
# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 42/04

# Uma Introdução ao Uso da Programação em Lógica para Modelagem Conceitual

Antonio L. Furtado

Departamento de Informática

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO**

**RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900**

**RIO DE JANEIRO - BRASIL**

Monografias em Ciência da Computação, No. 42/04 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2004

# Uma Introdução ao Uso da Programação em Lógica para Modelagem Conceitual *

Antonio L. Furtado

furtado@inf.puc-rio.br

**Abstract:** Information systems with a database component should be modelled at three levels, at least, which we denominate *static*, *dynamic* and *behavioural*. The static level refers to the classes of *facts* to be represented in the database. The dynamic level deals with the repertoire of *operations*, which are provided in order to achieve state transitions, by adding and/or removing facts. Taken together, these first two levels cover the object-oriented aspects of the information system being specified. The behavioural level indicates how the various agents are expected to use he system, so as to reach their *goals*, through the execution of authorized operations. This third level extends the specification to cover agent-oriented aspects. This work describes a method, supported by a simple tool written in **Prolog**, for the three-level specification of information systems. The specifications are executable, which allows the designers to run experiments before implementation. The tool includes a version of the WARPLAN plan-generation algorithm.

**Keywords:** information systems, conceptual modelling, logic programming.

**Resumo:** Sistemas de informação com um componente de banco de dados devem ser modelados em três níveis, pelo menos, os quais denominamos *estático*, *dinâmico* e *comportamental*. O nível estático refere-se às classes de *fatos* a serem representados no banco de dados. O nível dinâmico trata do repertório de *operações*, que são fornecidas para efetuar transições entre estados, através da adição/remoção de fatos. Considerados em conjunto, estes dois primeiros níveis cobrem os aspectos orientados a objeto do sistema de informações sendo especificado. O nível comportamental indica de que forma se espera que os vários agentes usem o sistema, de modo a atingir seus *objetivos*, por meio da execução de operações autorizadas. Este terceiro nível estende a especificação para cobrir os aspectos orientados a agente. O presente trabalho descreve um método, apoiado por uma ferramenta simples escrita em Prolog, para a especificação em três níveis de sistemas de informação. As especificações são executáveis, o que permite aos projetistas fazer experiências antes da implementação. A ferramenta inclui uma versão do algoritmo WARPLAN para geração de planos.

**Palavras-chaves:** sistemas de informação, modelagem conceitual, programação em lógica.

---

* This work has been partially supported by CNPq.

**In charge for publications:**

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

## 1. Introdução

No início do uso de computadores, os arquivos de dados armazenados externamente, em geral em fitas ou discos, eram considerados simplesmente como um recurso à disposição de algum programa. Tais arquivos só eram necessários porque (1) ao contrário dos dados localizados na memória principal, esses arquivos não se perdiam ao terminar a execução do programa, e/ou porque (2) esses arquivos continham um volume de dados que não caberia na memória principal.

Logo foi percebido que, em muitos casos, os mesmos dados desses grandes arquivos eram necessitados por mais de um programa. Manter cópias em arquivos separados para uso de cada programa era um desperdício de recursos, além de criar um risco de inconsistência: ao se alterar um dado em um arquivo, as demais cópias ficariam desatualizadas.

Com o advento da tecnologia de *bancos de dados*, tornou-se possível eliminar esse tipo indesejável de redundância. Vários programas poderiam compartilhar arquivos, através de algum software de SGBD (Sistema de Gerência de Bancos de Dados), que viria oferecer uma funcionalidade variada, incluindo a declaração e manipulação dos arquivos, geração de relatórios, controle de concorrência, preservação de restrições de integridade, etc.

O emprego da nova tecnologia levou, no começo, a uma inversão exagerada quanto à importância relativa de programas e arquivos. Os arquivos que constituiam um banco de dados, por guardar informação de alto valor para a empresa, passavam a ser considerados em primeiro lugar: projetava-se e instalava-se um banco de dados para só depois pensar nos programas que iriam utilizá-lo. A prática não tardou a corrigir essa atitude; hoje os projetistas geralmente procedem de forma equilibrada, conduzindo basicamente em paralelo a especificação de arquivos e programas, ambos componentes essenciais de um *sistema de informação*.

