UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
INSTITUTO DE COMPUTAÇÃO - ICOMP
DEPTO. DE ENGENHARIA DE SOFTWARE

MUNDO DOS BLOCOS VARIÁVEL EM MODEL CHECKING

FERNANDA SOUZA DE FREITAS

GABRIEL DA SILVA GLÓRIA

JULIO CESAR CERRATE CASTRO

LUANE DOS SANTOS LOPES

MUSKAAN RAMCHANDANI

PEDRO HENRIQUE BARROS MENDONÇA

RUAN COSTA DE MAGALHÃES

Manaus - AM

2025

FERNANDA SOUZA DE FREITAS

GABRIEL DA SILVA GLÓRIA

JULIO CESAR CERRATE CASTRO

LUANE DOS SANTOS LOPES

MUSKAAN RAMCHANDANI

PEDRO HENRIQUE BARROS MENDONÇA

RUAN COSTA DE MAGALHÃES

MUNDO DOS BLOCOS VARIÁVEL EM MODEL CHECKING

Relatório do 3º Trabalho Prático de Fundamentos de Inteligência Artificial-2025-2, como requisito para obtenção de nota parcial, ministrada pelo Prof.: Edjard Mota

Manaus - AM

2025

Sumário

1. Introdução

Este trabalho tem como objetivo modelar formalmente o clássico problema do Mundo dos Blocos (Blocks World), utilizando três abordagens complementares: (i) um modelo de transição de estados especificado em *NuSMV*, (ii) uma representação simbólica do domínio em lógica de predicados (*First-Order Logic*), e (iii) um sistema de regras auxiliares para verificação e planejamento.

O Mundo dos Blocos é um domínio amplamente utilizado em pesquisas de Inteligência Artificial e Raciocínio Automatizado, devido à sua simplicidade estrutural aliada à complexidade combinatória que envolve ações como mover blocos entre si, empilhar, desempilhar e gerenciar restrições de clareza, ocupação e estabilidade.

No primeiro módulo (main), especificado em *NuSMV*, modelamos o estado do mundo por meio de variáveis que representam a posição e o nível de cada bloco (position_X e level_X), bem como sua relação com outros blocos ou com a mesa (on_X). As ações permitidas (move) representam transições entre esses estados, sujeitas a condições como a clareza dos blocos, ausência de colisões e validade estrutural do empilhamento. O modelo inclui também invariantes (como valid_state) para garantir propriedades essenciais, como a não sobreposição de blocos no mesmo nível e a ausência de ciclos de suporte entre blocos. A propriedade temporal especificada em *CTL* (EF goal) busca garantir que um determinado estado objetivo é alcançável a partir do estado inicial.

A segunda parte do trabalho reimplementa o domínio em lógica de predicados, utilizando uma sintaxe inspirada em Prolog. São definidos predicados como on(X,Y,T), clear(X,T) e occupied(Pos,Level,T), bem como os efeitos das ações move em termos de adição e remoção de fatos. Essa abordagem permite raciocinar simbolicamente sobre as condições de aplicação das ações e sobre a evolução do sistema ao longo do tempo. Um planejador recursivo (planner/4) é construído com base nessas definições, permitindo gerar planos válidos que levam o sistema do estado inicial ao estado objetivo.

Por fim, a terceira parte do código define um conjunto de regras auxiliares que codificam as pré-condições das ações e os efeitos de cada move. Essas regras são utilizadas tanto na verificação da aplicabilidade das ações quanto na geração dos estados subsequentes.

A combinação dessas três abordagens permite uma análise completa e integrada do problema, conciliando técnicas de verificação formal (via model checking) com raciocínio simbólico e planejamento automático. Essa integração é particularmente útil para explorar a relação entre lógica temporal e lógica de predicados no contexto da Inteligência Artificial, além de fornecer uma base sólida para extensões futuras, como a introdução de blocos de tamanhos variáveis, mais agentes ou restrições complexas de domínio.

2. Exemplo mundo dos blocos.smv

Este código define um sistema com três blocos (a, b e c) e quatro mesas (table1 a table4), modelado no NuSMV com a sintaxe MODULE main. Cada bloco pode estar sobre uma mesa ou sobre outro bloco. Isso é representado por três variáveis: on_a, on_b e on_c, que assumem valores do tipo enumerado. O conjunto de valores possíveis para essas variáveis inclui todas as mesas e os dois outros blocos. Por exemplo, on_c = a significa que o bloco c está empilhado sobre o bloco a.

A variável move é a mais importante no que diz respeito ao controle do sistema. Ela representa uma ação de movimento entre posições seja um bloco indo para cima de outro, ou para cima de uma mesa. Ela também é um tipo enumerado, contendo 19 valores possíveis além de none (que significa "sem movimento"). Cada valor segue o padrão move_X_Y, que indica mover o bloco X para cima de Y. Por exemplo, move_b_table2 representa o bloco b sendo movido para cima da mesa table2. O valor de move determina a ação a ser executada em cada passo de verificação do modelo.

A seção DEFINE estabelece condições auxiliares muito importantes. As definições clear_a, clear_b e clear_c verificam se cada bloco está livre (ou seja, nenhum outro bloco está sobre ele). Isso é fundamental porque no Mundo dos Blocos, você só pode mover um bloco que esteja livre. Da mesma forma, clear_table1 até clear_table4 verificam se as mesas estão vazias, ou seja, se nenhum bloco está atualmente sobre elas. Essas condições são utilizadas nas transições para garantir que as ações de movimentação sejam válidas.

A condição goal define o estado desejado que o sistema deve alcançar. No caso, o objetivo é colocar o bloco a diretamente sobre a mesa table1, b sobre a, e c sobre b, formando uma torre perfeitamente ordenada do maior para o menor (ou da base ao topo).

A cláusula INIT define o estado inicial do sistema. Aqui, o bloco a começa sobre a table1, o bloco b está sobre a table3, e o bloco c está sobre o bloco a. Isso representa uma configuração desordenada, que não satisfaz a condição de goal, ou seja, o sistema começa fora do estado objetivo e o verificador precisa encontrar um caminho de ações que leve ao estado desejado.

As regras de transição são implementadas usando a cláusula TRANS. Existem três blocos de transições, uma para cada bloco (on_a, on_b, e on_c). Para cada uma dessas variáveis, o código avalia qual ação está sendo solicitada via move, e se as condições para realizá-la estão

presentes: o bloco em questão deve estar livre (clear_a, por exemplo), o destino também deve estar livre (clear_table1, clear_b, etc.), e o bloco não deve já estar na posição alvo (por exemplo, on_a != table1 evita movimentos redundantes). Caso todas as condições estejam satisfeitas, a variável on_X é atualizada para refletir a nova posição do bloco. Se nenhuma das condições for satisfeita, a variável mantém seu valor anterior (TRUE : on_a, por exemplo).

Uma transição extra impõe uma regra de integridade lógica: se algum movimento for especificado (move != none), pelo menos uma das variáveis on_a, on_b, ou on_c precisa mudar. Isso previne que o sistema execute ações inválidas que não causem qualquer alteração no estado. Em outras palavras, essa linha obriga que qualquer ação tenha efeito concreto.

Por fim, a propriedade CTLSPEC EF goal; usa lógica temporal CTL para expressar a seguinte pergunta: "Existe um caminho (E) tal que, em algum momento no futuro (F), o estado do sistema satisfaça a condição goal?". Isso é uma verificação típica em sistemas formais — queremos saber se é possível, partindo da configuração inicial, alcançar a configuração desejada seguindo as regras definidas no modelo.

