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 7

1 Atividade 1

Este tutorial visa a aplicação do conhecimento obtido em atividades anteriores, sobre arquitetura cognitiva SOAR, no mundo virtual *"WorldServer3D"*. A atividade 1, aborda detalhadamente a análise dos códigos dos projetos: *"WorldServer3D", "DemoSOAR"* e *"WS3DProxy"*.
Analisando a classe *SimulationSOAR* presente no projeto *DemoSOAR,* pode-se observar que é utilizada uma chamada de sistema via *java* para identificar dois itens. O primeiro é a identificação do sistema operacional que a aplicação está sendo executado no momento. O segundo item, é a identificação da arquitetura que o sistema operacional (SO) está rodando, ou seja, mesmo que a estação de trabalho seja de 64 *bits* mas o sistema operacional instalado seja de 32 *bits,* o retorno da arquitetura será referente a instalação do SO, Figura 1.

```
NativeUtils.setLibraryPath(".");
System.out.println("OS:" + System.getProperties().getProperty("os.name")
        + " Architecture:" + System.getProperties().getProperty("os.arch"));

String osName = System.getProperty("os.name").toLowerCase(Locale.ENGLISH);
String osArch = System.getProperty("os.arch").toLowerCase(Locale.ENGLISH);
```

Figura 1 – Identificando SO e arquitetura.

Um exemplo para a execução do código presente na Figura 1 pode ser o caso de uma estação de trabalho contendo um SO Linux Ubuntu de 64 *bits*. As variáveis *osName* e *osArch* irão conter respectivamente Linux e amd64.
Esta tarefa é necessária devido a necessidade da importação de arquivos para a configuração da interface de comunicação do *java* com o controlador SOAR. Identificado o SO e a arquitetura, é possível carregar as bibliotecas e *dlls* específicas e necessárias para a criação desta interface de comunicação. Esta configuração é realizada utilizando a classe *NativeUtils*. Com a função *"setLibraryPath(".")"* é setado o diretório onde serão importados os arquivos (*.lib, .dll e/ou .so*). Os arquivos são importados através da função *"loadFileFromJar("path")"*, que os importa para o diretório, escolhido anteriormente, utilizando as classes padrões de manipulação de arquivos presentes na *java.io* respectivamente *InputStream* e *OutputStream,* Figura 2. Para o exemplo citado anteriormente, os caminhos dos arquivos importados seriam respectivamente: *"/linux64/libSoar.so", "/linux64/libJava_sml_ClientInterface.so"* e *"/soar-rules.soar",* Figura 3.

```
// Open and check input stream
InputStream is = NativeUtils.class.getResourceAsStream(path);
if (is == null) {
    throw new FileNotFoundException("File " + path
            + " was not found inside JAR.");
}

// Open output stream and copy data between source file in JAR
OutputStream os = new FileOutputStream(temp);
try {
    while ((readBytes = is.read(buffer)) != -1) {
        os.write(buffer, 0, readBytes);
    }
} finally {
```

Figura 2 – Modo de importação dos arquivos de configuração.

```
} else if (osName.contains("nix") || osName.contains("nux")) {
    if (osArch.contains("64")) {
        System.out.println("Linux 64 bits");
        NativeUtils.loadFileFromJar("/linux64/libSoar.so");
        NativeUtils.loadFileFromJar("/linux64/libJava_sml_ClientInterface.so");
    } else {
        System.out.println("Linux 32 bits");
        NativeUtils.loadFileFromJar("/linux32/libSoar.so");
        NativeUtils.loadFileFromJar("/linux32/libJava_sml_ClientInterface.so");
    }
} else {
```

Figura 3 – Importação dos arquivos de configuração.

