UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Elétrica e de Computação

Danilo Fernando Lucentini

A comparison among cognitive architectures: theoretical and practical analysis

Uma comparação entre arquiteturas cognitivas: análise teórica e prática

Campinas

2017

Danilo Fernando Lucentini

A comparison among cognitive architectures: theoretical and practical analysis

Uma comparação entre arquiteturas cognitivas: análise teórica e prática

Thesis presented to the School of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering.

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Ricardo Ribeiro Gudwin

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Danilo Fernando Lucentini, e orientada pelo Prof. Dr. Ricardo Ribeiro Gudwin

Campinas

2017

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

L963c

Lucentini, Danilo Fernando, 1986-
A comparison among cognitive architectures : theoretical and practical analysis / Danilo Fernando Lucentini. – Campinas, SP : [s.n.], 2017.

Orientador: Ricardo Ribeiro Gudwin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Inteligência artificial. 2. Ciências cognitivas. 3. Agentes inteligentes (Software). 4. Simulação (Computadores). I. Gudwin, Ricardo Ribeiro, 1967-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma comparação entre arquiteturas cognitivas : análise teórica e prática
Palavras-chave em inglês:
Artificial intelligence
Cognitive science
Intelligent agents (Software)
Simulation (Computer)
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Ricardo Ribeiro Gudwin [Orientador]
Esther Luna Colombini
Fernando Antonio Campos Gomide
Data de defesa: 20-12-2017
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Danilo Fernando Lucentini RA: 032112

Data da defesa: 20/12/2017

Dissertation Title: "A comparison among cognitive architectures: theoretical and practical analysis"

Título da Dissertação: "Uma comparação entre arquiteturas cognitivas: análise teórica e prática"

Prof. Dr. Ricardo Ribeiro Gudwin (Presidente, FEEC/UNICAMP)

Profa. Dra. Esther Luna Colombini (IC/UNICAMP)

Prof. Dr. Fernando Antonio Campos Gomide (FEEC/UNICAMP)

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no processo de vida acadêmica do aluno.

Acknowledgements

My mom Célia and my dad Dante for all the support, love and understanding. This work is dedicated to them.

Prof. Dr. Ricardo Ribeiro Gudwin for his guidance. To show me the way and to stay by my side in all the moments, supporting and encouraging me.

My girlfriend, Nathércia, for all the love, support, patient and tireless reviews. I love you.

My colleagues from CogSys group at FEEC.

All the professors and employees at FEEC for all they have taught me.

All my friends who, directly or indirectly, were part of this moment.

"I have noticed even people who claim everything is predestined, and that we can do nothing to change it, look before they cross the road."

(Stephen Hawking)

Abstract

This work presents a theoretical and practical comparison of three popular cognitive architectures: SOAR, CLARION, and LIDA. The theoretical comparison is performed based on a set of cognitive functions supposed to exist in the human cognitive cycle. The practical comparison is performed applying the same experiment in all architectures, collecting some data and comparing them using a set of software quality metrics as a basis. The aim is to emphasize similarities and differences among the models and implementations, with the purpose to advise a newcomer on how to choose the appropriated architecture for an application.

Keywords: Cognitive architectures; Cognitive science; SOAR; CLARION; LIDA.

Resumo

Este trabalho apresenta uma comparação teórica e prática entre três das mais populares arquiteturas cognitivas: SOAR, CLARION e LIDA. A comparação teórica é realizada com base em um conjunto de funções cognitivas supostamente existentes no ciclo cognitivo humano. A comparação prática é realizada aplicando-se um mesmo experimento em todas as arquiteturas, coletando alguns dados e comparando-as usando como base algumas métricas de qualidade de software. O objetivo é enfatizar semelhanças e diferenças entre os modelos e implementações, com o objetivo de aconselhar um novo usuário a escolher a arquitetura mais apropriada para uma certa aplicação.

Palavras-chaves: Arquiteturas cognitivas; Ciência Cognitiva; SOAR; CLARION; LIDA.

List of Figures

List of Tables

Contents

1 Introduction

What will be my lunch today? Should I go to work by bus or by car? Should I use a short or a long sleeve shirt? These questions are examples of some decisions that we take every day. We live in a dynamic environment, where everything changes constantly and where, at every moment, we face a situation that requires a decision to be taken. We take all kinds of decisions, even for minor things and, sometimes, we are not even realizing we are making a choice.

Artificial intelligence (AI) is a research area in computer science that aims to reproduce the human intelligent behavior in artificial agents, helping them to choose the best decision to be taken at a given moment. Through the years, a lot of approaches were developed: first we had expert systems (originally using classic logic, and later including fuzzy logic), which were based on a symbolic approach correlating conditions or facts to actions or results (expressed in form of rules); later, neural networks appeared with a sub-symbolic approach, following an analogy based on the way that neurons work in the brain, translating a problem-solving issue in a functional mapping, using inductive learning techniques to find the parameters of this mapping; finally, seeking to fill some gaps, emerged the cognitive systems, based on models of human or animal cognitive processing.

Cognition is a process related to how a person or animal understands the world and acts on it. The sequence of understanding and further acting is called a *cognitive cycle* and is performed using a set of functions (called cognitive functions), each one responsible for a specific task, contributing to the effectiveness of the cycle. So, if we want to develop an artificial agent that can operate in dynamic environments, it is crucial for them to exhibit a refined and malleable problem-solving capability, and nothing better than to mirror in the cognitive functions of humans or animals (forged by natural selection along the years). If we are able to replicate these functions (perception, memory, attention, motivation, consciousness, etc) in artificial agents, we can get amazing results by combining the velocity of the current processors with the versatile way that humans or animals solve problems. In this context, the concept of cognitive architecture emerged.

A cognitive architecture (CA) is, basically, a multi-domain computational model that can be used for several purposes in different levels of applications. It tries to describe, based on different models of human or animal cognition, the way that problems are solved, knowledge is acquired, goals are determined, actions are taken and sensorial data is processed. Ron Sun (SUN, 2004) lists some important characteristics that any CA should have, for instance: bio-evolutionary realism, cognitive realism, the eclecticism of methodologies and techniques, reactivity, trial-and-error adaptation, and so on.

Each CA tries to model the functions required to create an autonomous intelligent agent, based on biologically inspired foundations. These foundations supporting the architecture are referred as being the *architectural model* or *conceptual model* of the architecture. Given a conceptual model, several computer implementations can be created, being designated as *architecture implementations*, that tend to be very complex, due to the number of concepts and theories involved. An architecture implementation is, in short, a software that is used as a platform for the creation of other software, this one responsible, in fact, to meet a set of goals. In other words, the architecture implementation can be viewed as the "skeleton" of the application, because it provides the foundations of a reusable code that can be applied in different situations.

Over the years, several paradigms appeared in cognitive science, bringing new theoretical foundations, and the number of CA's considerably increased (SOCIETY, 2012). There are many examples of architectures under development that continually receive updates: ACT-R, CLARION, SOAR, LIDA, EPIC, COGPRIME, ART, BECCA, etc. Due to this diversity, it is complex to choose any of them in particular to make experiments or to apply in real problems, in such a way that a user, that is beginning the study in this area, will have a herculean task in order to determine which architecture will best fit his needs or even if a cognitive architecture is the best approach to solve the desired problem.

This diversity of approaches and the difficulty in properly mapping the similarities and differences among them is the motivation for this work. Our main goal is to facilitate a researcher interested in this subject in performing his first steps, providing a better overview of the area and highlighting some important aspects. To achieve this goal, we performed a detailed analysis of some CA's and a deep comparison among them under

two perspectives: architectural modeling and architectural implementation. The models were compared using biological functions as a basis and implementations were compared using a practical experiment applied for all the architectures.

Considering this proposal and that a large number of cognitive architectures is available, three CA's are focused in this work: SOAR (State Operator And Result), CLARION (Connectionist Learning with Adaptive Rule Induction ON-line) and LIDA (Learning Intelligent Distribution Agent). These architectures were chosen using the following criteria: to be traditional and emblematic exponents in the field, to have available a considerable number of publications and tutorials, to have their implementation available for free in the Internet, and to be constantly updated (making the analysis of the implementation feasible). Besides that, it is important to notice that each of the chosen architectures is related to a specific paradigm in cognitive science (cognitivism, connectionism and embodied cognitive science).

In the next 3 chapters, these cognitive architectures will be explored, detailing the foundations of their conceptual models and also exposing the functionality of each module in the architectures. In chapter 5, we try to establish a strategy of comparison, to be used in the sequence, based on the architecture's models and implementations, and also elucidating the details of the practical experiments. Finally, chapters 6 and 7 consolidate the analysis providing the obtained results.

2 The SOAR Cognitive Architecture

Created at the University of Michigan by John E. Laird, Paul Rosenbloom and Allen Newell, SOAR is a cognitive architecture that typically represents the cognitivist ideology, predominant in the 70's and 80's, which defined the mind as a digital computer, or as a set of symbolic representations handled according to syntactical rules and following Searle's Chinese Room argument (SEARLE, 2001).

Despite its symbolic origins, new modules started to be added to SOAR, giving it a hybrid flavor. In 2008, Laird proposed an extended SOAR architecture with new modules and features (LAIRD, 2008). Nevertheless, as Laird said, all of these modules are in continuous development and gradually being integrated into the architecture. Figure 2.1 helps to describe all the components that make up the actual architecture.

In SOAR's website[1], we can find both the executable and source code of SOAR, which are available for different operational systems. In the package available for download, there are some tools that make the development and the debugging easier (SoarDebugger and VisualSoar, for example), documentation, code samples and the SOAR APIs. SOAR uses an integration language, Soar Markup Language (SML), to embed SOAR within another piece of code, allowing the use of SOAR in the construction of artificial agents. Along with this work, we will be highlighting some examples of this language in order to explain how to create an agent with SOAR embedded.

First of all, it is important to provide a brief explanation of the principles of the architecture. SOAR is a tool envisioned for the exploration and search of a problem space. The idea is that, given a problem, we are able to construct a model for this problem finding a proper problem space and encoding the problem in terms of a current *state*, which might be changed through the application of *operators*, leading the system to further *states*, until a *goal state* is reached. From this perspective, two primary concepts in SOAR are: *operators* and *states*.

A *state* is an encoding of a situation in a given time, encoded in SOAR's Working Memory using a set of WME's (Working Memory Elements). As the SOAR

[1] URL: <http://soar.eecs.umich.edu>

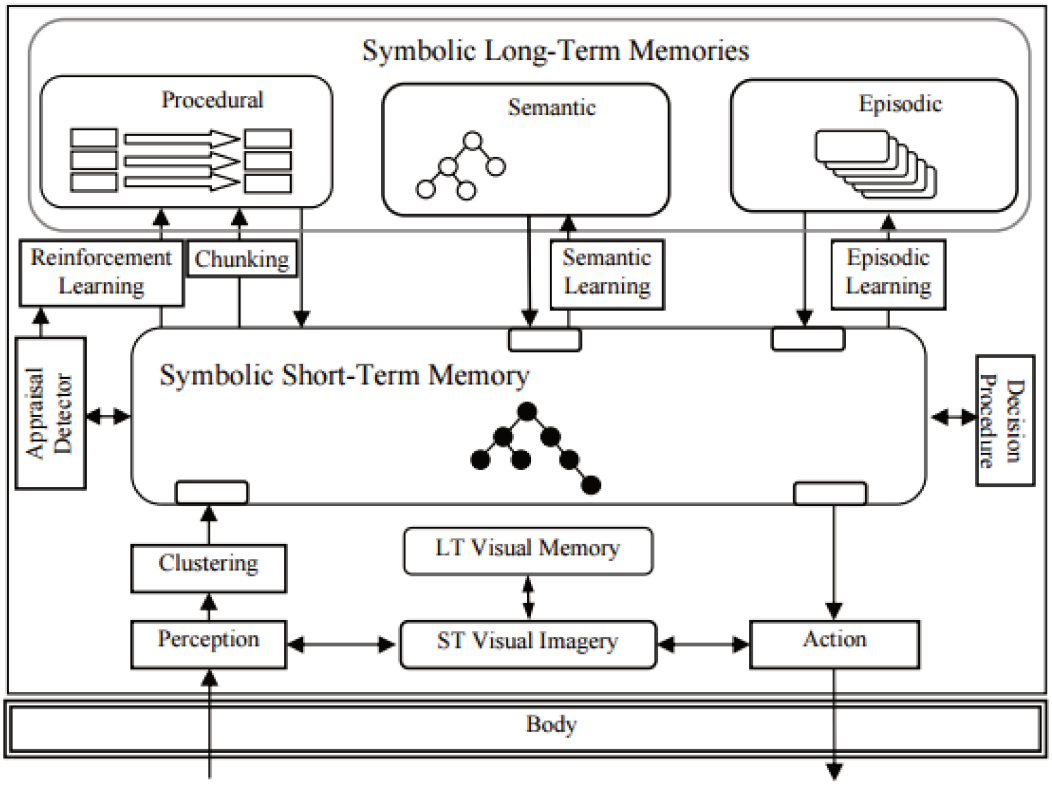


Figure 2.1 – SOAR Modules - Extracted from (LAIRD, 2008)

manual describes:

> "Each WME contains a very specific piece of information; for example, a WME might say that "B1 is a block". Several WME's collectively may provide more information about the same object, for example, "B1 is a block", "B1 is named A", "B1 is on the table", etc. These WME's are related because they are all contributing to the description of something that is internally known to Soar as B1" (LAIRD; CONGDON, 2014).

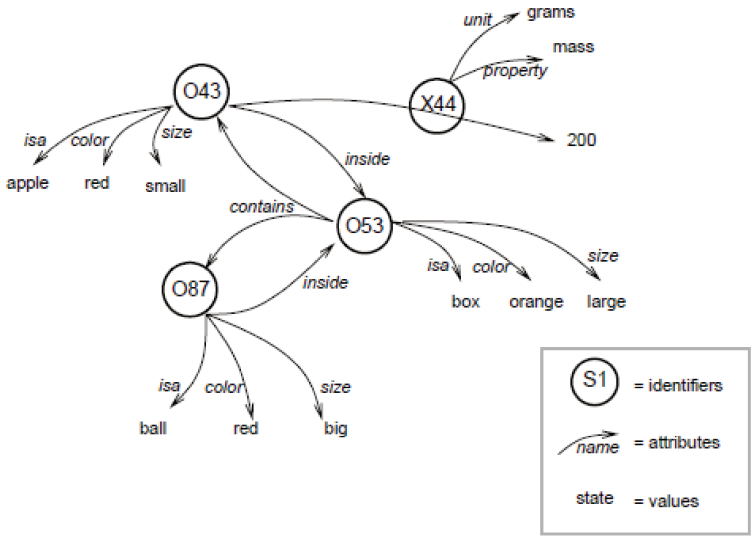
So, a WME is an entity that helps to describe an object, which is important to model a situation in SOAR problem-solving context. In short, each WME is an identifier-attribute-value triple, and all WME's with the same identifier are part of the same object (LAIRD; CONGDON, 2014).

A canonical representation for a WME is as follows:

```
(identifier ^attribute value)
```
(2.1)

In SOAR, an *identifier* is always a string formed by a capital letter followed by an integer number. An *attribute* is any string (appended by a circumflex symbol), and

a *value* can be either a string, a number or even other identifiers. Intrinsically, a set of WME's can be viewed as a graph where each branch refers to an attribute and each leaf to a value (or identifier) as illustrated in figure 2.2. The example below defines some WME's in the Working Memory and, as previously described, each element is composed of an identifier (a node in the graph), an attribute (an arc) and a value (either a final leaf in the graph, or another identifier).




```
(O43 ^isa apple)
(O43 ^color red)
(O43 ^size small)
(O43 ^X44 200)
(O43 ^inside O53)
(X44 ^unit grams)
(X44 ^property mass)
(O53 ^isa box)
(O53 ^color orange)
(O53 ^size large)
(O53 ^contains O43)
(O53 ^contains O87)
(O87 ^isa ball)
(O87 ^color red)
(O87 ^size big)
(O87 ^inside O53)
```

Figure 2.2 – Working memory representation in SOAR - Extracted from (LAIRD; CONGDON, 2014)

States are objects in the Working Memory, with a pre-defined set of attributes which can be augmented by the user in order to model any kind of problem. During SOAR's initialization process, an initial state S1 is always created, being represented by a pre-defined set of WME's. After that, SOAR starts a sequence of reasoning cycles, where WME's are matched against a set of rules in Procedural Memory, in order to seek for a proper *operator* to be applied, and once an operator is selected, it is applied, creating changes in the system state.

Operators are abstract modifiers that make changes in the current state and are represented also by means of WMEs. At a minimum, an operator is an object with just one attribute: its name. But they might also have other parameters (defined by further WMEs), which can be useful to characterize the operator's effective change in the current state. To complement its representation as a SOAR object, an operator needs to describe its effect by means of a set of rules. Rules are another kind of elementary entity in SOAR,

stored in the Procedural Memory. Each rule has basically two parts: LHS (Left Hand Side), usually describing a condition to be met, and RHS (Right Hand Side), usually prescribing an action to be performed while the condition given in the LHS is met. The LHS is constructed by a set of *patterns* which are matched against WME's in the Working Memory, possibly with variables which can be instanced during the pattern-match phase. The RHS of a rule is constructed by a set of *commands*, which can do one of the following things:

- Add a new WME to the Working Memory.
- Remove a WME from the Working Memory.
- Propose an operator to be applied to current state
- Define preferences among operators.
- Perform system actions like printing, saving files, etc.

Thus, rules in the Procedural Memory can be of different types, depending on their conditions and actions (refer to Box 2.1 to visualize an example of usage of each type of rule).

- **Operator Proposition Rules**: Rules which detect a set of conditions in the current state and propose the application of a particular operator to the current state.
- **Elaboration Rules**: Rules which detect a set of conditions in the current state and propose the creation of WMEs to further elaborate the state (e.g. calculate derived variables, etc).
- **Preferences Definition Rules**: Rules which detect a set of situations in their condition part and propose some preference among operators in their actions part.
- **Operator Application Rules**: Rules which detect if a particular operator was selected by SOAR and perform a set of removals and additions of WMES in order to change the state.
- **Monitoring Rules**: Rules which detect a set of situations in the current state and perform some system debugging actions like printing message or saving logs.