Em resumo, este código NuSMV implementa um modelo simbólico do problema de movimentação de blocos, define todas as regras e condições de movimentação válidas, e usa verificação formal para determinar se o sistema pode atingir um objetivo predeterminado. É um exemplo típico de uso de model checking em problemas de planejamento, com grande utilidade tanto no ensino quanto na análise de sistemas automatizados.

```
MODULE main

VAR
  on_a : {table1, table2, table3, table4, b, c};
  on_b : {table1, table2, table3, table4, a, c};
  on_c : {table1, table2, table3, table4, a, b};
  move : {none,
        move_a_table1, move_a_table2, move_a_table3, move_a_table4, move_a_b,
move_a_c,
        move_b_table1, move_b_table2, move_b_table3, move_b_table4, move_b_a,
move_b_c,
        move_c_table1, move_c_table2, move_c_table3, move_c_table4, move_c_a,
move_c_b};

DEFINE
  clear_a := (on_b != a) & (on_c != a);
  clear_b := (on_a != b) & (on_c != b);
  clear_c := (on_a != c) & (on_b != c);
```

```
  clear_table1 := (on_a != table1) & (on_b != table1) & (on_c != table1);
  clear_table2 := (on_a != table2) & (on_b != table2) & (on_c != table2);
  clear_table3 := (on_a != table3) & (on_b != table3) & (on_c != table3);
  clear_table4 := (on_a != table4) & (on_b != table4) & (on_c != table4);
  goal := (on_a = table1) & (on_b = a) & (on_c = b);

INIT
  on_a = table1 &
  on_b = table3 &
  on_c = a;

-- Transições para on_a (padrão para outros blocos)
TRANS
  next(on_a) =
    case
      move = move_a_table1 & clear_a & clear_table1 & on_a != table1 : table1;
      move = move_a_table2 & clear_a & clear_table2 & on_a != table2 : table2;
      move = move_a_table3 & clear_a & clear_table3 & on_a != table3 : table3;
      move = move_a_table4 & clear_a & clear_table4 & on_a != table4 : table4;
      move = move_a_b & clear_a & clear_b & on_a != b : b;
      move = move_a_c & clear_a & clear_c & on_a != c : c;
      TRUE : on_a;
    esac;

-- Transições para on_b (similar)
TRANS
  next(on_b) =
    case
      move = move_b_table1 & clear_b & clear_table1 & on_b != table1 : table1;
      move = move_b_table2 & clear_b & clear_table2 & on_b != table2 : table2;
      move = move_b_table3 & clear_b & clear_table3 & on_b != table3 : table3;
      move = move_b_table4 & clear_b & clear_table4 & on_b != table4 : table4;
      move = move_b_a & clear_b & clear_a & on_b != a : a;
      move = move_b_c & clear_b & clear_c & on_b != c : c;
      TRUE : on_b;
    esac;

-- Transições para on_c (similar)
TRANS
  next(on_c) =
    case
      move = move_c_table1 & clear_c & clear_table1 & on_c != table1 : table1;
      move = move_c_table2 & clear_c & clear_table2 & on_c != table2 : table2;
      move = move_c_table3 & clear_c & clear_table3 & on_c != table3 : table3;
      move = move_c_table4 & clear_c & clear_table4 & on_c != table4 : table4;
      move = move_c_a & clear_c & clear_a & on_c != a : a;
      move = move_c_b & clear_c & clear_b & on_c != b : b;
      TRUE : on_c;
    esac;
```

```
-- Garante mudança em algum on se move != none
TRANS
  (move != none) -> (next(on_a) != on_a | next(on_b) != on_b | next(on_c) != on_c);

-- Propriedade: objetivo é alcançável?
CTLSPEC EF goal;
```

3. Mundo dos blocos em prolog

Esse código modela o clássico problema dos blocos onde você tem vários blocos e mesas, e deseja manipular a posição e a pilha desses blocos para alcançar um estado objetivo. Aqui, ao invés de usar uma linguagem de modelagem como NuSMV, o problema é descrito usando fatos atômicos e regras lógicas, típicos de linguagens como Prolog.

Começando pelos predicados básicos, temos fatos que descrevem a situação dos blocos: o predicado on(Bloco, Sobre) indica que um bloco está sobre outro bloco ou sobre uma mesa, por exemplo, on(a, table) significa que o bloco a está diretamente sobre a mesa. position(Bloco, Pos) define a posição horizontal do bloco na mesa, representada por um número entre 0 e 6. level(Bloco, Nivel) determina a altura ou nível do bloco na pilha o nível 0 significa estar diretamente sobre a mesa, níveis maiores indicam estar empilhado sobre outros blocos. O predicado clear(Bloco) indica que não há nenhum bloco em cima dele, ou seja, está livre para ser movido, e occupied(Pos) representa se uma posição da mesa está ocupada por algum bloco, levando em conta o tamanho (largura) dos blocos.

O estado do sistema é representado como uma lista de fatos cada fato é um predicado que define uma propriedade atual dos blocos (como posição, nível, se está sobre algo). A lista deve conter todas essas informações para todos os blocos envolvidos, e é verificada com estado(S) para garantir que é uma lista.

Um exemplo de estado inicial é definido com o predicado initial_state/1, onde são listados os fatos que descrevem a configuração inicial dos blocos. Nessa configuração, por exemplo, o bloco a está sobre a mesa na posição 2 e nível 0; b está sobre o bloco d, que por sua vez está sobre o bloco a. Os tamanhos fixos de cada bloco também são listados, o que é importante para calcular ocupação e estabilidade.

Para representar a regra de quando um bloco está livre (clear), o código usa a negação da existência de outro bloco que esteja sobre ele (\+ member(on(_, Bloco), S)). Ou seja, para que

um bloco esteja claro, não pode haver outro bloco acima dele. Para saber se uma posição na mesa está ocupada (occupied(Pos, S)), o código verifica se algum bloco está sobre a mesa e se a posição dada está dentro do intervalo coberto pelo bloco isso depende da posição horizontal e do tamanho do bloco.

O estado objetivo é definido por goal_state/1, onde são especificadas as posições e relações desejadas dos blocos para o final do plano, como por exemplo, a na posição 0 da mesa, c sobre a, d sobre c, e assim por diante.

As ações possíveis no sistema são definidas com o predicado action(move(Bloco, De, Para)), que modela o movimento de um bloco de uma posição ou bloco (De) para outro (Para), desde que De e Para sejam diferentes. Os blocos possíveis são listados explicitamente, e a função pos_ou_bloco aceita tanto posições na mesa (de 0 a 6) quanto blocos para mover sobre eles.

Antes de executar uma ação, o sistema verifica as pré-condições para garantir que a movimentação é válida. O predicado can(move(Bloco, De, Para), S) checa se o bloco está no local correto, está livre para mover, se o destino está livre (sem ocupar o espaço necessário), se há espaço horizontal suficiente para acomodar o bloco na mesa ou sobre outro bloco, e se a pilha será estável após o movimento.

O conceito de estabilidade é especialmente importante: um bloco só pode ser colocado sobre outro bloco se for menor ou igual em tamanho, e se a posição for tal que o centro de massa do bloco empilhado fique alinhado corretamente para evitar quedas. Esse cálculo é feito usando a diferença de tamanhos e um offset para centralizar o bloco sobre o debaixo.

Quando uma ação é aplicada, ela remove fatos antigos (deletes) relacionados ao estado anterior do bloco e adiciona fatos novos (adds) que refletem a nova posição, nível e clareza do bloco, calculando também a nova posição horizontal e o nível da pilha.

O código ainda implementa uma verificação de estado válido (valid_state) que garante que não existam colisões de blocos na mesma posição e nível, que não haja ciclos (exemplo: bloco A sobre B e B sobre A simultaneamente), e que as posições e níveis estejam dentro dos limites permitidos.

Finalmente, o planejador recursivo, implementado no predicado plano, busca encontrar uma sequência de ações (Plano) que transforme o estado inicial no estado objetivo. Ele faz isso

verificando se o estado corrente satisfaz o objetivo, e caso contrário, tenta aplicar uma ação válida que leva a um novo estado válido, chamando recursivamente até encontrar uma solução.

Como um todo, esse código exemplifica uma forma declarativa de modelar e resolver o problema dos blocos, usando predicados para descrever estados e regras, e um motor lógico para planejar sequências de ações. Diferente do modelo NuSMV, que trabalha com transições de estados explicitamente e verifica propriedades temporais, aqui o foco está na geração do plano de ações por meio de regras lógicas e busca recursiva. É uma abordagem que reflete bem o uso de lógica para representação do conhecimento e planejamento em inteligência artificial.

Análise do Código SMV em Lógica de Primeira Ordem Usando Prolog

O código SMV modela um mundo dos blocos com tamanhos variáveis (a, b, c tamanho 1; d tamanho 2), em uma grade de posições 0 a 6 na mesa, com restrições de vacância, estabilidade (centro de massa), níveis de pilha e movimentos. Em FOL/Prolog, isso pode ser representado como um sistema de planejamento baseado em STRIPS (como no Capítulo 17 de Bratko), onde:

- Estados são conjuntos de fatos atômicos (predicados como on/2, position/2, level/2, clear/1, occupied/1).
- Ações são transições (move/3), com pré-condições (can/2), adições (adds/2) e deleções (deletes/2).
- Invariantes (INVAR) são axiomas que devem ser verdadeiros em todos os estados.
- Objetivo (goal) é um conjunto de fatos a serem satisfeitos.
- Verificação (CTLSPEC EF goal) corresponde a uma busca para encontrar um plano que alcance o objetivo (usando recursão em Prolog para simular busca em profundidade ou largura).

A análise foca em correção lógica, completude e restrições físicas (vacância horizontal/vertical, estabilidade).