Após cada importação é realizado o carregamento do arquivo importado para a execução do *java*, utilizando o comando *"System.load(temp.getAbsolutePath())"*.
Ainda na classe *SimulationSOAR* são realizadas as configurações para a execução da classe *SimulationTask* onde se inicializa a comunicação com o *WorldServer3D* utilizando as regras do SOAR. Em seguida é realizada a criação do avatar que utilizada as regras do operador SOAR e também é iniciada a classe *SoarBridge,* para a execução do controlador SOAR. Então, seguindo a execução inicia-se uma iteração de *loop* infinito para a realizar as atividades propostas pelo código presente no arquivo *soar-rules.soar,* através da execução da função *runSimulation()* presente na classe *SimulationTask,* Figura 4.

```
SimulationTask simulationTask = new SimulationTask();
simulationTask.initializeEnviroment(Boolean.FALSE);
simulationTask.initializeCreatureAndSOAR(soarRulesPath, true,
        soarDebuggerPath, soarDebuggerPort);
// Run Simulation until some criteria was reached
Thread.sleep(3000);
while (true) {
    simulationTask.runSimulation();
    Thread.sleep(100);
}
```

Figura 4 – Inicia as atividades no *WorldServer3D*.

A classe *SimulationTask* possui um objeto, chamado *proxy*, instanciado com a classe *WS3DProxy*. Através do mesmo, é possível realizar operações no mundo virtual (execução do projeto *WorldServer3D*), dentre as operações pode-se citar a criação da criatura (que retorna o objeto

Creature) e adiquirir ("getar") o objeto *World*, o mundo virtual corrente, Figura 5.

```
c = proxy.createCreature(100, 100, 0);
w = proxy.getWorld();
c.start();
w.grow(1);
```

Figura 5 – Manipulação através do *WS3DProxy*.

A classe *SimulationTask* também possui um objeto para a comunicação com o controlador SOAR, o mesmo é declarado como *soarBridge* e instanciado com a classe *SoarBridge*. Utilizando este objeto é possível realizar a interação com controlador SOAR. Isto é realizado a partir do *loop* infinito presente na Figura 4, de modo que a cada iteração são realizadas as seguintes tarefas, Figura 6:

- Através do objeto *Creature* são adquiridos todos os estados da criatura no *WorldSever3D;*
- Possuindo as informações, são preparadas as estruturas a serem inseridas no controlador SOAR, informações de coisas (*things*) presentes no mundo virtual e informações referentes a criatura;
- Preparada as estruturas a serem inseridas no controlador SOAR, são enviadas através do objeto *soarBridge*;
- Após o envio para o controlador SOAR, são obtidas as respostas do controlador;
- As respostas são processadas e gerando decisões enviadas ao mundo virtual.

```
public void runSimulation() throws SoarBridgeException, NullPoin
    if (soarBridge != null) {
        c.updateState();
        //Status currentStatus = c.getState();
        List<Thing> v = c.getThingsInVision();
        logger.info("Objetos no Ambiente: " + v.size());
        System.out.println("Objetos no Ambiente: " + v.size());
        for (Thing t : v) {
            logger.info(t.toString());
        }
        if (c != null) {
            // Prepare Creature To Simulation
            prepareAndSetupCreatureToSimulation(c, v);
            // Run simulation
            soarBridge.runSimulation();
            // Process Responde Commands
            processResponseCommands();
        } else {
            throw new NullPointerException("There are no creatur
        }
```

Figura 6 – Ciclo de execução.

1.1 Update State

A primeira chamada para a atualização é a da função presente na classe *Creature* (objeto c), *c.updateState()*. A atualização é realizada utilizando uma função estática da classe *CommandUtility*, utilizando a chamada *"CommandUtility.sendGetCreatureState(myselfName)"*, onde a variável *myselfName* é o ID da criatura. Através desta chamada é possível obter todas as informações da criatura e objetos em sua visão (*c.getThingsInVision()*), Figura 7 e 8.

```
StringTokenizer st = CommandUtility.sendGetCreatureState(myselfName);

//////////////////Creature data:
if (!st.hasMoreTokens()) {
    logger.log(Level.SEVERE, "Error - myName missing!");
} else {
    command = st.nextToken();
    myselfName = command; //string
}
if (!st.hasMoreTokens()) {
    logger.log(Level.SEVERE, "Error - index is missing!");
} else {
    index = st.nextToken();
}
if (!st.hasMoreTokens()) {
```

Figura 7 – Atualizando informações sobre o mundo virtual – I.

```
public static synchronized StringTokenizer sendGetCreatureState(String robotNameID)
    String controlMessage = "getcreaturestate " + robotNameID;
    return sendCmdAndGetResponse(controlMessage);
}
```

Figura 8 – Atualizando informações sobre o mundo virtual – II.

