UNICAMP
Universidade Estadual de Campinas

FEEC
Faculdade de Engenharia Elétrica e de Computação

Aluno: Mateus Neves Barreto
R.A.: 142358
Disciplina: IA006
Professor: Ricardo R. Gudwin

Relatório – Aula 5

1 Atividade 1

Neste tópico é apresentado o desenvolvimento do Tutorial 4 do SOAR. Nele é tratado o clássico problema dos missionários e canibais. Para o desenvolvimento deste tutorial, foi criado um projeto nomeado de *mac* no VisualSoar, Figura 1.

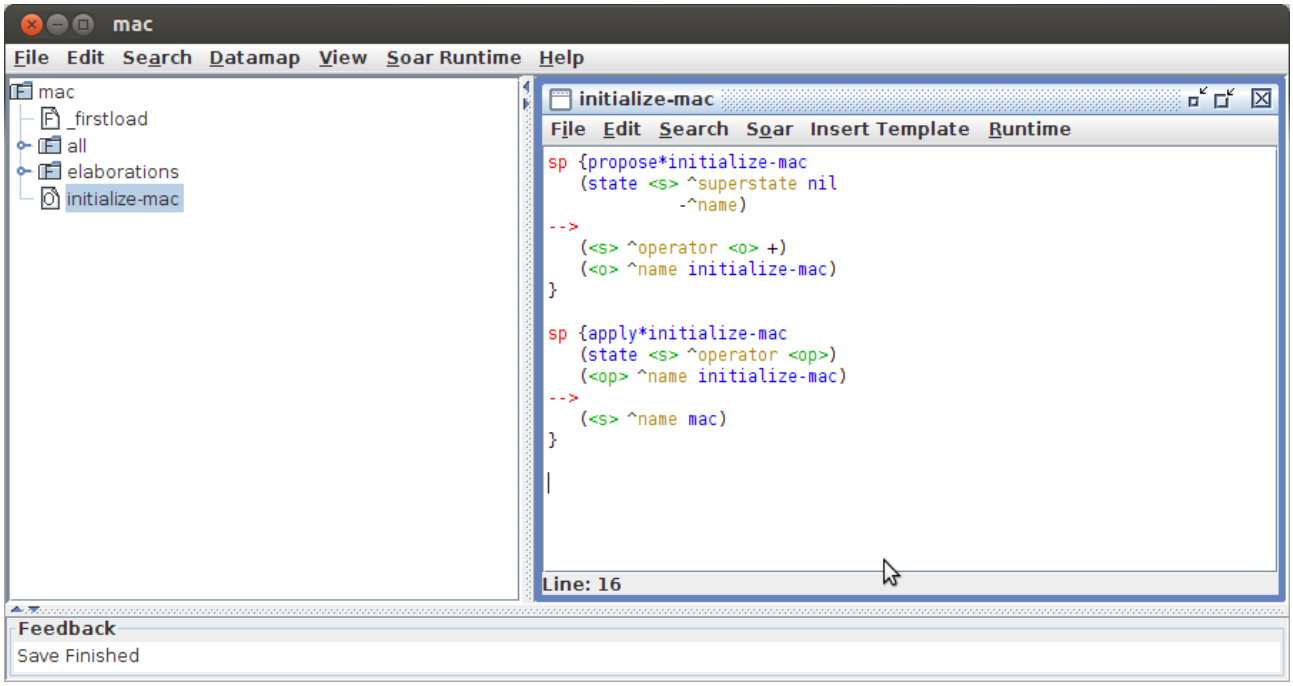

Figura 1 – Criação do projeto no VisualSoar.

A inicialização do estado inicial é bem similar ao do tutorial realizado com o problema dos jarros, Figura 2.

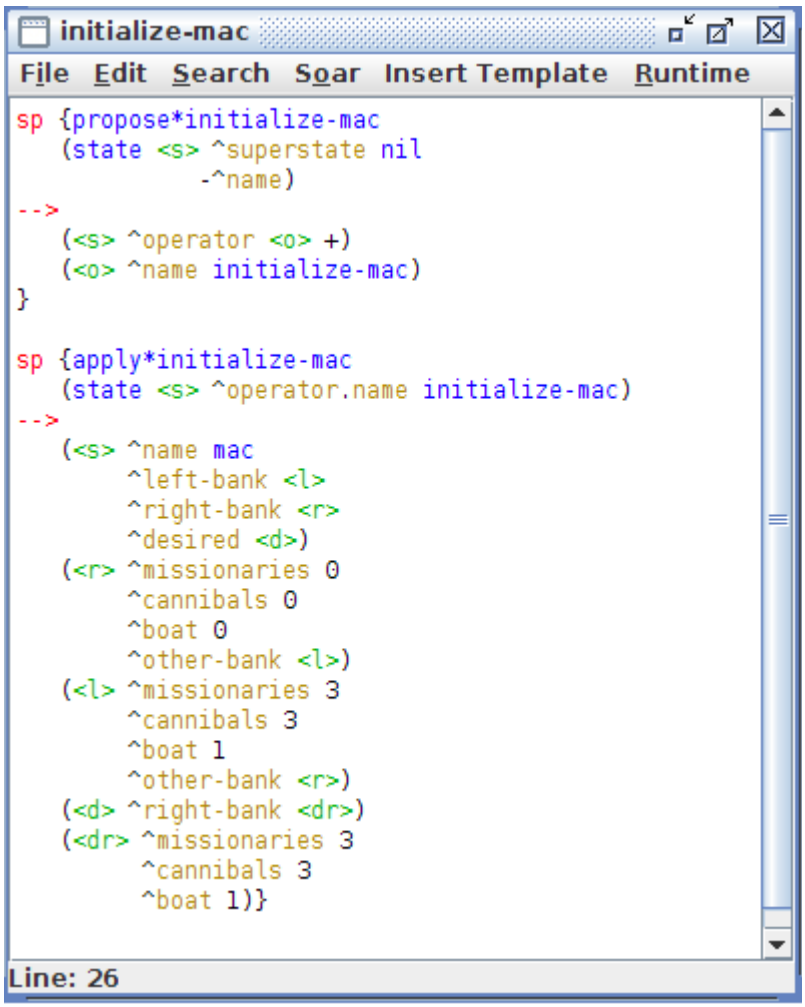

Figura 2 – Inicialização do projeto.

1.1 Proposições de operadores *mac*

Neste sub-tópico é apresentado as proposições para criação de operadores que realizem as três possíveis tarefas:

- Mover um missionário ou um canibal;
- Mover dois missionários ou dois canibais;
- Mover um missionário e um canibal juntos;

A implementação destas três regras esta presente na Figura 3.

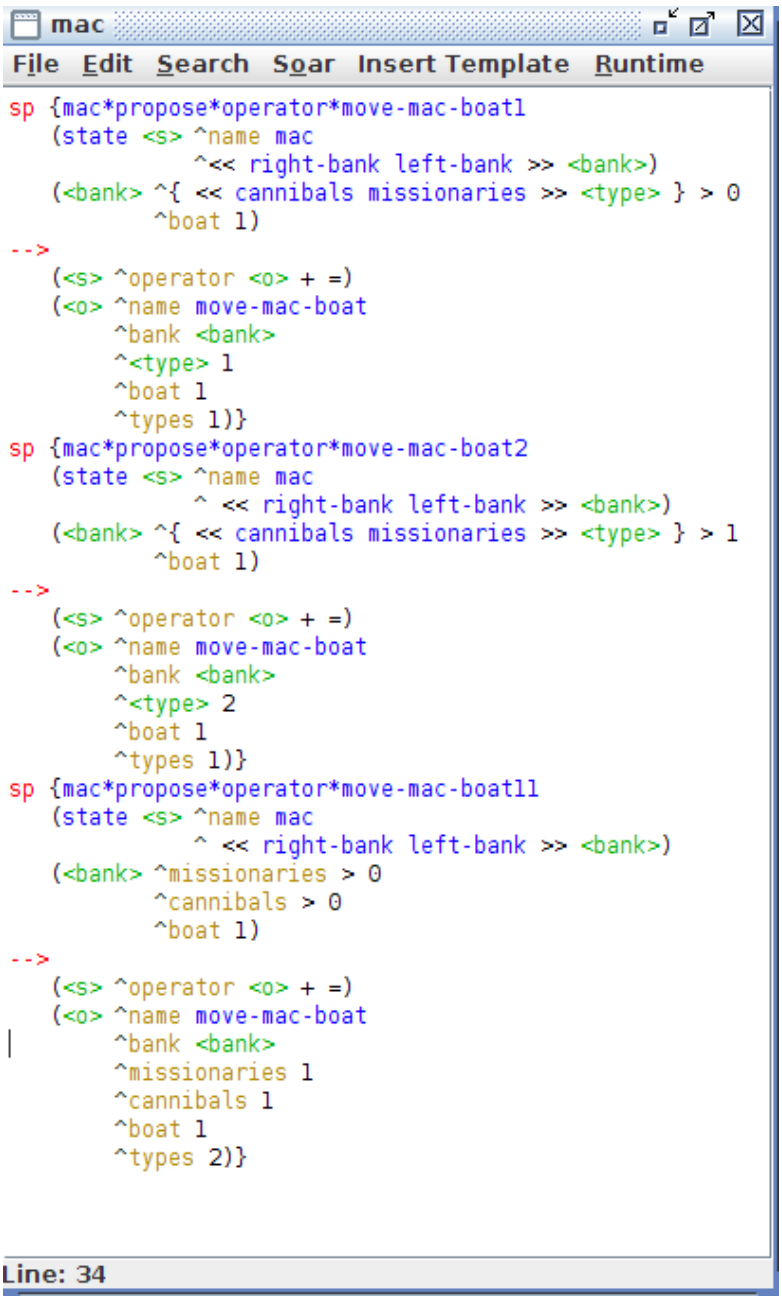

Figura 3 – Proposição de operadores *mac*.

1.2 Aplicação dos operadores

Neste sub-tópico são criadas as regras que implementam as propostas realizadas no item anterior, Figura 4.

