R1,R2) :-
        findall(Gs2, (resolve(G1,R1,Gs3),append(Gs3,Gs1,Gs2)), Gss),
        abdemo_nafs(Gss,R1,R2).

( 9) resolve(G,R,[]) :-
        member(G,R).
(10) resolve(G,R,Gs) :-
        axiom(G,Gs).
```

Figura 9: *Meta-interpretador abdutivo estendido com negação*

4.3.2 Compilando os axiomas do cálculo de eventos

O meta-interpretador trivial pode ser especializado para o cálculo de eventos através da compilação dos axiomas EC em cláusulas do metanível. Por exemplo, uma cláusula λ :- $\lambda_1, \ldots, \lambda_n$ pode ser compilada na definição de *demo* através da adição da seguinte cláusula ao meta-interpretador:

$demo([\lambda|Gs1])$:-
$axiom(\lambda_1, Gs2)$,
$append(Gs2, [\lambda_2, \ldots, \lambda_n|Gs1], Gs3)$,
$demo(Gs3)$.

O resultado é equivalente àquele obtido pelo meta-interpretador trivial na presença da cláusula do programa-objeto que foi compilada. Considere agora a seguinte cláusula-objeto que corresponde ao axioma (EC2).

```
holds_at(F,T3) :-
   happens(A,T1,T2),
   T2<T3,
   initiates(A,F,T1),
   not clipped(T1,F,T2).
```

Sua compilação resulta na metacláusula a seguir, onde o predicado $before$ é usado para representar ordenação temporal.

```
demo([holds_at(F,T3)|Gs1]) :-
   axiom(initiates(A,F,T1),Gs2),
   axiom(happens(A,T1,T2),Gs3),
   axiom(before(T2,T3),[]),
   demo([not clipped(T1,F,T3)]),
   append(Gs3,Gs2,Gs4),
   append(Gs4,Gs1,Gs5),
   demo(Gs5).
```

A execução dessa metacláusula não simula precisamente a execução PROLOG da cláusula-objeto correspondente. Isso porque o grau extra de controle disponível no metanível permite ajustar a ordem na qual as submetas de *holds_at* são resolvidas. Embora *initiates* seja resolvido imediatamente, a prova de suas submetas é adiada até que *happens* e $before$ tenham sido resolvidos. Essa manobra é necessária para evitar laços. Ademais, tratar *initiates* antes de *happens* resulta num espaço de busca bem menor que aquele obtido se essas submetas fossem tratadas na ordem oposta [4]; já que, nesse caso, eventos irrelevantes para o fluente F seriam então considerados.

Para representar o axioma (EC5), que não é uma cláusula definida, introduzimos a função *neg*. Essa função permite que fórmulas do cálculo de predicados da forma $\neg HoldsAt(f,t)$ sejam escritas como $holds\_at(neg(F), T)$. Então, obtemos a cláusula-objeto

```
holds(neg(F),T3) :-
   happens(A,T1,T2),
   T2<T3,
   terminates(A,F,T1),
   not declipped(T1,F,T2).
```

cuja compilação resulta na seguinte metacláusula:

