Nome: Fabio Grassiotto

RA: 890441

Disciplina: IA941A, 1º S 2018

Aula 9 – Clarion: Controlando o WorldServer3D

**Objetivo**

Utilizar o Clarion para controlar uma aplicação externa por meio de código-exemplo em C# da aplicação DemoClarion.

**Atividade 1**

Na atividade 1, foi estudado em sala um exemplo de um controlador que utiliza o SOAR de forma reativa para a tomada de decisões. Foram constatadas as seguintes características relativas ao funcionamento do código Java do DemoJSOAR:

* Funcionamento do loop principal em Main.java:

O loop principal efetua inicialmente a leitura de um arquivo de regras do Soar.

A partir daí inicializa o ambiente de simulação e entra em um loop infinito que executa as regras lidas do arquivo soar passo a passo através da chamada soarBridge.step().

* Acesso ao WorldServer3D através do Proxy:

O método step() executa os seguintes passos:

* + Prepara o input link, criando o ambiente no WS3D.
  + Executa as regras do Soar.
  + Processa o output link, criando uma lista de comandos.
  + Envia os comandos para o WS3D utilizando o método processCommands() com a lista de comandos retornados anteriormente.

O método mstep é utilizado de forma similar, mas quebrando os passos nas fases de execução do Soar (micro-step).

* Leitura do Estado do WS3D: a leitura do estado do WS3D é realizada através dos métodos do SoarBridge:
  + prepareInputLink() - cria elementos de memória de trabalho WMEs relacionadas ao estado do ambiente.
  + processOutputLink() - envia comandos de saída do output link do Soar para controlar o WS3D.
* Como os dados do Soar são utilizados para controlar a criatura: através da execução do método processCommands().
* Arquivo soar-rules.soar:

São propostas regras com operadores distintos para cada passo da criatura. Regras de preferência são utilizadas para selecionar cada um dos operadores.

**Atividade 2**

Nesta atividade, é proposto o desenvolvimento de um conjunto de regras no SOAR para implementar uma estratégia deliberativa de comportamento para o controle da criatura. Esta estratégia deverá deliberar todas as ações intermediárias que são necessárias para que o objetivo seja atingido.

Os seguintes passos foram implementados nesta atividade:

1. **Alterações no código Java do DemoJSOAR:**

O código do método SoarBridge::PrepareInputLink foi alterado para:

* + Adicionar ao input link uma estrutura contendo um somatório dos objetivos dos leaflets da criatura sumarizados por cor, LEAFLET.
  + Adicionar ao input link uma estrutura com o somatório do objetivo corrente da criatura, ou seja, os totais do leaflet subtraidos das jóias já coletadas, TARGET.

// Create the creature leaflets in the input link.

List<Leaflet> leafletList = c.getLeaflets();

Identifier leaflet = CreateIdWME(creature, "LEAFLET");

int leafletRed = 0, leafletGreen = 0, leafletBlue = 0, leafletYellow = 0,

leafletMagenta = 0, leafletWhite = 0;

for (Leaflet l: leafletList)

{

// Get what to collect from leaflet.

HashMap<String, Integer> h = l.getWhatToCollect();

for (String key: h.keySet())

{

// Count all jewel ocurrences in the leaflets.

if (key.equals(COLOR\_RED)) {

leafletRed++;

} else if (key.equals(COLOR\_GREEN)) {

leafletGreen++;

} else if (key.equals(COLOR\_BLUE)) {

leafletBlue++;

} else if (key.equals(COLOR\_YELLOW)) {

leafletYellow++;

} else if (key.equals(COLOR\_MAGENTA)) {

leafletMagenta++;

} else {

leafletWhite++;

}

}

}

// Create leaflet in the inputlink. All three leaflets are summed up

// as a single list.