Reconheceu-se, também, que era preciso iniciar a especificação em termos *conceituais*, ou seja, levantar os requisitos do sistema como ele existiria no mundo real, e expressá-los na própria linguagem do *domínio de aplicação* (tal como administração de pessoal, gerência de vendas, etc.), antes de considerar como seria a implementação em alguma plataforma computacional.

Em termos conceituais, cabe definir que classes de *fatos* poderão existir no mini-mundo do domínio de aplicação. Adotando-se o modelo **Entidades-Relacionamentos** (**ER**) [Ch,He], um fato pode-se referir à existência de uma instância de uma classe de entidades (e.g. empregado), à existência de um relacionamento entre instâncias de entidades (e.g. um empregado lotado em um departamento), e aos valores de atributos dessas entidades ou relacionamentos (e.g. salário de empregado, data em que o empregado passou a estar lotado no departameno). O conjunto de fatos que valem em um dado momento corresponde a um *estado* do mini-mundo.

Para a definição conceitual de *operações*, capazes de efetuar a transição entre estados, através da adição e/ou remoção de fatos, é também necessário atentar para como funciona o domínio de aplicação. No domínio de administração de pessoal, tem-se operações para contratar empregados, vinculá-los a departamentos, etc. O modelo **STRIPS** (Stanford University Research Institute Ploblem-Solver) [FN] permite definir operações por suas *pré-condições* e *pós-condições*. A pré-condição de uma operação é uma expressão lógica envolvendo fatos positivos e/ou negativos, a qual deve ser verdadeira para que a operação seja executada; a pós-condição caracteriza os efeitos da operação, e consiste em dois conjuntos de fatos: os que a operação adiciona e os que a operação remove.

No mini-mundo do domínio de aplicação atuam pessoas (físicas ou jurídicas). Tais *agentes*, diante de determinadas *situações* que observam no estado corrente, passam a ter certos *objetivos*. Para atingir tais objetivos, executam uma ou mais dentre as operações definidas, desde que estejam autorizados – para operações que não estão autorizados a executar, recorrem a outros agentes que estejam. Pode-se dizer que um dos modos de caracterizar o comportamento dos agentes é enumerar para cada um deles o conjunto de *regras situação-objetivo* que os leva a agir.

A abordagem que temos adotado para a especificação conceitual de sistemas de informação [CF], compreende três níveis de esquemas: o nível *estático* para a caracterização das classes de fatos, o *dinâmico* para as operações e o *comportamental* para a atuação dos agentes. Tomados juntos, os dois primeiros níveis cobrem os aspectos *orientados a objetos* [KL] do sistema de informação sendo especificado. O terceiro estende a especificação para englobar os aspectos *orientados a agentes* [RG]. Utilizamos o modelo **ER** para o nível estático e o **STRIPS** para o dinâmico. Para o comportamental, recorremos a *regras situação-objetivo* (e também a *planos típicos*, conceito que não será tratado aqui – cf. [CF]).

Para tornar concreta a discussão, introduziremos o método através de um exemplo. A notação usada é a da programação em lógica, especificamente a da linguagem **Prolog**.

## 2. Nível estático

As entidades são `empregados` e `departamentos`, cujos atributos identificadores são, respectivamente, nome e sigla. Entre empregados e departamentos há dois relacionamentos: `lotado` e `dirige`, sendo o primeiro 1:n (cada empregado só pode estar vinculado a um departamento) e o segundo 1:1 (cada departamento só pode ser dirigido por um único empregado, um empregado não pode dirigir mais de um departamento). Outra *restrição de integridade* é que, para dirigir um departamento, o empregado tem de estar vinculado a ele. Empregados têm ainda o atributo `salário`, cujo valor mínimo é 100.

```
% ESQUEMA ESTATICO

entity(empregado,nome).
entity(departamento,sigla).

attribute(empregado,salario).

relationship(lotado,[empregado,departamento]).
relationship(dirige,[empregado,departamento]).
```