```
demo([hols(neg(F),T3)|Gs1]) :-
   axiom(terminates(A,F,T1),Gs2),
   axiom(happens(A,T1,T2),Gs3),
   axiom(before(T2,T3),[]),
   demo([not declipped(T1,F,T3)]),
   append(Gs3,Gs2,Gs4),
   append(Gs4,Gs1,Gs5),
   demo(Gs5).
```

4.3.3 Tratamento de informação incompleta

No contexto de informação incompleta sobre um predicado, não queremos assumir o seu completamento e, portanto, não podemos usar negação por falha para provar literais negados para esse predicado. A saída é reter literais negados de tal predicado no metanível, dando tratamento especial a eles. Em geral, se sabemos $\neg\phi \leftarrow \psi$, para provar $\neg\phi$ é suficiente provar ψ. Similarmente, se sabemos $\neg\phi \leftrightarrow \psi$, para provar $\neg\phi$, é necessário e suficiente provar ψ.

No caso de planejamento de ordem parcial, temos informação incompleta sobre o predicado $before$, o que permite que narrativas parcialmente ordenadas de eventos sejam representadas. Assim, quando o meta-interpretador encontra um literal negativo da forma $before(X, Y)$, ele tenta provar $before(Y, X)$. Uma forma de conseguir isso é adicionar $before(Y, X)$ ao resíduo, se o resíduo resultante for consistente.

O predicado $before$ não é a única fonte de informação incompleta. Considerações similares afetam o tratamento do predicado $holds\_at$. Quando o provador encontra um literal $not(holds\_at(F, T))$, onde F é um fluente livre de variáveis, ele tenta provar $holds\_at(neg(F), T)$ e, inversamente, quando encontra $not(holds\_at(neg(F), T))$, ele tenta provar $holds\_at(F, T)$. Em ambos os casos, podemos ter novas adições ao resíduo.

4.3.4 O sistema de planejamento AECP

O AECP[16], desenvolvido por *Shanahan* [35], nada mais é que o resultado da compilação dos axiomas do cálculo de eventos, diretamente, em cláusulas do meta-interpretador *abdemo*. Para resolver um problema de planejamento, descrevemos o domínio com cláusulas *initiates*, *terminates* e *releases*, a situação inicial com cláusulas *initially* e, então, fornecemos ao AECP uma lista de literais *holds_at* representando a situação meta. Como resposta, ele devolverá um resíduo, contendo literais *happens* e $before$, que é o plano desejado.

Shanahan [35] estabelece uma correspondência entre o AECP e um algoritmo de planejamento de ordem parcial através de uma simples inspeção de código, conforme se observa na figura 10. Para compreender melhor essa correspondência, precisamos definir o conceito de *intervalo de proteção* e entender como ele é usado no tratamento de ameaças pelo AECP.

Definição 4.10 *Um intervalo de proteção é uma tripla da forma $\langle\tau_1, \phi, \tau_2\rangle$, onde ϕ é um fluente e τ_1 e τ_2 especificam um intervalo no qual nenhuma ação que termine ϕ ocorre.* □

Note que um intervalo de proteção da forma $\langle\tau_1, \phi, \tau_2\rangle$ equivale a um vínculo causal $\alpha_p \xrightarrow{\phi} \alpha_c$ tal que $Happens(\alpha_p, \tau_1)$, $Happens(\alpha_c, \tau_2)$ e ϕ é um efeito de α_1 e uma precondição de α_2. O propósito de um intervalo de proteção, assim como o de um vínculo causal, é garantir que a adição de uma nova ação não ameaça a validade de um fluente já estabelecido por alguma outra ação do plano. Sendo assim, um literal $clipped(\tau_1, \phi, \tau_2)$, representando um intervalo de proteção, é adicionado à lista de suposições

[16] *Abductive Event Calculus Planner.*