Representação de Estados (VAR e DEFINE em SMV → Predicados em Prolog)

No SMV, variáveis como on_a, position_a, level_a representam o estado de cada bloco. Definições como clear_a e occupied_table_0 são fórmulas computadas.
Em Prolog (FOL), estados são listas de fatos [Fact1, Fact2, ...], onde:

- Predicados: on(Bloco, Sobre) (Sobre pode ser table ou outro bloco), position(Bloco, Pos), level(Bloco, Nível), clear(Bloco), occupied(Pos), size(Bloco, Tamanho).
- Justificativa: Isso estende a representação abstrata de Bratko (apenas on/2 e clear/1) para incluir posições espaciais (grade 0..6) e níveis (para estabilidade vertical), como exigido no Ponto 1 do assignment (comparando com Fig. 17.1: lugares abstratos viram posições numéricas; blocos ganham tamanho e centro de massa para vacância/estabilidade).

Código Prolog para representação:

```
% Predicados para fatos atômicos (FOL: ∀ Bloco, Pos, etc.) on(Bloco, Sobre).  % Ex.: on(a, table) ou on(b, a) position(Bloco, Pos).  % Pos in 0..6 level(Bloco, Nivel).  % Nivel in 0..10 clear(Bloco).  % Nada sobre o Bloco
occupied(Pos).  % Pos ocupada por algum bloco na mesa size(Bloco, Tam).  % Tam fixo: size(a,1), etc.

% Estado é uma lista de fatos estado(S) :- is_list(S).

% Exemplo de estado inicial (do INIT no SMV)
initial_state([on(a, table), position(a, 2), level(a, 0),  on(b, d), position(b, 3), level(b, 2),  on(c, table), position(c, 0), level(c, 0),  on(d, a), position(d, 2), level(d, 1),  size(a,1), size(b,1), size(c,1), size(d,2)]).

% Definições computadas (como DEFINE no SMV)
clear(Bloco, S) :- \+ member(on(_, Bloco), S).  % FOL: ¬∃ X on(X, Bloco)

occupied(Pos, S) :-  member(on(Bloco, table), S),  member(position(Bloco, BPos), S), member(size(Bloco, Tam), S),
   BPos =< Pos, Pos =< BPos + Tam - 1.

% Objetivo (goal no SMV) goal_state([on(a, table), position(a, 0),  on(c, a), position(c, 0),  on(d, c), position(d, 0),  on(b, d), position(b, 1)]).
```

Essa representação é completa para FOL, pois quantificadores (∀/∃) são implícitos em queries Prolog (ex.: clear(a, S) usa negação como falha para ¬∃). Comparado a Bratko: Foi adicionado position/2 e occupied/1 para vacância horizontal (grade onde menor bloco cabe em 1 espaço, como na dica do assignment); level/2 para vacância vertical e estabilidade (centro de massa calculado em ações).

Ações e Transições (TRANS e ASSIGN em SMV → can/adds/deletes em Prolog)

No SMV, move é uma variável enum com guards em case para pré-condições; next(...) atualiza estados.

Em Prolog, ações são move(Bloco, De, Para), onde De/Para pode ser table_Pos ou bloco. Use can(Acao, Estado) para pré-condições (clear origem/destino, vacância, estabilidade); adds/deletes para efeitos; apply(Acao, Estado, NovoEstado) para transição.
Justificativa: Foi modificado o planner de Fig. 17.6 (Ponto 2 do assignment): foi adicionado variáveis para goals (subset check) e ações (incluindo table_Pos para grade, estabilidade via centro de massa). Explicação da mudança: Em Bratko, can(move(X,Y,Z), [on(X,Y), clear(X), clear(Z)]); aqui, foi colado checks espaciais (vacant_horizontal/3, stable/3).

Código Prolog para ações:

```
% Ações possíveis (enum move no SMV)
action(move(Bloco, De, Para)) :- bloco(Bloco), pos_ou_bloco(De), pos_ou_bloco(Para), De \= Para.

bloco(B) :- member(B, [a,b,c,d]). pos_ou_bloco(table_Pos) :- between(0,6,Pos); bloco(Pos).

% Pré-condições: can(Acao, Estado) - com vacância e estabilidade can(move(Bloco, De, Para), S) :- member(on(Bloco, De), S), clear(Bloco, S), clear_destino(Para, S), vacant_horizontal(Bloco, Para, S),
  stable(Bloco, Para, S).

clear_destino(table_Pos, S) :- \+ occupied(Pos, S).  % Para table_Pos
clear_destino(DestBloco, S) :- clear(DestBloco, S).  % Para outro bloco

vacant_horizontal(Bloco, Para, S) :- member(size(Bloco, Tam), S),
  (Para = table_Pos ->  % Move para mesa
     forall(between(0, Tam-1, Offset), \+ occupied(Pos + Offset, S))
  ; Para = DestBloco ->  % Move para bloco (vacância horizontal implícita via position)
member(position(DestBloco, DPos), S), member(size(DestBloco, DTam), S), member(position(Bloco, BPos), S),
     BPos >= DPos, BPos + Tam -1 =< DPos + DTam -1
  ).

stable(Bloco, Para, S) :-
```

```
   Para = DestBloco,  % Só para pilhas     member(size(Bloco, BTam), S),
member(size(DestBloco, DTam), S),
   BTam =< DTam,  % Bloco menor ou igual sobre maior     member(position(DestBloco,
DPos), S),
   Delta = DTam - BTam,
   Offset = case(Delta >= 0, Delta / 2, (Delta -1) / 2),  % Centro de massa (como no SMV)
   Offset >= 0, Offset + BTam -1 =< DTam -1.  % Estável se centro alinhado

% Adições e Deleções (adds/deletes no SMV via next)
adds(move(Bloco, De, Para), [on(Bloco, Para), clear(De), position(Bloco, NewPos),
level(Bloco, NewLevel)]) :-
   % Calcula NewPos e NewLevel baseado em Para (similar a next no SMV)
   (Para = table_Pos -> NewPos = Pos, NewLevel = 0
   ; Para = DestBloco -> member(position(DestBloco, DPos), S), NewPos = DPos + Offset,
member(level(DestBloco, DLevel), S), NewLevel = DLevel +1).

deletes(move(Bloco, De,        Para), [on(Bloco, De), clear(Para), position(Bloco, OldPos),
level(Bloco, OldLevel)]).

% Aplica ação: apply(Acao, Estado, NovoEstado) apply(A, S, S1) :- deletes(A, D),
subtract(S, D, S2), adds(A, A1), append(S2, A1, S1).
```

Em FOL, pré-condições são conjunções (& no SMV vira , em Prolog). A mudança para variáveis em goals (sessão 17.5 de Bratko) é manejada por subset(Goal, S) no planejador abaixo. Estabilidade é nova restrição: FOL ¬(size(Bloco) > size(Dest) ∧ ¬alinhado_centro), evitando instabilidade.

Invariantes e Restrições (INVAR em SMV → Axiomas em Prolog)

No SMV, INVAR previne colisões em níveis/posições e ciclos em pilhas. Em Prolog, invariantes são checados em estados válidos (valid_state(S)), usando negação para ¬.

Código Prolog:

```
% Invariante: Sem colisões em mesmo nível e posição sobreposta valid_state(S) :-
forall((bloco(B1), bloco(B2), B1 \= B2),
        (member(level(B1, L), S), member(level(B2, L), S) ->            \+
(member(position(B1, P1), S), member(size(B1, S1), S),                member(position(B2, P2),
S), member(size(B2, S2), S),
            P1 =< P2 + S2 -1, P2 =< P1 + S1 -1))),
   % Sem ciclos (exemplos do INVAR)
   \+ (member(on(a,b), S), member(on(b,a), S)),
   \+ (member(on(a,c), S), member(on(c,a), S)),
   % ... (todos os ! ciclos do SMV)
   % Limites de posição e nível
```

forall(bloco(B), (member(position(B, P), S), member(size(B, Tam), S), P >= 0, P + Tam -1 =< 6)), forall(bloco(B), (member(level(B, N), S), N =< 10)).
Invariantes são fórmulas FOL universais (∀ B1,B2 ¬colisão(L,P)), garantindo consistência. Sem eles, planos poderiam violar física (ex.: bloco maior sobre menor sem estabilidade).

Planejador e Verificação (CTLSPEC em SMV → plano/3 em Prolog)

No SMV, EF goal verifica existência de plano via model checking. Em Prolog, plano/3 gera o plano recursivamente (como no diagrama da página 2: base se goal satisfeito, recursivo via ação aplicável).