## 3. Nível dinâmico

As operações permitem `contratar` um empregado, `vinculá-lo` a um departamento, `transferí-lo` para outro departamento, `nomeá-lo` para diretor do departamento a que está vinculado, e `promovê-lo` com relação ao salário. Um empregado que dirige um departamento não pode ser transferido para outro. Ao ser contratado, o empregado tem o salário inicial 100. Como os salários não podem decrescer, o salário novo fixado pela promoção tem de ser maior do que o anterior. Note que não é fornecida operação para criar departamentos, de modo que os únicos que existem são Informática (INF) e Elétrica (ELE), a serem indicados no estado inicial (descrito mais adiante).

Embora uma alternativa possível, e freqüentemente usada, seja declarar explicitamente as restrições de integridade no formalismo de especificação, optamos neste exemplo por manter as

restrições através do mecanismo de pré e pós-condições. Como se pode verificar, se apenas as operações aqui definidas forem utilizadas para manipular o banco de dados, as seguintes restrições serão preservadas:

- Os atributos nome e sigla têm um único valor
- O atributo salário tem um único valor
- O relacionamento lotado é 1:n
- O relacionamento dirige é 1:1
- O valor mínimo de salário é 100
- O valor do salário não pode decrescer
- Só pode estar lotado em departamento quem for empregado
- Só pode dirigir um departamento um empregado lotado nesse departamento
- Só existem os departamentos INF e ELE
- Empregado que dirige um departamento não pode passar para outro departamento

```
% ESQUEMA DINAMICO

operation(contrata(E)).
added(empregado(E),contrata(E)).
added(salario(E,100),contrata(E)).
precond(contrata(E),true, _).

operation(vincula(E,D)).
added(lotado(E,D),vincula(E,D)).
precond(vincula(E,D),(departamento(D),empregado(E)),S) :-
   not (departamento(D1), holds(lotado(E,D1),S)).

operation(transfere(E,D1,D2)).
deleted(lotado(E,D1),transfere(E,D1,D2)).
added(lotado(E,D2),transfere(E,D1,D2)).
precond(transfere(E,D1,D2),departamento(D2),S) :-
   departamento(D1),
   not holds(dirige(E,D1),S).

operation(nomeia(E,D)).
added(dirige(E,D),nomeia(E,D)).
precond(nomeia(E,D),lotado(E,D),S) :-
  not (empregado(E1),holds(dirige(E1,D),S)).

operation(promove(E,S1,S2)).
deleted(salario(E,S1),promove(E,S1,S2)).
added(salario(E,S2),promove(E,S1,S2)).
precond(promove(E,S1,S2),(salario(E,S1),S2 > S1), _).
```

## 4. Nível comportamental

Há duas regras situação-objetivo. A primeira refere-se à motivação dos empregados: se ocorrer a situação em que um empregado esteja dirigindo seu departamento e ainda tenha salário 100, isso despertará nele o objetivo de ter salário 200. A segunda é do gerente de pessoal: se um empregado estiver ganhando mais do que o respectivo diretor, o gerente vai querer vê-lo em outro departamento (para simplificar, não incluímos a exigência óbvia de que o diretor desse novo departamento ganhe mais do que o empregado transferido).

```
% ESQUEMA COMPORTAMENTAL

sit_obj(empregado(E), (dirige(E,_),salario(E,100)), salario(E,200)).
sit_obj(gerente, (lotado(E,D),salario(E,S1),
                  dirige(C,D),salario(C,S2),
                  S2<S1,
                  departamento(D1),not (D1 = D)),
                  lotado(E,D1))
```

## 5. Estado inicial

O estado inicial é constituído apenas pelos departamentos de Elétrica e Informática, identificados pelas siglas.

```
% estado inicial do banco de dados

departamento(inf).
departamento(ele).
```

## 6. Executando através da especificação

Depois de carregar os arquivo **imc.ari** (disponível em www-di.puc-rio.br/~furtado/imc.ari) através de `consult`, podemos utilizar o predicado

```
mostra_fatos.
```

para exibir todos os fatos existentes de início, que são apenas `departamento(ele)` e `departamento(inf)`.