CreateFloatWME(leaflet, COLOR\_RED, leafletRed);

CreateFloatWME(leaflet, COLOR\_GREEN, leafletGreen);

CreateFloatWME(leaflet, COLOR\_BLUE, leafletBlue);

CreateFloatWME(leaflet, COLOR\_YELLOW, leafletYellow);

CreateFloatWME(leaflet, COLOR\_MAGENTA, leafletMagenta);

CreateFloatWME(leaflet, COLOR\_WHITE, leafletWhite);

// Initialize current target structure, leaflet - collected jewels.

int targetRed = leafletRed - collectedRed;

int targetGreen = leafletGreen - collectedGreen;

int targetBlue = leafletBlue - collectedBlue;

int targetYellow = leafletYellow - collectedYellow;

int targetMagenta = leafletMagenta - collectedMagenta;

int targetWhite = leafletWhite - collectedWhite;

Identifier target = CreateIdWME(creature, "TARGET");

CreateFloatWME(target, COLOR\_RED, targetRed);

CreateFloatWME(target, COLOR\_GREEN, targetGreen);

CreateFloatWME(target, COLOR\_BLUE, targetBlue);

CreateFloatWME(target, COLOR\_YELLOW, targetYellow);

CreateFloatWME(target, COLOR\_MAGENTA, targetMagenta);

CreateFloatWME(target, COLOR\_WHITE, targetWhite);

O código do método SoarBridge::ProcessOuputLink() foi alterado para:

* Processar comandos de saída do SOAR para adicionar e remover entidades em memória.
* Atualizar os dados de target corrente da criatura com a adição do método updateCollectedJewels().

case GET:

String thingNameToGet = null;

String colorToGet = null;

command = new Command(Command.CommandType.GET);

CommandGet commandGet = (CommandGet)command.getCommandArgument();

if (commandGet != null)

{

thingNameToGet = GetParameterValue("Name", idx);

if (thingNameToGet != null) commandGet.setThingName(thingNameToGet);

commandList.add(command);

colorToGet = GetParameterValue("Color", idx);

updateCollectedJewels(colorToGet);

}

break;

case ADD\_MEM:

memoryEntityInit = true;

memoryEntityName = GetParameterValue("Name", idx);

memoryEntityX = tryParseFloat(GetParameterValue("X", idx));

memoryEntityY = tryParseFloat(GetParameterValue("Y", idx));

break;

case REMOVE\_MEM:

memoryEntityInit = false;

memoryEntityName = null;

memoryEntityX = 0;

memoryEntityY = 0;

break;

void updateCollectedJewels(String color) {

if (color.equals(COLOR\_RED)) {

collectedRed++;

} else if (color.equals(COLOR\_GREEN)) {

collectedGreen++;

} else if (color.equals(COLOR\_BLUE)) {

collectedBlue++;

} else if (color.equals(COLOR\_YELLOW)) {

collectedYellow++;

} else if (color.equals(COLOR\_MAGENTA)) {

collectedMagenta++;

} else {

collectedWhite++;

}

}

1. **Criação de um novo conjunto de regras no arquivo planning.soar**

*2.1 Proposta 1: Implementação de look-ahead planning conforme o tutorial 5 do SOAR.*

Inicialmente o conjunto de regras foi alterado para integrar as regras *default,* copiando a pasta do mesmo nome dos exemplos do tutorial e adicionando uma regra para o carregamento do arquivo selection.soar. O software VisualSoar foi utilizado para edição das regras para possibilitar a divisão das regras em arquivos separados, simplificando o fluxo do trabalho.

Seguindo o tutorial 5 do Soar, os seguintes passos foram seguidos para implementar a estratégia deliberativa para solução do problema:

* Criação de um estado inicial (executado no arquivo initialize-planning.soar).
* Criação de condições para sucesso, definido como o momento em que a quantidade de jóias no knapsack se torna a mesma quantidade de jóias especificadas como target dos 3 leaflets.
* Criação de condição de falha. Utilizei para tanto a repetição de estados já presentes na pilha de estados.
* Reutilização de proposta de operadores exemplo.Reutilizei no caso os operadores *wander*, os dois operadores de memorização (*see entity)* e os operadores de movimentação e obtenção de jóias (*move e get jewel)*.
* Remoção de priorização de operadores, para provocar impasses e utilizar a simulação de operadores em etapas de simulação para resolver o problema.

