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 8 e 9

1 Utilizando o ACS – Básico

Este relatório abordará os tutorias sobre o CLARION, a moderna arquitetura de inteligência cognitiva híbrida. A versão que será abordada é a 6.1, a mais recente (6.1.0.7). Esta arquitetura é feita para linguagem C#, para os tutoriais será utilizado o ambiente de desenvolvimento *"mono develop"* que auxilia no desenvolvimento (C#) no sistema operacional Linux.
Após a instalação do ambiente de desenvolvimento, o primeiro *hello world* é sugerido. Uma simples utilização da arquitetura já é notada no primeiro código, Figura 1.

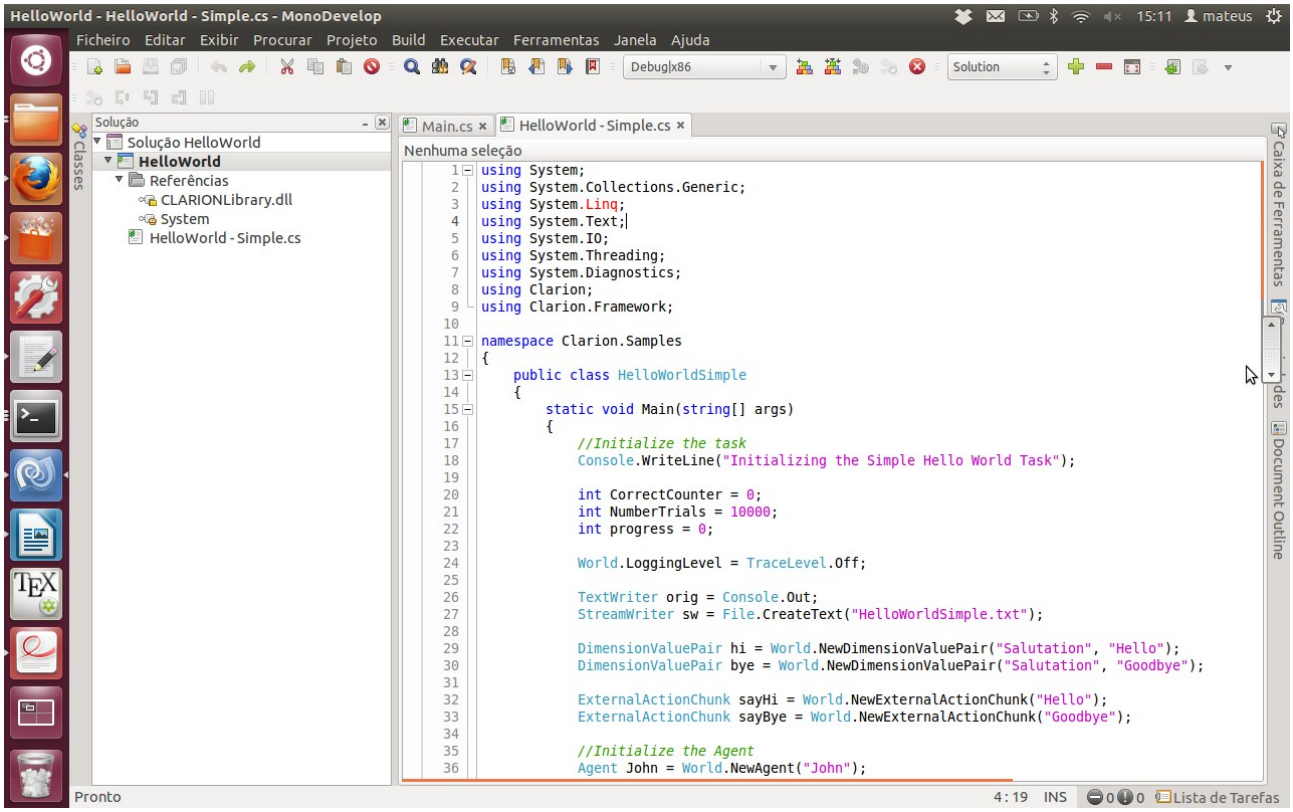

Figura 1 – Primeiro HelloWorld - I.

A execução do código presente na Figura 1 gera duas saídas, a primeira através do console, e a segunda em um arquivo *HelloWorldSimple.txt*, respectivamente Figura 2 e 3.