Em seguida, experimentemos criar dois empregados, Maria e José, executando a operação de `contratar`, além de `vincular` José a INF, e novamente relacionando os fatos armazenados:

```
executa(contrata(maria)),executa(contrata(jose)),executa(vincula(jose,inf)),
mostra_fatos.
```

Além da informação sobre os departamentos, vão aparecer `empregado(maria)`, `empregado(jose)`, `salario(maria,100)`, `salario(jose,100)`, `lotado(jose,inf)`.

Como não apenas as instâncias de fatos que constituem o banco de dados mas também os três esquemas são disponíveis "on line", podemos fazer consultas sobre esses meta-dados, tais como:

```
listing(entity), listing(operation), listing(sit_obj).
```

obtendo em resposta, respectivamente, as classes de entidades, as operações e as regras situação-objetivo definidas.

Uma vantagem ainda maior de ter dados e meta-dados simultaneamente disponíveis é a possibilidade de combiná-los em consultas mais complexas. O predicado `frame`, por exemplo, permite reunir todos os valores de atributos possuídos por uma instância de entidade. A consulta:

```
frame(maria,X).
```

devolve na variável X a expressão `empregado(nome: maria)/[salario: 100]`. Note que o atributo identificador `nome` é indicado em separado, enquanto os demais atributos aparecem na lista que se segue à barra. Se executarmos:

```
assert(attribute(empregado,profissao)), assert(profissao(maria,engenheiro)).
```

estaremos ampliando o esquema estático, e logo adicionando um fato referente ao novo atributo. Agora submetendo:

```
frame(maria,A/F).
```

obteremos em A `empregado(nome:maria)` e, em F, teremos a lista de pares atributo-valor `[profissao:engenheiro,salario: 100]`.

Outro uso combinado de dados e meta-dados, dessa vez envolvendo as operações, é a geração de planos. Para isso, o arquivo **imc.ari** inclui uma versão estendida do algoritmo WARPLAN. Suponhamos que se queira ter uma certa Laura a dirigir o departamento INF. Um plano para atingir esse objetivo será atribuído à variável P através de:

```
plano(dirige(laura,inf), P).
```

Note que, nesse momento, Laura nem sequer é empregada. O encadeamento de pré e pós-condição produzirá em P a seqüência: `start => contrata(laura) => vincula(laura,inf) => nomeia(laura,inf)`. Muitas vezes a execução de um plano produz outros efeitos além dos que correspondem ao objetivo indicado. Para saber com antecedência quais seriam todos os resultados do plano acima, repetimos o comando com um adendo:

```
plano(dirige(laura,inf), P), mostra_resultado(P).
```

o que faz aparecerem na tela os novos fatos que seriam adicionados, bem como a informação de quais seriam removidos (o que não ocorre neste exemplo). Um dos fatos adicionados será `salario(laura,100)`.

Depois de conferir esses resultados do plano e nos certificarmos de que não haveria efeitos indesejáveis, podemos executar efetivamente o plano, assim mudando o estado do banco de dados:

```
plano(dirige(laura,inf), P), executa(P), mostra_fatos.
```

Laura dirige um departamento e ganha o menor salário permitido, como é de praxe na contratação. Vimos que uma regra situação-objetivo, no esquema comportamental, declara que qualquer empregado, que se veja na situação de dirigir um departamento e continuar com o salário base, terá o objetivo de passar a ganhar o dobro. Na linha abaixo veremos como se pode, de uma só vez, ativar essa regra, gerar um plano para atingir o objetivo despertado, e apurar qual seria seu resultado:

```
pode_querer(A,S,O), plano(O,P), mostra_resultado(P).
```

sendo a varável A instanciada com o agente empregado(laura), S com a conjunção `dirige(laura,inf),salario(laura,100),` e O com `salario(laura,200)`. P será simplesmente `start => promove(laura,100,200)`. Na tela aparecerão os fatos adicionados e removidos:

```
salario(laura,200)
not salario(laura,100)
```