Utilizando a estratégia acima, não obtive sucesso na execução do programa. Acredito que a falha ocorreu na implementação das rotinas de evaluation dos operadores. Considerei, no entanto, partir para uma nova proposta de implementação, utilizando planejamento combinado com uma proposta reativa.

* 1. *Proposta 2: Implementação de planejamento utilizando proposta reativa.*

Nesta segunda proposta procurei implementar regras para a movimentação da criatura procurando restringir as ações de acordo com o planejamento.

O planejamento a ser utilizado consiste em implementar regras para seguir o seguinte fluxo:

1. A criatura apenas está interessada em uma quantidade fixa de jóias de cada cor conforme especificado na estrutura do leaflet.
2. A criatura dispõe de 3 leaflets com quantidades de requisitos de jóias diferentes. A estratégia seguida será de coletar a soma dos requisitos dos 3 leaflets ao invés de coletar um leaflet por vez.
3. A criatura irá coletar as jóias percebidas pelo sistema visual, não será fornecida uma lista com conhecimento *a priori* antes da exploração do ambiente.
4. A criatura irá coletar uma jóia por vez e apenas irá reter um registro em memória do objetivo final.
5. A criatura irá coletar jóias ou comidas que encontrar bloqueando seu caminho, ainda que não façam parte do objetivo atual traçado.

Para tanto, implementei as seguintes propostas e aplicação de operadores:

* search-and-hold.soar

Um conjunto de regras que procura no ambiente da criatura a jóia mais próxima que faça parte do conjunto de objetivos de jóias a serem coletadas de acordo com a informação do leaflet.

Quando uma jóia com essa característica é encontrada, ela é adicionada como entidade da memória para posterior coleta através de um comando no output link. Note que o planejamento implementado aqui foi o de definir como objeto de busca apenas jóias que a criatura tenha interesse e coletar apenas uma jóia por vez.

As condições explicadas acima são implementadas como regras conforme abaixo:

# First condition is, is there a jewel in the visual field of the creature?

(<creature> ^SENSOR.VISUAL.ENTITY <entity>)

(<entity> ^TYPE <type> JEWEL)

(<entity> ^COLOR <color>)

(<entity> ^X <x>)

(<entity> ^Y <y>)

(<entity> ^NAME <name>)

# Second condition is, are there any entities in memory?

# The strategy is to only keep one entity in memory per round of search-and-get.

-(<creature> ^MEMORY.ENTITY <memoryEntity>)

# Third condition is, do we still need to get a jewel of that color?

(<creature> ^TARGET <target>)

(<target> ^<color> <tgtAmmount>)

(<target> ^<color> { <tgtAmmount> > 0 }) go.soar

…

-->

(<creature> ^MEMORY.ENTITY <memoryEntity>)

(<memoryEntity> ^X <x>)

(<memoryEntity> ^Y <y>)

(<memoryEntity> ^NAME <name>)

**(<ol> ^ADD\_MEM <command>)**

(<command> ^Name <name>)

(<command> ^X <x>)

(<command> ^Y <y>)

* go.soar

Regras para movimentação da criatura até a posição da jóia registrada em memória. A única particularidade desta regra é que apenas uma jóia por vez será coletada. A especificação das propriedades da jóia-alvo é passada como elemento de memória:

(<il> ^CREATURE.MEMORY <memory>)

(<memory> ^ENTITY <entity>)

(<entity> ^NAME <name>)

(<entity> ^X <entityX>)

(<entity> ^Y <entityY>)