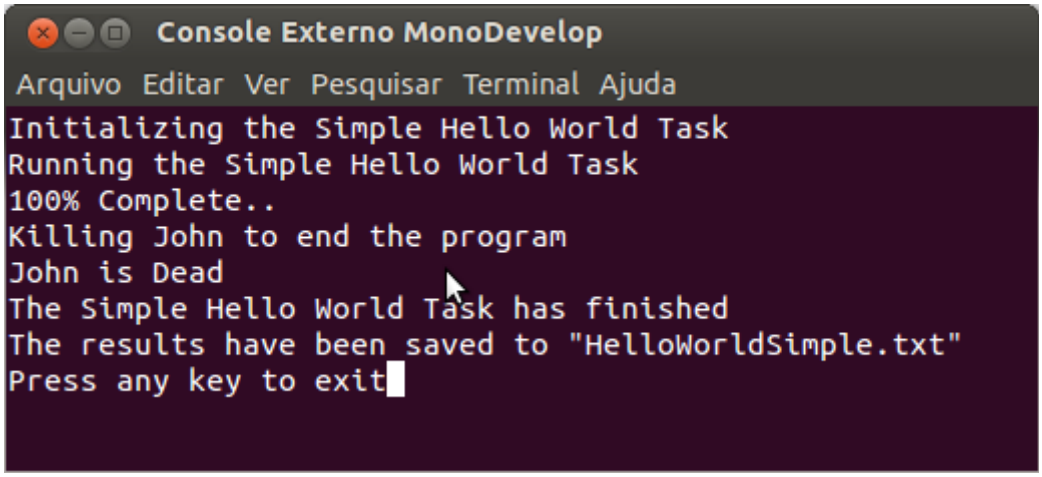

Figura 2 – Primeiro HelloWorld - II.