Box 2.1: Example of rules in SOAR

```
# Operator Proposition Rules
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)}

# Operator Application Rules
sp {apply*wander
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <ol>)
   (<o> ^name wander)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 1)
   (<command> ^VelR 1)
   (<command> ^VelL 1)}

#Elaboration Rules
sp {elaborate*block*increment
(state <s> ^problem-space blocks
                   ^block-counter <counter>
                   ^thing <element>)
(<element> ^type block)
-->
(<s> ^block-counter (+ 1 <counter>))}

#Preferences Definition Rules
sp {wander*preferences
(state <s> ^operator <o> +)
(<o> ^name wander)
-->
(<s> ^operator <o> <)}

#Monitoring Rules
sp {wander*debug
(state <s> ^operator.name wander
           ^io.output-link <out>)
-->
(write (crlf) |Operator applied: wander|)}
```

It is important to understand that the basic building blocks of SOAR are *WMEs* and *rules*. The concepts of *state* and *operators* are high-level structures which are constructed with the use of WMEs and rules. A *reasoning cycle* in SOAR is constituted of a sequence of *reasoning steps*, since the problem proposition in the form of an initial state, until the last step which invokes the *halt* command, indicating that the problem has been solved, and there is a solution.

Each *reasoning step* has the following phases:

- **Input:** During this phase, input information is collected from the environment, encoded into WMEs and appended to the `io.inputlink` node of the state.
- **Proposal:** During this phase, operator proposition rules are fired in order to propose a set of operators which might be applied to the current state.
- **Decision:** During this phase, SOAR evaluates all the proposed operators and select one of them to operate on the state.
- **Application:** During this phase, operator application rules are fired in order to change the current state, transforming it into a new state.
- **Output:** During this phase, any information to be output to the environment is added to the `io.outputlink` node of the current state, in the form of WMEs, which are then unencoded and made available for the user to be sent to system actuators.

In the beginning of the first *reasoning step* of a *reasoning cycle*, SOAR creates the basic structure of an initial *state*, which is then augmented by rules converting information from the `inputlink` to the current state. Then, a set of successive *reasoning steps* are performed, each of them applying just one operator and transforming a state in a new state until the command *halt* is called by an operator application rule, or an *impasse* is generated. There are many situations which could lead to an impasse. Basically, an impasse means that SOAR is not being able to proceed to the next *reasoning step* and further information is required before continuing. To try to solve an impasse, SOAR creates a new sub-state, where new kinds of rules are then used with the aim of eliminating the original impasse. This mechanism is used as a way to deal with sub-goals. For example, an impasse might be generated because there are more than one proposed operators, and operator preferences do not help in trying to define one of them. In this case, new rules are fired on the sub-state, trying new conditions which might allow a final decision, eliminating the impasse. There are also other kinds of impasses, like the `state-no-change` impasse or the `operator-no-change` impasse, where SOAR is not able to find further rules to apply proposing a new operator, and then SOAR also finish its *reasoning cycle*, without reaching a goal. During the development of a SOAR program, new rules should be added trying to eliminate all kinds of impasses, avoiding the situation in which SOAR needs to stop due to an impasse, without reaching the goal.

In the next sections, each module of SOAR is described in more details.

2.1 Decision procedure

One important algorithm we need to be aware in order to understand how SOAR works is the *Decision Procedure*. The *Decision Procedure* is responsible for managing the decision phase in the reasoning step. The decision phase can be characterized by three small steps, after which SOAR will select the best operator (only one) to be executed transforming the current state.

The first of these steps consists in collecting all the operators proposed as candidates for execution, as given by the proposal phase. If there is only one operator proposed, it will be selected with no further processing.

The second step consists of evaluating the *preferences* among all the proposed operators. These preferences are defined by *preferences definition rules*, providing a basis to the architecture to identify which is the best operator to be selected when there are multiple options. The SOAR programmer can create rules setting general preferences like operator A has the highest or lowest preference, or also specific preferences among two operators, like operator A has a higher preference than operator B.

The third and final step is related to the evaluation of *impasses*. If even after the first and second steps, there are still more than one plausible operator to execution (or none), an impasse will be raised. An impasse causes the creation of a sub-state where the main goal is to solve the impasse receiving some extra information provided by the architecture like: the type of impasse, which operators are involved, etc. Figure 2.3 summarizes all these steps.

It is important to emphasize that an impasse is not necessarily a result of a bad elaboration of operators and rules. Quite the reverse, it is a crucial feature of the architecture. For example: given an operator “move to object”, which is triggered when there is a specific kind of object in WME, if there are two (or more) instances of this type of object in the working memory, probably two (or more) instances of this operator “move to object” will be proposed and an impasse will be raised. However, SOAR can evaluate this impasse and take a better decision, like: “move to the closest object”, giving more

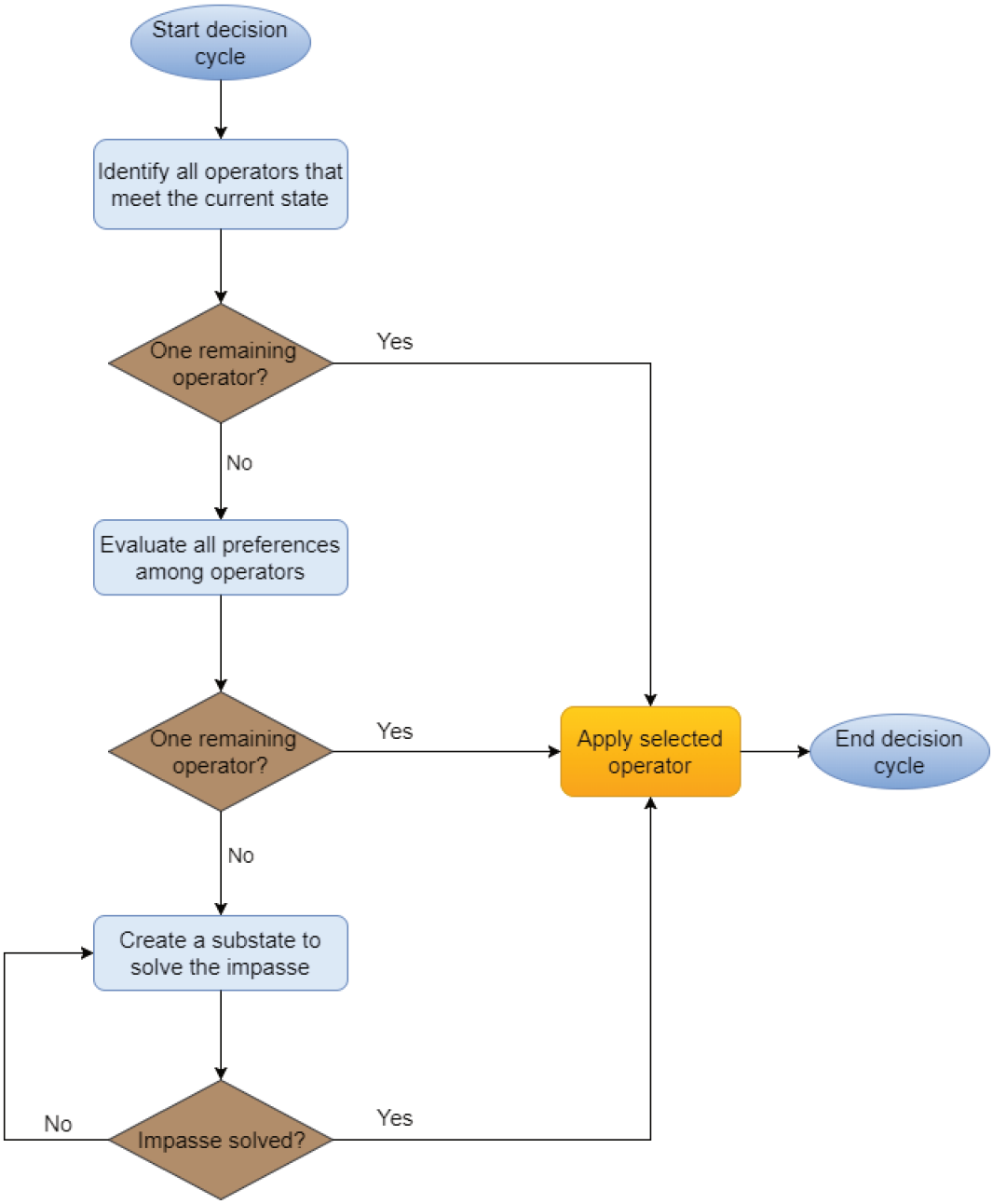


Figure 2.3 – SOAR Decision Procedure Flowchart

preference to one operator than the other.

The impasse sub-state is also solved using rules to refine the preferences of one or other operator under special circumstances. This allows the decision to be split into many separate rules, creating a more comprehensible set of rules. If the sub-state cannot still be solved, the process repeats, in a recursive way, with the creation of another impasse, and another sub-state, until the problem can be solved or a limit depth is reached, when the procedure stops anyway, even without finding a solution. With this hierarchy of

impasses, it is possible to plan the solution to impasses in layers: first one type of impasse and then the others, making the problem-solving capability more modular and robust.

2.2 Perception

As SOAR is a general purpose architecture, it is expected that it can be integrated into other environments to solve different kinds of problems. So, it is really important that this integration can be smooth and easily made, independently of the target environment.

The perception module is responsible for performing the link between the architecture and the real world in order to generate inputs to SOAR (in the form of WME's). The conversion between raw data (coming from the agent sensors) to WME's must be performed outside the scope of the architecture. At the end of the input phase of the *reasoning step* (where perception occurs), a set of WMEs are incorporated to the `^io.inputlink` node of the current state, where rules can be used to detect situations and further elaborate the state. Examples of how to implement perception in SOAR are covered in more details in section 7.1.

2.3 Action

The action module is very similar to the perception module because both are responsible for making the interface between the architecture and the environment, where the agent is inserted. However, the action module is responsible for providing the outputs of the architecture, i.e., given some inputs, the architecture will process them and generate the outputs which are the best for the current state. The mechanism is the opposite of the one at the Perception module. At the end of a *reasoning cycle* (when a goal is reached), a set of WMEs are incorporated into the `^io.outputlink` of the current state. From there, it is possible to unencode these WMEs into parameters which are then sent to the environment's actuator, causing some change in the environment. Also, we show some examples of how to implement action in more details in section 7.1.

2.4 Working memory

The *Working Memory* is the central repository of WME's, the place where all activity in SOAR is performed and stored. WME's describe the current state of the problem, binding together the many pieces of information necessary to represent a full problem-solving context (as illustrated in figure 2.2). WME's can also be added or removed from working memory due to the processing of other architecture modules (like e.g. the semantic memory, the episodic memory, the spatial visual system or the reinforcement learning module). Therefore, the working memory can be viewed as a theater of operations of the architecture, because all the other modules are linked through the working memory (as can be shown in figure 2.1). This modeling implements the short-term memory hypothesis, which is supposed to exist in several animals (including humans), having a limited storage capability (sometimes referred as the "magical number seven" (MILLER, 1956)) and with short-term duration.

All the WMEs in the Working Memory need to be linked to a parent node, creating a hierarchy of nodes (it is not possible to create a new node without connecting it to an existing node). The root node of all the hierarchy is always the S1 node, which holds the current state, and is generated automatically by SOAR during the beginning of a *reasoning cycle*. External inputs are connected to the `^io.inputlink` node and external outputs are connected to the `^io.outputlink` node of the current state S1. The state node S1 also has links to the episodic memory through a `^epmem` node, to the semantic memory through a `^smem` node, to the spatial visual system through a `^svs` node and to the reinforcement learning mechanism through a `^reward-link` node. Any sub-state node is connected to its parent state node through a `^superstate` node (the root S1 node has a `nil ^superstate` to indicate it is the root state). Every state node also has a `^type state` attribute to characterize it as a state node (the same happen with all sub-states created by impasses).

2.5 Procedural memory

In Cognitive Psychology, the *Procedural Memory* is the part of the Memory System responsible for storing how to do things or the place where actions are stored. In other words, this memory is related to abilities or habits that cannot be verbalized, for

example: drive a car, ride a bike, play piano and so on. In SOAR, the *Procedural Memory* is where rules are stored (also called productions). Usually, rules are described by text files and further loaded in SOAR for its operation. In such a case, we might say that in principle, rules are fixed, and can not change during SOAR operation. Nevertheless, it is possible to have programmatic access to these rules in procedural memory, and hypothetically, it is possible to include, modify and delete rules from SOAR during operation. This is not an easy task, though, and the support for this is minimal.

Generally, actions of a production either create preferences for operator selection or create/remove working memory elements (LAIRD; CONGDON, 2014). The actions of a production can have two types of support: *I-support* and *O-support*. The basic difference is: all the WME's modifications (described in the RHS) applied in the working memory by an I-support production will be retracted as soon as the LHS of that production does not match with the current state. In turn, O-support modifications do not retract, even when LHS no longer matches. O-support is given only to working memory elements created by operator-application productions. An operator-application production tests the current operator of a state and modifies the state (LAIRD; CONGDON, 2014). All the other productions that do not match this requirement receive I-support.

Box 2.2: Example of O-support

```
#Testing the selected operator
sp {apply*wander
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <ol>)
   (<o> ^name wander)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 1)}
```

Box 2.2 shows an example of O-support production. The LHS is checking if the operator `wander` is selected. If it is, the changes described in RHS will be permanently applied. This rule is also an example of operator application rules.

2.6 Episodic memory

In Cognitive Psychology, the *Episodic Memory* is a part of the memory system responsible for storing episodes: events experienced by a person (or animal), including itself and other entities in the world. Episodes are bounded together in a linear fashion, creating the feeling of past-present-future in the human mind, and the possibility of a *Mental Time Travel (MTT)* while accessing the Episodic Memory, reviving lived experiences again. Episodes can be retrieved consciously and allow the remembrance of many details of past events - like e.g. what we did this morning, last week or two years ago, and also what has happened during these time frames around me (things I have noticed to happen).

SOAR's Episodic Memory implements a simplified model of human episodic memory. In SOAR, episodes are simply samples (snapshots, full copies) of the Working Memory, taken in different time steps. According to SOAR's manual, "when episodic memory stores a new episode, it captures the entire top-state of working with a few exceptions" (LAIRD; CONGDON, 2014), so the most part of the WME's in working memory will be saved, creating what SOAR considers to be an episode.

This approach can look simple, but as previously said, the working memory integrates all the modules available in SOAR. Thereby, a working memory snapshot is a simple and effective way to provide a full and contextualized information about the current state.

2.6.1 Episodic learning and retrieval

In order to be able to use the episodic memory, it is important to know how the episodes are saved and retrieved in the episodic memory.

SOAR stores episodes in a very simple way: once the episodic memory system is turned on, new episodes are automatically recorded without any deliberate action. That is, the agent does not need to invoke any special command to store a new episode. The architecture will constantly record new episodes. The time and phase that new episodes are recorded can be customized.

The retrieving process is more complex. The most common way to return an

episode is using a *cue*. In SOAR, cues work like queries in SQL. A cue is composed of a set of WME's and SOAR will search the best episode in episodic memory matching with that cue. Several episodes might match, at a certain level, with a cue. Therefore, SOAR uses a heuristic that takes in account the number of WME matches per episode and the current WME activation (see section 2.9), grouping all in a single equation and returning only the best episodes.

During the creation of a new state in working memory, at the beginning of a *reasoning cycle* (or during the creation of a new sub-state while trying to solve an impasse), SOAR creates the following augmentations to facilitate the use of episodic memory:

Box 2.3: Episodic Memory Structure

```
(<s> ^epmem <e>)
    (<e> ^command <e-c>)
    (<e> ^result <e-r>)
    (<e> ^present-id #)
```

Box 2.3 describes the WME's structure to access the episodic memory content via node `^epmem`. In order to retrieve episodes, a rule might augment the `^command` structure with a cue. In response, SOAR's episodic memory system augments the `^result` structure in response. The `^present-id` is also returned by the episodic memory system and indicates a number identifying the time in which the episode was captured.

2.7 Semantic memory

In Cognitive Psychology, a *Semantic Memory* is a kind of *Declarative Memory* used to store declarative knowledge, general statements about the world not related to time, namely, meanings and understandings. Unlike episodic memory, this memory is context independent and stores statements like "the sky is blue" or "Earth is in the solar system", or "Campinas is in the state of São Paulo".

In SOAR, the Semantic Memory is a general repository of WMEs to be stored in a long-term, in order to be retrieved later. In this sense, it has a structure which is similar to a Working Memory, but instead of being stored in a short-term memory, it is

stored in a long-term memory, just like a file in a file system. We might think of a semantic memory as a part of a working memory which is stored in a file.

2.7.1 Semantic learning and retrieval

SOAR's semantic memory works in a quite similar way to an episodic memory, with the following difference. While in episodic memory a full snapshot of the Working Memory is automatically saved at each time step, in semantic memory, the user has to clearly specify what he wants to be stored in a new record.

Similarly to episodic memory, the retrieving process in the semantic memory involves the definition of a cue in order to retrieve the recorded information. But while in episodic memory the matching process requires an exact match, semantic memory allows a partial match between the cue and the information stored in long-term memory.

During the creation of a new state in working memory, at the beginning of a *reasoning cycle* (or during the creation of a new sub-state while trying to solve an impasse).

Box 2.4: Semantic Memory Structure

```
(<s> ^smem <smem>)
     (<smem> ^command <smem-c>)
     (<smem> ^result <smem-r>)
```

Box 2.4 describes the WME's structure to access the semantic memory content via node `^smem`. In order to retrieve information from the semantic memory, a rule might augment the `^command` structure with a cue. In response, SOAR's semantic memory system augments the `^result` structure in response.