```
abdemo([holds_at(F1,T3)|Gs1],R1,R5,N1,N4) :-  % seleciona uma meta F1 a ser atingida
  abresolve(initiates(A,F1,T1),R1,Gs2,R1),    % escolhe uma ação A que tenha F1 como efeito
  abresolve(happens(A,T1,T2),R1,[],R2),       % adiciona uma instância A' dessa ação no plano
  abresolve(before(T2,T3),R2,[],R3),          % adiciona uma restrição de ordenação no plano
  add_neg([clipped(T1,F1,T3)],N1,N2),         % adiciona um vínculo causal no plano
  abdemo_nafs(N2,R3,R4,N2,N3),                % realiza tratamento de ameaças
  append(Gs2,Gs1,Gs3),                        % estabelece as precondições de A' como novas metas
  abdemo(Gs3,R4,R5,N3,N4).                    % continua até que todas as metas tenham sido atingidas
```

Figura 10: *Correspondência entre o* AECP *e um planejador de ordem parcial*

negativas sempre que um literal $happens(\alpha_p, \tau_1)$, representando a ocorrência de uma ação α_p no instante τ_1, é incluído no resíduo, com a finalidade de garantir uma precondição ϕ de uma ação α_p cuja ocorrência no instante τ_2 também já foi postulada no resíduo. A partir daí, cada vez que uma nova ação tem que ser adicionada ao resíduo, esses intervalos de proteção são checados para determinar se a adição é possível. Eventualmente, caso um conflito seja detectado durante essa checagem, o resíduo pode ser incrementado com a adição de cláusulas de ordenação temporal extras, para evitar que algum intervalo de proteção seja ameaçado pela nova ação. Essas restrições de ordenação visam antecipar ou postergar a ação que ameaça o intervalo considerado.

4.3.5 Exemplos de análises que serão feitas

Diferentemente da análise feita por *Shanahan* [35], que basicamente mostra a correspondência entre a metacláusula da figura 10 e o POP, propomos uma análise baseada em comparações de *passos de planejamento* [2] ou *tarefas de refinamento* [13], i.e. inserção de ações, adição de restrições de ordenação, adição de restrições de (não-)codesignação, proteção de vínculos causais e retrocesso , etc. Por exemplo, uma análise interessante é verificar que impacto a sistematicidade do SNLP pode ter na eficiência do AECP. Para isso, podemos modificar a axiomatização do cálculo de eventos, implementar as alterações correspondentes no AECP, de modo a torná-lo sistemático, e testá-lo em alguns domínios específicos.

Em seu artigo [35], *Shanahan* não chega a mostrar a correspondência que ele afirma existir entre o AECP e o UCPOP. Na verdade, ele mostra apenas uma correspondência com o POP, que é baseado em STRIPS proposicional. Assim, uma outra análise interessante é verificar como estender o AECP de modo que ele suporte a representação ADL, i.e. efeitos condicionais e quantificação universal.

Além dessas, outras técnicas de planejamento serão estudadas, implementadas no AECP e analisadas; entre elas, multicontribuidores [12], decomposição hierárquica [7], planejamento de ordem total e planejamento com ações de percepção.

5 Metodologia

Para atingir os objetivos propostos nesse trabalho, será empregada a seguinte metodologia:

- Estudar e implementar algoritmos eficientes (corretos e completos) da literatura de planejamento.
- Alterar o meta-interpretador AECP de modo a implementar esses algoritmos, mantendo o propósito original de sua especificação:
  1. Raciocínio correto sobre representação correta.
  2. Independência de detalhes algoritmos sem o comprometimento da eficiência do planejador.

Método de Avaliação: Comparar os passos de planejamento observados nos sistemas algorítmicos com aqueles observados nos sistemas produzidos segundo a abordagem lógica.

6 Cronograma

A seguir apresentamos o cronograma com as atividades que serão desenvolvidas visando alcançar os objetivos propostos para esta dissertação.

Relação das Atividades:

1. Estudar e implementar alguns algoritmos de planejamento (Snlp, Htn e outros).
2. Estender o meta-interpretador Aecp de modo a implementar esses algoritmos de planejamento.
3. Avaliar a eficiência dessas extensões em alguns domínios específicos.
4. Redigir a dissertação.
5. Defender a dissertação.

Cronograma (Mar/2001 - Dez/2001):

	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
1	x	x	x	x						
2		x	x	x	x	x	x			
3				x	x	x	x	x		
4					x	x	x	x	x	x
5										x

Prazo máximo para defesa: Março/2003

A O meta-interpretador AECP

A.1 Uma implementação em SWI-PROLOG

```
/*
  Abductive Event Calculus Planner,
  baseado na versão 4.2 de M. P. Shanahan
*/

abdemo(Gs,[H,B]) :-
   initsym(t),
   abdemo(Gs,[[],[]],[H,B],[],N),
   write('Passos do plano: '), nl,
   writelst(H),
   write('Restricoes de ordenacao: '), nl,
   writelst(B),
   write('Intervalos de protecao: '), nl,
   writelst(N).

abdemo([],R,R,N,N).

/*
(EC1) holds_at(F1,T) <- initiallyP(F) & ~clipped(0,F,T)
*/

abdemo([holds_at(F1,T)|Gs1],R1,R3,N1,N4) :-
   F1 \= neg(F2),
   abresolve(initially(F1),R1,Gs2,R1),
   append(Gs2,Gs1,Gs3),
   add_neg([clipped(0,F1,T)],N1,N2),
   abdemo_naf([clipped(0,F1,T)],R1,R2,N2,N3),
   abdemo(Gs3,R2,R3,N3,N4).