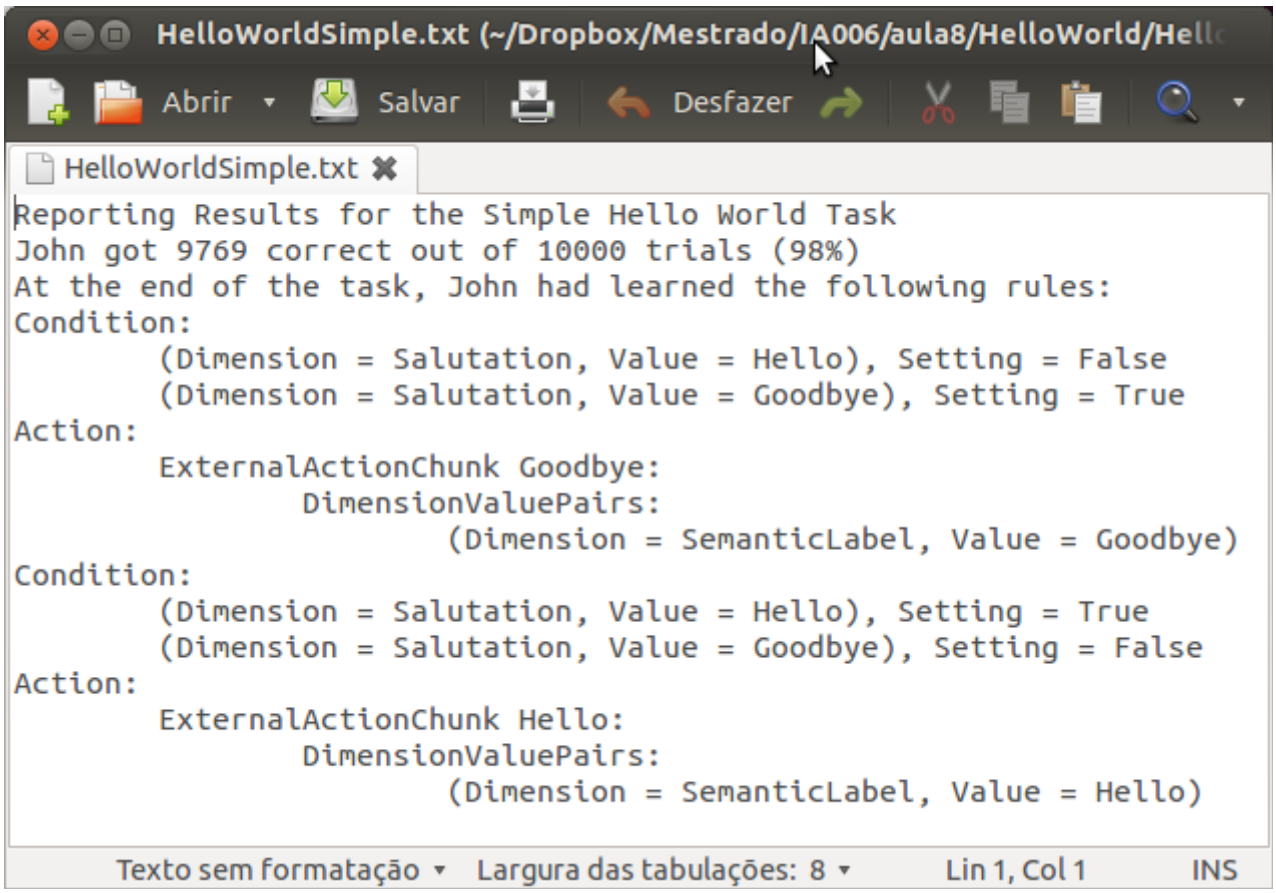

Figura 3 – Primeiro HelloWorld - III.

Para realizar está execução, foi necessária a criação de um projeto C# de modo texto no *mono develop* e a adição da biblioteca *CLARIONLibrary.dll* ao mesmo, similar a uma biblioteca *java*. A utilização desta biblioteca, se enquadra na utilização de recursos (bibliotecas, assemblys) de objetos (classes, interfaces e outros). A *CLARIONLibrary.dll* fornece ferramentas necessárias para a criação de agentes baseados em CLARION no próprio ambiente de simulação, ou seja, não há a separação que o SOAR utiliza.
A divisão de hierarquia da biblioteca *CLARIONLibrary* pode ser descrita da seguinte forma:

- **Clarion –** esta é a base da biblioteca e possui três classes: *World, AgentInitializer* e *ImplicitComponentInitializer*;
- **Clarion.Framework –** este sub item contem a maioria das classes necessárias para a

inicialização e execução da simulação;

- **Clarion.Framework.Core –** este item possui os construtores para o núcleo de operação do sistema;
- **Clarion.Framework.Extensions** – contém as extensões (módulos metacognitivos, componentes) para o CLARION, que embora não é especificada dentro do Clarion, podem ser usadas com agentes da mesma forma das classes encontradas em Clarion.Framework;
- **Clarion.Framework.Templates** – contém as classes abstratas, as classes de interface, delegações para construir a customização de alguns objetos;
- **Clarion.Plugins** – fornece vários *plugins* e ferramentas que podem ser utilizadas no ambiente de simulação para melhorar a capacidade e aplicação dos agentes baseados em CLARION;
- **Clarion.Samples –** contém exemplos que possibilitam a aprendizagem do CLARION.

As classes do CLARION são referenciadas estatisticamente, ou seja, as mesmas são referenciadas diretamente pelo nome da classe e a chamada do método ou objeto desejado.
O objetivo do agente presente no código da Figura 1, é responder a comprimentos, como exemplo se o mesmo receber *hello* ou *goodbye* ele deverá responder com *hello* ou *goodbye*. O agente utiliza apenas o ACS com reforço de uma rede backpropagation e regras de extração e refinamento (RER).
A importação dos pacotes Clarion e Clarion.Framework será necessária na maioria das classes que manipulam os agentes CLARION, utilizados nas simulações.

1.1 Inicializando o mundo

Neste momento, será iniciado a explicação mais detalhada do *helloWorld* apresentado anteriormente. Pode-se observar que a linguagem C# é bem próxima de C++ e java, por isso as declarações iniciais serão omitidas, na explicação. Mas, antes de iniciar as simulações é necessário a inicialização da arquitetura cognitiva, que pode ser utilizada através do objeto *world*, Figura 4.

```
DimensionValuePair hi = World.NewDimensionValuePair("Salutation", "Hello");
DimensionValuePair bye = World.NewDimensionValuePair("Salutation", "Goodbye");
```

Figura 4 – Inicialização *world - I*.