2.8 Spatial Visual System

In Cognitive Psychology, Baddeley (2012) proposed the *Visuo-Spatial Sketchpad* as a part of working memory meant to organize objects in space and establishing the spatial relations among them. In SOAR the idea of a *Visuo-Spatial Sketchpad* gave rise to the construction of a *Spatial Visual System*, which was made available only on version

9.5.0 of SOAR, released in 2015. Spatial visual reasoning is the ability to understand the visual consequences of manipulating objects in a scene. So, for example, given an initial scene where there are a car and a person, it is possible to indicate precisely where the car and the person are positioned; but, after a short period of time, the car and the person start to move. Where will be their positions now? Are they near to each other? Imagine this in an environment where there are multiple objects with different types of interactions among them (above, below, inside, etc).

Before the implementation of this module, the environment was responsible to provide all the information and relations among all objects to the architecture. However, as the Soar manual states:

> "(Spatial Visual System) allows the agent to build more flexible symbolic representations without requiring modifications to the environment code. Furthermore, it allows the agent to manipulate internal copies of the scene graph and then extract spatial relationships from the modified states, which is useful for look-ahead search and action modeling" (LAIRD; CONGDON, 2014).

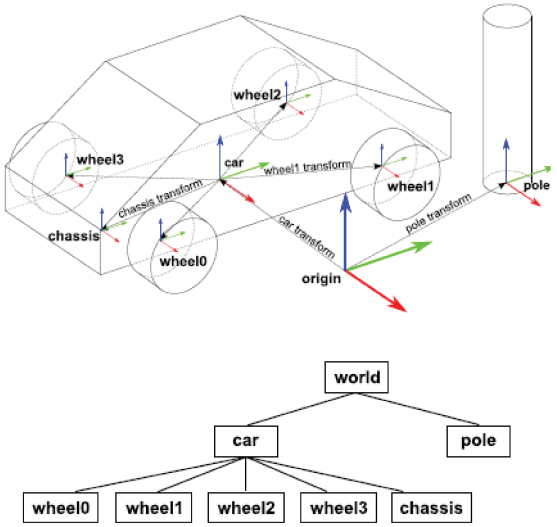


Figure 2.4 – SOAR Visual State - Extracted from (LAIRD; CONGDON, 2014)

Figure 2.4 helps to clarify the question. In this case, there is a world that is composed by a pole and a car (made of a chassis and four wheels) and it is possible to specify which object is the "parent" of some entity and also its properties (Cartesian points, circle or polyhedron, etc). Besides that, the module provides the ability to apply transforms in the objects like offsets, rotations, scales and, more than that, these trans-

forms are automatically applied to the children nodes, so it makes easy to the architecture understands what were the consequences of the environment changes.

During the creation of a new state in working memory, at the beginning of a *reasoning cycle* (or during the creation of a new sub-state while trying to solve an impasse).

Box 2.5: Spatial Visual System Structure

```
(<S> ^svs <SVS>)
  (<SVS> ^command <SVS-C>)
  (<SVS> ^spatial-scene <SSC>)
      (<SSC> ^id world)
```

Box 2.5 describes the WME's structure to access the spatial visual system via node `^svs`. The SVS maintains an internal representation of the environment as a collection of discrete objects with simple geometric shapes, called the *scene graph*. SOAR rules can query for spatial relationships between the objects in the scene graph through a working memory interface similar to that of episodic and semantic memory. SVS represents the scene graph structure in working memory under the `^spatial-scene` link. The SVS provides a Scene Graph Edit Language (SGEL), a simple, plain text, line-oriented language that is used by SVS to modify the contents of the scene graph. SOAR rules might use the `^command` link, similar to semantic and episodic memory, which is used to post queries using SGEL. These commands allow the agent to modify the scene graph, helping the system to perform visual-spatial reasoning.

2.9 Working Memory Activation (WMA)

Based on similar mechanisms available in other cognitive architectures like ACT-R or LIDA, WMA (Working Memory Activation) is a recent feature that was integrated into SOAR and, as Nuxoll (NUXOLL *et al.*, 2004) said, it is very useful for two scenarios: to retrieve which stored episodes are the best match for the current situation and to support forgetting in working memory.

To forget something is equally important as remembering something. Given that there is a finite space for storage, a forgetting process might consider the importance

of different pieces of information, privileging those which have more value. In this manner, when the WMA mechanism is turned on, each WME in working memory gains an activation value and this activation might decay or increase any time that a WME was tested by a production: if there is a match it will increase, otherwise decay. The architecture will constantly check each WME activation in working memory and if one of them has an activation value below a defined threshold, it will be discarded. So, this is an artificial forgetting process because only the WMEs with a higher activation will prevail.

Besides that, one of the toughest tasks in episodic memory is to retrieve a relevant episode, due to the large amount of data which is stored (NUXOLL *et al.*, 2004). One way of biasing this match is using working memory activation. If an episode has a high WME match, but each WME has a low activation, probably it will not be selected. Once again the activation acts as a filter that returns just the most important things.

WMA can be turned on and off in SOAR, using the `wma` command. Many configuration parameters and options are available to tune the mechanism.

2.10 Chunking

There are basically two kinds of learning in SOAR: *Chunking* and *Reinforcement Learning*. *Chunking* was the first learning mechanism to be introduced in the architecture and its function is basically to optimize the overhead generated by the sub-goals mechanism while reaching an impasse situation, creating new rules for avoiding the generation of sub-goals, by summarizing the acquired knowledge while processing a sub-goal in order to solve an impasse.

As explained in section 2.1, when an impasse is triggered, a new sub-state is created, becoming a new sub-problem to be solved by SOAR. When this sub-problem is solved, the SOAR chunking module is triggered, creating a new production, called a *chunk*, representing the sub-state solving process. In other words, when a chunk is created, SOAR will "take a picture" of the current working memory state and a new rule will be created in procedural memory. WME's that are either examined or created during the sub-state will be the preconditions of the chunk and the action of the sub-state will be the result of the chunk. The purpose is very simple: the created chunk will be matched in similar situations, avoiding the impasse and saving processing time (LAIRD; CONGDON,

2014).

However, in the same way that this module can improve the performance of the framework in some situations, it can also degenerate it in others. As pointed out by Kennedy (KENNEDY; JONG, 2003), an excessive number of new chunks impacts significantly the architecture performance, that is, if so many chunks are created, but just a few are constantly used, the time saved to solve the sub-state is wasted evaluating all the chunks in the decision cycle.

The chunking learning procedure is automatically run by SOAR while the feature is turned on. Chunking can be turned on or off using the `learn` command (e.g. at the RHS of a rule).

2.11 Reinforcement learning

This is another module that was integrated recently in the architecture and its function is to incorporate the classic mechanism of reinforcement learning in SOAR. Inspired by behaviorist psychology, it is based on the idea that for each action executed by an agent, a reinforcement value (positive or negative) might be obtained from environment, such that after some interactions, the agent might execute more times the actions associated with the positive reinforcement instead those receiving negative ones.

The reinforcement learning mechanism in SOAR relies on one specific kind of preference which can be set up in a rule's RHS: indifference preference. There are two different kinds of indifference preferences: symbolic indifference preference and numeric indifference preference, which are indicated by the following commands in the RHS of a rule:

- **Symbolic Indifference Preference:**

```
(<S> ^operator <O> =)
```

- **Numeric Indifference Preference:**

```
(<S> ^operator <O> = 1.8)
```

The only difference between symbolic and numeric indifference preferences is a constant number which is declared in the case of a numeric indifference preference. When

a symbolic indifference preference is declared, and there is no other kind of preferences in the proposed operators, instead of generating an impasse, the decision procedure selects a random operator, amongst those which were proposed, using a uniform distribution. In the case, all the preferences of proposed operators are numeric indifference preferences, the decision procedure uses the numeric value given to set the preference as a weight to generate a biased non-deterministic selection. So, the biggest the assigned numeric value, the higher the chance of an operator in being chosen.

This is a standard mechanism which can be used in SOAR to determine the preference in the selection of operators. Now, when the reinforcement learning mechanism is turned on, it is possible to use a reward value to modify these numeric values of numeric indifference preferences, resulting in a change on the probabilistic distributions used in the decision procedure which will select the operation to be applied. As described by box 2.6, this reward can be indicated using the `^reward-link` node in the current state, so, in this example, the creation of the following WMEs on the current state node will provide a reward of 1.2 to the reinforcement learning mechanism.

Box 2.6: Reinforcement Learning Example

```
(<S> ^reward-link <R>)
    (<R> ^reward <V>)
        (<V> ^value 1.2)
```

The exact change in the numeric value depends on many parameters which can be customized in the reinforcement learning module. This adjustment is better detailed in Laird (LAIRD; CONGDON, 2014) and Nason (NASON; LAIRD, 2005). At the beginning of each decision cycle, the architecture will grab the reward given in the `^reward-link` and use it to change the numeric preferences for the last executed operator, using a variation of Q-Learning or SARSA algorithms.

2.11.1 Q-Learning

Q-Learning is a reinforcement learning technique where the intent is to select the best action in a finite set of states that the agent can assume. It was proposed by Watkins in 1989, but only in 1992 its convergence was proved (WATKINS; DAYAN, 1992).

Basically, given a Q-matrix, where all possible agent states and all possible actions are described, the idea is to update this matrix with the obtained rewards in order to allow the agent to choose the most convenient action. Next, there is a description of the algorithm:

1. Initialize the Q-matrix (usually randomly but it is also possible to set all values to zero or use any other initialization policy) as exemplified below. The capital letters represent the states and the lowercase letters represent the actions that can be applied in that state. The values in the matrix will represent how good (or bad) it is to select that action in that state.

$$\mathrm{Q} = \begin{array}{c} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{matrix} x \\ y \\ z \end{matrix} \end{array}$$

2. Choose the action to be selected given the current state. Here, it is possible to use several approaches, for example:

   a) ε-greedy: the action with the highest Q-matrix value for that state will be chosen most of the times and, only in a few cases (with probability ε), the other actions will be randomly chosen (for those, it does not matter what are the Q-matrix values, the probability is uniformly distributed).

   b) softmax: the action is not selected with a uniformly distributed probability, but on the contrary, a weighted probabilistic distribution is applied, based on the Q-matrix value.

3. Evaluate the reward received after applying the action and the new state of the agent.

4. Update the Q-matrix according to the following equation:

$$Q(S_t, a_t) = Q(S_t, a_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, a_t)] \qquad (2.2)$$

where $Q(S_t, a_t)$ is the Q-matrix value for the previous state and for the selected action, α is the learning rate (between 0 and 1) meaning how fast the agent will learn based on the rewards, R_{t+1} is the reward obtained after applying the selected action, γ is the discount factor (between 0 and 1) that will be applied to the reward and $max_a Q(S_{t+1}, a)$ is the maximum Q-matrix value for the next state (in our example, it is the maximum value located in the column associated with the new state, considering all possible actions for that state).

2.11.2 SARSA

The SARSA (State Action Reward State Action) is also a reinforcement learning technique and it is very similar to the Q-Learning algorithm. The main difference is how the Q-matrix values are updated (all the other steps in section 2.11.1 remains the same).

In short, the Q-matrix value of the previous state is not updated using the maximum Q-matrix value of the next state (considering all possible actions), but it is updated using exclusively the Q-matrix value of the next state and action. So, the Q-matrix will only be updated after two iterations, i.e., after two action selections.

$$Q(S_t, a_t) = Q(S_t, a_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)] \tag{2.3}$$

In some cases, Q-Learning can lead to some issues due to its approach to selecting just the maximum value. For example, the greediness of selecting a short path instead of a more safe path (e.g. as described in an experiment with a robot and a cliff (SUTTON; BARTO, 1998)). So, SARSA tends to be more conservative and more slow to converge, but it provides other characteristics, like safety for example.

2.12 Summary

During this chapter, we described in details how each module of SOAR works and how they contribute to the cognitive cycle.

SOAR cognitive cycle starts with data being provided to the architecture as WME's (a symbolic entity that helps to describe the problem-solving context). These

WME's are sent to Workspace where they can be modified by other modules in architecture (e.g. memories). After that, SOAR tries to find the best operator (state modifiers described by WME's and rules) to apply in the current state. SOAR analyze all operators that meet the current state and after evaluating all the preferences and impasses, only one operator is selected to be applied. This process is repeated until the desired state is reached.

In the next chapter, we follow the same approach for CLARION.

3 The CLARION Cognitive Architecture

Developed at Rensselaer Polytechnic Institute by Ron Sun, CLARION (Connectionist Learning with Adaptive Rule Induction On-line) is a cognitive architecture with a clear inspiration in *Connectionism*, a strong paradigm in Cognitive Science which dominated the 90's and still today has a strong influence in the intelligent systems community. Despite its large inspiration on neural networks, CLARION is classified as a hybrid architecture. In opposition to cognitivist (symbolic) approaches, connectionism emphasizes the sub-symbolic characteristics of the reasoning process. CLARION tries to balance this connectionist inspiration by including explicitly rule-based processing together with neural networks, providing somewhat an integration of symbolic and sub-symbolic approaches.

In (SUN, 2004), Ron Sun lists a set of characteristics that every cognitive architecture should have in order to create biologically inspired systems and one of them is the dichotomy of *explicit* versus *implicit* processes or, in other words, symbolic and sub-symbolic processes. According to him, the presence of this dichotomy in cognitive architectures is very important, because it values the union of different cognitive processes, where one is more precise and direct and the other is more holistic and imprecise, in such a way to represent more accurately the different processes involved in the human cognitive cycle.

This dichotomy is intrinsic in the design of CLARION where each module is divided into two parts: *top-level* responsible for the explicit knowledge (symbolic) and *bottom-level* responsible for the implicit knowledge (sub-symbolic). As can be visualized in figure 3.1, the architecture is composed of four big modules: Action Centered Subsystem (ACS), Non-Action Centered Subsystem (NACS), Motivational Subsystem (MS) and Meta-Cognitive Subsystem (MCS). These modules are connected to each other and the interaction among them generates the cognitive process.

From CLARION's website[1], an executable version of CLARION can be freely downloaded. Although the source code is not opened, the binaries are totally available. Recently, the implementation of this architecture has undergone big changes, moving from

[1] URL: <http://www.clarioncognitivearchitecture.com>

Java to C#.

In the next sections, we cover each module of CLARION, which will be explained in more details. Further on in this thesis, we provide examples showing up how the architecture can be used in a more pragmatic way.

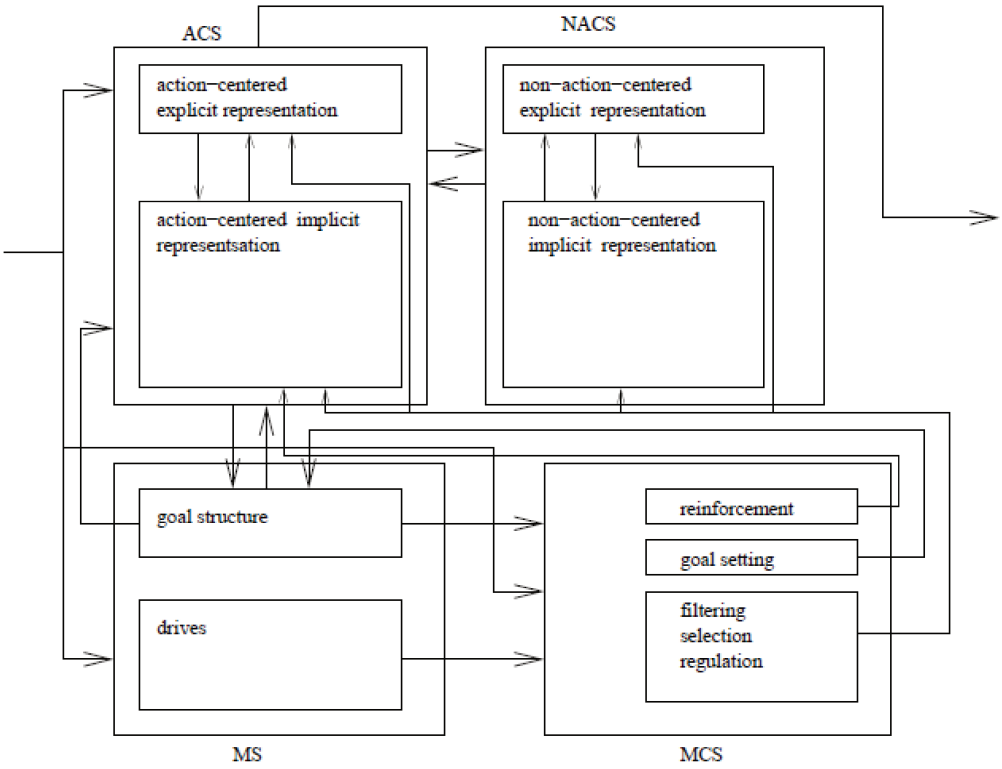


Figure 3.1 – CLARION Architecture - Extracted from (SUN, 2006)

3.1 Action centered subsystem - ACS

The most important module in CLARION is its Action-Centered Subsystem (ACS). As its name says, this module is responsible for selecting the best action to be applied to the environment, in a given situation. Looking to figure 3.1, the ACS receives data coming either from the environment, from NACS and from MS. On the other hand, it also acts on the environment, NACS and MS. Its operating cycle can be described as:

1. Get data from environment related to the current problem-solving state
2. Suggest an action, using the bottom-level mechanisms
3. Suggest an action, using the top-level mechanisms

4. Select the action that will be executed taking into account the suggestions from both bottom and top levels

5. Run learning algorithms on bottom and top levels

It is important to highlight that the sensory inputs provided to the architecture should be in a *dimension-value pair* format, i.e., a set of tuples where one element refers to the type of the sensory information and the other element refers to the value itself, for example: (temperature, 32); (distance, 15) or (presence of predators, true).

3.1.1 ACS Bottom level

The bottom level of each module in CLARION consists of implicit knowledge and, in ACS, this implicit knowledge is manifested mainly as a neural backpropagation network with a feedback provided by the environment (positive or negative). The current implementation of this architecture provides to the user the possibility to choose a pre-implemented Q-Learning neural network or use a basic generic equation (very useful for debugging) or even the possibility to implement a custom neural network.