/*
(EC2) holds_at(T,T3) <- happens(A,T1,T2) & initiates(A,F,T1) & T2<T3 & ~clipped(T1,F,T3)
*/

abdemo([holds_at(F1,T3)|Gs1],R1,R5,N1,N4) :-
   F1 \= neg(F2),
   abresolve(initiates(A,F1,T1),R1,Gs2,R1),
   abresolve(happens(A,T1,T2),R1,[],R2),
   abresolve(before(T2,T3),R2,[],R3),
   append(Gs2,Gs1,Gs3),
   add_neg([clipped(T1,F1,T3)],N1,N2),
   abdemo_nafs(N2,R3,R4,N2,N3),
   abdemo(Gs3,R4,R5,N3,N4).

/*
(EC4) ~holds(F,T) <- initiallyN(F) & ~declipped(0,F,T)
*/

abdemo([holds_at(neg(F),T)|Gs1],R1,R3,N1,N4) :-
   abresolve(initially(neg(F)),R1,Gs2,R1),
   append(Gs2,Gs1,Gs3), add_neg([declipped(0,F,T)],N1,N2),
   abdemo_naf([declipped(0,F,T)],R1,R2,N2,N3),
   abdemo(Gs3,R2,R3,N3,N4).

/*
```

```
(EC5) ~holds(F,T3) <- happens(A,T1,T2) & terminates(A,F,T1) & T2<T3 & ~declipped(T1,F,T3)
*/

abdemo([holds_at(neg(F),T3)|Gs1],R1,R5,N1,N4) :-
   abresolve(terminates(A,F,T1),R1,Gs2,R1),
   abresolve(happens(A,T1,T2),R1,[],R2),
   abresolve(before(T2,T3),R2,[],R3),
   append(Gs2,Gs1,Gs3),
   add_neg([declipped(T1,F,T3)],N1,N2),
   abdemo_nafs(N2,R3,R4,N2,N3),
   abdemo(Gs3,R4,R5,N3,N4).


% Resolve cláusulas não relacionadas ao EC

abdemo([G|Gs1],R1,R3,N1,N2) :-
   abresolve(G,R1,Gs2,R2),
   append(Gs2,Gs1,Gs3),
   abdemo(Gs3,R2,R3,N1,N2).


abresolve(terms_or_rels(A,F,T),R,Gs,R) :- axiom(releases(A,F,T),Gs).

abresolve(terms_or_rels(A,F,T),R,Gs,R) :- axiom(terminates(A,F,T),Gs).

abresolve(inits_or_rels(A,F,T),R,Gs,R) :- axiom(releases(A,F,T),Gs).

abresolve(inits_or_rels(A,F,T),R,Gs,R) :- axiom(initiates(A,F,T),Gs).


abresolve(happens(A,T),R1,Gs,R2) :-
   abresolve(happens(A,T,T),R1,Gs,R2).

abresolve(happens(A,T1,T2),[HA,BA],[],[HA,BA]) :-
   member(happens(A,T1,T2),HA).

abresolve(happens(A,T,T),[HA,BA],[],[[happens(A,T,T)|HA],BA]) :-
   executable(A),
   skolemise(T).

abresolve(before(X,Y),R,[],R) :-
   demo_before(X,Y,R).

abresolve(before(X,Y),R1,[],R2) :-
   \+ demo_before(X,Y,R1),
   \+ demo_beq(Y,X,R1),
   add_before(X,Y,R1,R2).

abresolve(diff(X,Y),R,[],R) :- X \= Y.

abresolve(G,R,Gs,R) :- axiom(G,Gs).


abdemo_nafs([],R,R,N,N).

abdemo_nafs([N|Ns],R1,R3,N1,N3) :-
   abdemo_naf(N,R1,R2,N1,N2),
   abdemo_nafs(Ns,R2,R3,N2,N3).
```

```
/*
(EC3) clipped(T1,F,T4) <-> happens(A,T2,T3) & T1<T3 & T2<T4 & (terminates(A,F,T2)|releases(A,F,T2)
*/

abdemo_naf([clipped(T1,F,T4)|Gs1],R1,R2,N1,N2) :-
   findall(Gs3,
           (abresolve(terms_or_rels(A,F,T2),R1,Gs2,R1),
           abresolve(happens(A,T2,T3),R1,[],R1),
           append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3)),Gss),
   abdemo_nafs(Gss,R1,R2,N1,N2).