Para o exemplo que está sendo apresentado, é necessário a criação de uma dimensão com valores possíveis associados. No caso, a dimensão é a saudação “Salutation” com duas possibilidades, “Hello” e “Goodbye”. Também são declaradas as possíveis ações externas que o agente pode realizar, neste caso apenas responder “Hello” ou “Goodbye” quando for apresentado a uma saudação, Figura 5.

```
ExternalActionChunk sayHi = World.NewExternalActionChunk("Hello");
ExternalActionChunk sayBye = World.NewExternalActionChunk("Goodbye");
```

Figura 5 – Inicialização *world - II*.

Repare que estas ações *chuncks* podem ser mais complexas se necessário, estão simples devido ao exemplo utilizado. A Figura 6 apresenta a criação do agente que é utilizado para a arquitetura cognitiva.

```
//Initialize the Agent
Agent John = World.NewAgent("John");
```

Figura 6 – Criação do agente.

O agente foi criado como John apenas por uma questão didática, pois nomear o agente (passando o

seu nome por parâmetro) é opcional; a não ser que seja necessário recuperar o agente no mundo depois de um tempo, caso a referencia de seu objeto se perca por algum motivo de manipulação de variável. Até o momento foi criado um agente vazio, pois o mesmo ainda não aprendeu nada sobre as ações que deve tomar, o mesmo ainda precisa ser vinculado com os *chunks* ou pares de dimensão e valor.

1.2 Inicializando o agente

A inicialização do agente e a vinculação dos pares de dimensão e valor podem ser observados nas Figuras 7 e 8 respectivamente.

```
SimplifiedQBPNetwork net = AgentInitializer.InitializeImplicitDecisionNetwork
      (John, SimplifiedQBPNetwork.Factory);
```

Figura 7 – Inicialização do agente.

```
net.Input.Add(hi);
net.Input.Add(bye);

net.Output.Add(sayHi);
net.Output.Add(sayBye);
```

Figura 8 – Vinculação dos pares DV.

Note que no momento da vinculação do agente é atribuída uma rede que auxilia na decisão do agente, a rede (SimplifiedQBPNetwork) está presente nos pacotes do CLARION. Neste exemplo, a fábrica especificada no parâmetro, Figura 7, proverá a criação da rede no nível baixo do ACS *(bottom level* do ACS*)* e a rede criada é de utilização exclusiva do agente.
Os vínculos realizados na Figura 8, são necessários para que o agente receba informações (externas e/ou internas) e possa interagir com o mundo. Então, podemos concluir que o objeto "net" recebeu os parâmetros de entrada e saída possíveis da rede inicializada anteriormente. O número de camadas internas e nós desta rede é calculado automaticamente.
Após as inicializações, é necessário realizar um *commit* no agente. Observe, que realizado o *commit,* não é possível mais realizar mudanças na estrutura da rede, consequentemente o agente fica imutável neste sentido, não no sentido de aprendizagem. O código que foi utilizado neste exemplo para o *commit* do agente foi o seguinte: *"John.Commit(net);"*.

1.3 Ajustando parâmetros

Os ajustes, serão tratados com mais detalhes posteriormente no próximo tutorial. Mas, sabe-se claramente que os ajuste de uma rede é específico de cada problema. Para resolver isto, estes ajustes foram utilizados de forma mais simples devido ao problema tratado neste exemplo. A rede foi setada com uma taxa de aprendizagem de 1 e a performance de refinamento da rede não foi requerida, Figura 9.

```
net.Parameters.LEARNING_RATE = 1;
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

Figura 9 – Ajustes.

1.4 Executando uma simulação

Neste momento, o agente está pronto para ser treinado. Já é possível realizar a simulação com o agente e o mundo, através de iterações repetidas (neste exemplo 10 mil).

Em geral a simulação segue os seguintes passos:

- Especificar ao agente informações sensoriais obtidas em um ciclo *perception-action*;
- Capturar e gravar a ação que é escolhida pelo agente;
- Atualizar o estado do mundo, se necessário, com base nas ações tomadas pelo agente;
- Fornecer um *feedback* para o agente, gratificando ou penalizando de acordo com suas ações;
- Acompanhar a performance do agente, para poder tirar conclusões finais (estatísticas).

O primeiro item apresentado nos passos da simulação, pode ser entendido como a utilização de informações da memória de trabalho, *drives*, objetivos e outros específicos do agente. Por este motivo, os objetos de informação sensorial não podem ser compartilhados entre os agentes. O código deste item pode ser observado na Figura 10.