1.2 Prepare Creature to Simulation

Em seguida é feita a preparação da estrutura da creatura para a simulação, através do comando: *"prepareAndSetupCreatureToSimulation(c, v)"*. Neste método, Figura 9, ocorre a preparação da estrutura da creatura através das coisas *(things)* passadas como parâmetros. Além das informações do vetor *"things"*, também se obtem informações sobre o combustível, através do objeto *creature (creature.getFuel())*, passado como parâmetro.

```
private void prepareAndSetupCreatureToSimulation(Creature creature, List<Thing> things)
    if (creature != null) {
        // Create Visual Sensor
        VisualSensor visualSensor = new VisualSensor();
        visualSensor.setSensorReadings(things);

        // Create Fuel Sensor
        FuelSensor fuelSensor = new FuelSensor();
        FuelSensorReadings fuelSensorReadings = new FuelSensorReadings();
        fuelSensorReadings.setCurrentFuel(creature.getFuel());
        fuelSensor.setSensorReadings(fuelSensorReadings);

        // Put things in SOAR output interface to start simulation
        SimulationRobot simulationCreature = new SimulationRobot(creature);
        simulationCreature.addSensorsAvailable(visualSensor);
        simulationCreature.addSensorsAvailable(fuelSensor);

        // Setup StackHolder for run simulation
        soarBridge.setupStackHolder(StakeholderType.CREATURE, simulationCreature);
    } else {
        throw new IllegalArgumentException("Arguments ares null");
    }
}
```

Figura 9 – Praparando e setando a creatura para execução.

Após organizar as informações, todas as estruturas são unidas na classe *SimulationRobot* para serem passadas a interface de saída SOAR. Utilizando o objeto *soarBridge (soarBridge.setupStackHolder)* todos os elementos são inseridos na memória de trabalho. Onde cada elemento é criado com um elemento da memória de trabalho (WME), dependendo do tipo: FUEL ou VISUAL, Figura 10 e 11 respectivamente.

```
case FUEL: {
    Identifier fuel = agent.CreateIdWME(creatureSensor, "FUEL");
    FuelSensorReadings sensorReadings = (FuelSensorReadings)
            creatureParameter.getSensorHandler(SensorType.FUEL).getSensorReadings();
    if (sensorReadings != null) {
        agent.CreateFloatWME(fuel, "VALUE", sensorReadings.getCurrentFuel());
    }
}
break;
```

Figura 10 – Criando elementos na memória de trabalho - FUEL.

```
case VISUAL: {
    Identifier visual = agent.CreateIdWME(creatureSensor, "VISUAL");
    List<Thing> thingsList = (List<Thing>)
            (creatureParameter.getSensorHandler(SensorType.VISUAL).getSensorReadings());
    if (thingsList != null) {
        for (int thingsInSensorCount = 0; thingsInSensorCount < thingsList.size();
                thingsInSensorCount++) {
            Identifier entity = agent.CreateIdWME(visual, "ENTITY");
            Thing t = thingsList.get(thingsInSensorCount);
            agent.CreateFloatWME(entity, "DISTANCE", GetGeometricDistanceToCreature(
                    t.getX1(), t.getY1(), t.getX2(), t.getY2(),
                    creatureParameter.getPosition().getX(),
                    creatureParameter.getPosition().getY()
                    ));
            agent.CreateFloatWME(entity, "X", t.getX1());
            agent.CreateFloatWME(entity, "Y", t.getY1());
            agent.CreateFloatWME(entity, "X2", t.getX2());
            agent.CreateFloatWME(entity, "Y2", t.getY2());
            agent.CreateStringWME(entity, "TYPE",
                    SimulationItem.getItemTypeFromThingCategory(t.category).toString());
            agent.CreateStringWME(entity, "NAME", t.getName());
            agent.CreateStringWME(entity, "COLOR", Constants.getColorName(
                    t.getMaterial().getColor()
                    ));
        }
    }
}
break;
```

Figura 11 – Criando elementos na memória de trabalho - VISUAL.

Até este ponto, tudo o esforço foi voltado a preparação para um ciclo de simulação do controlador SOAR. No próximo item, é apresentado a simulação e em seguida o processamento das respostas obtidas pelo controlador.