/*
(EC6) declipped(T1,F,T4) <-> happens(A,T2,T3) & T1<T3 & T2<T4 & (initiates(A,F,T2)|releases(A,F,T2)
*/

abdemo_naf([declipped(T1,F,T4)|Gs1],R1,R2,N1,N2) :-
   findall(Gs3,
           (abresolve(inits_or_rels(A,F,T2),R1,Gs2,R1),
           abresolve(happens(A,T2,T3),R1,[],R1),
           append([before(T1,T3),before(T2,T4)|Gs2],Gs1,Gs3)),Gss),
   abdemo_nafs(Gss,R1,R2,N1,N2).

abdemo_naf([holds_at(F1,T)|Gs],R1,R2,N1,N2) :-
   opposite(F1,F2),
   abdemo([holds_at(F2,T)],R1,R2,N1,N2).

abdemo_naf([holds_at(F,T)|Gs],R1,R2,N1,N2) :-
   abdemo_naf(Gs,R1,R2,N1,N2).

abdemo_naf([before(X,Y)|Gs],R,R,N,N) :-
   X = Y.

abdemo_naf([before(X,Y)|Gs],R,R,N,N) :-
   X \= Y, demo_before(Y,X,R).

abdemo_naf([before(X,Y)|Gs],R1,R2,N1,N2) :-
   X \= Y, \+ demo_before(Y,X,R1),
   abdemo_naf(Gs,R1,R2,N1,N2).

abdemo_naf([before(X,Y)|Gs],R1,R2,N,N) :-
   X \= Y, \+ demo_before(Y,X,R1),
   \+ demo_beq(X,Y,R1), add_before(Y,X,R1,R2).

abdemo_naf([G|Gs1],R,R,N,N) :-
   G \= clipped(T1,F,T2), G \= declipped(T1,F,T2), G \= holds_at(F,T),
   G \= before(X,Y), \+ abresolve(G,R,Gs2,R).

abdemo_naf([G1|Gs1],R1,R2,N1,N2) :-
   G1 \= clipped(T1,F,T2), G1 \= declipped(T1,F,T2),
   G1 \= holds_at(F,T), G1 \= before(X,Y),
   findall(Gs3,(abresolve(G1,R1,Gs2,R1),append(Gs2,Gs1,Gs3)),Gss),
   Gss \= [], abdemo_nafs(Gss,R1,R2,N1,N2).


demo_before(X,Y,[HA,BA]) :- demo_before(X,Y,BA,[]).

demo_before(0,Y,R,L) :- Y \= 0.

demo_before(X,Y,R,L) :- X \= 0, member(before(X,Y),R).
```

```
demo_before(X,Y,R,L) :- X \= 0, \+ member(before(X,Y),R), member(X,L).

demo_before(X,Y,R,L) :-
   X \= 0, \+ member(before(X,Y),R), \+ member(X,L),
   member(before(X,Z),R), demo_before(Z,Y,R,[X|L]).


demo_beq(X,X,R).

demo_beq(X,Y,R) :-
   X \= Y,
   demo_before(X,Y,R).


add_before(X,Y,[HA,BA]) :-
   member(before(X,Y),BA).

add_before(X,Y,[HA,BA],[HA,[before(X,Y)|BA]]) :-
   \+ member(before(X,Y),BA),
   \+ demo_beq(Y,X,[HA,BA]).

add_neg(N,Ns,Ns) :-
   member(N,Ns).

add_neg(N,Ns,[N|Ns]) :- \+ member(N,Ns).

skolemise(T) :- gensym(T).

opposite(neg(F),F).

opposite(F1,neg(F1)) :- F1 \= neg(F2).

diff(X,Y)  :- not equal(X,Y).

equal(X,X).

initsym(T) :-
   abolish(sym,2),
   asserta(sym(T,1)).

gensym(T) :-
   sym(X,Y), N is Y+1,
   retract(sym(X,Y)),
   asserta(sym(X,N)),
   name(X,Xs),
   name(Y,Ys),
   append(Xs,Ys,Zs),
   name(T,Zs).

writelst([]) :- nl.
writelst([X|Y]) :- write(X), nl, writelst(Y).
```

A.2 Um exemplo de uso: o domínio das compras

O *domínio das compras* [31] consiste em construir um plano para se obter uma furadeira, leite e banana. As ações que o agente pode executar são ir até um local, $ir(X)$, e comprar algo, $comprar(X)$.