Código Prolog completo para planejador:

% Planejador recursivo (plano(Estado, Goal, Plano)) plano(S, Goal, []) :- subset(Goal, S), valid_state(S). % Base: goal satisfeito e estado válido plano(S, Goal, [A|Plan]) :- action(A), can(A, S), apply(A, S, S1), valid_state(S1), % Checa invariantes no novo estado plano(S1, Goal, Plan).

% Consulta exemplo: encontre plano do inicial ao goal consulta_plan :- initial_state(Init), goal_state(Goal), plano(Init, Goal, Plan), write(Plan), nl.

Isso é a formulação recursiva do diagrama (plano de S1 a Sgoal via ações A1..Ai). Em FOL, é equivalente a ∃ Plano ∀ i (can(Ai, Si) ∧ apply(Ai, Si, Si+1) ∧ subset(Goal, Sfinal)). Para o SMV, EF goal é true se existe tal Plano.

Conclusão Geral da Análise

O código SMV é uma representação precisa do mundo dos blocos com restrições físicas, mas em Prolog/FOL, torna-se mais declarativo e fácil para planejamento regressivo.

```
% Predicados para fatos atômicos (FOL: ∀ Bloco, Pos, etc.)
on(Bloco, Sobre).  % Ex.: on(a, table) ou on(b, a)
position(Bloco, Pos).  % Pos in 0..6
level(Bloco, Nivel).  % Nivel in 0..10
clear(Bloco).  % Nada sobre o Bloco
occupied(Pos).  % Pos ocupada por algum bloco na mesa
size(Bloco, Tam).  % Tam fixo: size(a,1), etc.

% Estado é uma lista de fatos
estado(S) :- is_list(S).

% Exemplo de estado inicial (do INIT no SMV)
initial_state([on(a, table), position(a, 2), level(a, 0),
            on(b, d), position(b, 3), level(b, 2),
            on(c, table), position(c, 0), level(c, 0),
            on(d, a), position(d, 2), level(d, 1),
            size(a,1), size(b,1), size(c,1), size(d,2)]).

% Definições computadas (como DEFINE no SMV)
clear(Bloco, S) :- \+ member(on(_, Bloco), S).  % FOL: ¬∃ X on(X, Bloco)

occupied(Pos, S) :-
   member(on(Bloco, table), S),
   member(position(Bloco, BPos), S),
   member(size(Bloco, Tam), S),
   BPos =< Pos, Pos =< BPos + Tam - 1.

% Objetivo (goal no SMV)
goal_state([on(a, table), position(a, 0),
          on(c, a), position(c, 0),
          on(d, c), position(d, 0),
          on(b, d), position(b, 1)]).

% Ações possíveis (enum move no SMV)
action(move(Bloco, De, Para)) :- bloco(Bloco), pos_ou_bloco(De), pos_ou_bloco(Para),
De \= Para.

bloco(B) :- member(B, [a,b,c,d]).
pos_ou_bloco(table_Pos) :- between(0,6,Pos); bloco(Pos).

% Pré-condições: can(Acao, Estado) - com vacância e estabilidade
can(move(Bloco, De, Para), S) :-
```

```
    member(on(Bloco, De), S),
    clear(Bloco, S),
    clear_destino(Para, S),
    vacant_horizontal(Bloco, Para, S),
    stable(Bloco, Para, S).

clear_destino(table_Pos, S) :- \+ occupied(Pos, S).  % Para table_Pos
clear_destino(DestBloco, S) :- clear(DestBloco, S).  % Para outro bloco

vacant_horizontal(Bloco, Para, S) :-
    member(size(Bloco, Tam), S),
    (Para = table_Pos ->  % Move para mesa
        forall(between(0, Tam-1, Offset), \+ occupied(Pos + Offset, S))
    ; Para = DestBloco ->  % Move para bloco (vacância horizontal implícita via position)
        member(position(DestBloco, DPos), S),
        member(size(DestBloco, DTam), S),
        member(position(Bloco, BPos), S),
        BPos >= DPos, BPos + Tam -1 =< DPos + DTam -1
    ).

stable(Bloco, Para, S) :-
    Para = DestBloco,  % Só para pilhas
    member(size(Bloco, BTam), S),
    member(size(DestBloco, DTam), S),
    BTam =< DTam,  % Bloco menor ou igual sobre maior
    member(position(DestBloco, DPos), S),
    Delta = DTam - BTam,
    Offset = case(Delta >= 0, Delta / 2, (Delta -1) / 2),  % Centro de massa (como no SMV)
    Offset >= 0, Offset + BTam -1 =< DTam -1.  % Estável se centro alinhado

% Adições e Deleções (adds/deletes no SMV via next)
adds(move(Bloco, De, Para), [on(Bloco, Para), clear(De), position(Bloco, NewPos),
level(Bloco, NewLevel)]) :-
    % Calcula NewPos e NewLevel baseado em Para (similar a next no SMV)
    (Para = table_Pos -> NewPos = Pos, NewLevel = 0
    ; Para = DestBloco -> member(position(DestBloco, DPos), S), NewPos = DPos + Offset,
member(level(DestBloco, DLevel), S), NewLevel = DLevel +1).

deletes(move(Bloco, De, Para), [on(Bloco, De), clear(Para), position(Bloco, OldPos),
level(Bloco, OldLevel)]).

% Aplica ação: apply(Acao, Estado, NovoEstado)
apply(A, S, S1) :- deletes(A, D), subtract(S, D, S2), adds(A, A1), append(S2, A1, S1).

% Invariante: Sem colisões em mesmo nível e posição sobreposta
valid_state(S) :-
    forall((bloco(B1), bloco(B2), B1 \= B2),
          (member(level(B1, L), S), member(level(B2, L), S) ->
           \+ (member(position(B1, P1), S), member(size(B1, S1), S),
               member(position(B2, P2), S), member(size(B2, S2), S),
```

```
            P1 =< P2 + S2 -1, P2 =< P1 + S1 -1))),
    % Sem ciclos (exemplos do INVAR)
    \+ (member(on(a,b), S), member(on(b,a), S)),
    \+ (member(on(a,c), S), member(on(c,a), S)),
    % ... (todos os ! ciclos do SMV)
    % Limites de posição e nível
    forall(bloco(B), (member(position(B, P), S), member(size(B, Tam), S), P >= 0, P + Tam
-1 =< 6)),
    forall(bloco(B), (member(level(B, N), S), N =< 10)).

% Planejador recursivo (plano(Estado, Goal, Plano))
plano(S, Goal, []) :- subset(Goal, S), valid_state(S).  % Base: goal satisfeito e estado válido

plano(S, Goal, [A|Plan]) :-
    action(A),
    can(A, S),
    apply(A, S, S1),
    valid_state(S1),  % Checa invariantes no novo estado
    plano(S1, Goal, Plan).

% Consulta exemplo: encontre plano do inicial ao goal
consulta_plan :- initial_state(Init), goal_state(Goal), plano(Init, Goal, Plan), write(Plan), nl.
```

4. Planning as model checking.svm

Este código é um modelo formal escrito na linguagem SMV (Symbolic Model Verifier), que descreve uma versão do clássico problema de inteligência artificial conhecido como "Mundo dos Blocos". Sua finalidade não é executar uma solução diretamente, mas sim criar uma representação precisa de um ambiente com suas regras, um estado inicial e um objetivo. A partir deste modelo, uma ferramenta de software chamada verificador de modelos pode explorar automaticamente todas as sequências de movimentos possíveis para determinar se o objetivo pode ser alcançado e, em caso afirmativo, como. O cenário específico envolve quatro blocos (a, b, c, d) com tamanhos variados e uma mesa com sete posições distintas.

O estado do sistema em qualquer momento é definido por um conjunto de variáveis declaradas na seção VAR. As variáveis on_a, on_b, on_c e on_d rastreiam o que está imediatamente abaixo de cada bloco, que pode ser a mesa (table) ou outro bloco. As variáveis position indicam a coordenada horizontal de cada bloco, enquanto level representa sua altura em uma pilha. A variável move atua como a entrada do sistema, representando a ação que pode ser tentada a cada passo, como mover um bloco para a mesa ou para cima de outro. Para simplificar a lógica, a seção DEFINE cria apelidos para condições complexas. Isso inclui

definir os tamanhos dos blocos (onde d é maior que os outros), determinar se um bloco está "livre" (clear) para ser movido, e verificar se uma posição na mesa está ocupada. Mais importante, é aqui que a condição de goal (a configuração final desejada) é precisamente especificada. O ponto de partida para a verificação é estabelecido na seção INIT, que descreve a configuração exata de todos os blocos no início do problema.