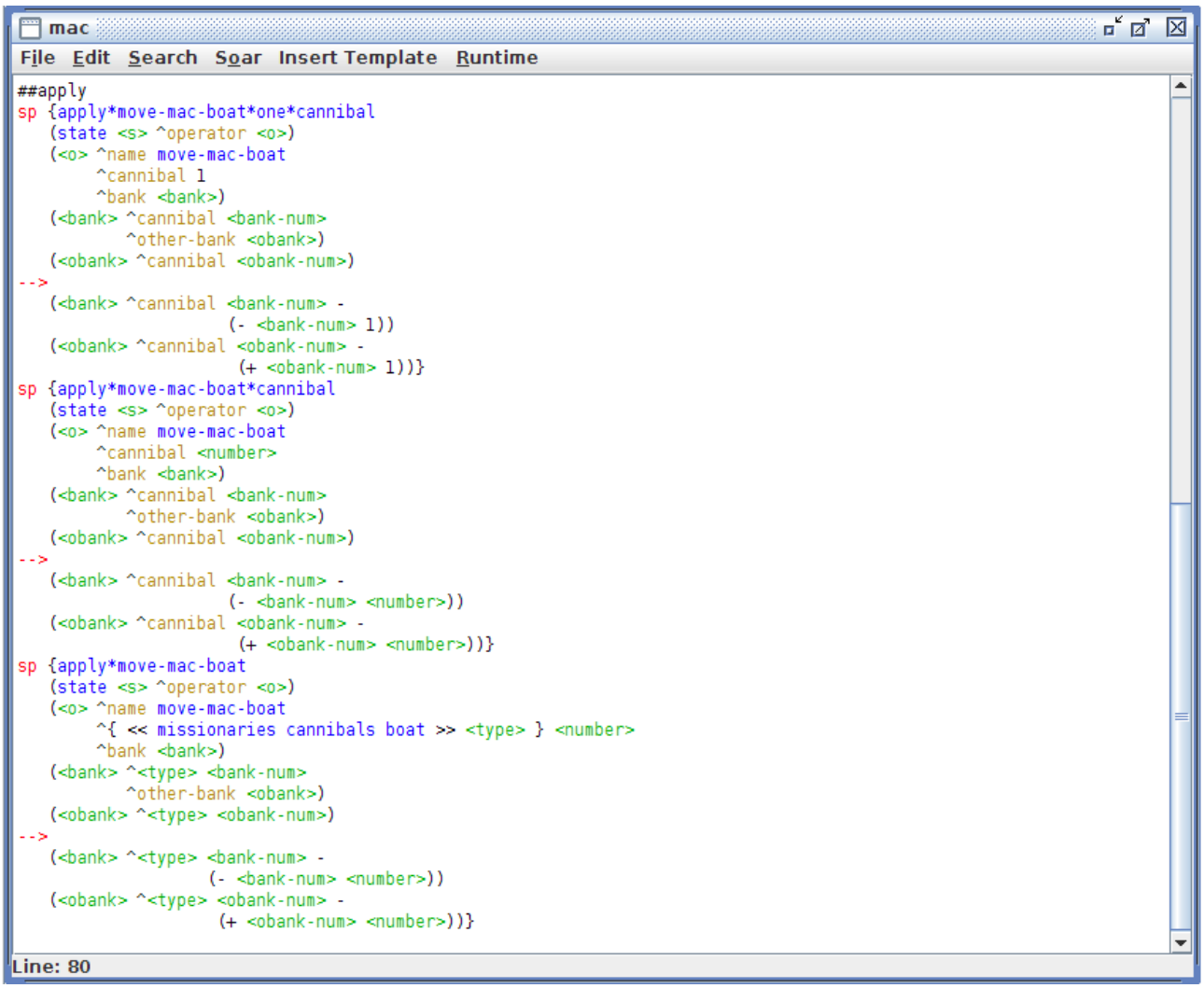

Figura 4 – Aplicação dos operadores *mac*.

1.3 Monitorando operadores

Neste sub-tópico, são apresentadas as regras que monitoram os operadores selecionados e os estados dos mesmos, Figura 5.

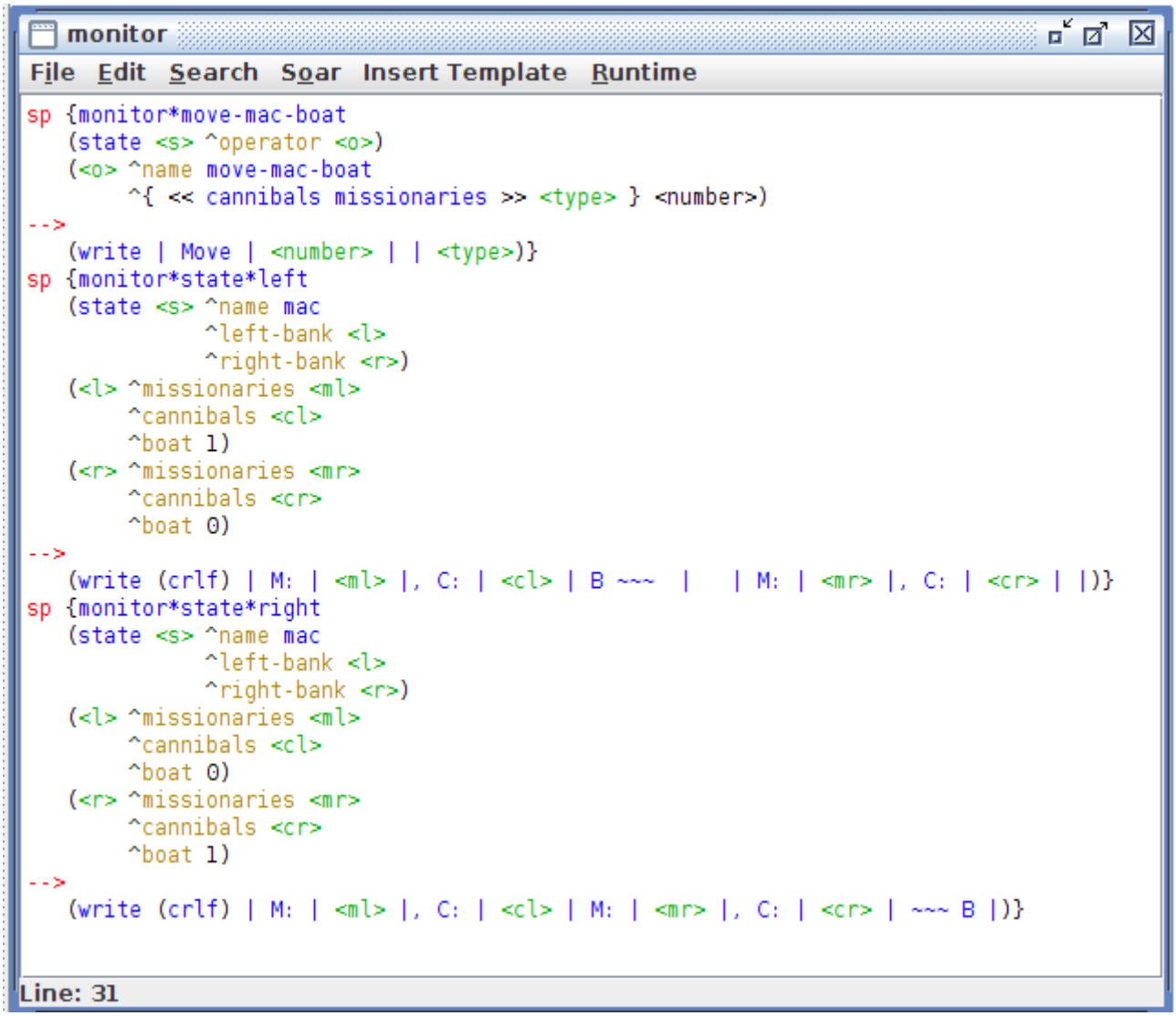

Figura 5 – Monitorando estado de operadores selecionados.

A execução das regras criadas até o momento, pode ser observada na Figura 6.

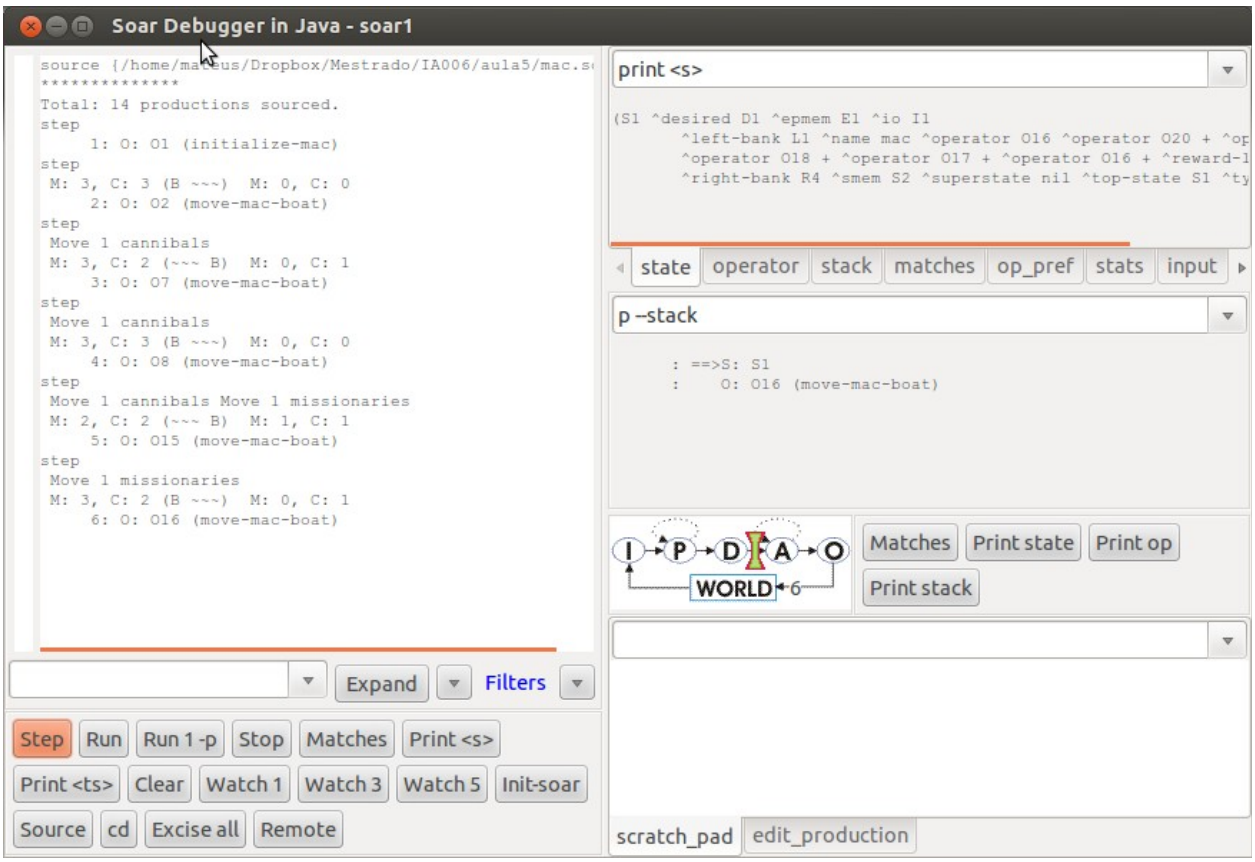

Figura 6 – Execução código parcial.

1.4 Estado de reconhecimento

1.4.1 Secesso

A Figura 7 apresenta a regra de detecção de estado de sucesso, ou seja, o estado final de resolução do problema dos missionários e canibais. Este estado é atingido quando todos os missionários e os canibais passam a estar do outro lado do rio.

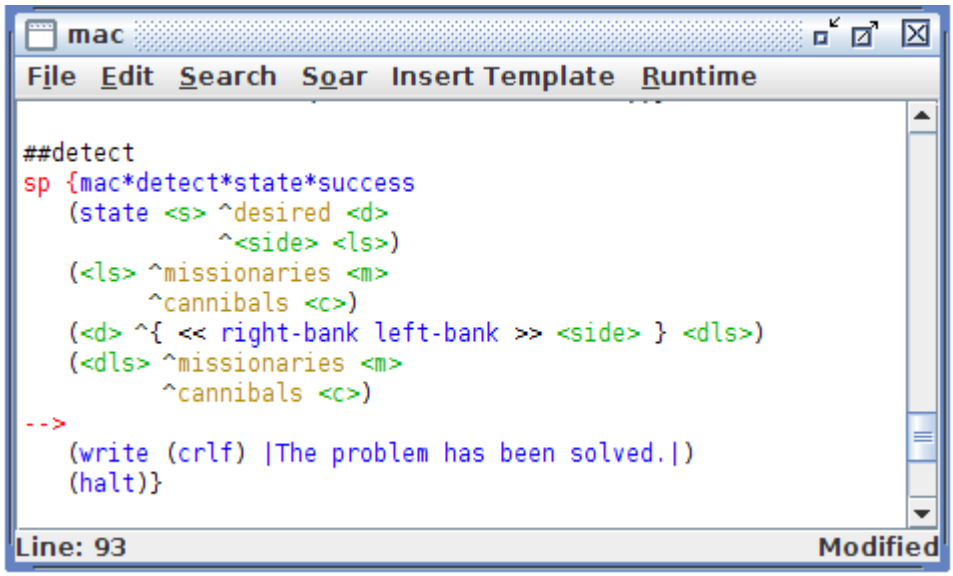

Figura 7 – Reconhecimento de estado de sucesso.