It is important to emphasize that the Q-Learning neural network used in CLARION uses the basic algorithm already described for SOAR (see section 2.11.1), but applied on a neural network. If we have a large number of possible states and actions, it can be very difficult to update all the Q-value matrix, so Q-Learning using neural network can be a good approximation for the Q-value matrix, being more simple and requiring a short period of learning. Figure 3.2 helps us to better understanding the scenario: a neural network w will receive as input the current state of the problem x and the current action selected by the agent a. The neural network will generate as output a numeric result, represented by $Q(x, a)$ that will be compared with the best possible value represented by $Q^{target}(x, a)$, the same algorithm as the one presented in equation 2.2. The idea is to train this neural network in such a way to minimize the error for $Q(x, a) - Q^{target}(x, a)$ adjusting the internal weights of the network.

The bottom-level can receive data and actuate in three locations: environment, NACS and MS. So, a network in the bottom-level can take into consideration the current dimension-value pairs, the activated chunks and also the current goals in order to suggest

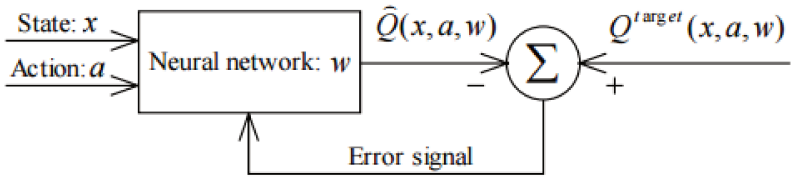


Figure 3.2 – Q-Learning in neural networks - Extracted from (HUANG *et al.*, 2005)

an action. On the other hand, the output can actuate in these three places independently, i.e., a network can always use the data independently what is the source of the information, but it should actuate only in one specific location (environment, NACS and MS), as illustrated in figure 3.3.

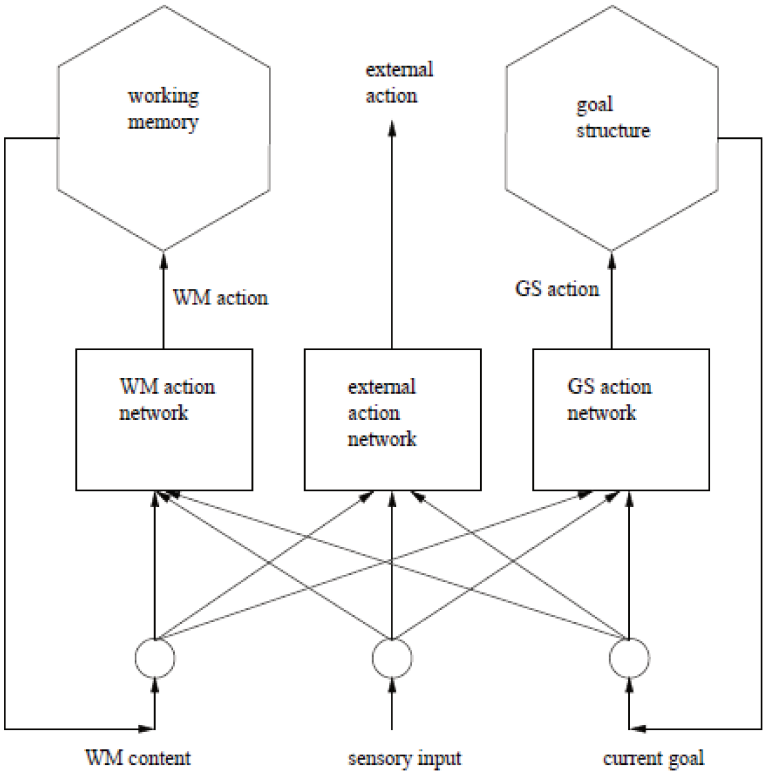


Figure 3.3 – CLARION ACS Bottom Level - Extracted from (SUN, 2003)

3.1.2 ACS Top Level

The *Top Level* sub-module of ACS deals with explicit knowledge, i.e. declarative knowledge which can be expressed with symbols (propositions or predicates) and rules. Rules follow the pattern condition/action, where the condition will match the current state of the agent (as in the bottom-level, dimension-value pairs, and chunks activated

in NACS) with an expected situation, and the action will describe the changes that will be applied if the rule is selected for execution. As in the bottom-level, an action in the top-level can actuate only in one of three different outputs: environment, NACS, and MS.

Rules are divided into three different categories (SUN, 2003):

- **RER *(Rule Extraction Refinement)* Rules:** These rules are generated using the bottom-up learning mechanism (more details in section 3.1.4). Basically, if an action suggested by the bottom-level is selected for execution, the learning mechanism will generate a rule in the top-level with the preconditions and actions used in the bottom-level.
- **IRL *(Independent Rule Learning)* Rules:** IRL rules are quite similar to RER rules. The main difference is the origin of the rule. RER rules have their origin in the bottom-level, and then the rule is propagated to the top-level. In IRL, rules are independently generated directly in the top-level using some heuristic or even wired-up. After some interactions, IRL rules are tested according to their efficacy and, as a result, they can be removed or refined (similar to RER).
- ***Fixed Rules (FR)*:** This is the most simple type of rules because they represent the prior knowledge acquired by the agent (pre-wired) and they are not subjected to a refinement process (like RER and IRL).

3.1.3 Action selection

In the *action selection* stage, ACS has the suggestions provided by both the top and bottom levels and it must select the most appropriated action. It is important to remember that the action can be applied in three different domains: environment, NACS or MS. Basically, the first step in action selection is to separate the suggestion from the top and bottom levels which are applicable to the same domain, so, there will be three different groups including the suggestions from the top and bottom level for a specific domain.

In the context of each group, it is still necessary to select just one action. CLARION provides two algorithms to select the final outcome of each group: *stochastic* and *combination*.

For the stochastic method, there is a pre-defined probability of selection for each type of source suggesting an action in CLARION (if that source suggests at least one possible action). Therefore RER rules will have a probability P_{RER}, IRL will have a probability P_{IRL}, FR rules will have a probability P_{FR} and, finally, the bottom-level will have a probability:

$$P_{BL} = 1 - P_{RER} - P_{IRL} - P_{FR} \tag{3.1}$$

These probabilities might be static (pre-set) or variable. If they are static, they should be specified before the beginning of the simulation with constant values. However, if they are variable, they are dynamically re-calculated during the simulation, based on the success rate of the chosen actions. So, for example, if an action from bottom-level was chosen as the winner and its result generated a positive reward, then the probability to choose bottom-level actions will increase (when compared to the other possible sources). Once the probability of each source is defined, a selection algorithm, like the "roulette wheel algorithm", is used to define the "winner source", but it is possible to have multiple suggested actions in the same "winner source", so it is time to a tie-break, selecting just one action.

If the selected source of action is from ACS top-level, the *rule utility* will be used as a tiebreaker in a Boltzmann distribution. Equation 3.2 describes how the rule utility is calculated. Basically, it is the rule benefit minus its cost with a scale factor. As Sun states (SUN, 2003) the benefit of a rule is calculated based on the positive match ratio, i.e., how many positive matches a rule produces within the context of all the possible matches by the rule (a positive match criterion needs to be set). The cost of a rule is set based on the execution time considerations.

$$U_j^r = benefit_j - v * cost_j \tag{3.2}$$

Thus, the probability for selecting a rule i in the top-level given the current problem state in a moment x will be:

$$p(i|x) = \frac{e^{U_i^r/\tau}}{\sum_j e^{U_j^r/\tau}} \tag{3.3}$$

Where U_i^r is the utility for rule i, τ is the temperature of the Boltzmann function, representing the randomness of selection, and the sum in the denominator represents the sum of the utility functions of all rules that meet the current problem state in a moment x.

It is important to note the role of the τ constant in equation 3.3. The lower the value of τ, the higher the utility function will be predominant to select a rule, i.e., if $\tau \to 0^+$ the best action will always be selected. On the other hand, if τ has a high value, more randomness will be obtained, i.e., if $\tau \to \infty$ the actions are chosen with the same probability. This is the softmax algorithm (WHITESON *et al.*, 2007) (already described in section 2.11.1).

If the source of action is from the bottom level, instead of the utility function, the return of the Q-Learning function, $Q(x, a)$, is used (the same described in section 3.1.1) as a utility function.

Finally, it is time to focus our attention on the second way to select an action: *combinational*. The combinational way is quite similar to the stochastic way. The main difference is: in the stochastic way the source of an action (RER, IRL, FR or BL) is selected first by a probabilistic distribution and then the action that belongs to that source is selected using a Boltzmann distribution. However, in a combinational way, the maximum utility for each possible source of action is computed (obviously we only consider the sources that have suggested at least one action at that time), i.e., this number represents the highest utility for an action that matches the current state of a given source. So, for RER, IRL, FR, and BL, there will, respectively, u_a^{RER}, u_a^{IRL}, u_a^{FR} and q_a (this one represents the Q-function return instead of the rule utility function), representing the maximum utility function at that time.

$$u_a = w_{RER} * u_a^{RER} + w_{IRL} * u_a^{IRL} + w_{FR} * u_a^{FR} + w_{BL} * q_a \tag{3.4}$$

After that, a weighted sum is performed, as demonstrated in equation 3.4, gathering all utilities together and generating a combined value. Finally, this combined value u_a will be used as a threshold, so all the suggested actions with a utility function lower than this value will be eliminated and all the suggested actions greater or equal to

this value will be picked and a Boltzmann distribution will be applied on those (using the same principles as described in equation 3.3).

At the end, there may be three final actions, each one applicable to a different domain (of course that, in a given situation, not all domains are going to necessarily have an action, so it is possible to have just one, two or three domains with possible actions). CLARION can use one of the following approaches to deal with this multi-domain actions (SUN, 2003):

- Random: each domain will have a specific probability and CLARION will select just one action (among the three possibles) based on the probability of each domain.
- External first: if there is an action to be applied in the environment, it will be always chosen. Otherwise, an action from the other domains will be picked (based on the probability for that domain).
- All: apply the actions in all domains simultaneously (although, this approach can impact the performance).

3.1.4 Learning

CLARION provides learning between ACS levels (top-bottom and bottom-up) and also within the same level. Considering learning at the same level, CLARION has an intrinsic reinforcement learning algorithm implemented in the bottom level of ACS based on *backpropagation* networks. Generally, the bottom level has a Q-Learning neural network, whose reinforcement is provided by MCS (see section 3.4.2). Thus, combining the reinforcement provided by MCS, the neural network can adjust its weights in such a way to maximize the reward.

Regarding learning between levels, the bottom-up learning is responsible for generating RER rules (see section 3.1.2). The basic algorithm is described below (SUN, 2003):

1. If the action selected to execute is from the bottom level, a new rule in the top-level is created with exactly the same inputs and outputs suggested by the bottom-level.

2. In the next interactions, if the created rule was selected for execution, check the outcome of the rule.

    a) If the outcome is successful, try to *generalize* the rule in order to make it more universal (for example: removing some conditions)

    b) If the outcome is not successful, try to *specialize* the rule (for example: adding some conditions)

It is important to note that the RER rules are not stored forever. Each rule has a set of statistics and if these values become below a defined threshold, the rule is removed from the top-level.

The top-down learning is simpler. It consists of a classic neural network learning and is generally made using FR rules (innate knowledge). However, IRL rules can be used too. So, the neural network will have a supervised training in order to define its functional mapping based on the knowledge provided by the top-level rules.

3.2 Non-action centered subsystem - NACS

Several studies have already detailed the key role of memory in intelligent behaviors, and this is exactly the main functionality of the NACS: storing and retrieving knowledge. This system is close to what has been referred as semantic memory or declarative memory in the literature (SUN, 2003). The NACS structure (top and bottom level) is described in figure 3.4 and detailed in the next sub-sections.

3.2.1 NACS Bottom level

The NACS bottom-level is responsible for the processing of implicit knowledge, being composed by associative memory networks (AMN), having as inputs and outputs a set of dimension-value pairs. To accomplish this goal, two types of neural networks can be used: auto-associative or hetero-associative.

For auto-associative memories, the same set of dimension-value pairs are presented as inputs and also as outputs. The idea is that, after some training cycles, the internal network weights are adjusted in order to retrieve a piece of data from only a tiny

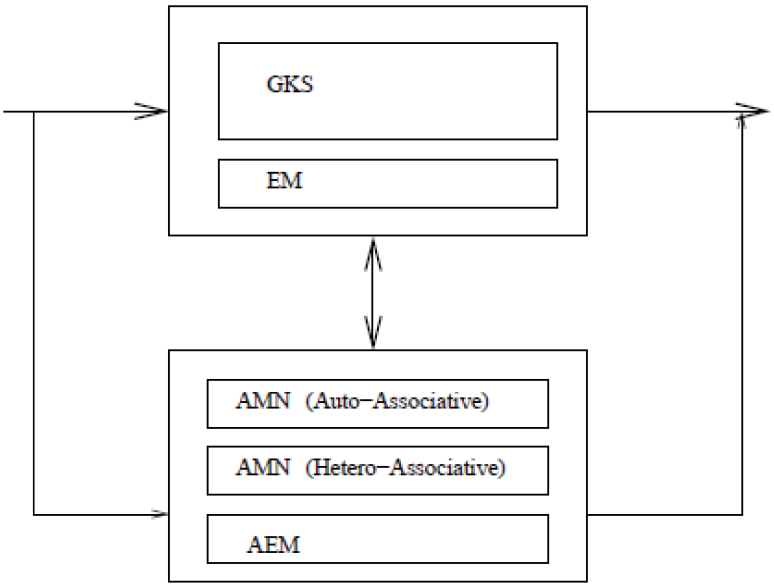


Figure 3.4 – CLARION NACS - Extracted from (SUN, 2003)

sample of itself. In other words, the net can be used to recover the trained output from a distorted input (recall familiar inputs). This is very useful for data validation when we have only partial data being presented (as e.g. in security systems where the control is made by voice detection).

On the other hand, for hetero-associative memories, a set of inputs is trained with a specific output in order to allow the detection of a pattern, given the current set of inputs. So, after some training, if an input is provided, the network will try to detect the same category in this set of data. This technique is very useful when we have a noisy data being presented and we would like to extract some patterns (as e.g. in the detection of known geometric forms in a visual image).

In CLARION implementation, there are a set of pre-implemented networks like the Hopfield network (auto-associative memory), but it is also possible for the user to have his own implementation in the bottom level.

3.2.2 NACS Top level

At the top-level, the explicit knowledge representation is the *General Knowledge System (GKS)*. It is responsible for encoding the symbolic representation of concepts, referred as *chunks*.

A chunk can be faced as a set of features (dimension-value pairs) that together will have an abstract meaning, a symbolic representation, and it can use AND/OR logical functions. For example, "cold" AND "white" AND "wet" can specify "snow". Besides that, the links across chunks can be generated and also stored in GKS, e.g., "snow" AND "human shape" can specify "snowman". Each chunk has a strength value associated, i.e., a number between zero and one, that represents its activation level. This value is calculated based on different sources of activations: other chunks and dimension-value pairs related to that chunk; an explicit activation coming from ACS (the ACS can change the activation of a chunk deliberately) and an associative mapping coming from the bottom-level (see section 3.2.3). The chunk activation will be the maximum value among all the sources.

3.2.3 Integrating the two levels

One important thing to note is the integration between the bottom and the top levels in this module. As previously discussed, the AMN inputs and outputs are in the form of dimension-value pairs. In turn, chunks located in the top-level can be viewed as an outcome of a set of dimension-value pairs and/or other chunks. So, the outputs of AMNs can trigger the activation of some chunks in the top-level (bottom-up activation). However, it is also possible for a chunk to be activated, but their dimension-value pairs in bottom-level are not. This can happen because a chunk can be associated to other dimension-value pairs or even to other chunks, so it is possible to activate a chunk via another "branch". When this scenario happens, a top-down activation takes place in order to activate the associated dimension-value pairs that are not activated in the bottom-level.

Basically, NACS receives data from ACS and from external sensors. In parallel, the top-level calculates the strengths of each chunk and the AMNs also generate their outputs. First, the bottom-up activation occurs and then the top-down activation takes place. After that, a new round of interactions can occur until a pre-defined number of cycles is reached or until no more chunks are created.

3.3 Motivational subsystem

If a cognitive architecture aims to be faithful to the way humans identify and solve problems, it is necessary to think about incorporating goals and motivations in an

artificial agent. Having goals is one of the many factors distinguishing intelligent beings from merely reactive ones and, as Ron Sun pointed out (SUN, 2004), this consideration is important to the development of any kind of cognitive architecture that aims to be biologically inspired. This concept is strongly related to the notion of *final cause* held by Aristotle, i.e., that there is a purpose guiding some kinds of actions.

CLARIONS' Motivational Subsystem (MS) is not a standalone subsystem because it is closely tied to the other modules. Its main functionality is to help the agent to choose the best action for the current situation and, as in any other module in CLARION, it has a symbolic top-level and a sub-symbolic bottom-level (as can be viewed in figure 3.5).

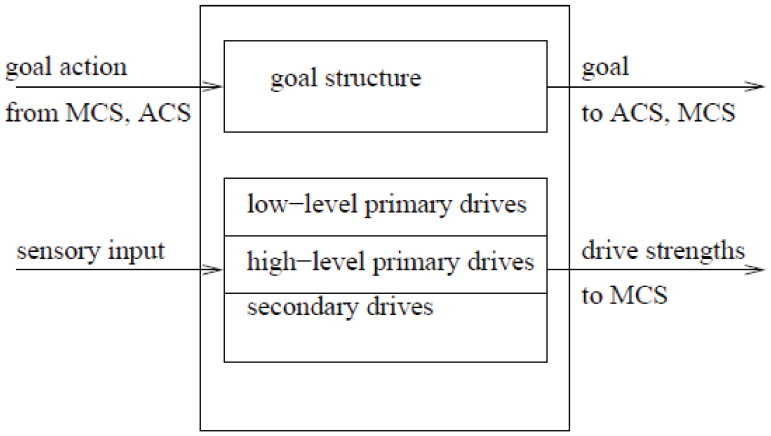


Figure 3.5 – CLARION Motivational Subsystem - Extracted from (SUN, 2003)

3.3.1 MS Bottom level

The MS bottom level is composed of three different kinds of *drives*, a concept initially proposed by Hull (1943). According to Hull, organisms have internal and psychological *needs* like e.g. an energetic need, a water need and so on. Drives are measurements of how intense are these needs, in a given instant of time, and work as impulses leading to actions which will help the satisfaction of them. Along time, drives levels become unbalanced due to changes in the environment or in the organism's body. For instance, when an agent's energetic level goes down, this increases the hunger drive. When the hunger drive becomes higher than all other drives, this leads to an action like look for food, and then, in finding it, eating this food, resulting in the normalization of the energetic balance and

the hunger drive going down again. Thus, the basic idea behind drives is a competition among the different drives, leading to a winning drive and then a further action which leads to the satisfaction of the implicit needs, and the return to a state of homeostasis and relaxation.