As regras que governam o ambiente são definidas nas seções INVAR e TRANS. A seção INVAR estabelece as leis físicas do modelo — condições que devem ser verdadeiras em todos os momentos. Essas invariantes garantem que os blocos não se sobreponham, permaneçam dentro dos limites da mesa e não formem pilhas cíclicas impossíveis (como a sobre b e b sobre a). A seção TRANS é o núcleo do modelo, detalhando como o estado do sistema evolui. Para cada ação de move possível, ela define como os valores das variáveis on e position mudarão. Um movimento só é permitido se suas pré-condições forem atendidas, como o bloco a ser movido estar livre e seu destino estar desocupado. O modelo inclui até mesmo a lógica para centralizar um bloco menor sobre um maior. As variáveis de level (altura) são então atualizadas na seção ASSIGN, calculando a nova altura de cada bloco com base em sua posição na pilha.

Finalmente, a linha CTLSPEC EF goal instrui o verificador de modelos sobre o que provar. Esta é uma especificação em Lógica de Árvore Computacional (CTL) que se traduz na pergunta: "Existe (E) algum caminho de execução possível onde, em algum ponto no futuro (F), o estado do sistema satisfaça a condição goal?". Em vez de testar um caminho, o verificador explora sistematicamente a árvore de todos os estados alcançáveis a partir da configuração inicial. O resultado será a confirmação de que o objetivo é alcançável (geralmente acompanhado pela sequência de movimentos para chegar lá) ou a prova de que é impossível atingir a meta a partir do estado inicial, dadas as regras do modelo.

```
MODULE main

VAR
  on_a : {table, b, c, d};
  on_b : {table, a, c, d};
  on_c : {table, a, b, d};
  on_d : {table, a, b, c};
  position_a : 0..6;
  position_b : 0..6;
  position_c : 0..6;
```

```
  position_d : 0..6;
  level_a : 0..10;
  level_b : 0..10;
  level_c : 0..10;
  level_d : 0..10;
  move : {none,
        move_a_to_table_0, move_a_to_table_1, move_a_to_table_2, move_a_to_table_3,
move_a_to_table_4, move_a_to_table_5, move_a_to_table_6,
        move_a_to_b, move_a_to_c, move_a_to_d,
        move_b_to_table_0, move_b_to_table_1, move_b_to_table_2, move_b_to_table_3,
move_b_to_table_4, move_b_to_table_5, move_b_to_table_6,
        move_b_to_a, move_b_to_c, move_b_to_d,
        move_c_to_table_0, move_c_to_table_1, move_c_to_table_2, move_c_to_table_3,
move_c_to_table_4, move_c_to_table_5, move_c_to_table_6,
        move_c_to_a, move_c_to_b, move_c_to_d,
        move_d_to_table_0, move_d_to_table_1, move_d_to_table_2, move_d_to_table_3,
move_d_to_table_4, move_d_to_table_5, move_d_to_table_6,
        move_d_to_a, move_d_to_b, move_d_to_c};

DEFINE
  size_a := 1;
  size_b := 1;
  size_c := 1;
  size_d := 2;
  clear_a := (on_b != a) & (on_c != a) & (on_d != a);
  clear_b := (on_a != b) & (on_c != b) & (on_d != b);
  clear_c := (on_a != c) & (on_b != c) & (on_d != c);
  clear_d := (on_a != d) & (on_b != d) & (on_c != d);
  occupied_table_0 := (on_a = table & position_a <= 0 & 0 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 0 & 0 <= position_b + size_b - 1) |
                      (on_c = table & position_c <= 0 & 0 <= position_c + size_c - 1) |
                      (on_d = table & position_d <= 0 & 0 <= position_d + size_d - 1);
  occupied_table_1 := (on_a = table & position_a <= 1 & 1 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 1 & 1 <= position_b + size_b - 1) |
                      (on_c = table & position_c <= 1 & 1 <= position_c + size_c - 1) |
                      (on_d = table & position_d <= 1 & 1 <= position_d + size_d - 1);
  occupied_table_2 := (on_a = table & position_a <= 2 & 2 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 2 & 2 <= position_b + size_b - 1) |
                      (on_c = table & position_c <= 2 & 2 <= position_c + size_c - 1) |
                      (on_d = table & position_d <= 2 & 2 <= position_d + size_d - 1);
  occupied_table_3 := (on_a = table & position_a <= 3 & 3 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 3 & 3 <= position_b + size_b - 1) |
                      (on_c = table & position_c <= 3 & 3 <= position_c + size_c - 1) |
                      (on_d = table & position_d <= 3 & 3 <= position_d + size_d - 1);
  occupied_table_4 := (on_a = table & position_a <= 4 & 4 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 4 & 4 <= position_b + size_b - 1) |
                      (on_c = table & position_c <= 4 & 4 <= position_c + size_c - 1) |
                      (on_d = table & position_d <= 4 & 4 <= position_d + size_d - 1);
  occupied_table_5 := (on_a = table & position_a <= 5 & 5 <= position_a + size_a - 1) |
                      (on_b = table & position_b <= 5 & 5 <= position_b + size_b - 1) |
```

```
                (on_c = table & position_c <= 5 & 5 <= position_c + size_c - 1) |
                (on_d = table & position_d <= 5 & 5 <= position_d + size_d - 1);
  occupied_table_6 := (on_a = table & position_a <= 6 & 6 <= position_a + size_a - 1) |
                (on_b = table & position_b <= 6 & 6 <= position_b + size_b - 1) |
                (on_c = table & position_c <= 6 & 6 <= position_c + size_c - 1) |
                (on_d = table & position_d <= 6 & 6 <= position_d + size_d - 1);
  goal := (on_a = table & position_a = 0) & (on_c = a & position_c = 0) & (on_d = c &
position_d = 0) & (on_b = d & position_b = 1); -- Adjust as needed for specific goal

INIT
  on_a = table & position_a = 2 &
  on_b = d & position_b = 3 &
  on_c = table & position_c = 0 &
  on_d = a & position_d = 2 &
  level_a = 0 &
  level_b = 2 &
  level_c = 0 &
  level_d = 1;

INVAR
  !(level_a = level_b & position_a <= position_b + size_b - 1 & position_b <= position_a +
size_a - 1) &
  !(level_a = level_c & position_a <= position_c + size_c - 1 & position_c <= position_a +
size_a - 1) &
  !(level_a = level_d & position_a <= position_d + size_d - 1 & position_d <= position_a +
size_a - 1) &
  !(level_b = level_c & position_b <= position_c + size_c - 1 & position_c <= position_b +
size_b - 1) &
  !(level_b = level_d & position_b <= position_d + size_d - 1 & position_d <= position_b +
size_b - 1) &
  !(level_c = level_d & position_c <= position_d + size_d - 1 & position_d <= position_c +
size_c - 1) &
  position_a + size_a - 1 <= 6 & position_a >= 0 &
  position_b + size_b - 1 <= 6 & position_b >= 0 &
  position_c + size_c - 1 <= 6 & position_c >= 0 &
  position_d + size_d - 1 <= 6 & position_d >= 0 &
  -- Prevent cycles
  !(on_a = b & on_b = a) &
  !(on_a = c & on_c = a) &
  !(on_a = d & on_d = a) &
  !(on_b = c & on_c = b) &
  !(on_b = d & on_d = b) &
  !(on_c = d & on_d = c) &
  !(on_a = b & on_b = c & on_c = a) &
  !(on_a = b & on_b = d & on_d = a) &
  !(on_a = c & on_c = d & on_d = a) &
  !(on_b = c & on_c = d & on_d = b) &
  !(on_a = b & on_b = c & on_c = d & on_d = a) &
  !(on_a = b & on_b = d & on_d = c & on_c = a) &
  !(on_a = c & on_c = b & on_b = d & on_d = a) &
```

```
  !(on_a = c & on_c = d & on_d = b & on_b = a) &
  !(on_a = d & on_d = b & on_b = c & on_c = a) &
  !(on_a = d & on_d = c & on_c = b & on_b = a);

TRANS
  next(on_a) =
    case
      move = move_a_to_table_0 & clear_a & !(on_a = table & position_a = 0) &
!occupied_table_0 : table;
      move = move_a_to_table_1 & clear_a & !(on_a = table & position_a = 1) &
!occupied_table_1 : table;
      move = move_a_to_table_2 & clear_a & !(on_a = table & position_a = 2) &
!occupied_table_2 : table;
      move = move_a_to_table_3 & clear_a & !(on_a = table & position_a = 3) &
!occupied_table_3 : table;
      move = move_a_to_table_4 & clear_a & !(on_a = table & position_a = 4) &
!occupied_table_4 : table;
      move = move_a_to_table_5 & clear_a & !(on_a = table & position_a = 5) &
!occupied_table_5 : table;
      move = move_a_to_table_6 & clear_a & !(on_a = table & position_a = 6) &
!occupied_table_6 : table;
      move = move_a_to_b & clear_a & clear_b & on_a != b & (position_b + case (size_b -
size_a) >= 0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_a) >= 0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_b | ((position_b + case (size_b - size_a) >=
0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_b & (position_b + case (size_b - size_a) >= 0 : (size_b - size_a) / 2 ; TRUE :
((size_b - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_b + size_b - 1)) : b;
      move = move_a_to_c & clear_a & clear_c & on_a != c & (position_c + case (size_c -
size_a) >= 0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_a) >= 0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_c | ((position_c + case (size_c - size_a) >=
0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_c & (position_c + case (size_c - size_a) >= 0 : (size_c - size_a) / 2 ; TRUE :
((size_c - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_c + size_c - 1)) : c;
      move = move_a_to_d & clear_a & clear_d & on_a != d & (position_d + case (size_d -
size_a) >= 0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_a) >= 0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_d | ((position_d + case (size_d - size_a) >=
0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_d & (position_d + case (size_d - size_a) >= 0 : (size_d - size_a) / 2 ; TRUE :
((size_d - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_d + size_d - 1)) : d;
      TRUE : on_a;
    esac;

TRANS
  next(position_a) =
    case
      move = move_a_to_table_0 & clear_a & !(on_a = table & position_a = 0) &
!occupied_table_0 : 0;
```

```
    move = move_a_to_table_1 & clear_a & !(on_a = table & position_a = 1) &
!occupied_table_1 : 1;
    move = move_a_to_table_2 & clear_a & !(on_a = table & position_a = 2) &
!occupied_table_2 : 2;