1.4.2 Falha

A Figura 8 apresenta a regra de reconhecimento de estado de falha. Este estado é atingido quando

em qualquer lado do lago, fique presente um número maior de canibais em relação ao número de missionários. Os canibais acabam comendo os missionários.

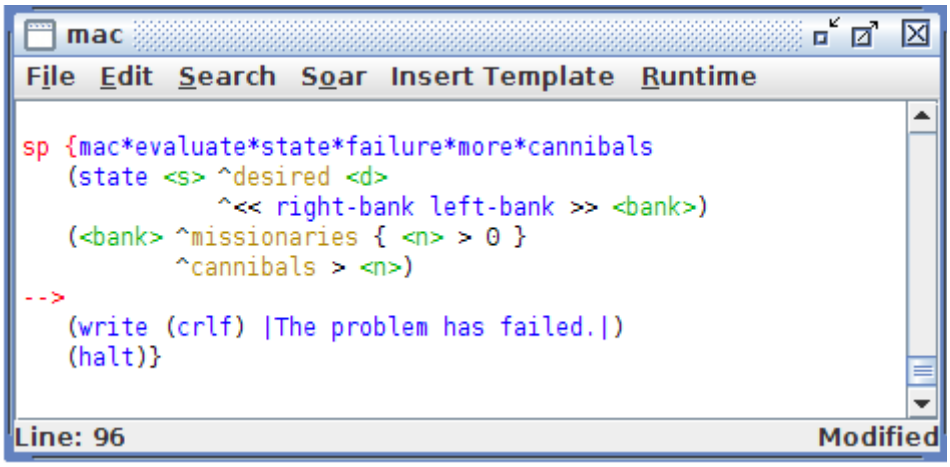

Figura 8 – Reconhecimento de estado de falha.

1.5 Controle de pesquisa

Neste sub-tópico são apresentadas as regras que identificam o estado de infactibilidade para o problema e retornam ao estado anterior. Para realizar tal ação é necessário persistir o operador e depois realizar a correção do estado e apagar o último operador utilizado. As Figuras 9 e 10 apresentam a implementação deste tópico.

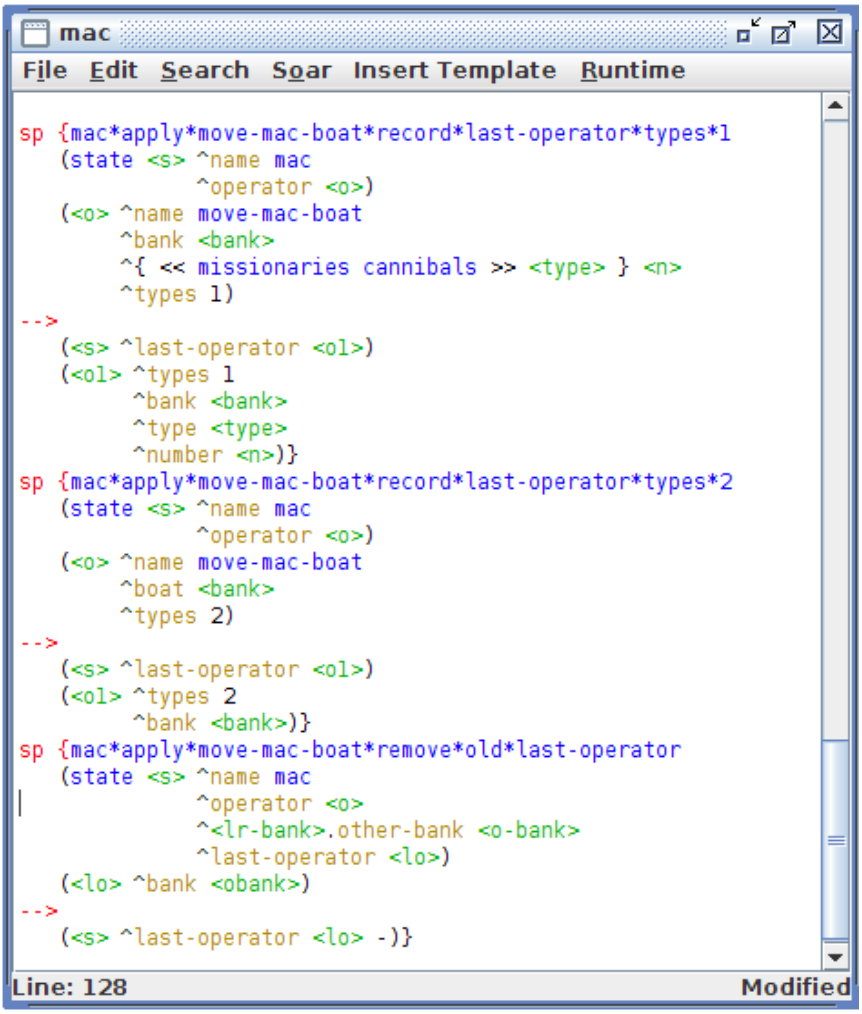

Figura 9 – Recuperando o último operador - 1.

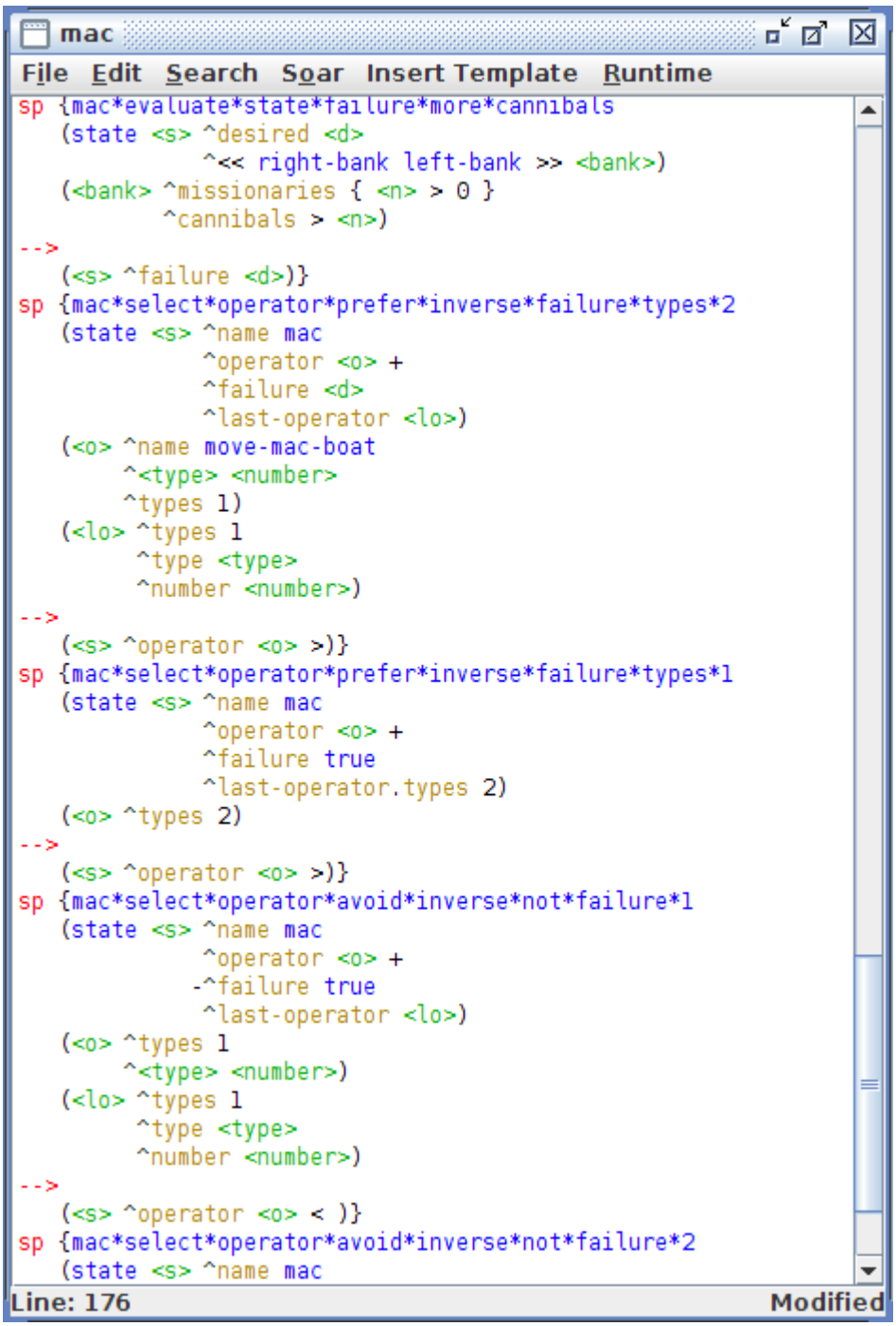

Figura 10 – Recuperando o último operador – 2.

A Figura 11 apresenta a execução do programa realizado no tutorial deste tópico.