The architecture provides support for the following kinds of drives:

1. **Primary low-level drives**: represent physiological needs. They are "hard-wired" in the agent and usually acquired during evolution. For example: eat, drink, sleep.

2. **Primary high-level drives**: these are not related to physiological needs, but represent high-level needs, in the sense of Maslow (1943). They are mostly related to social behaviors, as e.g. honor, justice, curiosity, but equally to primary low-level drives, they are "hard-wired"

3. **Secondary drives**: derived from other drives, they are acquired during the process of satisfying particular compositions of primary drives. Differently from primary drives, secondary drives are usually subject to some sort of learning, using as a base primary drives and input from the environment.

In CLARION, the determination of a drive level can be realized by means of two different approaches: by neural networks or by explicit equations. A CLARION modeler can create a custom neural network and associate the drive level to its output. Or, he can use equations (in a totally customized way) in order to specify the drive levels. More details on this are available on section 7.2.

In the Clarion Motivational System, the bottom level is not directly used to generate actions on the ACS. Motivational system's bottom level is highly integrated with its top level in order to affect ACS. Once drives levels are computed, they are sent to the MCS subsystem (see section 3.4). Drives are used together with other pieces of information by the MCS in order to trigger updates in the goal structure (MS top-level). Only the output of MS top level is used to affect ACS and generate actions in the system. The role of MS Top Level is explained in next section.

3.3.2 MS Top level

The *goal structure* is the main component of MS top level, and it is shared with the ACS in order to generate actions. Two different kinds of goal structures can be used: a *goal stack* or a *goal list*. Both hold internally a set of *goal chunks*, from where a single goal chunk is selected. A *goal chunk* is basically a symbol describing an objective, which together with parameters given by dimension-value pairs, can be used to select the most appropriate action for the current cognitive cycle. Each goal chunk has an associated numeric value called base-level activation (BLA), which will be used in order to select the winning goal. This selection is performed using a Boltzmann distribution (the same one described on equation 3.3) based on the BLA. Equation 3.5 describes how BLA is computed for each goal chunk.

$$B_i = B_i^0 + c * \sum_{l=1}^{n} t_l^{-d} \tag{3.5}$$

Goal chunks enter the Goal Structure by means of *goal actions*, performed by the MCS goal settings submodule (see subsection 3.4.1). Goal actions can basically set or reset a goal chunk within the goal structure. A goal chunk set will insert the goal chunk in the goal structure, if it is not already there, or update its BLA, if it is. A goal chunk reset will delete the goal chunk from the goal structure. In equation 3.5, B_i represents the BLA for the goal i. B_i^0, c and d are constants, n is the number of times that the goal was set in the goal structure and t_l is the time since the last l^{th} set of the goal in goal structure (measured in milliseconds). The default values are: $c = 2$, $d = 0.5$ and $B_i^0 = 0$. Based on this equation, it is possible to note that the BLA of each goal decays during time and if a goal is set multiple times, it receives a boost in its BLA.

So, the goal structure works as a repository of many goals, from which a single goal is selected and presented as an input to the ACS, where it can be used to define an external action. The working of a *goal stack* is simpler. It uses a simple strategy of putting goal chunks on a stack (the last goal chunk becoming the current goal), and once a goal chunk finishes its usefulness, it is taken out from the stack and the next goal action in the stack becomes the current goal. The *goal list* has a more complex behavior, using the BLA activation in order to select the current goal chunk. As Sun pointed out (SUN, 2003), the *goal structure* provides specific and tangible motivations for actions, in the form of

goal chunks, which can be used by the ACS in order to have more behavioral flexibility and formation of expediencies, also being more persistent and longer when compared to a simple drive mechanism as proposed by Hull (sometimes this persistence is important for the agent). So, the main role of the goal structure is to provide to the agent persistence and continuity in its actions.

A CLARION modeler will explicitly define *goal chunks*, creating a set of possible goals for the agent. Using drives from the MS bottom level, a modeler can describe goals based on the internal needs of the agent and also on a desirable state for the environment. For example, a top-level goal for "to eat" can be composed by the "hunger" drive and also for all the required dimension-value pairs used to identify a "food".

It is important to emphasize the tight connection between MS and MCS such that MS can work properly. Figure 3.6 gives an overview of the relation between these modules. The many drives in MS will calculate their excitations and these values will be sent to MCS. Within MCS's Goal Setting submodule, a *GoalEquation* will compute a *goal strength* for each goal, based on drives excitations and state dimension-value pairs. After evaluating all the strengths and using a predefined strategy (see section 3.4.1), the *GoalSelection* generates a *goal action*, which is sent to the Goal Structure. Finally, while receiving a goal action setting a goal chunk which is already in the Goal Structure, its BLA is updated accordingly. Besides that, ACS can also set (or remove) a goal chunk in the goal structure (see section 3.1, particularly figure 3.3), without using drives and MCS, providing the possibility of directly changing its goals based on the current or previous actions.

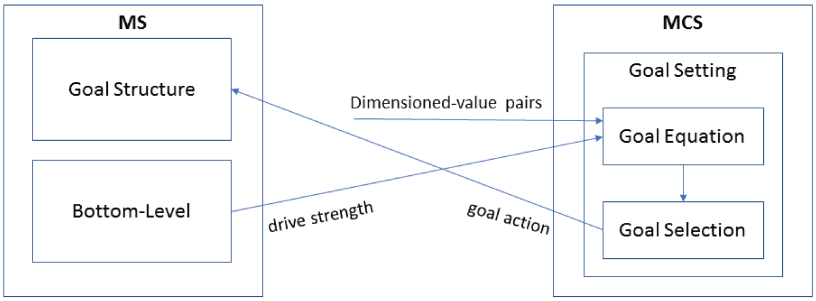


Figure 3.6 – CLARION Goal Setting Overview

3.4 Meta-cognitive subsystem - MCS

From cognitive psychology, meta-cognition refers to an individual's knowledge about its own cognitive processes, or anything related to them (COX, 2005). In Clarion, the Meta-Cognitive Subsystem (MCS) acts essentially as a "mini-ACS", except that instead of acting on the external world, the actions are directed to the internal aspects of the architecture, providing an agent with self-regulation. As in other modules in the architecture, MCS processes can be performed by rules (explicit knowledge) or equations/neural networks (implicit knowledge). However, in this subsystem bottom-level will prevail, because meta-cognitive processes are usually faster and effortless, being mostly implicit (SUN, 2003).

This module is divided into different sub-modules (as detailed in figure 3.7), each one responsible for a specific regulation process, which will be described in next sections.

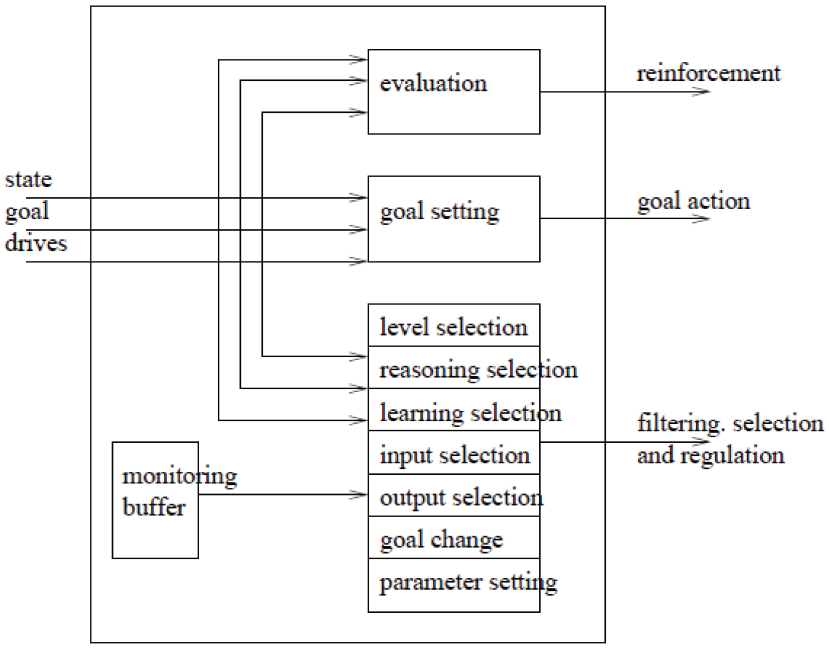


Figure 3.7 – CLARION Meta Cognitive Subsystem - Extracted from (SUN, 2003)

3.4.1 Goal setting

As discussed in section 3.3, the MS subsystem relies on MCS in order to set up the goals in the *Goal Structure*. Basically, the *Goal Setting* task evaluates the many

drives coming from MS, together with other factors coming from the environment, and define *Goal Actions*, which affect the Goal Structure. These actions are basically to set and reset a *Goal Chunk* in the Goal Structure. If a goal action sets a specific goal chunk, and this goal chunk is not already in the Goal Structure, then it is included. If the goal chunk is already in the Goal Structure, a set action will boost the BLA activation of the goal chunk, which usually decays with time while within the Goal Structure. If a goal chunk is within the goal structure, and the MCS orders a reset goal action, then the goal chunk is deleted from the goal structure.

In order to define goal actions, a *Goal Equation* computes a goal strength for each goal chunk, using as input drive levels and state information in the form of dimension-value pairs, weighted by relevance factors, as indicated in equation 3.6. This relation between goals, drives and sensory information must be specified by the CLARION modeler and is described in more details in section 7.2.

$$GE_g = \sum_{d=1}^{n} DL_d * Rel_{d,g} + \sum_{p=1}^{n} DV_p * Rel_{p,g} \tag{3.6}$$

The goal equation GE computes the goal strength for a specific goal chunk g as the sum of the drive levels DL of each drive d, multiplied by the relevance factor $Rel_{d,g}$ between g and d. This is combined with the sum of the activation DV of the several dimensioned-value pairs p multiplied by the relevance factor $Rel_{p,g}$ between each p and g.

The MCS can opt for two strategies while choosing the next goal:

1. A balance of interests: goals strengths are calculated according to equation 3.6 and the goal chunk with the highest score becomes the goal to be set in the next *goal action*. In this strategy, drives can be attached to more than one goal chunk (differently from next strategy).

2. Winner takes all: drives are attached to just one goal chunk. The drive with the highest level wins (deterministically or stochastically) and the new goal action considers the unique goal chunk attached to it.

The architecture also provides means to remove previously set goal, if this is required.

3.4.2 Reinforcement

As discussed in section 3.1, ACS has an intrinsic bottom level learning process, based on Q-Learning Backpropagation networks. This learning process relies on the evaluation of a reward measuring "how good" or "how bad" was an act. This sub-module in MCS has the role of providing to ACS bottom-level the reinforcement value that will be used for adjusting the internal weights of the network (as can be viewed in figure 3.1, the output of this module will be used as input in ACS bottom-level).

As highlighted by Sun (SUN, 2003), the world does not provide the agent with a scalar reinforcement signal, as usually assumed in the literature. Instead, the world simply changes its state after an action is performed. Thus, the reinforcement signal should be calculated internally by the agent in order to verify how appropriate the performed action was.

In order to accomplish this task, sensory information (internal and/or external), drives levels and goal chunks in the goal structure are used as inputs. Using these inputs, the degree of satisfaction of drives and goals are evaluated. At the end, a scalar reinforcement value is generated, which is used as feedback.

3.4.3 Filtering, selection, and regulation

Filtering is a mechanism allowing the agent to focus its attention on the most relevant aspects of the sensory information, excluding undesired inputs. Similarly to the reinforcement function (see section 3.4.2), a filtering function takes in account the drive levels, the current goal, and some sensory information in order to filter the most appropriate dimension-value pairs, while others have their activation multiplied by a scalar number close to zero. Therefore, they will have less impact on the next CLARION cognitive cycle. This sub-module tries to reproduce a typical situation where our concentration is totally directed to a specific task and trivial things around us do not take our attention.

On the other hand, selection and regulation are related to the ability of self-regulation or, in other words, the ability to change internal gains, parameters or even to select different algorithms for the purpose of maximizing agent's performance in a given situation. Consider the following examples of possible types of regulations:

- to change the learning mechanism (Q-learning with Backpropagation, simplified Q-learning, disable it, etc),
- to change the probability of selecting an action from top-level or bottom-level,
- to change the temperature in the Boltzmann distribution,
- to change the learning thresholds and other aspects.

All these types of self-regulation are accomplished using the same approach of other MCS sub-modules, as already highlighted, i.e., using a selection/regulation function taking into account all the relevant information (drive levels, goals, inputs/outputs) and, based on that, changing the internal aspects of the agent, optimizing its performance. In current CLARION implementation, all these parameters or algorithms that can be changed are represented either as a constant or as a class variable that can be easily changed.

3.5 Summary

During this chapter, we described in details how each module of CLARION works and how they contribute to the cognitive cycle.

CLARION cognitive cycle starts with data being provided to each module of architecture as dimension-value pairs. NACS module is responsible for storing and retrieving knowledge working as a semantic memory or declarative memory. MS module is responsible for providing new goals based on the Hull's drives theory. MCS module is responsible for the self-regulation aspects of the agent. Finally, all data from the other modules are sent to ACS where it will select the most appropriated action to be taken based on the suggestions provided by the top and bottom levels.

In the next chapter, we follow the same approach for LIDA.

4 The LIDA Cognitive Architecture

Stan Franklin and other contributors started the development of the LIDA cognitive architecture at the University of Memphis in the 90's. LIDA is an enhancement of a previous architecture developed by the group, IDA (Intelligent Distribution Agent), where many different learning processes were incorporated (LIDA means Learning IDA). LIDA implements several modules aiming at reproducing cognitive features presented by living beings, like memory, attention, perception, consciousness, and others (see figure 4.1).

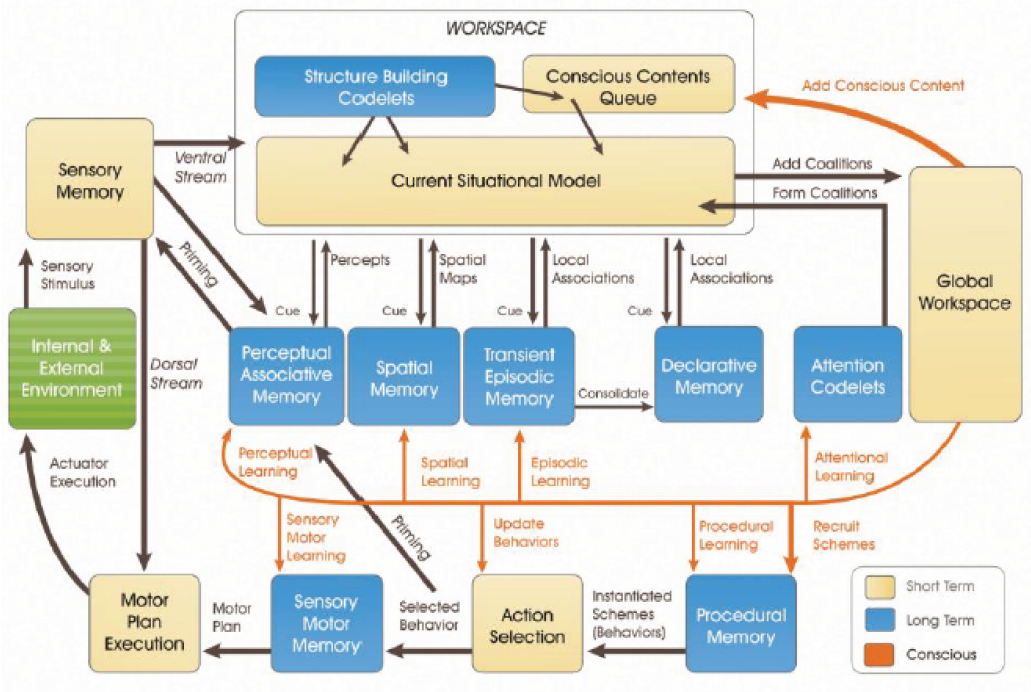


Figure 4.1 – LIDA Architecture - Extracted from (FRANKLIN *et al.*, 2016)

An important feature of this architecture is its implementation of Bernard Baars' model of consciousness, the *Global Workspace Theory*, turning it into one of a few architectures claiming to implement a model of consciousness for artificial agents. According to Gamez (GAMEZ, 2008), the basic idea of this theory is that a number of separate and parallel processes compete to set their information in the global workspace, a kind of virtual memory region. Once this information reaches the global workspace, it is broadcast to all other processes, which can analyze the data and potentially create

coalitions with the information producer, working towards a common goal or action. The global workspace is not a physical memory region, but a privileged condition that any memory can momentarily experience when it is allowed to make the *broadcast*.

Another way to understand Baars' concept of consciousness is appealing to the "Interactive Theater Metaphor". This metaphor is described while imagining a theater totally covered by dark (the theater of consciousness) where there is an audience (also in the dark), a stage and a spotlight pointing to some place in the stage. In this interactive theater, actors are a part of the audience, which momentarily decides to go to the stage and perform some action, triggered by what they are seeing at the stage. The enlightened spot can have one or more actors, playing and being watched by all the audience (the performance becoming the contents of consciousness). Seeing this performance, members of the audience can be excited and suddenly decide to go to the stage to also play and, according to their performance, receiving the spotlight (the consciousness), and potentially affecting all the others in the theater, in an endless cycle. Besides that, behind the scenes, there are a lot of other people, like the director, stagehands, scriptwriters, scene designers, working without being noticed by the audience, but helping to determine who will be illuminated by the spotlight in the next moment.