```
si = World.NewSensoryInformation(John);
```

Figura 10 – Inicializando o sensor de informação.

O objeto de sensor de informação (si) criado precisa ser configurado. Como o objetivo deste tutorial é simples, basta escolher aleatoriamente se o agente receberá uma saudação de *hello* ou *goodbye*, Figura 11.

```
//Randomly choose an input to perceive.
if (rand.NextDouble() < .5)
{
    //Say "Hello"
    si.Add(hi, hi.MAXIMUM_ACTIVATION);
    si.Add(bye, bye.MINIMUM_ACTIVATION);
}
else
{
    //Say "Goodbye"
    si.Add(hi, hi.MINIMUM_ACTIVATION);
    si.Add(bye, bye.MAXIMUM_ACTIVATION);
}
```

Figura 11 – Escolha de entrada aleatória.

A Figura 12, apresenta o recebimento do sensor de informação pelo agente e o armazenamento da escolha tomada.

```
//Perceive the sensory information
John.Perceive(si);

//Choose an action
chosen = John.GetChosenExternalAction(si);
```

Figura 12 – Percepção e escolha do agente.

A Figura 13, apresenta o processamento da escolha do agente (saída) e o envio do *feedback*.

```
if (chosen == sayHi)
{
    //The agent said "Hello".
    if (si[hi] == John.Parameters.MAX_ACTIVATION)
    {
        //The agent responded correctly
        Trace.WriteLineIf(World.LoggingSwitch.TraceWarning, "John was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent responded incorrectly
        Trace.WriteLineIf(World.LoggingSwitch.TraceWarning, "John was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
else
{
    //The agent said "Goodbye".
    if (si[bye] == John.Parameters.MAX_ACTIVATION)
    {
        //The agent responded correctly
        Trace.WriteLineIf(World.LoggingSwitch.TraceWarning, "John was correct");
        //Record the agent's success.
        CorrectCounter++;
        //Give positive feedback.
        John.ReceiveFeedback(si, 1.0);
    }
    else
    {
        //The agent responded incorrectly
        Trace.WriteLineIf(World.LoggingSwitch.TraceWarning, "John was incorrect");
        //Give negative feedback.
        John.ReceiveFeedback(si, 0.0);
    }
}
```

Figura 13 – Processando a escolha e eviando *feedback*.

Mesmo que não estivesse sendo usada uma rede simples, por exemplo uma *Q-Learning* no lugar de uma *Q-Learning* simplificada, ainda seria apenas necessário o envio de *feedback*, não sendo necessário mais nada além disso, pois o sistema cuidará das necessidades restantes. Note que o *feedback* enviado ao agente está na faixa de 0 à 1, com 0 para uma maior penalização, 1 para uma maior gratificação e neutro podendo ser representado por 0.5, mas neste exemplo não foi necessário.

Para finalizar a simulação é necessário "matar" o agente. Está ação é feita chamando o método "Die()", Figura 14.

```
John.Die();
```

Figura 14 – Finalizando o agente.

Após a execução do método "Die()" todos os processos internos do agente são finalizados. Porém embora a execução dos processos do agente tenham sido finalizadas, a configuração interna do agente ainda é mantida, podendo ser recuperada e/ou salvada.

2 ACS - Intermediário

Neste item será apresentado a execução do tutorial intermediário ACS, pelo exemplo *"Full Hello World"*. Frequentemente as tarefas que serão executadas podem ter resultados razoáveis, porém nem sempre poderá ser o resultado esperado ou ótimo. Devido a este motivo, existe a possibilidade de "tunar" ou modificar os parâmetros do agente com o intuito de possuir um resultado mais eficaz. A teoria CLARION especifica parâmetros para vários mecanismos. Estes parâmetros são implementados nas bibliotecas CLARION em duas linhas: tanto parâmetros globais (estático), quanto parâmetros locais (instância). Como exemplo a classe *RefineableActionRule* implementa um tipo de regra aprimorável. Estas regras ser generalizadas e especificadas, onde cada uma pode ser alterada para otimizar a frequência com que o processo ocorre. A Figura 15 apresenta o código de exemplo.