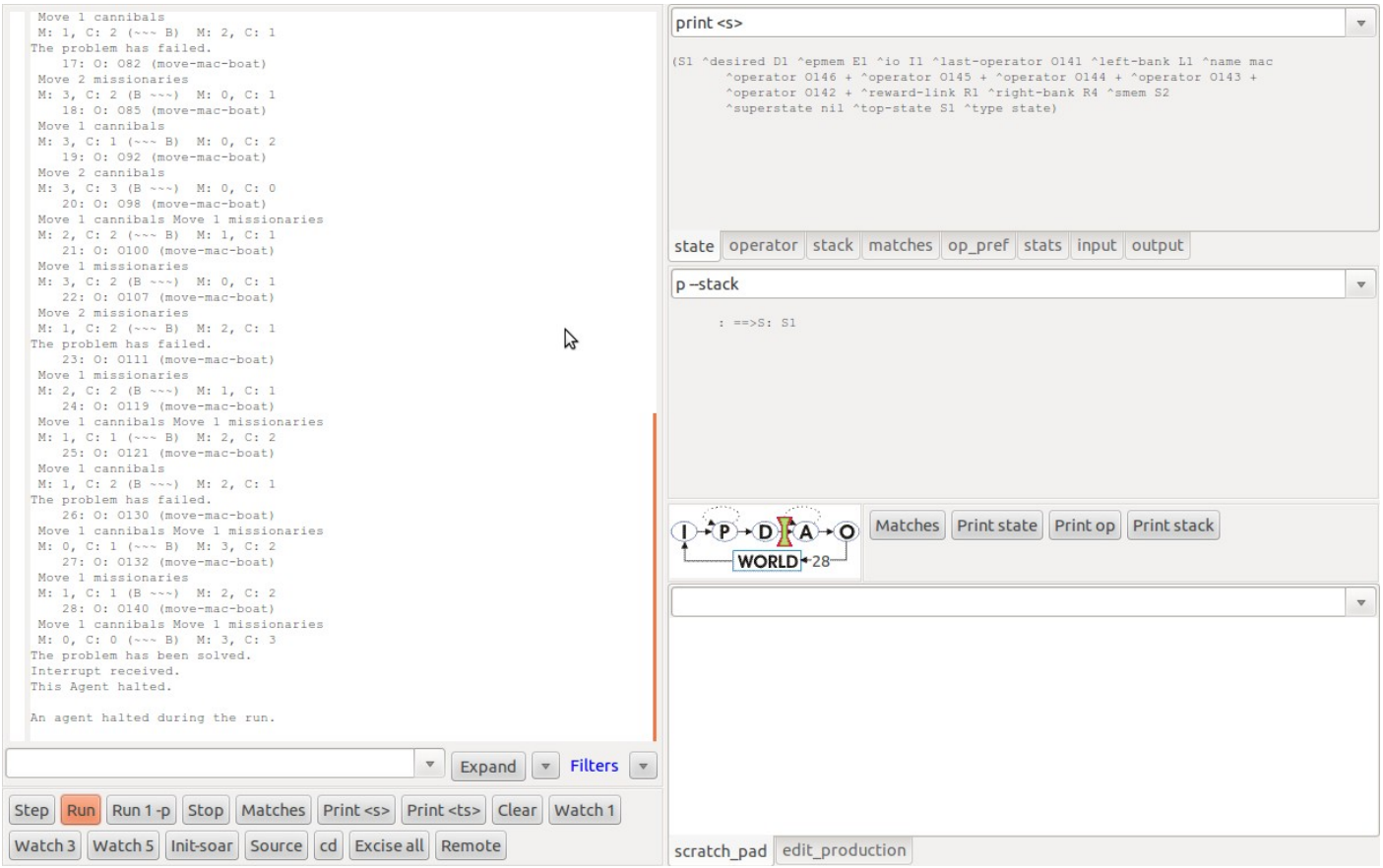

Figura 11 – Solução do problema dos missionários e canibais.

2 Atividade 2

Neste tópico é apresentado o desenvolvimento do Tutorial 5 do SOAR (parte 5). Nele é tratado novamente os problemas dos missionários e canibais e dos jarros de água, porém sub-rotinas de planejamento e aprendizado são apresentados.

2.1 O problema do jarro de água

2.1.1 Estado *tie*

Quando tiramos o operador de indiferença da proposição *fill* do programa *"water-jug"* o SOAR automaticamente gera novos sub-estados onde as preferências do operador são insuficientes para selecionar uma única opção, pois a proposição faltante causa um estado de conflito. Esse caso gera o estado de empate, levando ao operador de impasse *"tie"*, Figuras 12 e 13.

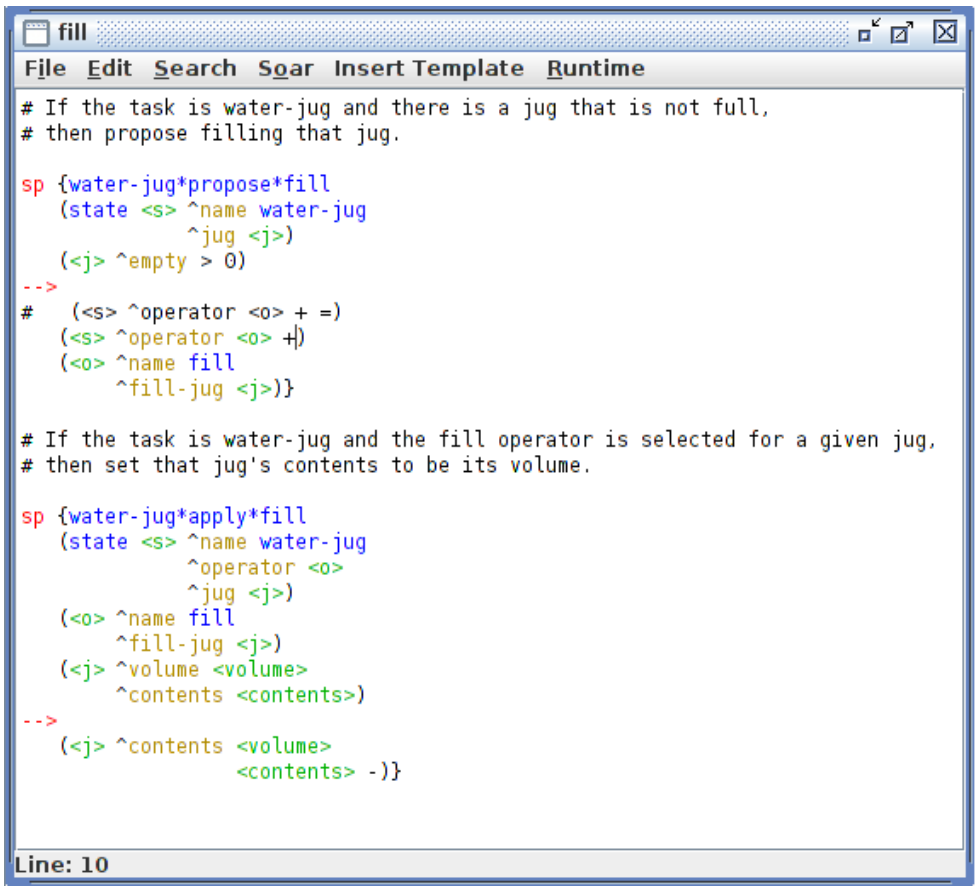

Figura 12 – Comentando proposição de indiferença.

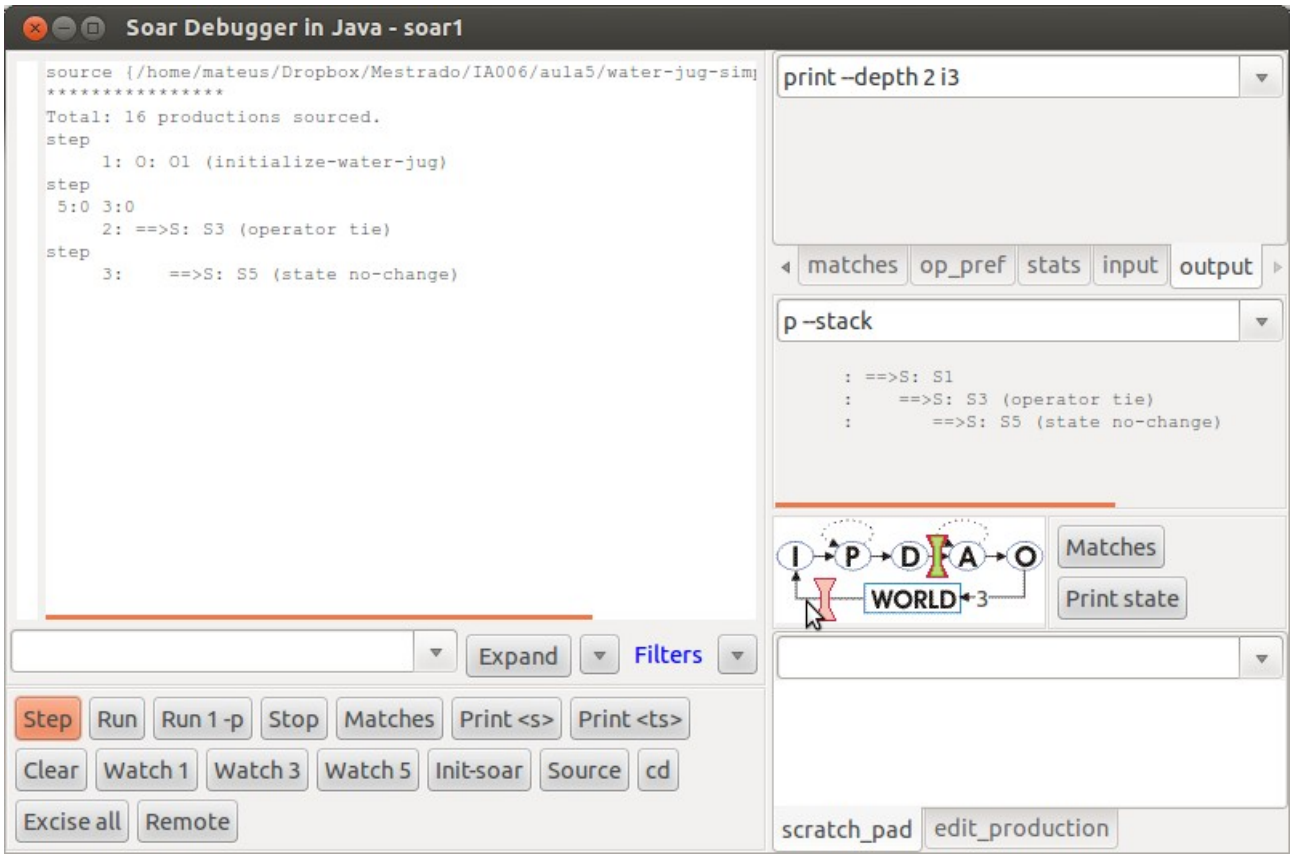

Figura 13 – Execução sem o operador de indiferença.

Para resolver o problema do impasse, o SOAR apresenta em conjunto com suas ferramentas, um conjunto de regras genéricas que realizam a avaliação e comparação de operadores de forma global;

podendo ser utilizado em problemas não muito específicos adicionando a importação das regras ao código, Figura 14.

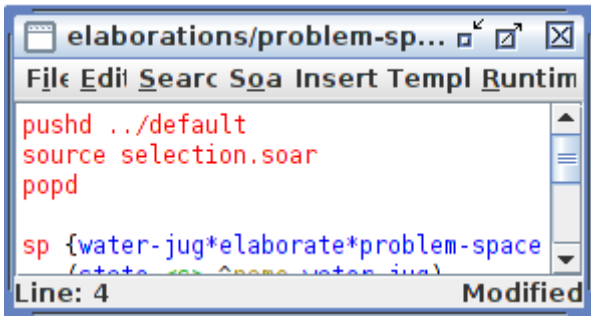

Figura 14 – Importação das regras de avaliação de operadores.

O tutorial começa a ensinar deste ponto, como utilizar estas regras gerais.

2.1.2 Representação do estado de seleção

O estado de seleção deve ser composto por um conjunto de atributos que possibilite uma avaliação complementar, com cada objeto de avaliação com a seguinte estrutura:

- **operador:** operador a ser avaliado;
- **symbolic-value:** sucesso/sucesso parcial/ falha parcial/ falha/ indiferente;
- **numeric-value:** [número];
- **value true:** indica qualquer existência de valor simbólico ou numérico;
- **desired <d>:** indicador desejado de um estado para ser utilizado na avaliação, se existir.

Outra opção de avaliação de estado, é a criação das avaliações direto nos operadores. A vantagem de criar avaliações sobre o estado, é que elas são automaticamente removidas quando o estado de impasse *(tie)* é resolvido.

2.1.3 Criação do estado inicial de seleção

Nesta parte do tutorial, é criado o estado de seleção a partir que a regra encontrar um estado de impasse *(tie)*, Figura 15.

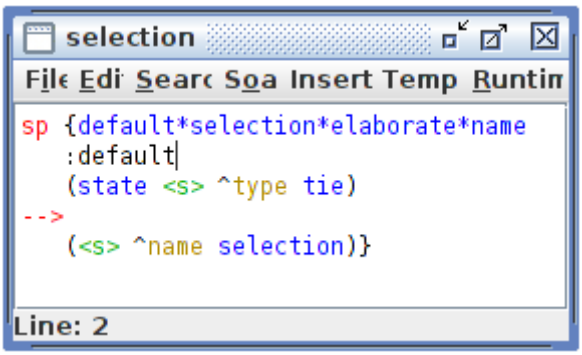

Figura 15 – Estado de seleção.