* collect-mem.soar / collect.soar / eat.soar

Regras para a coleta da jóia em memória, jóias encontradas no caminho mas não em memória e comidas. Quando uma coleta é realizada de forma bem sucedida, é enviado um comando no output link para remover a entidade de memória.

sp {apply\*collect\*memory

(state <s> ^operator <o>

^io.input-link <il>

^io.output-link <ol>)

(<o> ^name collect\_mem)

(<o> ^parameter.NAME <entityName>)

(<o> ^parameter.COLOR <entityColor>)

(<il> ^CREATURE <creature>)

(<creature> ^MEMORY <memory>)

(<memory> ^ENTITY <entity>)

-(<ol> ^GET <something>)

-(<ol> ^REMOVE\_MEM <something>)

-->

(<memory> ^ENTITY <entity> -)

(<ol> ^GET <command>)

(<command> ^Name <entityName>)

(<command> ^Color <entityColor>)

**(<ol> ^REMOVE\_MEM <entityName>) # Remove entity from memory**

* go-delivery.soar

Regra que detecta que a coleta das jóias especificada no leaflet foi finalizada e direciona a criatura na direção do delivery spot.

sp {propose\*go\*to\*delivery

(state <s> ^name planning

^deliverySpot <deliverySpot>

^io.input-link <il>)

(<deliverySpot> ^X <dX> ^Y <dY>)

(<il> ^CREATURE.TARGET <target>)

(<target> ^Red <tRedAmmount> <= 0

^Green <tGreenAmmount> <= 0

^Blue <tBlueAmmount> <= 0

^Yellow <tYellowAmmount> <= 0

^Magenta <tMagentaAmmount> <= 0

^White <tWhiteAmmount> <= 0)

-->

(<s> ^operator <o> +)

(<o> ^name goToDeliverySpot)

(<o> ^parameterDelivery <delivery>)

(<delivery> ^X <dX>)

(<delivery> ^Y <dY>)

(write (crlf) | propose\*all\*collected |)}

1. **Resultados**

A simulação do sistema proposto no visual debugger do SOAR mostra que o fluxo das operações é correto até o final da coleta das jóias. O mesmo fluxo pode ser observado executando o sistema com o WS3D.

Uma dificuldade que foi observada na implementação foi não ter percebido inicialmente que as estruturas de memória de estado são resetados pelo código Java a cada execução de step do Soar. Para contornar esse problema, adicionei estruturas para manter no inputlink a jóia de objetivo corrente (MEMORY.ENTITY) e o somatório do objetivo (TARGET).

1. **Conclusões**

Ao longo do desenvolvimento deste trabalho cheguei às seguintes conclusões relativas à solução da Atividade 2 utilizando uma estratégia deliberativa:

* A documentação do Tutorial 5 do Soar é deficiente. Seriam necessários exemplos mais claros de como implementar o processo de planning no Soar.
* O sistema oferecido para simulação é mais intuitivo que o debugger do Soar. No entanto, existem alguns bugs que precisam ser evitados para seu uso.
* A estratégia deliberativa não é facilmente implementada no Soar. Para tanto, é necessário se utilizar de criação de estados de simulação internos.
* A implementação do DemoJSOAR utiliza uma rotina de reset() ao final do ciclo de simulação do SOAR (step). Uma consequência dessa implementação é que os elementos em memória do estado não são preservados entre ciclos. Para contornar este problema implementei comandos no output link para adicionar e remover elementos em memória para nortear a busca pelas jóias no ambiente.
* É possível implementar uma estratégia deliberativa utilizando regras, sem se utilizar o processo de look-ahead do SOAR. Para tanto, uma estratégia deve ser traçada desde o início do processo de coleta.
* Para uma criatura no mundo real, a estratégia de uso de regras me parece estar mais próximo de como a tomada de decisão ocorre para seres humanos: com informação incompleta e estimação de caminhos à medida que o processo de busca é realizado. Baseei nesta suposição a minha estratégia de implementação final.