Para terminar, devemos reconhecer que este gerador de planos tem poder limitado. Ainda assim, é capaz de gerar planos alternativos em vários casos. Além disso, aceita conjunções de vários objetivos positivos e negativos. Por exemplo o comando

plano((lotado(maria,inf),not lotado(jose,inf)), P).

produz em P, inicialmente, `start => vincula(maria,inf) => transfere(jose,inf,ele)`. Digitando ";" para solicitar alternativa, obteremos em P as mesmas operações com a ordem trocada: `start => transfere(jose,inf,ele)) => vincula(maria,inf)`.

## 7. Conclusão

Embora simples, as facilidades proporcionadas por este pequeno protótipo bastam para sugerir uma estratégia viável para iniciar um projeto de sistemas de informação inteligentes. Os elementos essenciais desta abordagem são:

O uso de programação em lógica
A dispnibilidade combinada de esquemas e fatos "on line"
O uso do paradigma de geração de planos

Em [CF], demos uma visão mais completa dessas possibilidades, mencionando, entre outros pontos, as vantagens de se dispor também do processo dual de *reconhecimento de planos*. O protótipo **IPG**, descrito no artigo, implementa algoritmos bastante poderosos com essa dupla finalidade: o Abtweak [YTW] para geração e o método de Kautz [Ka] para o reconhecimento de planos.

## Referências Bibliográficas


[Ch] P. P. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems* 1, 1, 1976.

[CF] A. E. M. Ciarlini e A. L. Furtado. Understanding and Simulating Narratives in the Context of Information Systems. In *Proc. of the 21th International Conference on Conceptual Modeling*, 2002.

[FN] R. E. Fikes e N. J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* , 2(3-4), 1971.

[He] C. A. Heuser. *Projeto de Banco de Dados.* Ed. Sagra Luzzatto, 5a ed., 2004.

[Ka] H. A. Kautz. A Formal Theory of Plan Recognition and its Implementation. In *Reasoning about Plans.* J. F. Allen et al (eds.), Morgan Kaufmann, 1991.

[KL] W. Kim e F. H. Lochocsky (eds.), *Object-Oriented Concept, Databases, and Applications.* ACM Press, Addison-Wesley Publishing Company, 1989.

[RG] A. S. Rao e M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of the International Conference on Principles of Knowledge Representation and Reasoning*, 473-484, 1991.

[Wa] D. H. D. Warren. WARPLAN: a system for generating plans. Memo 76. Department of Artificial Intelligence, University of Edinburgh, 1974.

[YTW] Q. Yang, J. Tenenberg, S. Woods. On the Implementation and Evaluation of Abtweak. In *Computational Intelligence Journal*, Vol. 12, N. 2, Blackwell Publishers, 1996.

## Apêndice I: facilidades para rodar experiências com a especificação

```
% FACILIDADES PARA RODAR EXPERIENCIAS

% mostra fatos do estado corrente

mostra_fatos :-
  forall((fact(F), F),
         (nl,write('    '),write(F),nl)).

fact(F) :- not var(F), !,
   F =.. [P|_],
   (entity(P,_);
    attribute(_,P);
    relationship(P,_)).

fact(F) :-
   (entity(P,_), F =.. [P,_];
    attribute(_,P), F =.. [P,_,_];
    relationship(P,_), F =.. [P,_,_]).


% forma frame com atributos de entidade

frame(I,F) :-
  entity(E,C),
  T=..[E,I],
  T,
  findall(A:V,(attribute(E,A),Ta=..[A,I,V],Ta),Afr),
  sort(Afr,Afr1),
  T1=..[E,C:I],
  F = T1/Afr1.


% acha objetivo ativo (a situação motivadora está ocorrendo)

pode_querer(E, S, O) :-
  sit_obj(E, S, O),
  (atom(E), !; E),
  S.


% gera plano para atingir objetivo

plano(O,P) :- plans(O,P).


% mostra qual seria o resultado de operacao ou de plano

:- op(650,yfx,=>).

mostra_resultado(start) :-!.
mostra_resultado(R => O) :- !,
  forall((added(F,O),not F),
    (nl,
     write('    '),
     write(F),
     nl)),
  forall((deleted(F,O),F),
    (nl,
     write('    '),
     write(not F),
     nl)),
```

```
  mostra_resultado(R).
mostra_resultado(O) :-
  mostra_resultado(start => O).


% executa operacao ou plano

executa(start) :- !.
executa(R => O) :- !, exec_plan(R => O).
executa(O) :- exec_plan(start => O).
```