Complementing his *Global Workspace Theory*, Baars also proposed a general framework (see figure 4.2) where consciousness appears together with other functions composing a model for human cognition (BAARS; GAGE, 2010). This framework was also an inspiration in the development of LIDA and will be referred while making a parallel to LIDA's modules.

The development of LIDA put together, in an integrated way, many different computational mechanisms, some of them quite old but out of the mainstream, like e.g. Hofstadter's Copycat architecture (HOFSTADTER *et al.*, 1994), Kanerva's Sparse Distributed Memory (KANERVA, 1988), Drescher's Schema Mechanism (DRESCHER, 1991) and Maes's Behaviour Networks (MAES, 1989). According to Franklin *et al.* (2007), most tasks in LIDA are performed by means of *codelets*, a concept introduced by Hofstadter designating small pieces of independent code, focused on specific tasks. A kernel concept in LIDA is its notion of cognitive cycle, composed of three phases: *perception/understanding*, *attention* and *action/learning* (FRANKLIN *et al.*, 2016). Each of these steps is detailed

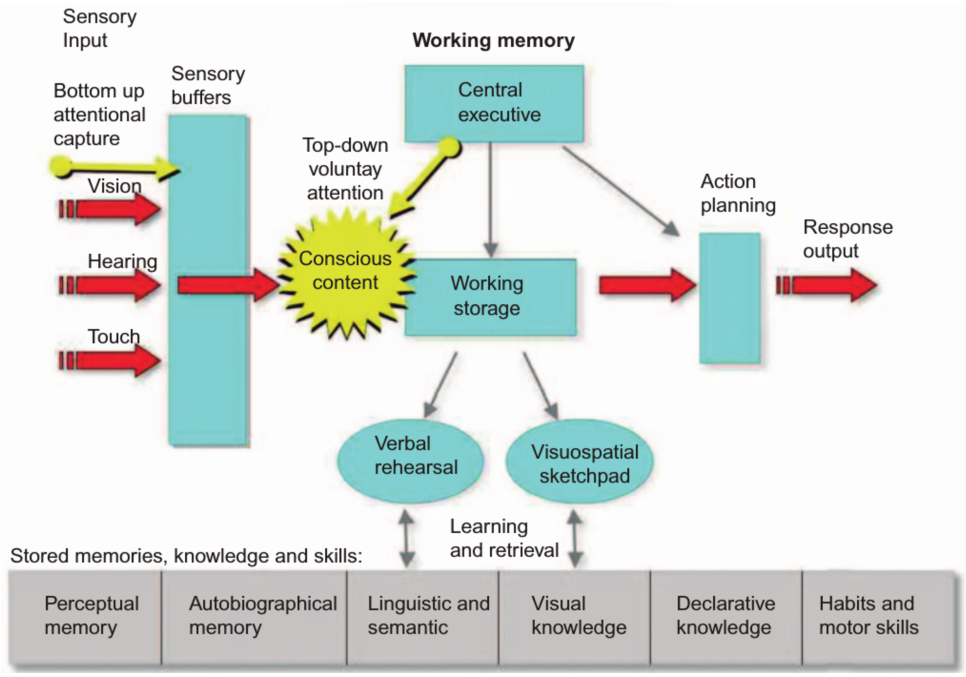


Figure 4.2 – Baars' framework - Extracted from (BAARS; GAGE, 2010)

in the next sections, together with details regarding LIDA's many modules.

A Java-based implementation of the LIDA architecture (called the "LIDA Framework") can be found at LIDA's website[1] (currently it is in version 1.2 - the one analyzed in this work), together with tutorials and many papers describing the evolution of the architecture. Especially good references providing a general overview of the LIDA architecture are (FRANKLIN *et al.*, 2014) and (FRANKLIN *et al.*, 2016).

4.1 The LIDA Cognitive Cycle

The *Cognitive Cycle* is the fundamental hypothesis of the LIDA model of cognition (MADL *et al.*, 2011). According to it, human cognition consists of cascading cycles of recurring brain events, where each cognitive cycle senses the current situation, interprets it with reference to ongoing goals, and then selects an internal or external action in response. Even though most aspects of the cognitive cycle are unconscious, each cycle also yields a moment of consciousness broadcasting. According to LIDA's model, perception occurs 80-100 ms after the sensory stimulus, followed by a conscious episode 200-280 ms after the sensory stimulus, and then action selection 60-110 ms after the start of the conscious phase. The whole cognitive cycle duration should amount to 260-390 ms (MADL *et*

[1] URL: <http://ccrg.cs.memphis.edu/>

al., 2011). Following LIDA's proponents, such cognitive cycles are the fundamental building blocks of all human cognition. According to them, they work as "cognitive atoms" for building the mind. Complex cognitive tasks, such as non-routine problem solving, deliberation, volitional decision making, higher-level perception or imagination, can require many of these cycles, several of which can cascade as long as the seriality of consciousness is preserved. Within each cognitive cycle, a number of modules and processes operate, varying with the current situation or task. The cognitive cycle has the following components (MADL *et al.*, 2011):

1. **Perception:** Sensory stimuli, external or internal, are received and interpreted by perception, producing the beginnings of meaning.

2. **Percept to preconscious buffer:** The percept (including some of the data plus the meaning, as well as possible relational structures) is stored in the preconscious buffers of LIDA's working memory (workspace). Temporary structures are built.

3. **Local associations:** Using the incoming percept and the residual contents of working memory (including emotional content) as cues, local associations are automatically retrieved from transient episodic memory and from declarative memory, and stored in long-term working memory

4. **Competition for consciousness:** Attention codelets view long-term working memory and bring novel, relevant, urgent, or insistent events to consciousness.

5. **Conscious broadcast:** A coalition of codelets, typically an attention codelet and its related informational content, gains access to the global workspace and has its content broadcast consciously. Thus consciousness solves the relevancy problem in recruiting resources.

6. **Learning:** Multiple learning mechanisms are initiated, following the broadcast of conscious content. The conscious content determines what is to be learned.

7. **Recruitment of resources:** Relevant schemes in procedural memory respond to the conscious broadcast. These are typically schemes whose context is relevant to information in the conscious broadcast.

8. **Setting goal context hierarchy:** The recruited schemes use the contents of consciousness, including feelings/emotions, to instantiate new goal context hierarchies (copies of themselves) into the action selection system, bind their variables and increase their activation. Other, environmental conditions determine which of the earlier behaviors (goal contexts) also receive variable binding and or additional activation.

9. **Action chosen:** The action selection module chooses a single behavior (scheme, goal context) from a just instantiated behavior stream or possibly from a previously active stream. Each selection of a behavior includes the generation of an expectation codelet.

10. **Action taken:** The execution of a behavior (goal context) results in the behavior codelets performing their specialized tasks, having external or internal consequences, or both. LIDA is taking an action. The acting codelets also include at least one expectation codelet whose task is to monitor the action, bringing to consciousness any failure in the expected results.

In the sequence, we detail the most important blocks in figure 4.1, following more or less the sequence employed by the cognitive cycle.

4.2 Sensory Memory

The cognitive cycle initiates with data acquisition. Sensors located in the agent will acquire the raw data and send them to sensory memory (SM). In this location, low-level feature detectors will extract meaningful information. In other words, specialized codelets will search for patterns or common characteristics in the raw data.

This module can be viewed as the first level of processing located in the "Sensory Buffer" in Baars' framework (refer to figure 4.2). Making a parallel with human beings, the eyes provide the raw data and the primary visual cortex (V1) in the brain will make the first processing of this data, identifying low-level features like vertical lines, 20-degree tilted lines, horizontal lines, upward motion, leftward motion, downward motion, particular colors, color differences and so forth (BAARS; GAGE, 2010). At the end,

this first layer of processing will convey this processed data to the Perceptual Associative Memory (see section 4.3).

4.3 Perceptual Associative Memory

Once the sensory memory is done in its processing, the output is sent to the Perceptual Associative Memory (PAM), where higher-level feature detectors (codelets) will process this data in order to obtain more meaningful information in the form of objects, categories, and events (SNAIDER *et al.*, 2011).

PAM has the form of a *Slipnet*, a kind of "semantic network" first introduced in the Copycat architecture. As Hofstadter defines

> "The basic image for the Slipnet is that of a network of interrelated concepts, each concept being represented by a node, and each conceptual relationship by a link having a numerical length, representing the "conceptual distance" between the two nodes involved. The shorter the distance between two concepts is, the more easily pressures can induce a slippage between them" (HOFSTADTER *et al.*, 1994).

The name *Slipnet* comes from the concept of *slippage*, i.e., the shifting of perceived boundaries inside structures, descriptive concepts chosen to apply to structures and features perceived as salient or not. The *slippage* of a concept happens when some particular input in need of classification has features in the frontier among two or more classes of concepts, and a final classification depends on a small slippery to one or another concept. In short, each node in the Slipnet will be a symbol that represents objects, categories, or events recognized by the agent, as is illustrated in figure 4.3, which shows an example of a Slipnet used in the Copycat architecture to categorize sequences of letters (a similar idea is applied in LIDA's PAM). Besides its symbolic meaning, each node of the Slipnet has an activation level, measuring how strong is the relevance of that symbol in that situation. This activation is spread through the net to all its neighbors, influencing in their activation.

One important concept in the Slipnet is the deepness of a node in the net or, in other words, how many layers of links are necessary to reach a node. For example, a "sheet of paper" node might be linked to a "book" node and the "book" node might be linked to a "library" node. So, it takes two layers of the "sheet of paper" node to reach the

"library" node. This number of layers captures the generality and abstractness of a given concept. Once the aspects of greater depth are perceived, they should have more influence on the ongoing perception of the situation than aspects of lesser depth (HOFSTADTER *et al.*, 1994).

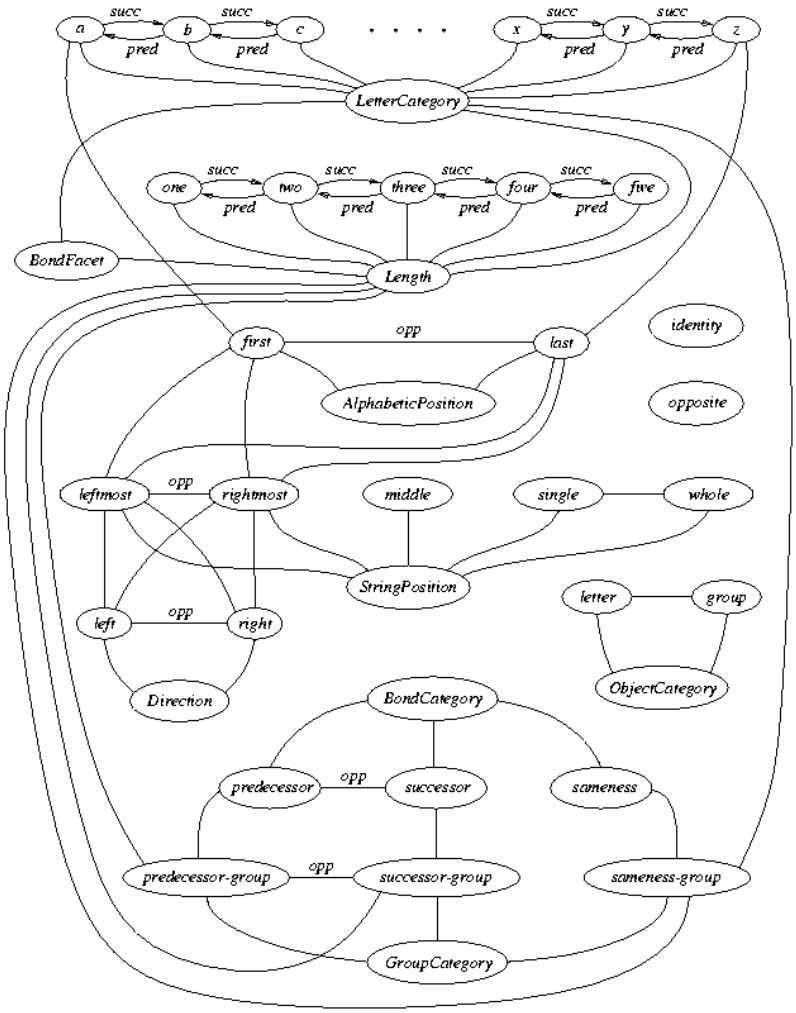


Figure 4.3 – Slipnet representation - Extracted from (HOFSTADTER *et al.*, 1994)

Again making an analogy with human beings, PAM represents the second level of processing in the "Sensory Buffer" in Baars' framework (refer to figure 4.2). As an example, it is possible to correlate the task done by PAM with the job performed by lateral occipital complex (LOC) that seems to have a general role in the visual object recognition, representing higher-level object shape information (BAARS; GAGE, 2010).

As mentioned, each node in PAM is a high-level symbolic representation of objects, categories or events and each symbol has an associated activation level measuring how strong is this representation. Although it is possible to customize how the activation

level is calculated, it is generally evaluated using a saturated sum between of factors: a current level activation, a base-level activation and the activation coming from the neighbors. The nodes whose activation level is greater than a certain threshold are selected and sent to Workspace (see section 4.4).

The current level activation of a node in PAM is calculated by a codelet responsible for detecting that specific high-level feature. This value measures how close the data coming from Sensory Memory matches with the expected pattern evaluated by the codelet and the current level is the value that will be spread through the Slipnet to all the neighbors of that node (the neighbors will get the level multiplied by a scale factor). So, for a new high-level pattern to be detected (i.e, a new node in PAM), a LIDA's modeler should code a new codelet responsible for evaluating this current level value. Finally, the base-level measures how usefulness was that node in the past. After receiving the consciousness broadcast, PAM can increase the base-level activation of some nodes that were in Global Workspace in the last cognitive cycle (see more details on section 4.7), increasing their chances to be selected in next cognitive cycle, providing continued action over time. The base-level activation also decays over the time.

4.4 Workspace

All nodes in PAM whose activation level is greater than a certain threshold (from now on denominated *Percepts*) are sent to the Workspace. Once the nodes are there, Workspace will use this information to cue Transient Episodic Memory (TEM) and Declarative Memory (DM), looking for collateral information that could be useful, improving the capability to make decisions based on past events. These memories are combined with the current percept to generate the *Current Situational Model* of the agent, which represents the agent's understanding of what is going on right now (SNAIDER *et al.*, 2011). The *Current Situational Model* is a short-term memory which has its contents constantly being updated.

The Workspace also hosts a short-term memory called *Conscious Contents Queue*, which is a FIFO queue storing a sequence of the last broadcasts received from the Global Workspace (for more information about this broadcast, refer to section 4.7). Although this structure is size-limited and their items have short-term duration, it allows

LIDA to ground and operate upon time-related concepts (FRANKLIN *et al.*, 2016). Finally, Workspace also has some internal codelets called *Structure Building Codelets*, whose responsibility is looking for contents of interests in the Current Situational Model (CSM), as Franklin describes:

> "If this content is found, then the codelet will perform an action that will result in modifications to the CSM. Possible actions include creating new associations (links), creating new content (such as category nodes), or removing previous associations and content. For example, a structure building codelet that specializes in categorization might add an 'is-a-member-of' link between an object node and a category node, while another with a different specialization might add an affordance link from an object node to an action node." (FRANKLIN *et al.*, 2016).

The Workspace on LIDA can be compared to the "Working Storage" element in Baars' framework of cognition, which is responsible for allowing information to be actively maintained and manipulated, retaining a small amount of data for a short period (BAARS; GAGE, 2010). Working Storage also receives information from other two modules: Verbal Rehearsal and Visuo-Spatial Sketchpad. Verbal Rehearsal is another term for the *inner speech* used for rehearsing, memorizing information and keeping track of our current concerns. The Visuo-Spatial Sketchpad refers to our ability to temporarily hold visual and spatial information (BAARS; GAGE, 2010). Both are short-term memories, having limited size capability and fast access.

As we can see, LIDA's Workspace is quite similar to the Working Storage proposed on Baars' framework. There are several similarities:

- the nodes decay rate is faster in the Workspace than in other modules on LIDA,
- this is the place where all the information is gathered from several places (sensory information, memories, etc),
- the working storage in the framework will be the place where the attention codelets will detect features to possibly reach consciousness.

4.5 Declarative Memory and Transient Episodic Memory

In Cognitive Psychology, Declarative Memory (DM) is a kind of long-term memory which can be investigated due to verbal declarations employed by human sub-

jects. Researchers in human memory use spoken language to collect those declarations from human subjects and the investigation protocols usually try to measure how these declarations are connected to reality. Declarative Memory is usually split into two kinds: Semantic Memory and Episodic Memory (sometimes called Autobiographical Memory). The Semantic Memory stores declarations about temporally neutral facts like "Paris is the capital of France." or "Brazil is a country in South America". The Episodic Memory stores events happening in specific time, space, and life circumstances, forming a linear sequence of autobiographical events experienced by the individual, which can travel mentally back in time to relive the experience (BAARS; GAGE, 2010), and recover past experiences in order to use them in the present. Examples of declarations from Episodic Memory include a description of events during your breakfast this morning, the memories of your last birthday celebration, your first travel to Europe or other episodes you lived when you were a child. It includes events where you had an active role, by means of actions you performed and events you just watched while observing the environment. Both Semantic Memory and Episodic Memory are stored through years or even during the whole life of an agent.

The proponents of LIDA, following Conway (2001) and Baddeley (2000), propose that before being consolidated in Episodic Memory, episodes are first stored in a short-term memory (called Sensory-Perceptual Episodic Memory by Conway (2001), or Episodic Buffer by Baddeley (2000)), which usually spans no more than 24 hours, but is already formatted as a sequence of episodes which were consciously experienced by the human subject along its daily interaction with its environment. This short-term memory is called by them Transient Episodic Memory (TEM).