O ":default" diz que a regra é uma regra padrão. As regras do tipo padrão possuem o mesmo comportamentos das regras comuns, porém o SOAR separa suas estatísticas. Todas as regras marcadas como *default* estão presente no arquivo "selection.soar".

2.1.4 Proposta do operador de seleção

Esta regra é criada apenas para avaliar o operador de seleção. Ele é proposto no problema de seleção se existir um item ainda não avaliado, ou seja, sem valor atribuído, Figura 16.

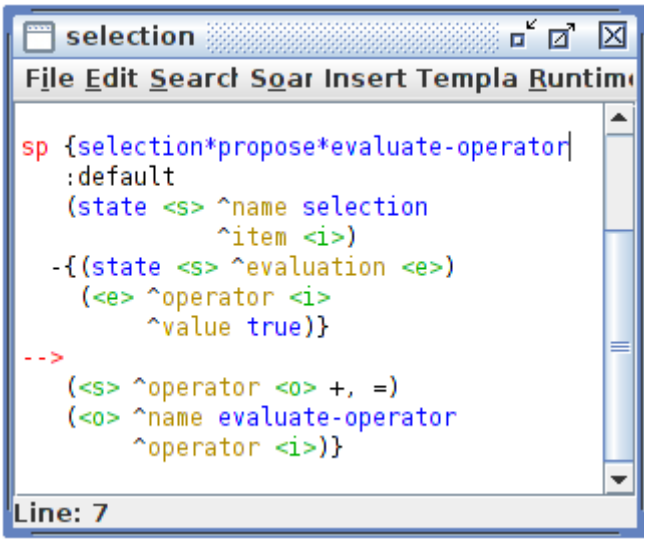

Figura 16 – Proposição do operador de seleção.

2.1.5 Criando o estado inicial

A proposta desta parte do tutorial é armazenar os estados anteriores a ocorrência do estado de impasse *(tie)*. Para realizar tal ação, é preciso decidir qual é o número de passos anteriores ao estado *tie* será gerado e observar a estrutura que esta sendo utilizada no problema. Porém no estado atual do problema do jarro os estados são gravados e apagados da memória de trabalho. Por exemplo a regra que aplica o preenchimento, presente na Figura 12, esta modifica os argumentos *"contents"* e *"empty"*. Por exemplo, se a regra de preenchimento for utilizada, no operador do jarro que contém 3 litros, com os atributos *"(j1 ^contents 0)"* e *(j1 ^empty 3);* os mesmos serão removidos da memória de trabalho e os atributos *"(j1 ^contents 3)"* e *(j1 ^empty 0)* serão inseridos,Figura 17.

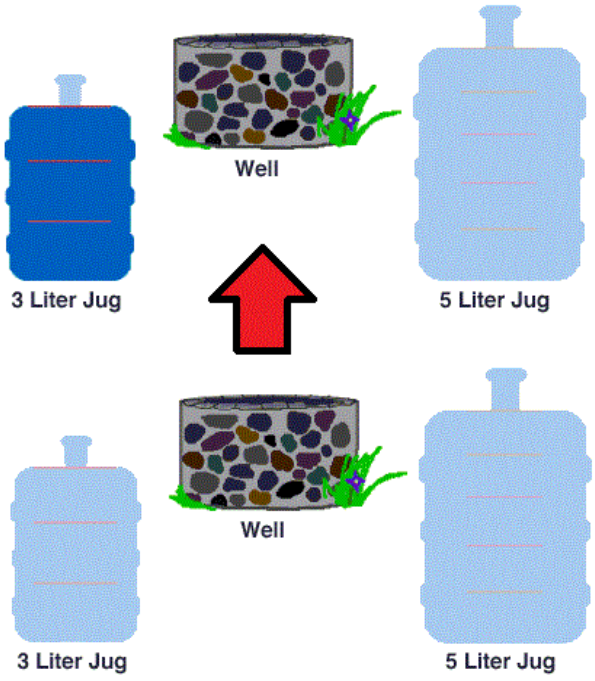

Figura 17 – Aplicando a regra fill no operador jarro-3litros.

O resultado desta operação não viabiliza o objetivo de alterar o operador sem afetar o estado original, como uma busca em profundidade, para depois aplicar o caminho. Existem duas possibilidades para solucionar este problema. A primeira é copiar dois níveis de atributos, adicionando a regra presente na Figura 18.

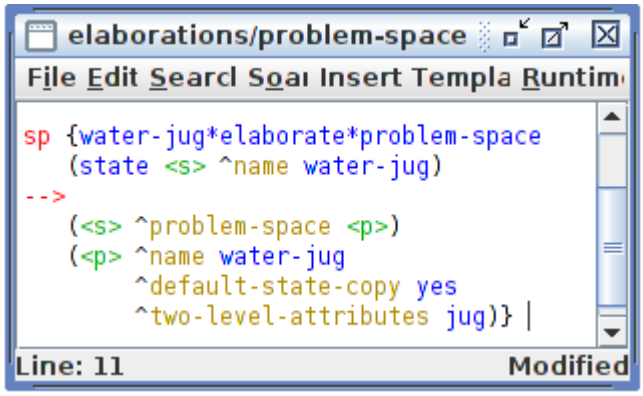

Figura 18 – Copiando niveis de atributos.

A segunda possibilidade de solução é a modificação da regra que aplica a operação de preenchimento, Figura 19.

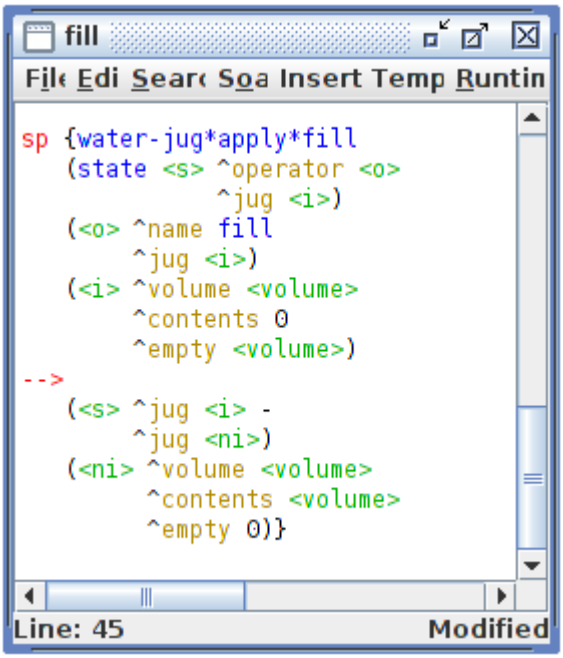

Figura 19 – Regra Fill modificada.

A primeira solução é preferida por duas razões; primeira: embora a segunda opção exija menos elementos na memória de trabalho, ela exige mais modificações na estrutura da memória de trabalho; segunda: esta abordagem é menos natural, pois propõe outro jarro ao invés de alterar o existente.

2.1.6 Aplicando e selecionando operador de avaliação

A seleção do operador é feita a partir de uma cópia. Assim que a cópia é selecionada as regras adicionais rejeitam todos os outro operadores, forçando a escolha para a cópia. Uma vez que o operador é selecionado, as regras são aplicadas no estado da cópia. Para realizar a seleção e a aplicação do operador, não precisa ser alterada nenhuma regra presente no *water-jug*.

2.1.7 Avaliando o resultado

Possuindo um novo estado, uma avaliação já pode ser realizada. Um argumento é adicionado ao operador a ser aplicado: *^tried-tied-operator <o>*. A regra para a detecção de sucesso foi modificada conforme a Figura 20.

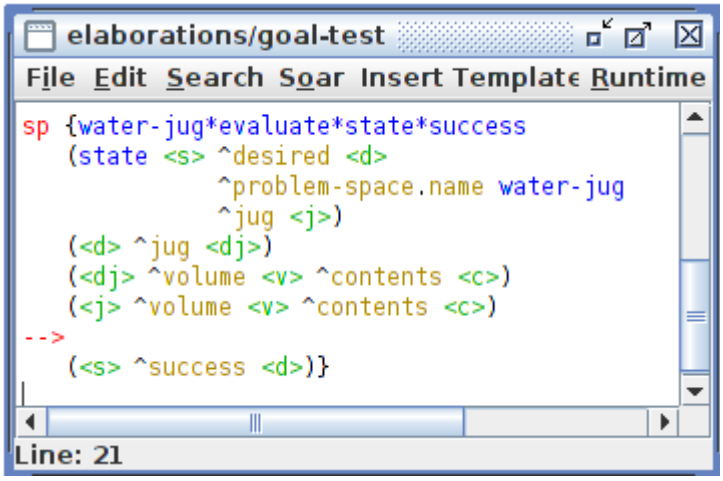

Figura 20 – Nova regra de avaliação.

Além das avaliações simbólicas as regras de seleção podem processar as regras com avaliações numéricas. As preferências simbólicas são: sucesso, sucesso parcial, falha, falha parcial e indiferente. Para avaliações numéricas é criado um atributo chamado *^numeric-value.*
É preciso ainda, criar regras que elaborem cada estado com seus respectivos super estados, Figura 21, e ainda uma regra para detectar falhas, Figura 22.

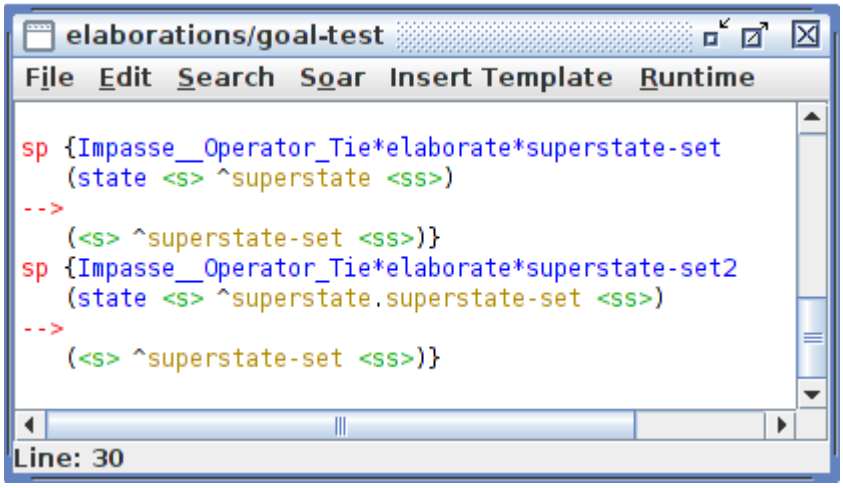

Figura 21 – Elaboração de super estado.