    move = move_a_to_table_3 & clear_a & !(on_a = table & position_a = 3) &
!occupied_table_3 : 3;
    move = move_a_to_table_4 & clear_a & !(on_a = table & position_a = 4) &
!occupied_table_4 : 4;
    move = move_a_to_table_5 & clear_a & !(on_a = table & position_a = 5) &
!occupied_table_5 : 5;
    move = move_a_to_table_6 & clear_a & !(on_a = table & position_a = 6) &
!occupied_table_6 : 6;
    move = move_a_to_b & clear_a & clear_b & on_a != b & (position_b + case (size_b -
size_a) >= 0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_a) >= 0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_b | ((position_b + case (size_b - size_a) >=
0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_b & (position_b + case (size_b - size_a) >= 0 : (size_b - size_a) / 2 ; TRUE :
((size_b - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_b + size_b - 1)) : position_b +
case (size_b - size_a) >= 0 : (size_b - size_a) / 2 ; TRUE : ((size_b - size_a) - 1) / 2 ; esac;
    move = move_a_to_c & clear_a & clear_c & on_a != c & (position_c + case (size_c -
size_a) >= 0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_a) >= 0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_c | ((position_c + case (size_c - size_a) >=
0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_c & (position_c + case (size_c - size_a) >= 0 : (size_c - size_a) / 2 ; TRUE :
((size_c - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_c + size_c - 1)) : position_c +
case (size_c - size_a) >= 0 : (size_c - size_a) / 2 ; TRUE : ((size_c - size_a) - 1) / 2 ; esac;
    move = move_a_to_d & clear_a & clear_d & on_a != d & (position_d + case (size_d -
size_a) >= 0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_a) >= 0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) -
1) / 2 ; esac) + size_a - 1 <= 6 & (size_a <= size_d | ((position_d + case (size_d - size_a) >=
0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 >=
position_d & (position_d + case (size_d - size_a) >= 0 : (size_d - size_a) / 2 ; TRUE :
((size_d - size_a) - 1) / 2 ; esac) + (size_a - 1) / 2 <= position_d + size_d - 1)) : position_d +
case (size_d - size_a) >= 0 : (size_d - size_a) / 2 ; TRUE : ((size_d - size_a) - 1) / 2 ; esac;
    TRUE : position_a;
  esac;

TRANS
 next(on_b) =
  case
    move = move_b_to_table_0 & clear_b & !(on_b = table & position_b = 0) &
!occupied_table_0 : table;
    move = move_b_to_table_1 & clear_b & !(on_b = table & position_b = 1) &
!occupied_table_1 : table;
    move = move_b_to_table_2 & clear_b & !(on_b = table & position_b = 2) &
!occupied_table_2 : table;
    move = move_b_to_table_3 & clear_b & !(on_b = table & position_b = 3) &
!occupied_table_3 : table;
```

```
    move = move_b_to_table_4 & clear_b & !(on_b = table & position_b = 4) &
!occupied_table_4 : table;
    move = move_b_to_table_5 & clear_b & !(on_b = table & position_b = 5) &
!occupied_table_5 : table;
    move = move_b_to_table_6 & clear_b & !(on_b = table & position_b = 6) &
!occupied_table_6 : table;
    move = move_b_to_a & clear_b & clear_a & on_b != a & (position_a + case (size_a -
size_b) >= 0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_b) >= 0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) -
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_a | ((position_a + case (size_a - size_b) >=
0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_a & (position_a + case (size_a - size_b) >= 0 : (size_a - size_b) / 2 ; TRUE :
((size_a - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_a + size_a - 1)) : a;
    move = move_b_to_c & clear_b & clear_c & on_b != c & (position_c + case (size_c -
size_b) >= 0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_b) >= 0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) -
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_c | ((position_c + case (size_c - size_b) >=
0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_c & (position_c + case (size_c - size_b) >= 0 : (size_c - size_b) / 2 ; TRUE :
((size_c - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_c + size_c - 1)) : c;
    move = move_b_to_d & clear_b & clear_d & on_b != d & (position_d + case (size_d -
size_b) >= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_b) >= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) -
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_d | ((position_d + case (size_d - size_b)
>= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_d & (position_d + case (size_d - size_b) >= 0 : (size_d - size_b) / 2 ; TRUE :
((size_d - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_d + size_d - 1)) : d;
    TRUE : on_b;
  esac;

TRANS
 next(position_b) =
  case
    move = move_b_to_table_0 & clear_b & !(on_b = table & position_b = 0) &
!occupied_table_0 : 0;
    move = move_b_to_table_1 & clear_b & !(on_b = table & position_b = 1) &
!occupied_table_1 : 1;
    move = move_b_to_table_2 & clear_b & !(on_b = table & position_b = 2) &
!occupied_table_2 : 2;
    move = move_b_to_table_3 & clear_b & !(on_b = table & position_b = 3) &
!occupied_table_3 : 3;
    move = move_b_to_table_4 & clear_b & !(on_b = table & position_b = 4) &
!occupied_table_4 : 4;
    move = move_b_to_table_5 & clear_b & !(on_b = table & position_b = 5) &
!occupied_table_5 : 5;
    move = move_b_to_table_6 & clear_b & !(on_b = table & position_b = 6) &
!occupied_table_6 : 6;
    move = move_b_to_a & clear_b & clear_a & on_b != a & (position_a + case (size_a -
size_b) >= 0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_b) >= 0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) -
```

```
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_a | ((position_a + case (size_a - size_b) >=
0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_a & (position_a + case (size_a - size_b) >= 0 : (size_a - size_b) / 2 ; TRUE :
((size_a - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_a + size_a - 1)) : position_a +
case (size_a - size_b) >= 0 : (size_a - size_b) / 2 ; TRUE : ((size_a - size_b) - 1) / 2 ; esac;
    move = move_b_to_c & clear_b & clear_c & on_b != c & (position_c + case (size_c -
size_b) >= 0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_b) >= 0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) -
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_c | ((position_c + case (size_c - size_b) >=
0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_c & (position_c + case (size_c - size_b) >= 0 : (size_c - size_b) / 2 ; TRUE :
((size_c - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_c + size_c - 1)) : position_c +
case (size_c - size_b) >= 0 : (size_c - size_b) / 2 ; TRUE : ((size_c - size_b) - 1) / 2 ; esac;
    move = move_b_to_d & clear_b & clear_d & on_b != d & (position_d + case (size_d -
size_b) >= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_b) >= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) -
1) / 2 ; esac) + size_b - 1 <= 6 & (size_b <= size_d | ((position_d + case (size_d - size_b)
>= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 >=
position_d & (position_d + case (size_d - size_b) >= 0 : (size_d - size_b) / 2 ; TRUE :
((size_d - size_b) - 1) / 2 ; esac) + (size_b - 1) / 2 <= position_d + size_d - 1)) : position_d
+ case (size_d - size_b) >= 0 : (size_d - size_b) / 2 ; TRUE : ((size_d - size_b) - 1) / 2 ;
esac;
    TRUE : position_b;
  esac;

TRANS
 next(on_c) =
  case
    move = move_c_to_table_0 & clear_c & !(on_c = table & position_c = 0) &
!occupied_table_0 : table;
    move = move_c_to_table_1 & clear_c & !(on_c = table & position_c = 1) &
!occupied_table_1 : table;
    move = move_c_to_table_2 & clear_c & !(on_c = table & position_c = 2) &
!occupied_table_2 : table;
    move = move_c_to_table_3 & clear_c & !(on_c = table & position_c = 3) &
!occupied_table_3 : table;
    move = move_c_to_table_4 & clear_c & !(on_c = table & position_c = 4) &
!occupied_table_4 : table;
    move = move_c_to_table_5 & clear_c & !(on_c = table & position_c = 5) &
!occupied_table_5 : table;
    move = move_c_to_table_6 & clear_c & !(on_c = table & position_c = 6) &
!occupied_table_6 : table;
    move = move_c_to_a & clear_c & clear_a & on_c != a & (position_a + case (size_a -
size_c) >= 0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_c) >= 0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_a | ((position_a + case (size_a - size_c) >=
0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_a & (position_a + case (size_a - size_c) >= 0 : (size_a - size_c) / 2 ; TRUE :
((size_a - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_a + size_a - 1)) : a;
```

```
    move = move_c_to_b & clear_c & clear_b & on_c != b & (position_b + case (size_b -
size_c) >= 0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_c) >= 0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_b | ((position_b + case (size_b - size_c) >=
0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_b & (position_b + case (size_b - size_c) >= 0 : (size_b - size_c) / 2 ; TRUE :
((size_b - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_b + size_b - 1)) : b;
    move = move_c_to_d & clear_c & clear_d & on_c != d & (position_d + case (size_d -
size_c) >= 0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_c) >= 0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_d | ((position_d + case (size_d - size_c) >=
0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_d & (position_d + case (size_d - size_c) >= 0 : (size_d - size_c) / 2 ; TRUE :
((size_d - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_d + size_d - 1)) : d;
    TRUE : on_c;
  esac;

TRANS
 next(position_c) =
  case
    move = move_c_to_table_0 & clear_c & !(on_c = table & position_c = 0) &
!occupied_table_0 : 0;