In LIDA, both DM and TEM are implemented using a Sparse Distributed Memory (SDM), a model proposed by Pentti Kanerva (DENNING, 1989). An SDM is an auto-associative memory, i.e., a content-addressable random access memory. Its input is represented in the form of a very long bit vector, containing thousands or tens of thousands bits. The memory responds to partial matches between the current input and previously stored patterns. SDM uses the Hamming distance as a measure of similarity. The output is also a vector of the same dimensionality as input, returning the stored pattern which better matches the input vector. Each bit in the input/output vector refers to a node in PAM, in a specific timeframe where the respective node of PAM was active.

Patterns stored in SDM are subject to a *decay* rate. In TEM, this decay rate is faster and the elements expire earlier. In DM, this decay is really slow, allowing sometimes that patterns remain stored in its SDM for the lifetime of an agent. Also, while items can enter directly in TEM, in DM new contents need to pass first by TEM and can only be stored in DM after a consolidation process of the undecayed TEM contents, using a periodic offline mechanism (RAMAMURTHY; FRANKLIN, 2011). Making a parallel with a human being, the process is similar to the role of a REM dream, which enhances memory consolidation (BAARS; GAGE, 2010). Both TEM and DM might receive as input a pattern, related to the agent's perception at the current instant, and provide as output the stored data which has the better partial match with the provided input. So, for example, if "wet", "white" and "cold" are presented to the memory, probably it can return "snow" as a possible match.

Ideally, assuming a PAM with 1000 concepts (implying a 1000 bits pattern in SDM as input/output), the storage of any possible combination of these 1000 concepts would require 2^{1000} slots of memory, each one holding one thousand bits. This is a very large number. This is not a realistic scenario, though, as most of the combinations of PAM nodes will never happen. Kanerva's SDM provides an affordable number of memory slots (yet a large number of possible patterns combinations), retrieving and storing them in an efficient way. This theory is mathematically complete and its effectiveness was also proved by computer simulations (KANERVA, 1988).

But how does an SDM work? First, each physical memory slot will have an assigned address (randomly designated at first) with the same size of the perceptual pattern. Besides that, there will be also a register for the cue (containing the pattern to be searched) and a register for the output (in the retrieving process, the result will be placed in this register). All of these registers have the same bit size of the perceptual pattern.

For the storing process, a cue bit array is sent to the memory, where each bit position refers to every node in PAM (if the node is activated it will be set as one, otherwise zero). Each memory slot will evaluate the Hamming distance between its own assigned address and the cue provided. In this case, the cue holds the data that we want to store (data register). If this distance is below a certain threshold (Kanerva recommends

451 for one million memory slots and one thousand bits), those slots will be selected, creating the *sphere* of the cue. At this point, it is important to remember that each slot will be part of other spheres for different cues, so it means that each slot stores more than one information, in such a way that a slot could be used for different cues (that is why the memory is sparse and distributed).

So, how is it possible to store more than one piece of data in the same slot? To accomplish it, Kanerva proposed that each memory slot will also have a set of registers that will work as counters (Kanerva states that an 8-bit register is enough for most of the applications). The number of counters is the size of the perceptual pattern. So, for a one thousand bits size, there will be one thousand counters for each memory slot. Thus, the storage process will be performed by adding or subtracting one from the counters, depending on the content and the cue's bit position. For example, in a three bits domain, two memory slots, $M1$ and $M2$, were selected given 101 as a cue. Each memory slot will have three counters: C_0, C_1 and C_2 (a three bits domain). So, at the end of the storage process, $C_{0_{M1}}$, $C_{0_{M2}}$, $C_{2_{M1}}$ and $C_{2_{M2}}$ will be incremented by one (because the cue has 1 at these indexes) and $C_{1_{M1}}$ and $C_{1_{M2}}$ will be decremented by one (because the cue has 0 at this index). After repeating this process, more spheres will consolidate more data, in such a way that when an information is requested to SDM, it will be possible to retrieve information related to the past experiences of the agent.

To retrieve information from SDM, the cue is sent to memory and the sphere is generated using the same algorithm described in the storage process. Once the sphere is defined, all the sphere counters are summed one by one, respecting the index order. The result is placed in another set of auxiliary counters, responsible for storing this math. So, for example, in a three bits domain, two memory slots were selected to compose the sphere, $M1$ and $M2$. Each memory slot has three counters: C_0, C_1 and C_2. The auxiliary counters, S_0, S_1 and S_2 are used to store the sum of memory slots counters in the sphere given by: $S_i = C_{i_{M1}} + C_{i_{M2}}$ where $0 \leq i \leq 2$. Since the memory output is also a bit array, this result is translated from the auxiliary counters to the output register considering that if the sum was nonnegative, that particular bit will be 1, otherwise 0.

Figure 4.4 gives an overview of a sparse distributed memory for 2000 memory slots, 256 bits perceptual input pattern and a Hamming distance of 119 as a threshold.

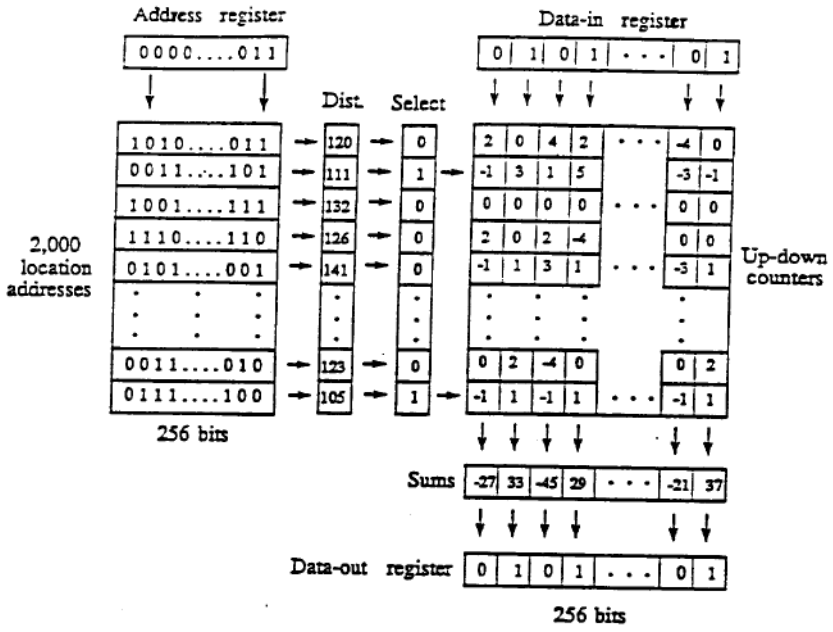


Figure 4.4 – Sparse Distributed Memory - Extracted from (DENNING, 1989)

The group of percepts, plus the data returned by both TEM and DM, is called the agent's *current situational model* including the current perceptual information and past related experiences, related to the current perception. The learning process (storing data into TEM) is triggered by the consciousness broadcast (see section 4.7).

It is important to point out that LIDA does not use the original SDM proposed by Kanerva, but a slightly adapted version of it. A "don't care" state besides the standard 0 and 1 values for each vector slot was included, in order to allow a flexible cuing with fewer features (RAMAMAURTHY *et al.*, 2004). As a consequence, an adjustment was made to the Hamming distance calculations such that the distance between a "don't care" and a 0 or 1 was set to 0.5. Another modification was the addition of a decay rate in each memory slot, which was not present in the original SDM proposal. Basically, the counters' contents in each of the hard locations were decremented based on the basis of the employed decay function (RAMAMURTHY *et al.*, 2006), which is very useful in the "TEM to DM" consolidation process, where only undecayed contents will be transferred to DM.

Finally, it is important to relate LIDA's implementation of DM and TEM to Baar's framework (figure 4.2). In Baars' framework, there is a gray area indicating the long-term memories where several types of non-conscious knowledge are stored, interacting constantly with the working memory through retrieving and the learning process. Clearly,

this behavior is similar to LIDA's model, where the content stored in the workspace will be cued in TEM and DM.

4.6 Attention Codelets

Attention codelets are one of the most important parts of the consciousness mechanism implemented in LIDA. Their job starts at the architecture's consciousness phase. Basically, they are constantly looking out for relevant, important or urgent information located in the current situational model (FRANKLIN *et al.*, 2014). Strictly speaking, attention codelets are interested in particular nodes or links located in the Workspace. If an attention codelet finds all the nodes and links it is supposed to look for, it creates a *coalition*[2] containing these nodes and links.

A coalition contains a portion of the Current Situational Model that is brought to the Global Workspace as a unit along with the attention codelet that created it (SNAIDER *et al.*, 2012). Once a coalition is placed in the Global Workspace, it will compete for consciousness with all other coalitions found by other attention codelets. Attention codelets also assign an activation level to the coalitions they promote, which is used in the competition process. This activation level depends on four factors (FRANKLIN *et al.*, 2016):

1. The activations of each node and link in the coalition.
2. The activation of the attention codelet itself.
3. How well the nodes and links match with the expected data looked for by the attention codelet.
4. When a winning coalition (chosen in the global workspace) has a strong activation, it will make the associated attention codelet to enter into a refractory period where it will be resistant to other processes and stimuli, until it gradually recovers. During this period, a lower activation is assigned to other coalitions.

[2] The notion of a coalition was first introduced by Koch (2004) as "a group of coupled neurons that encode one percept, event or concept". According to Koch, "coalitions are born and die at the time scale of a fraction of a second or longer ... members of a coalition reinforce each other and suppress members of competing coalitions ... every conscious percept must be a coalition of neurons explicitly expressing the perceived attributes". Koch's notion of coalition was further incorporated in Baars' theory of consciousness and embedded in LIDA

Baars' framework also provides a model for attention and its correlation to consciousness (see figure 4.5). This model includes both bottom-up attentional capture and top-down voluntary attention. Voluntary control over actions is performed by the "central executive", implementing: "higher-order purposeful behavior – identifying the objective, projecting the goal, forging plans to reach it, organizing the means by which such plans can be carried out, monitoring and judging the consequences to see that all is accomplished as intended" (BAARS; GAGE, 2010). The central executive focuses the agent's attention on specific events considered important for the current goal in a particular scenario.

As clarified by Baars (BAARS; GAGE, 2010), attentional selection often leads to conscious events and conscious experiences can influence attention in return. A good metaphor for understanding Baars' model of consciousness and understanding the role of attention in the construction of consciousness is the so-called "interactive theater metaphor" for consciousness. In this metaphor, consciousness works like a play running in a theater. But instead of a usual play, this play is an interactive play. In this kind of play, the audience watches what is going on at the stage, and while an audience member feels touched by what he/she is watching, he/she can join others on stage and starts performing. In this sense, all actors on stage are likewise members of the audience which felt invited to participate. All the performers in stage compete for being under the spotlight, to be seen by others. The theater director then chooses the best performances and point the spotlight to the selected actors in order to shine them to the rest of the audience, just as an attention mechanism selects something in a possible range of possibilities. On the other hand, consciousness is akin to one or more actors appearing in the spotlight and their performances being watched by everyone else in the theater. Consciousness and attention are different but inter-related mechanisms.

4.7 Global Workspace

Coalitions selected by attention codelets are sent to the global workspace where they start competing for consciousness. The winner coalition is selected in a straightforward way: the coalition with the highest activation wins. As already mentioned, LIDA is based on codelets that, by definition, are asynchronous and independent. However, the

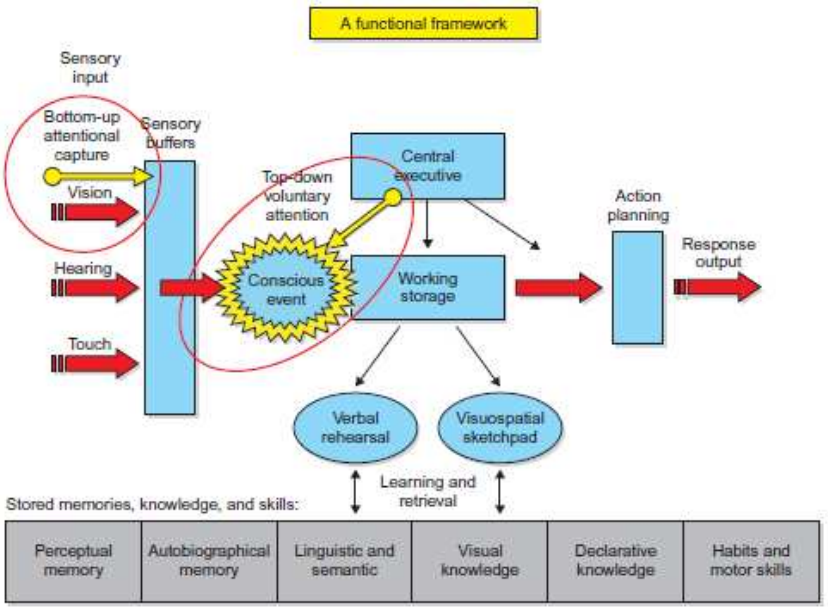


Figure 4.5 – Attention and consciousness in Baars' framework - Extracted from (BAARS; GAGE, 2010)

competition process provides the system with a point of synchronicity, running in a trigger format. It waits until a certain condition is true and then starts the competition. This triggering is effective in four different scenarios (FRANKLIN *et al.*, 2016):

1. At least one coalition has an activation level greater than a certain threshold.
2. The sum of the activation level of the coalitions is greater than another threshold.
3. No new coalitions arrived in the global workspace in a certain period of time.
4. A consciousness broadcast did not happen in a certain period of time.

According to the interactive theater metaphor, once one or more actors are illuminated by the spotlight, their performance can be watched by all others in the theater, possibly exciting other members of the audience and encouraging them to go to the stage and start their own performance. The same idea is applicable to LIDA because once the winner coalition is selected, it is broadcast to other modules in the architecture. The broadcast starts the learning/action phase in LIDA with different consequences depending on the module it reaches.

When the broadcast reaches TEM and DM, the winner coalition is stored as a new memory in the sparse distributed memory, where each coalition node is encoded in

an appropriate index of a bit array, using the algorithm already described in section 4.5. The broadcast is also sent to PAM and to the Workspace. In PAM, the broadcast can change the base level activation of some nodes or create new links between nodes. On the other hand, in Workspace, the coalition will be stored in the conscious contents queue, being available just for a few cognitive cycles, to help other codelets (e.g., the attention codelets) to have a time-related information, that could be helpful to the agent in order to make a decision based on previous actions or feelings.

The consciousness broadcast also affects procedural memory (PM), which is responsible for storing possible actions and their related context. The broadcast helps in selecting the agent's next action, in a process detailed in section 4.8.

As already mentioned, attention and consciousness are related but are different processes. When specific percepts are made salient by the attention mechanism, they compete for consciousness (see figure 4.5) and, in winning this competition, they can be spread around the brain, triggering a lot of cognitive functions like e.g. episodic and declarative memory learning. The broadcast is useful while dealing with unknown problems that require a collaborative and/or competitive activation of different specialized networks, planning the next steps to be taken in order to identify a solution (BAARS; GAGE, 2010). As we can see in figure 4.1, the process described in global workspace theory was implemented in LIDA, since the winner coalition will be broadcast to all other modules in the architecture, triggering different behaviors in different modules.

4.8 Procedural Memory

In human beings, procedural memory (PM) refers to sensory/motor habits or automatic skills, which are largely unconscious (BAARS; GAGE, 2010), i.e., automatic actions are performed without an explicit awareness by the human performer. For instance, when we are driving a car, we do not pay attention to all our movements. We are simply changing the gear, or stepping on the brake pedal, in an "automatic way". In Baars' framework, the PM is located in the gray area with the name "habits and motor skills" and is related to the agent's motor and verbal skills. They provide the means on how to do an action. In LIDA, the procedural memory will play the same function: to store the possible actions of an agent. The mechanism is based on Drescher's *schemes*.

As described by Drescher (1991), a scheme (or a schema, in Drescher's notation) is a representational unit expressing that a certain action has a specified result when certain conditions are met. Each scheme will be made of: the action context (the requirements for triggering the action), the action which refers to the perceptual symbol that is based on the internal and external sensory stimuli that accompany the execution of the action (e.g., for the "grabbing" action, the internal stimulus of one's hand grabbing an object, and the external stimulus of seeing one's hand grabbing) (MCCALL, 2014), and the result (which defines change expectations for the environment when that action is applied to it).

LIDA uses the same principles described by Drescher in the procedural memory. The context and results are represented by nodes and links, and the label is an identifier describing the action to be performed by the agent (like e.g. "turn right", "go ahead", etc.). Besides that, there is a numeric value associated with each scheme, defining its activation. This activation is related to the probability of a certain scheme to be selected at the end of the action selection process (more details in section 4.9). When PM receives the consciousness broadcast, the schemes with their contexts overlapping the nodes and links received in the broadcast are selected and sent to the action selection module. The consciousness broadcast also triggers a learning process in the procedural memory. Franklin clarifies that if a behavior is selected to execution and the event that triggered the behavior subsequently comes to consciousness again, the base-level activation of this scheme is reinforced (FRANKLIN *et al.*, 2016).

4.9 Action Selection

In Baars' framework (figure 4.2) the *Action Planning* component is responsible for generating outputs to the environment. This component begins considering general goals, influenced by emotional and motivational input from limbic regions of the brain, which trigger the frontal lobe, where physical actions are planned and motor system activity is initiated (BAARS; GAGE, 2010).

In LIDA, once a set of schemes is selected by PM, they are sent to the action selection module, where they give rise to *behaviors*. Behaviors compete with each other, based on Maes' behavior net algorithm (MAES, 1989), until just one is finally selected,

which is executed by the agent. The main question that any action selection mechanism has to answer is: how to select the most suitable action to take given a particular situation? Maes aims to answer this question proposing the behavior net mechanism.