## Apêndice II: versão estendida do algoritmo WARPLAN

```
% geração de planos

:- op(650,yfx,=>).

plans(P,T) :-
  var(P), !,
  net_effects(T,P),
  consistent(P,nil),
  chk(T).
plans(C,T) :-
  not consistent(C,true), !, fail.
plans(C,T) :-
  plan(C,true,start,T),
  net_effects(T,P),
  consistent(P,nil).

plan((X , C),P,T,T2) :- !,
  solve(X,P,T,P1,T1),
  plan(C,P1,T1,T2).
plan(X,P,T,T1) :-
  solve(X,P,T,P1,T1).

solve(not X,P,T,P,T) :-
  not fact(X), not X.
solve(not X,P,T,P1,T) :-
  fact(X),
  not_holds(X,T),
  and(not X,P,P1).
solve(not X,P,T,(not X , P),T1) :-
  fact(X),
  deleted(X,U),
  operation(U),
  achieve(not X,U,P,T,T1).
solve(not _,_,_,_,_) :- !, fail.
solve(X,P,T,P,T) :-
  not fact(X), X.
solve(X,P,T,P1,T) :-
  fact(X),
  holds(X,T),
  and(X,P,P1).
solve(X,P,T,(X , P),T1) :-
  fact(X),
  added(X,U),
  operation(U),
  achieve(X,U,P,T,T1).

achieve(X,U,P,T,T1 => U) :-
  precond(U,C,T),
  noloop(X,C),
  consistent(C,P),
  plan((true , C),P,T,T1),
  productive(U,T1),
  preserves(T1 => U,P).
achieve(X,U,P,T => V,T1 => V) :-
  retrace(P,V,P1,T),
  achieve(X,U,P1,T,T1),
  productive(V,T1),
  precond(V,C1,T1),
  preserves(T1,(true , C1)),
  preserved(X,T1 => V).
```

```
noloop(X,C) :-
  elem(Y,C),
  same(X,Y), !, fail.
noloop(_,_).

same(X,Y) :-
  copy(X,X1),
  copy(Y,Y1),
  name_var(X1),
  name_var(Y1),
  X1 == Y1.

preserves(U,(X , C)) :- !,
  preserved(X,U),
  preserves(U,C).
preserves(U,X) :-
  preserved(X,U).

preserved(not X,V) :-
  holds(X,V), !, fail.
preserved(not X,V) :- !.
preserved(true,V) :- !.
preserved(X,V) :-
  not_holds(X,V), !, fail.
preserved(_,_).

retrace((not P , C),V,X,T) :-
  deleted(P,V), !,
  retrace(C,V,X,T).
retrace((P , C),V,X,T) :-
  added(P,V), !,
  retrace(C,V,X,T).
retrace((P , C),V,(P , C1),T) :-
  retrace(C,V,C1,T).
retrace(true,V,true,T).

holds(X,T => V) :-
  added(X,V),
  operation(V).
holds(X,T => V) :- !,
  holds(X,T),
  not deleted(X,V).
holds(X,T) :-
  given(T,X).

not_holds(X,T => V) :-
  deleted(X,V),
  operation(V).
not_holds(X,T => V) :- !,
  not_holds(X,T),
  not added(X,V).
not_holds(X,T) :-
  not given(T,X).

given(start,X) :-
  fact(X),
  X.

productive(U,T) :-
  added(X,U),
  holds(X,T), !, fail.
productive(U,T) :-
  deleted(X,U),
  not holds(X,T), !, fail. /* mudei */
```

```
productive(_,_).

consistent(C,P) :-