1.3 Run Simulation

Após preparar a simulação, basta executá-la através do objeto *soarBridge (soarBridge.runSimulation()),* Figura 6. O agente executa a simulação através das informações preparadas anteriormente, Figura 12. Em seguida, verifica-se a ocorrência de erros, Figura 13.

```
public void runSimulation() throws SoarBridgeException {
    try {
        if (agent != null) {
            agent.RunSelfTilOutput();
            checkForKernelOrAgentError();
        }
    } catch (Exception e) {
        logger.error("Error while creating SOAR Kernel", e);
        throw new SoarBridgeException("Error while Creaing SOAR Bridge");
    }
}
```

Figura 12 – Executando a simulação.

```
private void checkForKernelOrAgentError() throws SoarBridgeException {
    if (kernel != null && kernel.HadError()) {
        throw new SoarBridgeException(kernel.GetLastErrorDescription());
    }

    if (agent != null && agent.HadError()) {
        throw new SoarBridgeException(agent.GetLastErrorDescription());
    }
}
```

Figura 13 – Verificação de erro.

1.4 Process Responde Commands

Seguindo a execução da Figura 6, a última etapa é o processamento da saída da simulação: *processResponseCommands()*. A cada simulação executada, gera-se um número de comandos que são obtidos através do método: *agent.GetNumberCommands()* e cada comando pode ser obtido pelo método: *agent.GetCommand(i),* onde *i* é o identificador de cada comando iniciando em 0 até o "número de comandos – 1". A partir deste métodos é possível obter a lista de comandos gerados na simulação: *soarBridge.getReceivedCommands(),* Figura 14. Cada comando pode possuir um tipo: MOVE, GET ou EAT.

```
private void processResponseCommands() throws SoarBridgeException, CommandExecException {
    // get simulation results
    ArrayList<SoarCommand> commandList = soarBridge.getReceivedCommands();

    if (commandList != null) {
        for (SoarCommand command : commandList) {
            switch (command.getCommandType()) {
                case MOVE:
                    processMoveCommand((SoarCommandMove) command.getCommandArgument());
                    break;

                case GET:
                    processGetCommand((SoarCommandGet) command.getCommandArgument());
                    break;

                case EAT:
                    processEatCommand((SoarCommandEat) command.getCommandArgument());
                    break;

                default:
                    // Do nothing
                    break;
            }
        }
    }
```

Figura 14 – Processando comandos da simulação.

Cada tipo de comando gera um tipo de ação no mundo virtual. Para o comando MOVE, a ação sempre será em relação a posição do avatar, Figura 15.

```
private void processMoveCommand(SoarCommandMove soarCommandMove) throws CommandExecExcept
    if (soarCommandMove != null) {
        if (soarCommandMove.getX() != null && soarCommandMove.getY() != null) {
            CommandUtility.sendGoTo("0", soarCommandMove.getRightVelocity(),
                    soarCommandMove.getLeftVelocity(),
                    soarCommandMove.getX(), soarCommandMove.getY());
        } else {
            CommandUtility.sendSetTurn("0", soarCommandMove.getLinearVelocity(),
                    soarCommandMove.getRightVelocity(),
                    soarCommandMove.getLeftVelocity());
        }
    } else {
        throw new NullPointerException("soarCommand is null");
    }
}
```

Figura 15 – Ação MOVE.

A ação referente ao comando GET, está relacionada com pegar algo, no caso possível atual do sistema pode-se citar uma jóia, Figura 16.

```
private void processGetCommand(SoarCommandGet soarCommandGet) t
    if (soarCommandGet != null) {
        c.putInSack(soarCommandGet.getThingName());
    } else {
        throw new NullPointerException("soarCommand is null");
    }
}
```

Figura 16 – Ação GET.

A última ação possível, EAT, está relacionada com comer frutas ao alcance do avatar, Figura 17.

```
private void processEatCommand(SoarCommandEat soarCommandEat) th
    if (soarCommandEat != null) {
        c.eatIt(soarCommandEat.getThingName());
    } else {
        throw new NullPointerException("soarCommand is null");
    }
}
```

Figura 17 – Ação EAT.