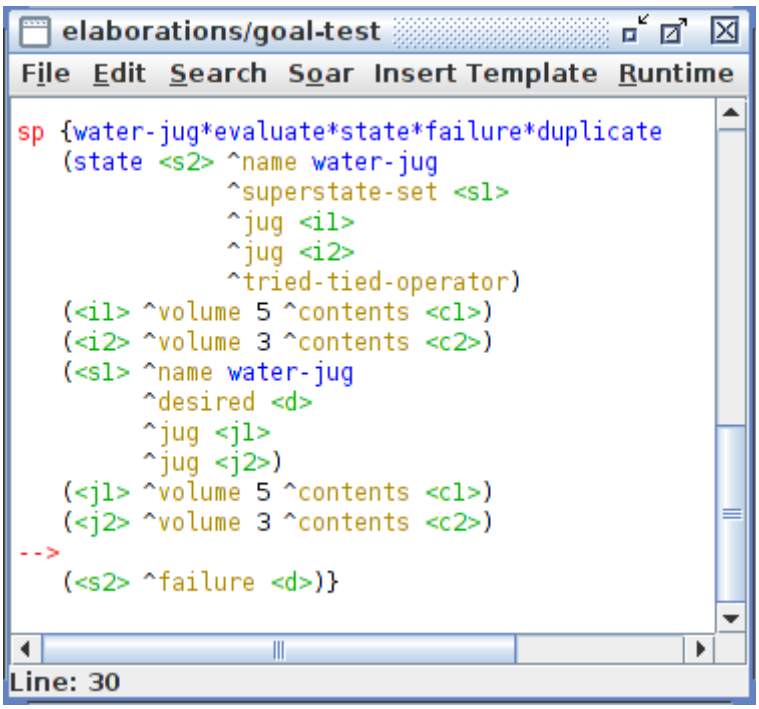

Figura 22 – Detecção de falhas.

Com estas regras a solução do problema é mais dirigida, porém demora um número excessivo para convergir ao resultado. Pois quando na busca para a solução do problema, não é marcado o estado já visitado, o estado pode ser visitado novamente.

2.1.8 Utilizando o *Chunk*

Para evitar a busca com repetição de estados utiliza-se a aprendizagem *chunkin.* Para ativar a aprendizagem, basta adicionar o comando "*learn –on*" na linha de execução ou nas regras. Para observar os *chunks* criados, basta adicionar o comando *"watch –chunks"* na linha de execução, Figura 23.

```
source {/home/mateus/Dropbox/Mestrado/IA006/aula5/water-jug-tie/water-j
*****************************
Total: 30 productions sourced.
learn --on
watch --chunks
step
     1: O: O1 (initialize-water-jug)
step
 5:0 3:0
     2: ==>S: S3 (operator tie)
step
     3:    O: O6 (evaluate-operator)
step
     4:    O: O5 (evaluate-operator)
step
     5:    O: O7 (create-preferences)
step
Firing chunk-13*d5*tie*1
     6: O: O3 (fill)
step
  FILL(5)
 5:5 3:0
Retracting chunk-13*d5*tie*1
Retracting chunk-13*d5*tie*1
     7: ==>S: S5 (operator tie)
```

Figura 23 – Utilização do *chunks* através do *learn on.*

Ativando a aprendizagem *chunk* pode-se observar que busca converge mais rápida, isso devido aos estados repetidos evitados na busca, Figura 24.

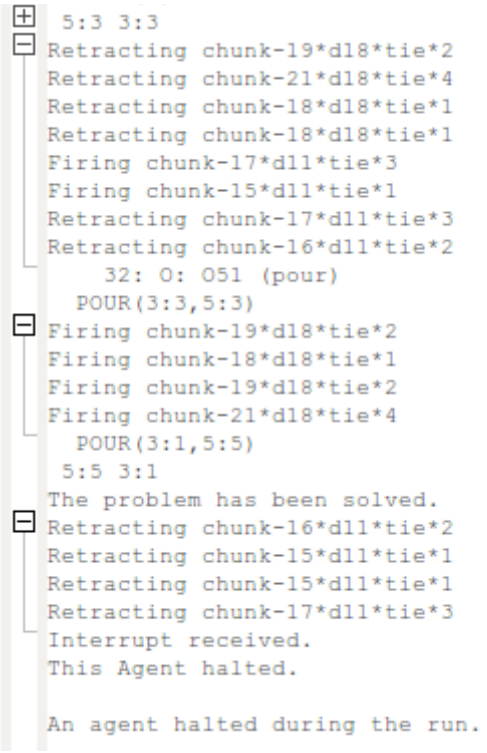

Figura 24 – Resultado utilizando a aprendizagem *chunk*.

2.2 O problema dos Missionários e dos Canibais

Neste sub-tópico são aplicadas as mesmas técnicas empregadas no problema do jarro de água. Também será explorado o uso de avaliações numéricas, além das simbólicas. O projeto utilizado nesta parte do tutorial se encontra na pasta "mac_parteV".

2.2.1 Conversão para planejamento e aprendizagem

Para inicialização deste tópico, é necessária a modificação das propostas "*move-mac-boat*" removendo todos os operadores de indiferença.

2.2.1.a) Adicionar regras de seleção

Para adicionar as regras de seleção, basta adicionar os comando presentes na Figura 25 ao projeto.

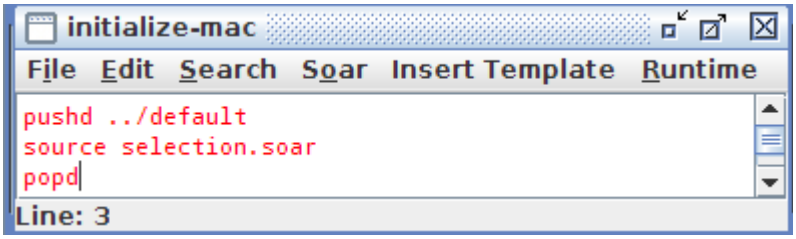

Figura 25 – Adicionando regras de seleção.

2.2.1.b) Adicionar informações de cópia de estado e espaço do problema

Devido a implementação original do problema, tratar regras para as duas margens do rio, será necessário o armazenamento de informações de estado em dois níveis. Pois se for necessário

desfazer a ação, o retorno para dois passos anteriores será possível. A figura 26, apresenta o código que viabiliza essa decisão.

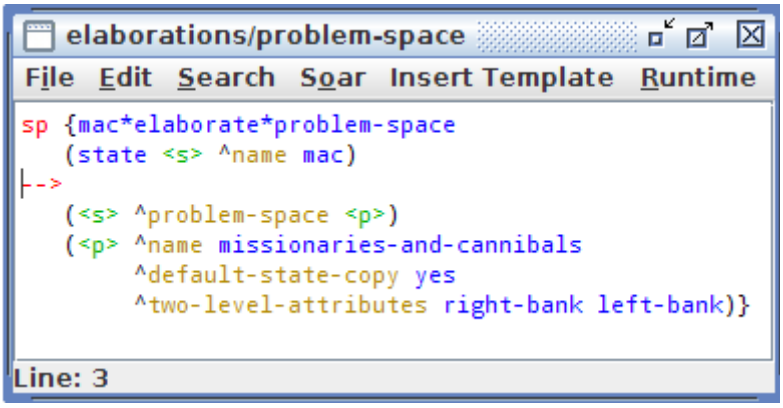

Figura 26 – Armazenamento em dois níveis.

2.2.1.c) Modificar a detecção de meta

Para gerar um estado de sucesso e não finalizar o agente, é necessário a modificação da detecção de meta conforme a Figura 27.

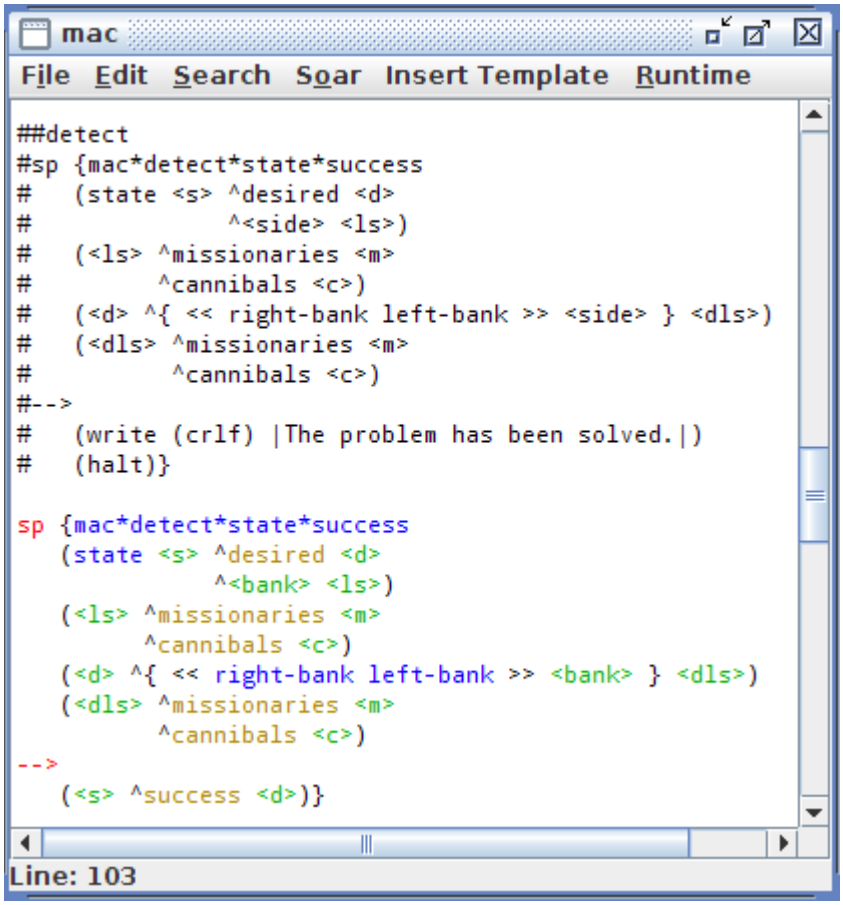

Figura 27 – Modificação de identificação de meta.

2.2.1.d) Modificar a detecção de falhas

Para gerar um estado de falha e não finalizar o agente, é necessário a modificação da regra de identificação de falha conforme a Figura 28.

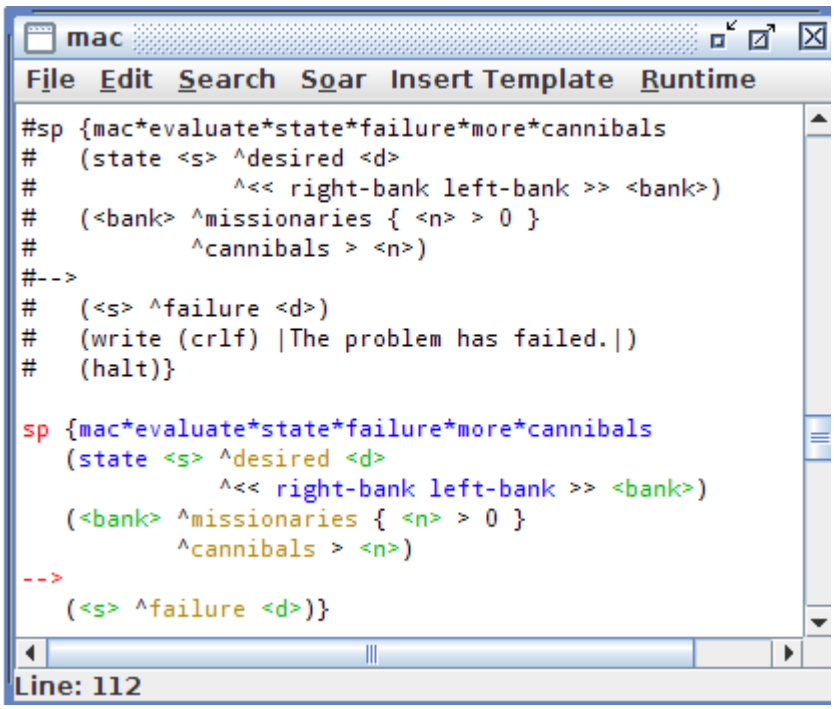

Figura 28 – Modificação de identificação de falha.