  imposs(S),
  not (not intersect(C,S)),
  (P == nil, implied1(S,C);
   not P == true, implied(S,(C , P))),
  !, fail.
consistent(C,P) :-
  elem(X,C),
  elem(not X,C),
  !, fail.
consistent(_,_).

implied((S1 , S2),C) :- !,
  implied(S1,C),
  implied(S2,C).
implied(not X,C) :-
  elem(X,C), !, fail.
implied(X,C) :-
  elem(not X,C), !, fail.
implied(X,C) :-
  elem(X,C).
implied(X,C) :-
  not fact(X),
  X.

implied1((S1 , S2),C) :- !,
  implied1(S1,C),
  implied1(S2,C).
implied1(not X,C) :-
  elem(X,C), !, fail.
implied1(X,C) :-
  elem(not X,C), !, fail.
implied1(X,C) :-
  elem(X,C).
implied1(X,C) :-
  X.

intersect(S1,S2) :- !,
  elem(X,S1),
  elem(X,S2).

and(X,P,P) :-
  elem(Y,P),
  X == Y, !.
and(X,P,(X , P)).

elem(X,(Y , C)) :-
  elem(X,Y).
elem(X,(Y , C)) :- !,
  elem(X,C).
elem(X,X).

sub_goals(X,not Y,S) :- !,
  xsetof(Y,holds(Y,S),Z),
  neg_conj_list(X,Z).
sub_goals(X,Y,S) :-
  xsetof(Y,holds(Y,S),Z),
  conj_list(X,Z).
```

```
% verificando e determinando efeitos de planos

chk(start) :- !.
chk(T => U) :-
  precond(U,C,T),
  effects(T,E),
  consistent(C,E),
  forall(elem(F,C),
      (not fact(F); (fact(F), holds(F,T)))),
  productive(U,T),
  chk(T).

sub_plan(S,P) :-
  ex1p(P,P1),
  append(S1,_,P1),
  one(ex1p(S,S1)).

part_plan(S,T) :-
  (not var(S), p_pl(S,T);
   var(S),
   op_plan(L, T => '$'),
   p_pl(S=>L,T=>'$')).

p_pl(O, T=>O) :-
  not O = _=>_ .
p_pl(S=>O, T=>O) :- !,
  p_pl1(S,T).
p_pl(S,T=>O) :-
  p_pl(S,T).

p_pl1(O, T=>O) :-
  not O = _=>_ .
p_pl1(S=>O,T=>O) :-
  p_pl1(S,T).

op_plan(O, T=>O).
op_plan(P, T=>O) :-
  op_plan(P,T).

effects(start,[]) :- !.
effects(T,P) :-
  xbagof(F, F1 ^(s_add1(F,T) ;
                 (s_del1(F1,T), F = (not F1))),
         C),
  conj_list(P,C).

net_effects(T,P) :-
  xbagof(F, F1 ^((s_add1(F,T), not holds(F,start)) ;
                 (s_del1(F1,T), holds(F1, start), F = (not F1))),
         L),
  conj_list(P,L).

s_added(T,S) :-
  xsetof(A, s_add1(A,T), S).

s_add1(X, T => V) :-
  added(X,V),
  operation(V).
s_add1(X, T => V) :- !,
  s_add1(X,T),
  not deleted(X,V).

s_deleted(T, S) :-
  xsetof(A,s_del1(A,T),S).
```

```
s_del1(X,T => V) :-
  deleted(X,V),
  operation(V).
s_del1(X, T=>V) :- !,
  s_del1(X,T),
  not added(X,V).


% execução de plano

exec_plan(T) :-
  plans(_,T),
  exec_plan1(T).

exec_plan1(T) :-
  ex1p(T,V),