1.5 Analisando as regras do controlador SOAR

Analisando o arquivo: *"soar-rules.soar"* presente no diretório *rules* do projeto *DemoSOAR,* é possível verificar que o mesmo desabilita o aprendizado e o aprendizado por memória episódica, Figura 18.

```
################################  CONFIGURATION ################################
watch 5
learn --off
epmem --set learning off
```

Figura 18 – Configuração inicial.

Se o agente está sem ação *(impasse no-change),* devido a uma interrupção em sua trajetória, ele

deve continuar andando para encontrar comida ou jóias. As regras presentes nas Figuras 19 e 20 propõe e aplicam a tarefa de desviar (WANDER). Observe que o aavatar *(creature)* sempre irá virar para à esquerda, devido a manter travada a roda da esquerda e acelerar a roda da direita.

```
sp {propose*wander
   (state <s> ^attribute state
              ^impasse no-change
              ^superstate <ss>)
   (<ss> ^io.input-link <il>)
   (<ss> ^superstate nil)
   (<il> ^CREATURE <creature>)
   (<creature> ^SENSOR.VISUAL <visual>)
-->
   (<ss> ^operator <o> +)
   (<o> ^name wander)}
```

Figura 19 – *WANDER propose*.

```
sp {apply*wander
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <ol>)
   (<o> ^name wander)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 0)
   (<command> ^VelR 2)
   (<command> ^VelL 0)}
```

Figura 20 – *WANDER apply*.

Para o agente se mover para a comida é proposto o operador *MOVE FOOD*. As Figuras 21 e 22 apresentam respectivamente a proposição e aplicação deste operador.
Note que na Figura 21 é realizado o cálculo da distância entre a comida e a criatura. Este cálculo pode ser facilmente interpretado pela quarta linha:

"(<food> ^distance (sqrt (+ ((- <creaturePositionX> <entityInMemoryPositionX>) (- <creaturePositionX> <entityInMemoryPositionX>)) (* (- <creaturePositionY> <entityInMemoryPositionY>) (- <creaturePositionY> <entityInMemoryPositionY>)))))"*

Esse cálculo é idêntico a calcular a distância euclidiana D de dois pontos (X1,Y1) e (X2,Y2):

$$D = \sqrt{((X1 - X2)^2 + (Y1 - Y2)^2)}$$

```
sp {propose*move*food
   (state <s> ^io.input-link <il>)
   (<il> ^CREATURE <creature>)
   (<creature> ^MEMORY <memory>)
   (<memory> ^ENTITY <entityInMemory>)
   (<creature> ^POSITION <creaturePosition>)
   (<creaturePosition> ^X <creaturePositionX>)
   (<creaturePosition> ^Y <creaturePositionY>)
   (<entityInMemory> ^TYPE FOOD)
   (<entityInMemory> ^X <entityInMemoryPositionX>)
   (<entityInMemory> ^Y <entityInMemoryPositionY>)
   (<entityInMemory> ^NAME <entityInMemoryName>)
   (<creature> ^PARAMETERS.MINFUEL <minFuel>)
-->
   (<s> ^operator <o> +)
   (<o> ^name moveFood)
   (<o> ^parameter <food>)
   (<food> ^distance (sqrt (+ (* (- <creaturePositionX> <entityInMemoryPositionX>)
                                 (- <creaturePositionX> <entityInMemoryPositionX>))
                              (* (- <creaturePositionY> <entityInMemoryPositionY>)
                                 (- <creaturePositionY> <entityInMemoryPositionY>))
                              )))
   (<food> ^X <entityInMemoryPositionX>)
   (<food> ^Y <entityInMemoryPositionY>)
   (<food> ^NAME <entityInMemoryName>)
   (<o> ^parameterFuel <minFuel>)}
```

Figura 21 – *MOVE FOOD propose*.

```
sp {apply*move*food
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^input-link <il>)
   (<io> ^output-link <ol>)
   (<o> ^name moveFood)
   (<o> ^parameter <food>)
   (<food> ^X <x>)
   (<food> ^Y <y>)
   (<food> ^NAME <entityInMemoryName>)
   (<il> ^CREATURE <creature>)
   (<creature> ^MEMORY <memory>)
   (<memory> ^ENTITY <entityInMemory>)
   (<entityInMemory> ^NAME <entityInMemoryName>)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 1)
   (<command> ^VelR 1)
   (<command> ^VelL 1)
   (<command> ^X <x>)
   (<command> ^Y <y>)}
```

Figura 22 – MOVE FOOD *apply*.