2.2.1.e) Adicionar a regra de detecção de estado duplicado

Esta regra tem o objetivo de identificar estados duplicados na pilha de estados e avaliar o mais recente dos duplicados como falha, Figura 29.

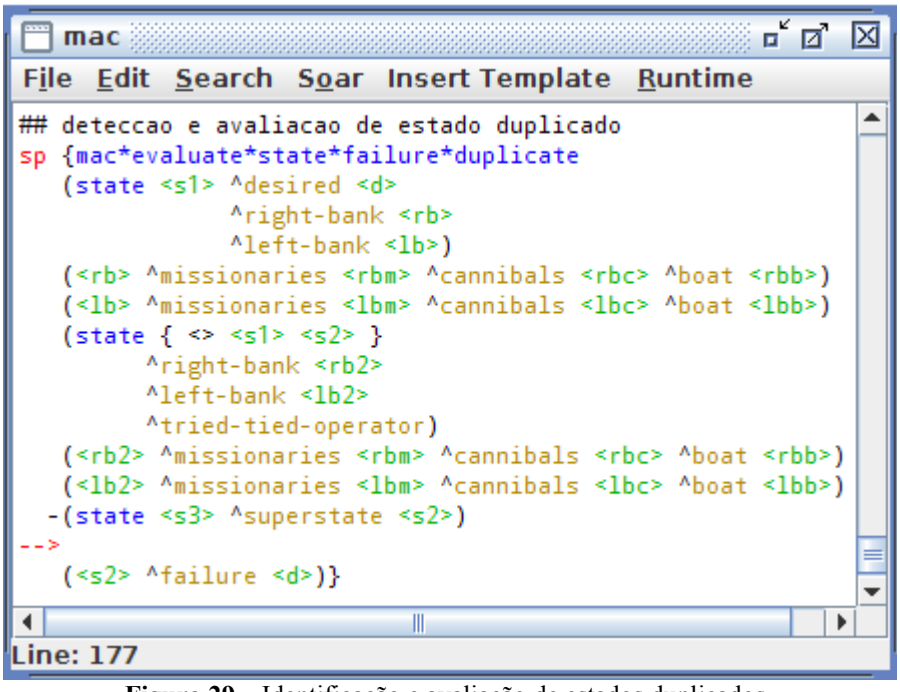

Figura 29 – Identificação e avaliação de estados duplicados.

2.2.1.f) Remover regras do último operador

As regras do último operador não são mais necessárias, pois o planejamento e a detecção de estados duplicados substituem o processamento onde estas regras eram designadas.
Após realizar estes seis itens, o programa é capaz de resolver o problema dos missionários e dos canibais por planejamento. Porém como aconteceu no problema do jarro de água, a solução pode

demorar a convergir. O próximo tópico abordará a aprendizagem *chunk,* voltada para o problema em questão.
A execução deste sub-tópico pode ser observado na Figura 30.

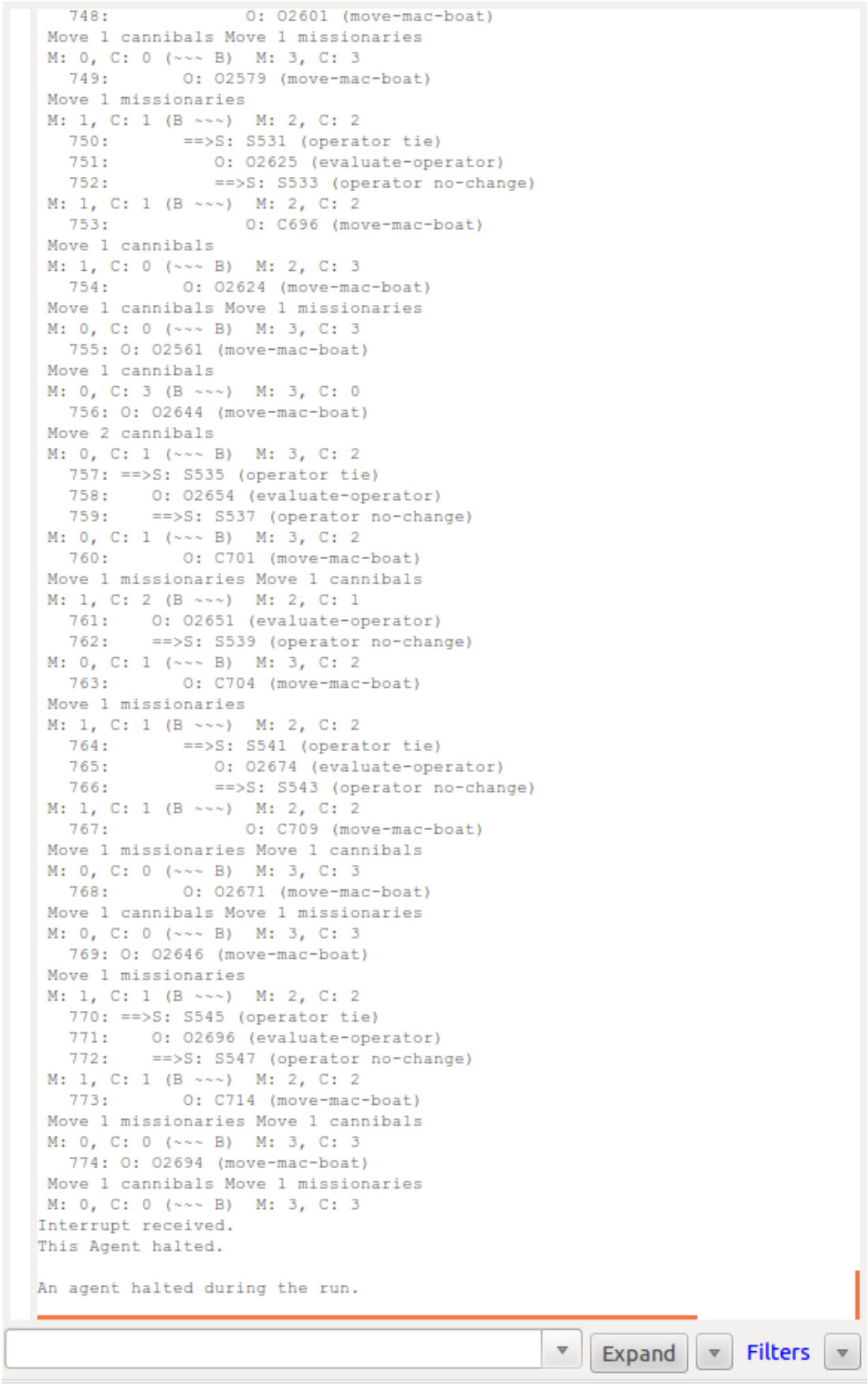

Figura 30 – Execução mac planejamento.

2.2.2 Problema sutil de aprendizagem *chunking*

Se for ativada a aprendizagem *chunk (learn –on,* por linha de comando*)* para a resolução do problema atual, a execução não irá chegar ao fim, Figura 31. Observe que além do comando *"learn –on"* foi utilizado o comando *"watch –chunk"* para análise da Figura 31.

```
 M: 3, C: 1 (B ~~~)  M: 0, C: 2
Firing chunk-34*d43*tie*2
    46:       O: C45 (move-mac-boat)
 Move 1 missionaries Move 1 cannibals
 M: 2, C: 0 (~~~ B)  M: 1, C: 3
Retracting chunk-34*d43*tie*2
    47: O: O105 (move-mac-boat)
 Move 1 missionaries
 M: 2, C: 1 (~~~ B)  M: 1, C: 2
Retracting chunk-34*d43*tie*2
Retracting chunk-37*d46*tie*3
Retracting chunk-36*d46*tie*2
Retracting chunk-38*d46*tie*4
Interrupt received.
This Agent halted.

An agent halted during the run.
```

Figura 31 – Execução com aprendizagem ativada.

A razão para a finalização da execução sem atingir o objetivo final é que a aprendizagem pode criar uma regra como exemplo:

"Se dois missionário, dois canibais e o barco estão a direita; rejeite o operador que move um missionário e o barco para esquerda"

Esta regra é aprendida pois leva a um sub-estado de falha, pois sobrariam dois canibais e um missionário a direita. Se o operador mover um canibal ao mesmo tempo que um canibal, esta regra aprendida seria evitada, não produzindo o estado de falha. No entanto as regras escritas até o momento, impossibilitam que o *chunk* aprenda isto. Uma solução satisfatória para este problema, é adicionar um teste para as regras de aplicação para observar o número de tipos que estão sendo modificados pelo operador, Figura 32.

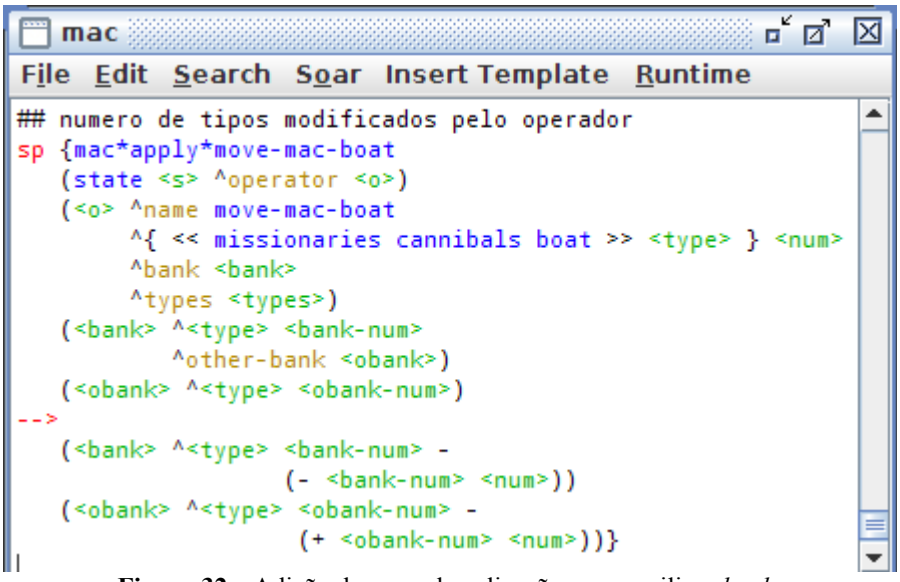

Figura 32 – Adição de regra de aplicação para auxiliar *chunk.*

Aplicando a regra na Figura 32 a aprendizagem *chunk* extrairá a seguinte regra:

"Se dois missionário, dois canibais e o barco estão a direita; rejeite o operador que move apenas um missionário e o barco para esquerda"