```
RefineableActionRule.GlobalParameters.SPECIALIZATION_THRESHOLD_1 = -.6;
RefineableActionRule.GlobalParameters.GENERALIZATION_THRESHOLD_1 = -.1;
```

Figura 15 – Otimizando parâmetros.

Para desativar as diferentes formas de aprendizagem que ocorrem no ACS (regras de refinamento, aprendizagem *bottom-up*, regras de extração, aprendizagem *top-down* e/ou aprendizagem em nível *bottom*), Figura 16.

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_RULE_EXTRACTION = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_TOP_DOWN_LEARNING = false;
ActionCenteredSubsystem.GlobalParameters.PERFORM_BL_LEARNING = false;
```

Figura 16 – Desativando formas de aprendizagem ou regras.

Todos estes exemplos estão relacionados com alterações de parâmetros globais.

2.1 Alterando parâmetros globais

Como já foi mencionado, os parâmetros globais são acessados estaticamente. Os parâmetros globais podem ser acessados a partir da classe *RefineableActionRule* , Figura 17.

```
RefineableActionRule.GlobalParameters
```

Figura 17 – Acessando parâmetros globais - I.

Os comandos para ativar e desativar a aprendizagem do ACS podem ser acessados a partir da classe *ActionCenteredSubsystem* , Figura 18.

```
ActionCenteredSubsystem.GlobalParameters
```

Figura 18 – Acessando parâmetros globais - II.

Os parâmetros globais são utilizados para inicializar os parâmetros locais das instâncias dos agentes, com isso, pode-se concluir que apenas alterações antes da inicialização do agente surtirão efeito. A alteração de parâmetro realizada no código da Figura 19 não terá efeito, pois a mudança foi realizada após a inicialização do agente. Esta alteração só terá efeito para as próximas inicializações de agentes. O caso correto pode ser observado na Figura 20.

```
Agent John = World.NewAgent("John");

ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;
```

Figura 19 – Alteração pós inicialização agente.

```
ActionCenteredSubsystem.GlobalParameters.PERFORM_RER_REFINEMENT = false;

Agent John = World.NewAgent("John");
```

Figura 20 – Alteração pré inicialização agente.

Todos os *bult-in,* classes da biblioteca CLARION, possuem parâmetros que podem ser alterados de qualquer ponto respeitando a hierarquia. A Figura 21 mostra um exemplo de mudança do parâmetro "POSITIVE_MATCH_THRESHOLD" de todas as regras.

```
Rule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

Figura 21 – Alterando parâmetros de regras globais.

O comando apresentado na Figura 21, mudará os parâmetros de todas as classes derivadas da classe *Rule (RefineableActionRule, IRLRule, AssociativeRule,* etc*)*. No entanto, suponha que o desejo seja alterar apenas os parâmetros relacionados as regras IRL. Essa alteração pode ser feita a partir da classe *IRLRule*, Figura 22.

```
IRLRule.GlobalParameters.POSITIVE_MATCH_THRESHOLD = .75;
```

Figura 22 – Alterando parâmetros de regras específicas.

Note que existe uma hierarquia entre os comandos da Figura 21 e 22. Pois o comando da Figura 21, é mais amplo que o comando da Figura 22. Porém se for desejado alterar apenas parâmetros de configurações de uma instância específica, isso deve ser feito localmente. Esta técnica é apresentada no próximo item deste relatório.

2.2 Alterando parâmetros locais

As alterações locais são realizadas especificamente nas instâncias de classes que contém os parâmetros. Por exemplo se deseja-se desativar o atributo ACS, "PERFORM_RER_ REFINEMENT", apenas do agente John, Figura 23.

```
John.ACS.Parameters.PERFORM_RER_REFINEMENT = false;
```

Figura 23 – Alterando parâmetros locais.

Em contrapartida dos parâmetros globais, os parâmetros locais podem ser alterados após a inicialização do objeto. Um exemplo, podendo citar a necessidade de alterar o parâmetro "LEARNING_RATE" da instância da classe *SimplifiedQBPNetwork* chamada *net*, Figura 24.