Várias outras regras estão presente no arquivo *"soar-rules.soar"* com o objetivo de mover a criatura através do mundo para pegar comida e jóias. Existem outras regras com o intuito de remover inconsistências como a de buscar uma comida ou jóia que já não existe.

No próximo item deste relatório, são propostas abordagens com o intuito de criar um agente capaz de seguir um *leaflet,* que viabilize a competição entre duas criaturas. O *leaflet* é uma sequência de objetivos que a criatura deve cumprir para ganhar pontos de acordo com o *leaflet* cumprido. Cada

criatura pode conter até três *leaflets*.

2 Atividade 2

O desenvolvimento da atividade 2 está disponibilizado no ambiente de entrega de atividades (aluno14-Mateus Neves Barreto - Aula7). Na mesma seção, encontra-se um video com a execução do agente inteligente com as regras modificadas para perseguir os *leaflets* (arquivo: *soar-rules.soar*), Figura 23.

```
sp {apply*move*jewel
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^input-link <il>)
   (<io> ^output-link <ol>)
   (<o> ^name moveJewel)
   (<o> ^parameter <jewel>)
   (<jewel> ^X <x>)
   (<jewel> ^Y <y>)
   (<jewel> ^NAME <entityInMemoryName>)
   (<il> ^CREATURE <creature>)
   (<creature> ^MEMORY <memory>)
   (<memory> ^ENTITY <entityInMemory>)
   (<entityInMemory> ^NAME <entityInMemoryName>)
   (<creature> ^LEAFLET <leaflet>)
   (<leaflet> ^ITEM <item>)
   (<item> ^COLOR <colorItem>)
   (<jewel> ^COLOR <colorItem>)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 1)
   (<command> ^VelR 1)
   (<command> ^VelL 1)
   (<command> ^X <x>)
   (<command> ^Y <y>)}
```

Figura 23 – Modificação da regra - *apply-move-jewel*.

Observe que foi criado uma estrutura dentro do criatura chamada *LEAFLET,* esse implementação pode ser observada a partir do código presente na Figura 24.

```
if (!creatureParameter.getLeafletList().isEmpty()) {
    creatureLeaflet = agent.CreateIdWME(creature, "LEAFLET");
    for (Leaflet leaflet : creatureParameter.getLeafletList()) {
        HashMap whatToCollect = leaflet.getWhatToCollect();
        for (Iterator it = whatToCollect.keySet().iterator(); it.hasNext();) {
            String numSearch = (String) it.next();
            System.out.println(numSearch + " = " + whatToCollect.get(numSearch));
            for (int i = 0; i < Integer.parseInt(whatToCollect.get(numSearch).toString()); i++) {
                Identifier entity = agent.CreateIdWME(creatureLeaflet, "ITEM");
                agent.CreateStringWME(entity, "COLOR", numSearch);
                if (!corToIdentifiers.containsKey(numSearch)) {
                    corToIdentifiers.put(numSearch, new ArrayList<Identifier>());
                }
                corToIdentifiers.get(numSearch).add(entity);
            }
        }
    }
    leafletListSoar = (Vector<Leaflet>) creatureParameter.getLeafletList().clone();
}
```

Figura 24 – Criação da estrutura LEAFLET dentro da criatura.

Note que apenas são iseridas as cores. A observação da estrutura criada pode ser feita através do SoarJavaDebburger, Figura 25.

```
(I2 ^CREATURE C3)
print C3
(C3 ^LEAFLET L1 ^MEMORY M1 ^PARAMETERS P3 ^POSITION P4 ^SENSOR S6)
print L1
(L1 ^ITEM I12 ^ITEM I11 ^ITEM I9 ^ITEM I7 ^ITEM I5 ^ITEM I4)
```

Figura 25 – *Input* SOAR.

CONCLUSÃO

Esse tutorial esclarece aparentemente as possíveis dúvidas que ficam sobre a arquitetura SOAR. Muito interessante devido ao desafio, que é possível tentar de várias formas de implementação do agente SOAR para resolver o problema. A busca do agente SOAR para os *leaflets* não para até a finalização da aplicação, o mesmo sempre irá preferir jóias que estejam presentes em seus objetivos de busca.