```
% objetivos do planejamento

goal :-
   abdemo([holds_at(tem(leite),t),
           holds_at(tem(furadeira),t),
           holds_at(tem(banana),t),
           holds_at(em(casa),t)], [H,B]).

% descrição do domínio

axiom(initially(em(casa)),[]).
axiom(initiates(ir(X),em(X),T),[]).
axiom(terminates(ir(X),em(Y),T),[diff(X,Y)]).
axiom(initiates(comprar(X),tem(X),T),[vende(Y,X), holds_at(em(Y),T)]).
axiom(vende(mercado,leite),[]).
axiom(vende(loja,furadeira),[]).
axiom(vende(mercado,banana),[]).

% politica de abducao

executable(ir(X)).
executable(comprar(X)).
```

A saída produzida pelo AECP, para esse problema, é a seguinte:

```
Passos do plano:
   happens(ir(casa), t6, t6)
   happens(comprar(banana), t5, t5)
   happens(ir(loja), t4, t4)
   happens(comprar(furadeira), t3, t3)
   happens(ir(mercado), t2, t2)
   happens(comprar(leite), t1, t1)
Restricoes de ordenacao:
   before(t1, t6)
   before(t5, t6)
   before(t2, t6)
   before(t4, t6)
   before(t6, t)
   before(t2, t5)
   before(t5, t)
   before(t3, t2)
   before(t4, t3)
   before(t3, t)
   before(t2, t1)
   before(t1, t)
Intervalos de protecao:
   [clipped(t6, em(casa), t)]
   [clipped(t2, em(mercado), t5)]
   [clipped(t5, tem(banana), t)]
   [clipped(t4, em(loja), t3)]
   [clipped(t3, tem(furadeira), t)]
   [clipped(t2, em(mercado), t1)]
   [clipped(t1, tem(leite), t)]
```

Referências

[1] Apt, K. R. *Logic Programming*, In Handbook of Theoretical Computer Science, J. van Leeuwen, Elsevier Science Publishers B.V., pages 493-573, 1990.

[2] Barros, L. N., and Santos, P. E. *The Nature of Knowledge in an Abductive Event Calculus Planner*, Proc. of 12th International Conference EKAW 2000, pages 328-343, 2000.

[3] Chapman, D. *Planning for Conjunctive Goals*, Artificial Intelligence, 32(3), pages 333-377, 1987.

[4] Chittaro, D. and Montanari, A. *Efficient Temporal Reasoning in the Cached Event Calculus*, Computacional Intelligence 12 (3), pages 359-382, 1996.

[5] Clark, K. *Negation as Failure*, In Herve Gallaire and Jack Minker, editors, Logic and Data Bases, pages 293-322, Plenum Press, NY, 1978.

[6] Cox, P. T., and Pietrzykowsky, T. *Causes for Events: Their Computation and Applications*, Proc. 8th International Conference on Automated Deduction, CADE'86, pages 608-621, 1986.

[7] Erol, K. *Hierarchical Task network Planning: Formalization, Analisys and Implementation.* PhD Thesis, University of Maryland, 1995.

[8] Eshghi, K. *Abductive Planning with Event Calculus*, In Proceedings of the Fifth International Conference on Logic Programming, pages 562-579, 1988.

[9] Fikes, R. E., and Nilsson, N. J. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence, vol. 2 (3/4), pages 189-208, 1971.

[10] Green, C. *Application of Theorem Proving to Problem Solving*, Proc. IJCAI-69, pp. 219-240, 1969.

[11] Kakas, A. C., Kowalski, R. A., and Toni, F. *Abductive Logic Programming*, Journal of Logic and Computation, vol. 2, (6):719-770, 1993.

[12] Kambhampati, S. *Multi-contributor Causal Structures for Planning: A Formalization and Evaluation*, Artificial Intelligence, vol. 69, pp. 235-278, 1992.

[13] Kambhampati, S. *Planning as Refinement Search: A Unified Framework for Evaluation Design Tradeoffs in Partial Order Planning*, Artificial Intelligence, vol. 76, pages 167-238, 1995.

[14] Korf, R. E. *Planning as Search: A Quantitative Approach*, Artificial Intelligence, vol. 33, pages 65-88, 1987.