    move = move_c_to_table_1 & clear_c & !(on_c = table & position_c = 1) &
!occupied_table_1 : 1;
    move = move_c_to_table_2 & clear_c & !(on_c = table & position_c = 2) &
!occupied_table_2 : 2;
    move = move_c_to_table_3 & clear_c & !(on_c = table & position_c = 3) &
!occupied_table_3 : 3;
    move = move_c_to_table_4 & clear_c & !(on_c = table & position_c = 4) &
!occupied_table_4 : 4;
    move = move_c_to_table_5 & clear_c & !(on_c = table & position_c = 5) &
!occupied_table_5 : 5;
    move = move_c_to_table_6 & clear_c & !(on_c = table & position_c = 6) &
!occupied_table_6 : 6;
    move = move_c_to_a & clear_c & clear_a & on_c != a & (position_a + case (size_a -
size_c) >= 0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_c) >= 0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_a | ((position_a + case (size_a - size_c) >=
0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_a & (position_a + case (size_a - size_c) >= 0 : (size_a - size_c) / 2 ; TRUE :
((size_a - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_a + size_a - 1)) : position_a +
case (size_a - size_c) >= 0 : (size_a - size_c) / 2 ; TRUE : ((size_a - size_c) - 1) / 2 ; esac;
    move = move_c_to_b & clear_c & clear_b & on_c != b & (position_b + case (size_b -
size_c) >= 0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_c) >= 0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_b | ((position_b + case (size_b - size_c) >=
0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_b & (position_b + case (size_b - size_c) >= 0 : (size_b - size_c) / 2 ; TRUE :
((size_b - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_b + size_b - 1)) : position_b +
case (size_b - size_c) >= 0 : (size_b - size_c) / 2 ; TRUE : ((size_b - size_c) - 1) / 2 ; esac;
```

```
    move = move_c_to_d & clear_c & clear_d & on_c != d & (position_d + case (size_d -
size_c) >= 0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) - 1) / 2 ; esac) >= 0 &
(position_d + case (size_d - size_c) >= 0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) -
1) / 2 ; esac) + size_c - 1 <= 6 & (size_c <= size_d | ((position_d + case (size_d - size_c) >=
0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 >=
position_d & (position_d + case (size_d - size_c) >= 0 : (size_d - size_c) / 2 ; TRUE :
((size_d - size_c) - 1) / 2 ; esac) + (size_c - 1) / 2 <= position_d + size_d - 1)) : position_d +
case (size_d - size_c) >= 0 : (size_d - size_c) / 2 ; TRUE : ((size_d - size_c) - 1) / 2 ; esac;
    TRUE : position_c;
  esac;

TRANS
 next(on_d) =
  case
    move = move_d_to_table_0 & clear_d & !(on_d = table & position_d = 0) &
!occupied_table_0 & !occupied_table_1 : table;
    move = move_d_to_table_1 & clear_d & !(on_d = table & position_d = 1) &
!occupied_table_1 & !occupied_table_2 : table;
    move = move_d_to_table_2 & clear_d & !(on_d = table & position_d = 2) &
!occupied_table_2 & !occupied_table_3 : table;
    move = move_d_to_table_3 & clear_d & !(on_d = table & position_d = 3) &
!occupied_table_3 & !occupied_table_4 : table;
    move = move_d_to_table_4 & clear_d & !(on_d = table & position_d = 4) &
!occupied_table_4 & !occupied_table_5 : table;
    move = move_d_to_table_5 & clear_d & !(on_d = table & position_d = 5) &
!occupied_table_5 & !occupied_table_6 : table;
    move = move_d_to_table_6 & clear_d & !(on_d = table & position_d = 6) & FALSE :
table;
    move = move_d_to_a & clear_d & clear_a & on_d != a & (position_a + case (size_a -
size_d) >= 0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_d) >= 0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_a | ((position_a + case (size_a - size_d) >=
0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_a & (position_a + case (size_a - size_d) >= 0 : (size_a - size_d) / 2 ; TRUE :
((size_a - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_a + size_a - 1)) : a;
    move = move_d_to_b & clear_d & clear_b & on_d != b & (position_b + case (size_b -
size_d) >= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_d) >= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_b | ((position_b + case (size_b - size_d)
>= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_b & (position_b + case (size_b - size_d) >= 0 : (size_b - size_d) / 2 ; TRUE :
((size_b - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_b + size_b - 1)) : b;
    move = move_d_to_c & clear_d & clear_c & on_d != c & (position_c + case (size_c -
size_d) >= 0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_d) >= 0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_c | ((position_c + case (size_c - size_d) >=
0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_c & (position_c + case (size_c - size_d) >= 0 : (size_c - size_d) / 2 ; TRUE :
((size_c - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_c + size_c - 1)) : c;
    TRUE : on_d;
```

```
    esac;

TRANS
  next(position_d) =
    case
      move = move_d_to_table_0 & clear_d & !(on_d = table & position_d = 0) &
!occupied_table_0 & !occupied_table_1 : 0;
      move = move_d_to_table_1 & clear_d & !(on_d = table & position_d = 1) &
!occupied_table_1 & !occupied_table_2 : 1;
      move = move_d_to_table_2 & clear_d & !(on_d = table & position_d = 2) &
!occupied_table_2 & !occupied_table_3 : 2;
      move = move_d_to_table_3 & clear_d & !(on_d = table & position_d = 3) &
!occupied_table_3 & !occupied_table_4 : 3;
      move = move_d_to_table_4 & clear_d & !(on_d = table & position_d = 4) &
!occupied_table_4 & !occupied_table_5 : 4;
      move = move_d_to_table_5 & clear_d & !(on_d = table & position_d = 5) &
!occupied_table_5 & !occupied_table_6 : 5;
      move = move_d_to_table_6 & clear_d & !(on_d = table & position_d = 6) & FALSE :
6;
      move = move_d_to_a & clear_d & clear_a & on_d != a & (position_a + case (size_a -
size_d) >= 0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) - 1) / 2 ; esac) >= 0 &
(position_a + case (size_a - size_d) >= 0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_a | ((position_a + case (size_a - size_d) >=
0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_a & (position_a + case (size_a - size_d) >= 0 : (size_a - size_d) / 2 ; TRUE :
((size_a - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_a + size_a - 1)) : position_a +
case (size_a - size_d) >= 0 : (size_a - size_d) / 2 ; TRUE : ((size_a - size_d) - 1) / 2 ; esac;
      move = move_d_to_b & clear_d & clear_b & on_d != b & (position_b + case (size_b -
size_d) >= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) - 1) / 2 ; esac) >= 0 &
(position_b + case (size_b - size_d) >= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_b | ((position_b + case (size_b - size_d)
>= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_b & (position_b + case (size_b - size_d) >= 0 : (size_b - size_d) / 2 ; TRUE :
((size_b - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_b + size_b - 1)) : position_b
+ case (size_b - size_d) >= 0 : (size_b - size_d) / 2 ; TRUE : ((size_b - size_d) - 1) / 2 ;
esac;
      move = move_d_to_c & clear_d & clear_c & on_d != c & (position_c + case (size_c -
size_d) >= 0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) - 1) / 2 ; esac) >= 0 &
(position_c + case (size_c - size_d) >= 0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) -
1) / 2 ; esac) + size_d - 1 <= 6 & (size_d <= size_c | ((position_c + case (size_c - size_d) >=
0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 >=
position_c & (position_c + case (size_c - size_d) >= 0 : (size_c - size_d) / 2 ; TRUE :
((size_c - size_d) - 1) / 2 ; esac) + (size_d - 1) / 2 <= position_c + size_c - 1)) : position_c +
case (size_c - size_d) >= 0 : (size_c - size_d) / 2 ; TRUE : ((size_c - size_d) - 1) / 2 ; esac;
      TRUE : position_d;
    esac;

ASSIGN
  next(level_a) := case next(on_a) != on_a : case next(on_a) = table : 0;
                                    next(on_a) = b : min(level_b + 1, 10);
```

```
                                next(on_a) = c : min(level_c + 1, 10);
                                next(on_a) = d : min(level_d + 1, 10);
                                TRUE : level_a;
                            esac;
              TRUE : level_a;
             esac;
  next(level_b) := case next(on_b) != on_b : case next(on_b) = table : 0;
                                next(on_b) = a : min(level_a + 1, 10);
                                next(on_b) = c : min(level_c + 1, 10);
                                next(on_b) = d : min(level_d + 1, 10);
                                TRUE : level_b;
                            esac;
              TRUE : level_b;
             esac;
  next(level_c) := case next(on_c) != on_c : case next(on_c) = table : 0;
                                next(on_c) = a : min(level_a + 1, 10);
                                next(on_c) = b : min(level_b + 1, 10);
                                next(on_c) = d : min(level_d + 1, 10);
                                TRUE : level_c;
                            esac;
              TRUE : level_c;
             esac;
  next(level_d) := case next(on_d) != on_d : case next(on_d) = table : 0;
                                next(on_d) = a : min(level_a + 1, 10);
                                next(on_d) = b : min(level_b + 1, 10);
                                next(on_d) = c : min(level_c + 1, 10);
                                TRUE : level_d;
                            esac;
              TRUE : level_d;
             esac;



TRANS
  (move != none) -> (next(on_a) != on_a | next(position_a) != position_a | next(on_b) !=
on_b | next(position_b) != position_b | next(on_c) != on_c | next(position_c) != position_c |
next(on_d) != on_d | next(position_d) != position_d);

CTLSPEC EF goal;
```