A Figura 33 apresenta a solução do problema com a aprendizagem *chunk* que converge de modo mais rápido.

```
Move 2 cannibals
M: 0, C: 1 (~~~ B)  M: 3, C: 2
   86:       O: O329 (move-mac-boat)
Move 1 missionaries
M: 1, C: 1 (B ~~~)  M: 2, C: 2
   87:       O: O337 (move-mac-boat)
Move 1 cannibals Move 1 missionaries
M: 0, C: 0 (~~~ B)  M: 3, C: 3
   88: O: O4 (move-mac-boat)
Move 2 cannibals
M: 3, C: 1 (~~~ B)  M: 0, C: 2
   89: O: O345 (move-mac-boat)
Move 1 cannibals
M: 3, C: 2 (B ~~~)  M: 0, C: 1
   90: O: O350 (move-mac-boat)
Move 2 cannibals
M: 3, C: 0 (~~~ B)  M: 0, C: 3
   91: O: O354 (move-mac-boat)
Move 1 cannibals
M: 3, C: 1 (B ~~~)  M: 0, C: 2
   92: O: O360 (move-mac-boat)
Move 2 missionaries
M: 1, C: 1 (~~~ B)  M: 2, C: 2
   93: O: O366 (move-mac-boat)
Move 1 cannibals Move 1 missionaries
M: 2, C: 2 (B ~~~)  M: 1, C: 1
   94: O: O371 (move-mac-boat)
Move 2 missionaries
M: 0, C: 2 (~~~ B)  M: 3, C: 1
   95: O: O374 (move-mac-boat)
Move 1 cannibals
M: 0, C: 3 (B ~~~)  M: 3, C: 0
   96: O: O380 (move-mac-boat)
Move 2 cannibals
M: 0, C: 1 (~~~ B)  M: 3, C: 2
Firing chunk-21*d58*tie*2
Firing chunk-17*d50*tie*2
   97: O: O382 (move-mac-boat)
Move 1 missionaries
M: 1, C: 1 (B ~~~)  M: 2, C: 2
Firing chunk-9*d30*tie*2
Retracting chunk-17*d50*tie*2
Retracting chunk-21*d58*tie*2
Firing chunk-19*d57*tie*2
Retracting chunk-9*d30*tie*2
   98: O: O390 (move-mac-boat)
Move 1 cannibals Move 1 missionaries
M: 0, C: 0 (~~~ B)  M: 3, C: 3
Retracting chunk-19*d57*tie*2
Interrupt received.
This Agent halted.

An agent halted during the run.
```

Figura 33 – Resolução com aprendizagem *chunk.*

2.2.3 Avaliações numéricas

Sem a aprendizagem do *chunk* o SOAR leva muito tempo para resolver certos problemas, devido a repetição de estados nas buscas. Uma abordagem para evitar uma busca longa é a adição de estados que não necessitam do estado final. Porém é necessário uma função de avaliação; mas para o problema em questão apenas é necessária a preferencia de estado que tem mais missionários e canibais em um lado desejado do rio. Assim, a avaliação é apenas o somatório do número de canibais e missionários do lado desejado do rio, com uma preferência elevada. Para viabilizar esta tarefa, é necessário a alteração da inicialização conforme a Figura 34.

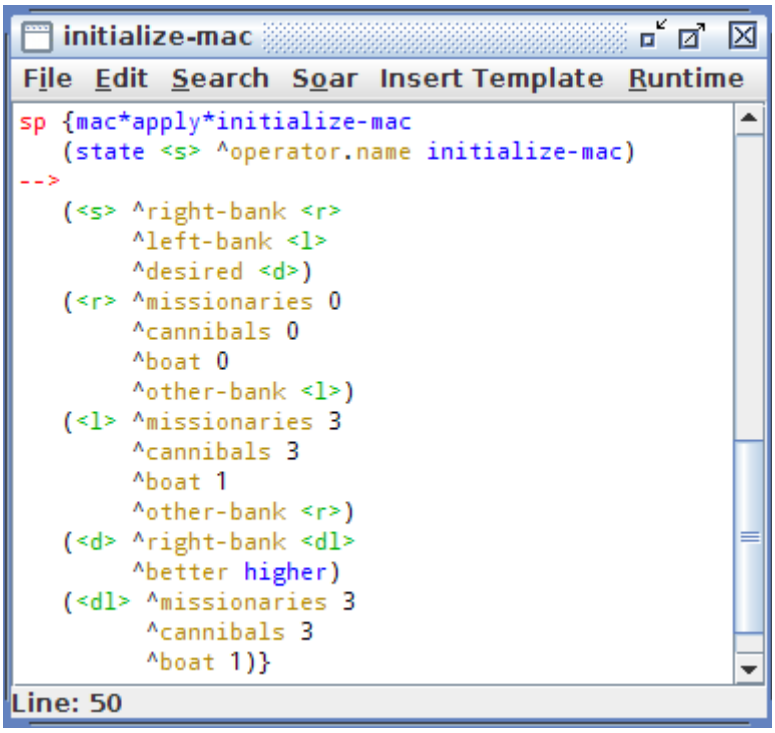

Figura 34 – Alterando inicialização para viabilizar avaliação numérica.

Existem regras de seleção que fazem todo o processamento de domínio independente, tais como comparar avaliações ou criar preferências.

A regra que computa a avaliação deve testar o estado desejado para determinar qual lado do rio deve ser avaliado. A regra também deve corresponder ao número de missionários e canibais do lado desejado do rio. A ação do operador é criar um acréscimo (de preferência) no estado que tem avaliação, Figura 35.

Note que a execução com a regra de avaliação numérica, executa significativamente mais rápido que a execução comum, sem a ativação da aprendizagem *chunk*. No entanto, quando a aprendizagem é utilizada coisas interessantes acontecem. Essa regra na verdade prejudica o desempenho, pois, a aprendizagem captura tanto aspectos bons quanto ruins, e a heurística pode estar errada. Então, quando essa avaliação não é utilizada, a aprendizagem baseia-se apenas em tarefas absolutas de acerto ou erro. A lição é que a aprendizagem de *chunk* é boa para resolução de problemas que as vezes não compensa inserir informações parcialmente corretas. A Figura 36 apresenta a execução da heurística junto com aprendizagem *chunk*.

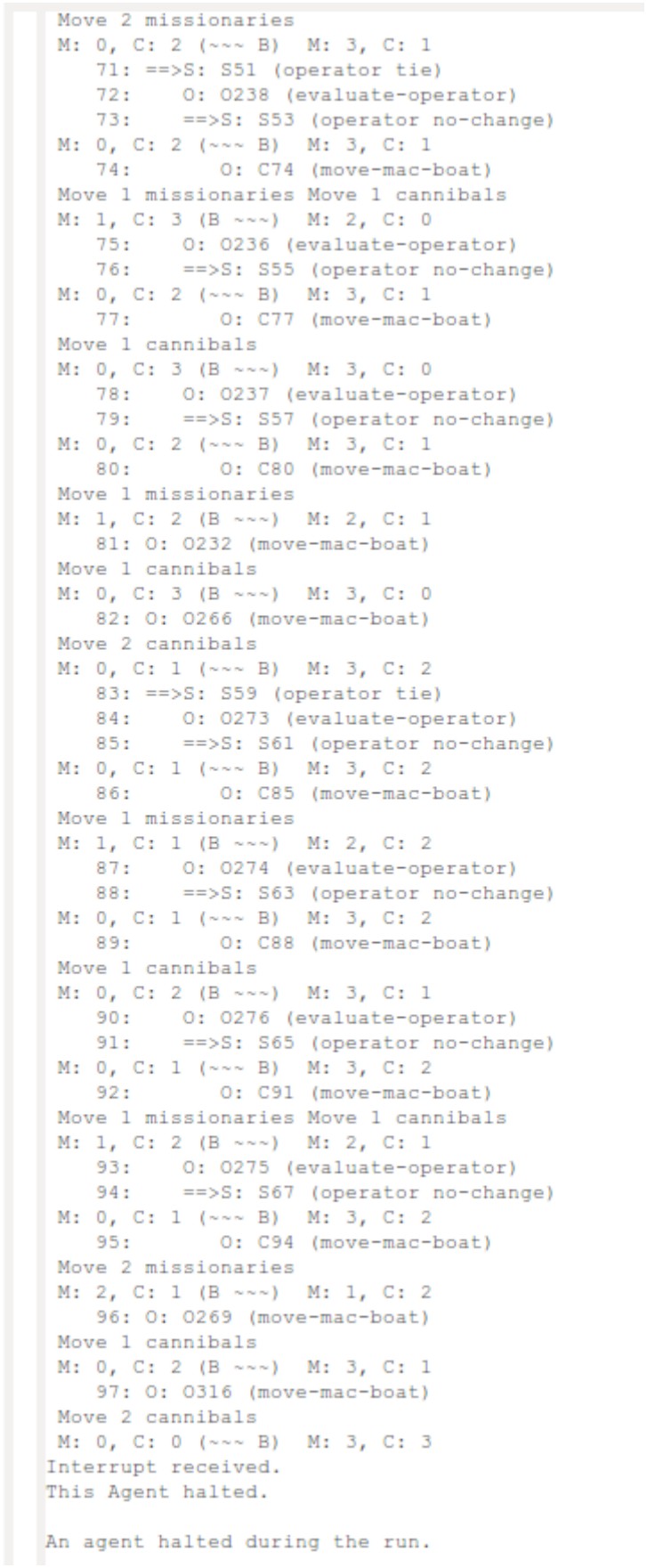

Figura 35 – Regra de avaliação numérica – sem aprendizagem *chunk.*

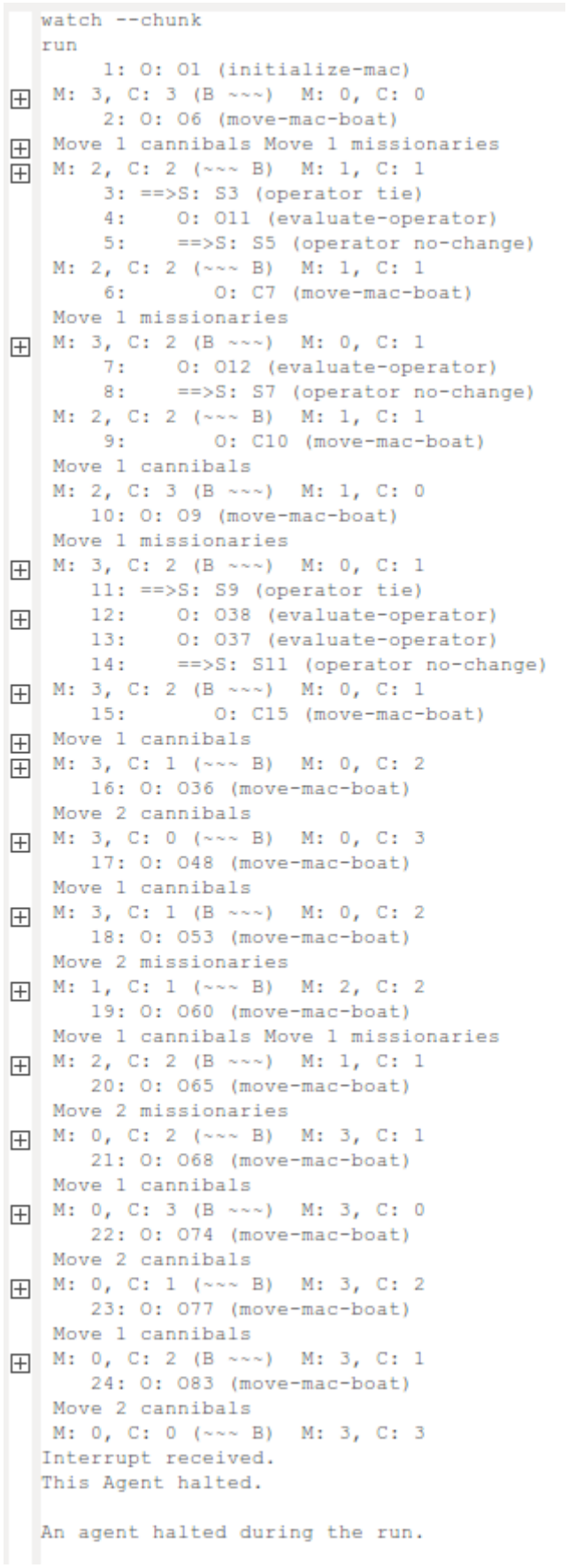

Figura 36 – Regra de avaliação numérica – com aprendizagem *chunk*.