```
//Retrieves the network from the bottom level of John's ACS
SimplifiedQBPNetwork net = (SimplifiedQBPNetwork)John.
   GetInternals(Agent.Internals.IMPLICIT_DECISION_NETWORKS).First();

net.Parameters.LEARNING_RATE = .5;
```

Figura 24 – Alterando parâmetros locais, após inicialização.

Observe que a alteração realizada no objeto *net* terá efeito para as próximas iterações de aprendizagens do mesmo.
Além destas duas técnicas, existe outra maneira de alterar parâmetros de uma forma mais automática, porém esta técnica esta além do tutorial tratado neste item.

2.3 Configurando a memória de trabalho

Nesta parte do tutorial, é apresentado o modo de configuração e utilização da memória de trabalho. Resumidamente, a memória de trabalho possibilita que o agente armazene seu conhecimento sobre o mundo. Tecnicamente, ela fica localizada dentro do ACS, porém toda interação realizada com a mesma é feita através da classe agente. O código a seguir, Figura 25, mostra como é possível retornar o conteúdo da memória de trabalho do agente.

```
IEnumerable<Chunk> wmContents =
    (IEnumerable<Chunk>)John.GetInternals
(Agent.InternalWorldObjectContainers.WORKING_MEMORY);
```

Figura 25 – Recuperando conteúdo da memória de trabalho.

Nota-se que a memória de trabalho pode conter qualquer tipo de *chunk*. Sempre que um *chunk* é adicionado, o mesmo passa a fazer parte da informação sensorial interna.

2.3.1 Configurando manualmente um *chunk* na memória de trabalho

A maneira mais simples de adicionar um *chunk* na memória de trabalho, é manualmente. A Figura 26, monstra essa tarefa através do agente John, adicionando o *chunk* e seu nível respectivo na memória de trabalho (com parâmetro de entrada).

```
John.SetWMChunk(ch, 1);
```

Figura 26 – Adicionando *chunk* a memória de trabalho.

Para remover ou desativar um *chunk*, basta chamar o método *resetMWChunk* apontando o objeto que se deseja retirar da memória de trabalho, Figura 27.

```
John.ResetWMChunk(ch);
```

Figura 27 – Removendo *chunk* a memória de trabalho.

2.3.2 Usando ação *chunks*

Ações que afetam a memória de trabalho, são referenciadas como ações da memória de trabalho. As ações *chunks* contém as informações sobre o tipo de atualização realizada. A Figura 28, apresenta a adição de um *chunk* (ch) a memória de trabalho.

```
WorkingMemoryUpdateActionChunk wmAct = World.NewWorkingMemoryUpdateActionChunk();

wmAct.Add(WorkingMemory.RecognizedActions.SET, ch);
```

Figura 28 – Adição de *chunk* a memória de trabalho.

Repare que é utilizada a classe *WorkingMemory.RecognizedActions*. Esta classe *RecognizedAcitons* é uma classe enumerada, que define quatro ações:

- SET: adiciona o *chunk* a memória de trabalho;
- RESET: remove o *chunk* da memória de trabalho;
- RESET_ALL: remove todos os *chunks* da memória de trabalho;
- SET_RESET: combina o terceiro e o primeiro item (RESET_ALL e SET).

Se for desejado um componente ACS para usar nesta ação, basta especificá-lo na camada de saída. A Figura 29 apresenta um exemplo de com configurar esta ação na rede de baixo nível do ACS.

```
... //Elided code performing additional initialization for the network

net.Output.Add(wmAct);
```

Figura 29 – Adição de *chunk* a rede de baixo nível do ACS.

Agora sempre que o ACS seleciona esta ação especificada, o sistema executará os comandos desta ação.

3 Setting Up & Using the Goal Structure

Neste item de tópico será discutido como utilizar as configurações da estrutura de meta. Considerando que um agente possui objetivos e está estrutura esta contida no agente. Toda a interação com a estrutura de meta, é realizada pela classe *Agent,* observe a Figura 30.

```
//Gets all of the items in the goal structure
IEnumerable<GoalChunk> gsContents =
    (IEnumerable<GoalChunk>)John.GetInternals
    (Agent.InternalWorldObjectContainers.GOAL_STRUCTURE);

//Gets the current goal
GoalChunk currentGoal = John.CurrentGoal;
```

Figura 30 – Estrutura de meta.

Como as ações e metas são utilizadas como *chunks* (usando a classe *GoalChunck*) as mesmas utilizam o mesmo objeto, Figura 31.