[15] Korf, R. E. *Search: A Survey of Recent Results*, In H. Shrobe, editor, Exploring Artificial Intelligence, pages 197-237. Morgan Kaufmann, San Mateo, CA, 1988.

[16] Korf, R. E. *Linear-space Best-first Search: Summary of Results*, In Proc. 10th Nat. Conf. on A.I., pages 533-538, July 1992.

[17] Kowalski, R. A., and Sergot, M. J. *A Logic-based Calculus of Events*, New Generation Computing 4, pages 67-95, 1986.

[18] Lifschitz, V. *On the Semantics of* Strips, In James Allen, James Hendler, and Austin Tate, editors, Readings in Planning, pp. 523-531. Morgan kaufman, 1990.

[19] Lifschitz, V. *Circumscription*, In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, The Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3, pp. 297-352, 1994.

[20] MacAllester, D., and Rosenblitt, D. *Systematic Nonlinear Planning*, In Proc. 9th Nat. Conf. on A.I., pages 634-639, July 1991.

[21] McCarthy, J. *Epistemological Problems of Artificial Intelligence*, In Proceedings of IJCAI-77, pages 1038-1044, 1977.

[22] McCarthy, J. *Applications of Circumscription to Formalizing Common Sense Knowledge*, Artificial Intelligence, 26(3):89-116, 1986.

[23] McCarthy, J., and Hayes, P. J. *Some Philosophical Problems from the Standpoint of Artificial Intelligence*, In B. Meltzer, D. Michie and M. Swann, ed., Machine Inteligence 4, pp. 463-502, 1969.

[24] McDermott, D. V. *Regression Planning*, Int. Journal of Intelligent Systems, 6:357-416, 1991.

[25] Pednault, E. P. D. *Formulating Multiagent, Dynamic-world Problems in the Classical Planning Framework*, In Georgeff, M. P. and Lansky, A. L., editors, Reasoning About Actions and Plans: Proceedings of the 1986 Workshop, pages 47-82, Timberline, Oregon. Morgan kaufmann, 1986.

[26] Peirce, C. S. *Collected Papers of Charles Sanders Pierce*, vol. 2, 1931-1958, Hartshorn et al., editors. Harvard University Press.

[27] Penberthy, J. S., and Weld, D. S. *UCPOP: A Sound, Complete, Partial Order Planner for ADL*, In Proceedings of KR-92, pages 103-114, 1992.

[28] Reiter, R. *On Closed Word Data Bases*, In H. Gallaire and J. Minker, Logic and Data Bases, pages 55-76, Plenum Press, NY, 1978.

[29] Robinson, J. A. *A Machine-oriented Logic based on the Resolution Principle*, Journal of the ACM, 12:23-41, 1965.

[30] Russell, S. *Efficient Memory Bounded Search Algorithms*, In Proceedings of the Tenth Conference on Artificial Intelligence, Vienna, Wiley, 1992.

[31] Russell, S., and Norvig, P. *Artificial Intelligence - A Modern Approach*, Prentice-Hall, 1995.

[32] Sacerdoti, E. *The Nonlinear Nature of Plans*, In Proceedings of IJCAI-75, pages 206-214, 1975.

[33] Shanahan, M. P. *Prediction is Deduction but Explanation is Abduction*, In Proceedings IJCAI-89, pages 1055-1060, 1989.

[34] Shanahan, M. P. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*, MIT Press, Cambridge, 1997.

[35] Shanahan, M. P. *An Abductive Event Calculus Planner*, The Journal of Logic Programming, 44:207-239, 2000.

[36] Tate, A. *Generating Projects Networks*, In Proc. 5th Int. Joint Conf. on A.I., pp. 888-893, 1977.

[37] Waldinger, R. *Achieving Several Goals Simultaneously*, In Machine Intelligence 8, Hellis Horwood Limited, Chichester, 1977.

[38] Weld, D. S. *An Introduction to Least Commitement Planning*, AI Magazine, 15(4), pages 27-61, Winter 1994.