  ex2p(V).

ex1p(start,[]).
ex1p(Ts => T,V) :- !,
  ex1p(Ts,Vs),
  append(Vs,[T],V).

ex2p([]) :- !.
ex2p([F|R]) :-
  forall(added(P,F), assert(P)),
  forall(deleted(P,F), retract(P)),
  ex2p(R).


% utilitários

on(X,[X|_]).
on(X,[Y|R]) :-
  on(X,R).

item(E,[E|R],1).
item(E,[X|R],J) :-
  var(J),
  item(E,R,I),
  J is I + 1 .
item(E,[X|R],J) :-
  not var(J),
  not J == 1,
  I is J - 1,
  item(E,R,I).

card([],0).
card([A|R],N) :-
  card(R,M),
  N is M + 1 .

append([],X,X).
append([X|R],Y,[X|S]) :-
  append(R,Y,S).

conc(Ls,S) :-
  conc1(Ls,Ls1),
  concat(Ls1,S).

conc1([],[]).
conc1([X|R],[Y|S]) :-
  string_term(Y,X),
```

```
  conc1(R,S).

reverse(Xs,Ys) :-
  rev1(Xs,[],Ys).

rev1([X|Xs],A,Ys) :-
  rev1(Xs,[X|A],Ys).
rev1([],Ys,Ys).

copy(X,Y) :-
  listvar(X,L1),
  cop1(L1,L),
  cop2(X,L,Y), !.

cop1([],[]).
cop1([V|R],[[V,W]|S]) :-
  cop1(R,S).

cop2(X,L,Y) :-
  case([atomic(X) -> Y = X,
        functor(X,F,N) -> (functor(Y,F,N), cop3(0,N,X,L,Y)),
        var(X) -> (cop4(X,X1,L), Y = X1)]).

cop3(N,N,X,L,Y) :- !.
cop3(I,N,X,L,Y) :-
  J is I + 1,
  arg(J,X,A),
  arg(J,Y,B),
  cop2(A,L,B),
  cop3(J,N,X,L,Y).

cop4(X,X1,[[Y,X1]|R]) :-
  X == Y, !.
cop4(X,X1,[P|R]) :-
  cop4(X,X1,R).

v_on(X,[Y|Z]) :-
  var(Y),!,
  (X == Y ;
   v_on(X,Z)).
v_on(X,[X|_]).
v_on(X,[Y|Z]) :-
  v_on(X,Z).

no_dup([X|Xs],Ys) :-
  v_on(X,Xs), !,
  no_dup(Xs,Ys).
no_dup([X|Xs],[X|Ys]) :-
  not v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([],[]).

v_flat(A,B,[A|B]) :-
  var(A), !.
v_flat(A,B,B) :-
  atomic(A), !.
v_flat([A|R],B,C) :- !,
  v_flat(R,B,T),
  v_flat(A,T,C).
v_flat(A,B,C) :-
  A =.. [F|R], !,
  v_flat(R,B,C).

listvar(X,Y) :-
```

```
  v_flat(X,[],T1),
  reverse(T1,T2),
  no_dup(T2,T3),
  reverse(T3,Y).

name_var(F) :-
  listvar(F,L),
  name_var1(L,1).

name_var(P,P1) :-
  copy(P,P1),
  name_var(P1).

name_var1([],I).
name_var1([X|R],I) :-
  J is I + 1,
  int_text(I,K),
  concat($V$,K,X),
  name_var1(R,J).

one(P) :- P, !.

forall(X,Y) :-
  not (X, not Y).

xsetof(X,Y,Z) :-
  (setof(X,Y,Z), !; Z = []).

xbagof(X,Y,Z) :-
  (bagof(X,Y,Z), !; Z = []).

neg_conj_list(true,[]) :- !.
neg_conj_list(not X,[X]) :- !.
neg_conj_list((not X , C),[X|Z]) :-
  neg_conj_list(C,Z).

conj_list(true,[]) :- !.
conj_list(X,[X]) :- not X =.. [',',_,_], !.
conj_list((X , C),[X|Z]) :-
  conj_list(C,Z).
```