```
GoalChunk g = World.NewGoalChunk();
```

Figura 31 – Objeto *GoalChunck*.

Também existem dois parâmetros que podem ser alterados para mudar o comportamento da estrutura de meta. Esses parâmetros estão presentes na classe *MotivationSubsystem* e especificam:

- O comportamento da estrutura de meta, podendo se comportar como uma pilha ou uma lista;
- Como configurar a ativação da meta atual (todos os objetivos ou objetivos parciais).

A Figura 32 apresenta um exemplo de utilização destes parâmetros.

```
John.MS.Parameters.CURRENT_GOAL_ACTIVATION_OPTION =
    MotivationalSubsystem.CurrentGoalActivationOptions.FULL;

John.MS.Parameters.GOAL_STRUCTURE_BEHAVIOR_OPTION =
    MotivationalSubsystem.GoalStructureBehaviorOptions.STACK;
```

Figura 32 – Parâmetros de configuração - meta.

O código presente na Figura 33, demostra como inicializar uma meta e como a mesma é utilizada como parte da entrada para um componente (neste exemplo é utilizado o nível *bottom* do ACS pelo *SimplefiedQBPNetwork*).

```
... //Elided code initializing other world objects

GoalChunk g = World.NewGoalChunk();
Agent John = World.NewAgent("John");

SimplifiedQBPNetwork net =
    AgentInitializer.InitializeImplicitDecisionNetwork(John,
    SimplifiedQBPNetwork.Factory);

net.Input.Add(g);

... //Elided code performing additional initialization for the network
```

Figura 33 – Inicializando uma meta.

Note que todas as metas do mundo são sempre especificadas como parte da informação sensorial interna e será automaticamente ativada a próxima vez que for executada no *SensoryInformation*. Existem duas formas de usar as metas em um agente, manualmente e pelo estrutura de meta de atualização do *chunck.* Os próximos itens expõe as duas formas de utilização.

3.1.1 Configuração de meta manualmente

O modo mais simples de adicionar ou ativar uma meta na estrutura de meta é da forma manual, chamando o método *SetGoal* pelo objeto agente, Figura 34.

```
John.SetGoal(g, 1);
```

Figura 34 – Setando uma meta em um agente.

Os dois métodos passados por parâmetro são respectivamente a meta e o nível de ativação da mesma. A Figura 34 adiciona a meta a estrutura de metas, do modo oposto do código presente na Figura 35 que remove ou desativa uma meta em um agente.

```
John.ResetGoal(g);
```

Figura 35 – Removendo uma meta de um agente.

Estes dois métodos permitem toda a utilização da estrutura de meta no CLARION. O próximo item apresenta o modo mais avançado de utilização usando ações de meta nos ACS.

3.1.2 Usando *Action Chuncks*

Na biblioteca CLARION as ações que executam as atualizações da estrutura de meta são definidas usando a classe *GoalStructureUpdateActionChunck*. A Figura 36 apresenta uma ação que seta uma meta *g* na estrutura de meta do agente.

```
GoalStructureUpdateActionChunk gAct = World.NewGoalStructureUpdateActionChunk();

gAct.Add(GoalStructure.RecognizedActions.SET, g);
```

Figura 36 – Adicionando uma meta em um agente.

Observe o primeiro parâmetro do método *Add* (segunda linha Figura 36), utiliza uma classe enumerada chamada *RecognizedActions*. Esta classe oferece uma lista de comandos que podem executar uma ação em uma instância de classe. A classe *GoalStructure* aceita 4 tipos de ações:

- SET – adiciona uma meta na estrutura de meta;
- RESET – remove uma meta na estrutura de meta;
- RESET_ALL – remove todas as metas da estrutura de meta;
- SET_RESET – combina as ações RESET_ALL e SET.

Se for desejado um componente ACS para usar esta ação é necessário especificá-lo na camada de saída. A Figura 37 apresenta um exemplo de como configurar esta ação em uma rede de nível *bottom* do ACS.

```
... //Elided code performing additional initialization for the network

net.Output.Add(gAct);
```

Figura 37 – Configurando ação na rede - ACS.

Agora sempre que o ACS selecionar está *GoalStructureUpdateActionChunck*, o sistema irá executar os comandos especificados nesta ação.