5. Tabela de Conceitos e Representação

Conceito	STRIPS	Prolog estendido	Proposta de modelo NuSMV	Justificativa para projeto NuSMV
Block Properties	block(X).	size(X, W).	DEFINE size_a := 1;	Codifica dimensões físicas imutáveis como constantes de tempo de compilação, não variáveis de estado, para eficiência.
Clear Condition	clear(X).	clear(X, S).	DEFINE clear_a := !(on_b=a | on_c=a | ...)	Verifica se nenhum bloco está sobre X, usado como guarda de mobilidade.
On Relation	on(X,Y).	on(X,Y,S).	VAR on_a : {b,c,table};	Modela posição relativa de cada bloco como variável de estado.
Table	table.	table(Pos).	VAR on_a : {table(1), table(2), b, c};	Modela a mesa com slots indexados para representar posições possíveis.

6. Tabela de Restrições no Mundo dos Blocos

Tipo de Restrição	Regra em Linguagem Natural	STRIPS	Prolog Estendido	Proposta de modelo NuSMV
Mobility	Um bloco só pode ser movido se estiver livre (clear).	precond(move(X,Y), clear(X))	can(move(B, Dest), S) :clear(B, S), ...	DEFINE clear_a := !(on_b=a | on_c=a | ...); usado como guarda em next(on_a).
Target Accessibility	O destino deve estar livre para receber o bloco.	precond(move(X,Y), clear(Y))	can(move(B, on(Target)), S) :- clear(Target, S), ...	DEFINE target_ok := clear(Target) como condição em TRANS.
Stability	Só é permitido colocar um bloco sobre outro de largura maior ou igual.	não modelado	size(B,W1), size(Target,W2), W1 ≤ W2	DEFINE size_check := (size_B <= size_Target) incluído no TRANS.
Spatial Occupancy	Todos os slots que o bloco ocupará na mesa devem estar livres.	não modelado	space_check(B, table(X), S) :- absolute_pos(B,X), ...	DEFINE space_ok_c_table2 := is_free(2) & is_free(3) ... usado em TRANS.
Logical Validity	Um bloco não pode ser colocado sobre si mesmo.	neq(X,Y)	checado em can(move(B, on(B)), S) → sempre falha	TRANS !(move = move_a_a) ou implícito.
Boundary Condition	Um bloco não pode ultrapassar os limites da mesa.	não modelado	absolute_pos(B, X), X+size(B)-1 < TableWidth	TRANS next(on_b)=table(k) permitido só se k+size_b-1 < table_width.

7. Situações

Situação 1:

Tarefas de Planejamento

A Situação 1 define o estado inicial S_0 e vários estados finais possíveis ($S_{f1}, S_{f2}, S_{f3}, S_{f4}, S_5, S_7$). O trabalho exige a geração manual do plano de ações para os seguintes caminhos:

1. Plano de $s_inicial=S0$ até um dos estados: s_f1, s_f2, s_f3 ou s_f4.
   ◦ Todos esses estados são visualmente representados na fonte.
2. Plano de $s_inicial=S0$ até $S5$.
   ◦ O estado $S5$ é visualmente representado.
3. Plano de $s_inicial=S0$ até $S7$.
   ◦ O estado $S7$ é visualmente representado.

O resultado esperado para a resolução da Situação 1 seriam sequências de ações na forma move(B, Pi, Pj) que transformam o estado $S0$ no estado objetivo.

Exemplo hipotético de formato do resultado (Não encontrado nas fontes): Para $S0 \rightarrow S_{f1}$, o plano seria:

1. move(c, 4, 3)
2. move(b, 3, 2)
3. ... (e assim por diante)

Situação 2:

A Situação 2 ilustra seis estados, de $S0$ a $S5$.

• Estados exibidos: $S0, S1, S2, S3, S4, S5$.

• Ação: A ação move(B, Pi, Pj) é permitida.

As fontes fornecidas exibem as representações visuais desses estados, mas não especificam um planejamento de $S_{inicial}$ para S_{goal} associado diretamente à Situação 2 no formato de lista de tarefas, como feito na Situação 1

Situação 3:

A Situação 3 ilustra oito estados, de $S0$ a $S7$.

• Estados exibidos: $S0, S1, S2, S3, S4, S5, S6, S7$.

• Ação: A ação move(B, Pi, Pj) é permitida.

Assim como na Situação 2, as fontes exibem os estados visualmente, mas não especificam um planejamento de $S_{inicial}$ para S_{goal} associado diretamente à Situação 3 no formato de lista de tarefas. Note, contudo, que a Situação 1 solicita planos que envolvem S_5 e S_7, cujos diagramas aparecem na seção de Situações 2 e 3

8. Conclusão

Em suma, este estudo consolida uma metodologia híbrida que articula verificação, lógica e planejamento para uma análise completa do domínio do Mundo dos Blocos. A principal contribuição foi demonstrar como a prova de alcançabilidade de um estado, obtida através da especificação temporal em NuSMV, pode ser complementada pela geração de um plano de ação explícito por um planejador baseado em lógica de predicados. A estrutura modular, com regras auxiliares que definem a semântica das ações, provou ser eficaz e extensível. Olhando para o futuro, o trabalho abre caminhos claros para investigações adicionais, incluindo a otimização de planos, a introdução de blocos com propriedades heterogêneas (tamanhos e pesos variáveis) e a modelagem de ambientes multiagente, confirmando o potencial da integração de lógicas formais para o avanço do Raciocínio Automatizado.