For Maes, any behavior i can be viewed as a tuple $(c_i, a_i, d_i, \alpha_i)$, where c_i defines all the conditions that should be met before a behavior can be executed, a_i is a reference for the add-list and d_i for the delete-list, where both specify propositions that are expected to become true or false when the behavior executes. The difference between them is that the add-list contains all the new states that the agent will experience and the delete-list contains all the states that the agent will not experience anymore after applying the behavior. For example, if a behavior is related to move an agent from point A to point B, probably its add-list will be "at location B" and its delete-list will be "at location A". Finally, α_i is a numeric value that indicates the strength level of the activation for that behavior. The activation, according to Maes, can also be impacted by the environment or by goals. The environment changes the current state (asserting propositions as true or false), and if these propositions are conditions of a behavior, this can decrease or increase the activation level of this behavior. Goals can also increase or decrease the activation of some behaviors if they are in accordance (or not) with the current agent goal.

Once all the behaviors are structured in this format, they will be linked in a network format (a graph). There are three possible types of links: successor, predecessor, and conflicter. Given two behaviors k and z, k has z as successor, when a proposition in a_k is also a member of c_z. For the second type of link the opposite is applied, in other words, k has z as predecessor when a proposition in a_z is in c_k. Lastly, k conflicts with z, when a proposition in c_k is also in d_z. It is important to emphasize that a link is applied to/from a proposition, thus, probably, there will be several links from one behavior to the others.

The basic idea of having links is to spread the behaviors activation in such a way that they will inhibit or activate their neighbor behaviors, such that, after a while, the energy will be accumulated in the best choices. Along with successor and predecessor links, one behavior increases the activation of other behaviors that are linked to it, by a fraction of its own activation level. However, along the conflicter links, a behavior decreases the activation level of the other behaviors that are linked to it also by a fraction of its

own activation. It is possible to note that this spread of activation follows an intuitive approach, i.e., a behavior increases the activation of other behaviors that are almost having condition to become executable (successor links) or it increases the activation of behaviors that, when executed, can help it to meet its preconditions (predecessor links) and, on the other hand, decrease the activation of the modules preventing it to become executable (conflicter links). After all the behaviors are linked, it is time to choose the best one and this is done according to algorithm 1.

Pseudocode 1 Selection of a winner behavior

```
1:  procedure GetWinnerBehavior
2:      threshold = ActivationThresholdValue
3:      while True do
4:          Process activation from environment and goals
5:          Spread the activation according to the links
6:          Apply a decay function to keep overall activation constant
7:          SelectedBehavior = None                    ▷ Hold the winner behavior
8:          for behavior b in all behaviors do
9:              if (c_b == True) and (α_b ≥ threshold) then
10:                 if (SelectedBehavior ≠ None) then
11:                     if (α_b > α_SelectedBehavior) then
12:                         SelectedBehavior = b
13:                     end if
14:                 else
15:                     SelectedBehavior = b
16:                 end if
17:             end if
18:         end for
19:         if (SelectedBehavior ≠ None) then
20:             α_SelectedBehavior = 0          ▷ Resetting activation of the chosen one
21:             return SelectedBehavior
22:         else
23:             threshold = threshold * 0.9              ▷ Reducing threshold in 10%
24:         end if
25:     end while
26: end procedure
```

As already mentioned in section 4.8, schemes hold four pieces of information: context, action name, result, and activation. Making a parallel with the behaviors used in the action selection module, it is possible to note that the context in the scheme will be used as the context in the behavior, the result in the schemes will be used as the add-list in the behavior, and the base level activation of the scheme will be used to evaluate the initial activation of the behavior (which will be a sum of several factors, like the activation of the

consciousness content, the base level activation of the scheme, the degree of coincidence of the conscious contents with the scheme's context and so on (FRANKLIN *et al.*, 2016)). As an important caveat, currently, the procedural memory does not deal with delete-lists, so although the original Maes' behavior can use this kind of information, it will be not be used by LIDA. As well as in Global Workspace, the action selection module works in a trigger format and it is triggered when a behavior is above a certain threshold or the sum of activations of all behaviors is above another threshold or no behavior was selected in a certain period of time (FRANKLIN *et al.*, 2016).

LIDA framework also provides a "basic action selection" method as an alternative way for Maes' behavior net where, in short, the behavior with the highest activation value is selected. Basically, all the initial activation of behaviors is evaluated and the selected behavior will be the one with the highest activation value among all the other behaviors and whose the activation value is greater or equal to the activation threshold. If there is no activation greater or equal to the activation threshold, a decay factor is applied to the threshold (and a different factor is also applied to behaviors) and a new action selection round is started. This is a very rudimentary mechanism, but it is very useful for debugging purposes, small agents or in the early stages of development.

4.10 Sensory Motor Memory

As Baars already pointed out (BAARS; GAGE, 2010), output and input processing have some striking parallels because they work as processing hierarchies. Inputs are received as raw data and then turned into high-level perception data. In turn, outputs receive influence from goals, emotions, and motivations which proceed down to motor skills. This is exactly the same philosophy used in LIDA, because after the action selection is done, the winner behavior is sent to sensory-motor memory where an associated algorithm (for that behavior) is executed, coordinating a high-level desire (the behavior) into a low-level action for that moment. This transformation is necessary because a high-level desire can be achieved through several tiny actions. For instance, if the behavior is "walk from position A to B", this will be translated in coordinated movements for each leg (step by step) and also a perceptual analysis that will tell if the B position was reached.

4.11 Summary

During this chapter, we described in details how each module of LIDA works and how they contribute to the cognitive cycle.

LIDA cognitive cycle starts in Sensory Memory where codelets detect low-level features from raw input data. After that, PAM will provide a high-level processing via an interconnected set of symbolic nodes. All nodes having the activation higher than a certain threshold are sent to the Workspace where they will interact with other modules (e.g. memories). Next, attention codelets will search for relevant features in Workspace. When an attention codelet finds a desired pattern, the result coalition of nodes is sent to the Global Workspace where all coalitions compete for consciousness. The winning coalition will be broadcast to the other modules where it will trigger several learning processes. The broadcast also reaches the Procedural Memory where it will filter the most relevant action to be taken using Mae's behavior net.

In the next chapter, we describe the strategies used for the comparison of the three cognitive architectures.

5 Looking for a Comparison Strategy

As already mentioned in the introduction, the study of cognitive architectures is becoming a very prolific area in the recent years, with new architectures appearing here and there, and consequently intriguing the reader for their advantages/disadvantages while compared to each other. This fact claims for some sort of comparison between them, triggering the appearance of works intending to compare CA's under different perspectives (SAMSONOVICH, 2010) (GOERTZEL *et al.*, 2014) (KOTSERUBA *et al.*, 2016). This kind of work is quite relevant because it helps new researchers in the field to identify what are the pros and cons of each CA, giving them a starting point and even clarifying if a cognitive architecture is the best way to solve a certain problem. Cognitive architectures are complex systems, with an inherent cost for this complexity. Depending on the problem, there might be simpler AI mechanisms more appropriate to the situation.

Most of these comparative papers focus exclusively on the architectural models, without paying attention to their software implementations. Even though several computer implementations can be created given a certain model, it is crucial to understand the capabilities and limitations of actual implementations, because this type of analysis can provide comments on their reuse in different situations. In practical terms, models that cannot supply suitable implementations might not be relevant concerning reuse. Besides that, biological aspects are sometimes left aside in this kind of study, i.e., issues like if the CA's addresses particular human or animal cognitive aspects are missing.

To fill these gaps, we performed two kinds of analysis in this work: the first related to the different ways the analyzed architectures were modeled and the second related to the implementation of such architectures. Models are compared using, as a reference, how they address a set of different cognitive functions considered relevant for the construction of an intelligent agent. This analysis provides a contrastive coverage of capabilities and limitations for each CA, imposed by design, with the aim of inspiring the emergence of new cognitive architectures combining the benefits of each of them.

In turn, architectures implementations are compared based on the proposition of a common problem, and the evaluation of how each cognitive architecture considered

in this work (SOAR, CLARION, and LIDA) can be used to solve it. It is not our goal to exercise all the features available in each architecture. On the contrary, a simple experiment is proposed as a benchmark, because the main intent is to evaluate important aspects in the implementations (response time, ease of integration on different domains, resolution of conflicting interests, complexity for coding the solution, code readability and scalability). Using a simple experiment makes this comparison more transparent, easy and reliable because each CA has its own design with different features, so it is very hard to design a complex experiment that can be equally applied in all three architectures. The main idea is to provide a simple walkthrough on each CA, warning users about limitations they might face during development, and features they might count with while using a specific architecture.

5.1 Models Comparison

Advances in neuroscience are changing our comprehension of the human brain, allowing a better understanding of the whole cognitive process and, as a result, driving the emergence of different theories trying to elucidate which are the major functions present in human cognition. Alexander and Dunmall (GAMEZ, 2008) developed an approach based around five axioms (depiction, imagination, attention, planning, and emotion), which they believe are the minimum required for consciousness. In turn, Sun (SUN, 2004) also listed some characteristics that every cognitive architecture should have in order to be biologically inspired, and Baars (BAARS; GAGE, 2010) combined a large body of brain functions into a single framework, helping to clarify the human cognitive cycle schematically (as detailed in figure 4.2).

Inspired by these theories, we selected a set of cognitive functions to be used as a reference while comparing the architecture models under the scope of this work. They are: *perception*, *goals*, *action selection*, *learning*, and *consciousness*. We analyze and evaluate how each cognitive architecture addresses each of them.

Perception is a process that comprehends how data is received from sensors, processed, organized and interpreted in order to allow the agent a better understanding of the environment.

Goals or motivations are what drives an agent to perform orchestrated actions

in order to achieve a prescribed future state. This future state might be precisely defined, or alternatively, can be any future state attending a given set of requirements which must be satisfied.

Action selection can be simply summarized as the answer to the question "what to do next?", in other words, when an agent is facing a situation with conflicting goals, emotions and memories, it is an action selection mechanism role to choose the best action for that moment.

Learning is the act of acquiring and storing a new knowledge or to modify an existing one, in such a way that it can be useful in the future agent iterations.

Consciousness, as a human cognitive function, just recently became worth to be investigated in a scientific way. This is a very controversial subject and some authors believe that consciousness is not related to just one phenomenon, but it is a mongrel concept, related to at least 4 different phenomena (ATKINSON *et al.*, 2000). Alexander and Dunmall stated that at least five different axioms should be present to allow consciousness and, on the other hand, Baars' Global Workspace Theory brought a fresh view for this topic (see section 4). In the AI community, an approach called "machine consciousness" started to appear investigating which elements should be synthesized for consciousness to be found in artificial agents.

5.2 Implementations Comparison

The definition of a common experiment to compare the three CAs under analysis in this work was not an easy task. This experiment should be complex enough to demand the usage of a cognitive architecture, but simple enough to focus on individual aspects of each architecture (remembering that it is not the goal of this work to exhaust all the capabilities of each architecture). We chose a kind of computer game (in fact a virtual environment) as a common platform of tests. In this platform, an non-player character (NPC) is placed in a 3D virtual scenario, where it shall perform a set of tasks autonomously, under the control of a cognitive architecture. We executed a total of three experiments, each one controlled by a different cognitive architecture (SOAR, CLARION and LIDA).

The game runs on a platform developed by our research group at DCA-FEEC-UNICAMP, called WorldServer3D (CASTRO; GUDWIN, 2013), a virtual environment for simulations of experiments with artificial agents. This platform follows a client-server architecture. The server is the virtual environment, where all world entities have their properties computed in real time, following a physics dynamics. There is a graphic user interface (viewer window) where the user can monitor what is going on in the world, keeping track of the simulation. It is possible to add, remove or modify entities directly on the user interface or through commands sent by the client using a network socket.

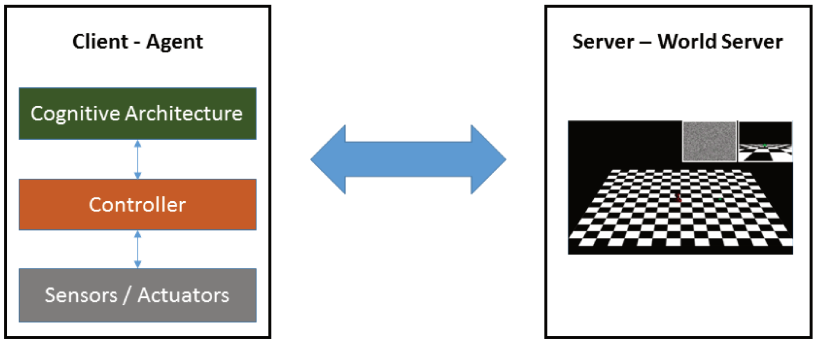


Figure 5.1 – Experiment Architecture

In order to have a simple, adjustable and maintainable application, the client follows a three-layer architecture as illustrated in Figure 5.1. The *Sensors/Actuators* layer is responsible for getting raw data from the sensors and also for sending commands to the actuators. There is no "business logic" in this layer, it is just a set of APIs (Application Programming Interface) making the interface to agent sensors or actuators. Since our experiment is, in fact, running on the server, this layer performs the communication to get data from the simulation (e.g. to get data from the agent's visual sensors, the client sends a command to the World Server requesting this data, receiving the information in return).

The *Controller* layer works as a "bridge" between the two other layers. It is responsible for processing the data received from sensors in a way that the cognitive architecture can receive its inputs, and send commands to the virtual world at each cognitive cycle. Finally, the *Cognitive Architecture* layer refers to the code of each cognitive architecture tested in our experiments.

Because SOAR, CLARION, and LIDA were developed in different program-

ming languages, even though a common software architecture was used for the Client, we had to use different strategies for building it in each case. Even though SOAR was built in C++, we used a Java binding available from their creators. We also used Java for the LIDA version and a customized C# version of the client was developed for CLARION.

Given the experiment structure, it is important to specify what the agent should do. The experiment consists of one single agent placed in a scenario surrounded by walls. After a period of time (T), the system distributes a given number of jewels (J) at random places in this area (as demonstrated in figure 5.2). The agent should divert the walls (avoiding an imminent collision) and collect as many jewels as possible, being able to store the location of some jewels in its memory.

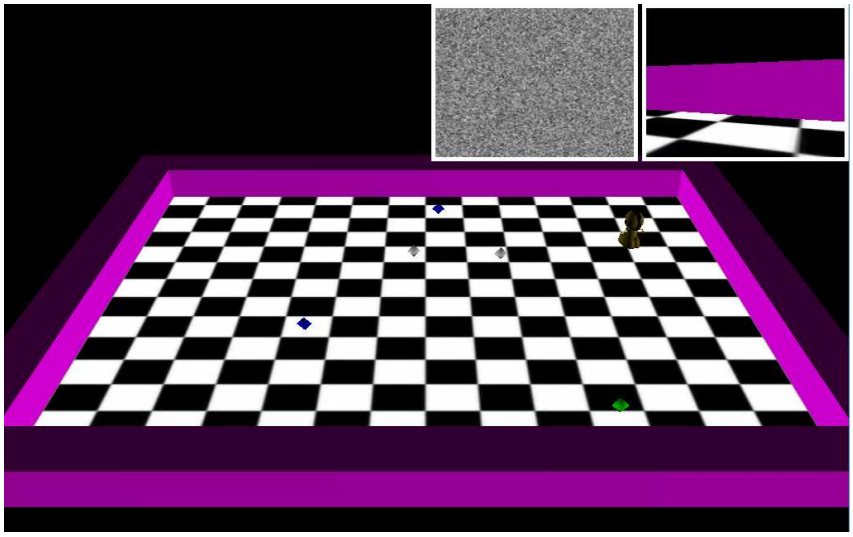


Figure 5.2 – Experiment illustration

The number of jewels J is specific for each experiment and new jewels are replaced at each T seconds, where a new batch of jewels is created to restore the original J. So, for example: if J is equal to 5 and T is 17 seconds, thus after 17 seconds, if the remaining number of jewels is 3 (because the agent collected 2 jewels), other 2 new jewels are replaced at random positions in order to set the maximum number of jewels of the experiment. The value of J varies from simulation to simulation in order to evaluate how the increase of jewels can impact in each cognitive architecture.

Again, the main intent of this experiment is to evaluate: response time, ease of integration on different domains, resolution of conflicting interests, complexity for codifying the solution, code readability, and scalability of the project.

5.3 Summary

In this chapter, we developed a strategy in order to provide a comparison among the cognitive architectures we wanted to analyze. We identified a set of cognitive functions supposedly existing in the human cognitive cycle with the intent to evaluate how each cognitive architecture deals with these functions. We also considered the same experiment (a computer game) in all three architectures, in order to develop a second kind of comparison. The main intent, in this case, is to evaluate how each architecture can be used to solve it. In the next chapter, we start performing the theoretical comparison, focusing our attention on the architecture models and how each selected cognitive function is implemented in each architecture.

6 A Theoretical Comparison

Before starting the analysis of each cognitive function, it is important to emphasize the differences in knowledge representation. Basically, there are two paradigms: symbolic and sub-symbolic. As Ron Sun has already pointed out (SUN, 2004), this dichotomy is crucial for the development of a biologically inspired cognitive architecture. Indeed, it is possible to infer some aspects just analyzing if an architecture is based on one approach, in another one or if it has a mixed approach.

Symbols are entities which make reference to other objects by means of a totally arbitrary convention, a law (CRAIG; MULLER, 2007). For example, the white flag is a symbol of peace, the red cross is a symbol for hospitals and so on. They are widely used, due to their flexibility and power. In the study of Semiotics (the science of representation), though, there are other kinds of representations besides symbols, e.g. indexes and icons. Indexes represent by drawing the attention of the sign-user to their objects, usually using an existential relation, like spatial-temporal relations or causal relations which refer them to their objects. Icons represent by standing in themselves the same properties of their objects (i.e. by being similar to their object in some sense).

At the beginning of Artificial Intelligence, Allen Newell and Herbert Simon (NEWELL; SIMON, 1976) explicitly claimed that a symbolic system might have everything necessary to generate intelligent behavior. This should be possible because the symbols might represent anything in the real world and if they can be manipulated to generate new symbols, then, considering unlimited resources, it should be possible, in principle, to envisage a computer as a classical *Turing machine* and, cumulatively, it will be feasible to represent all objects computationally, which will provide the necessary basis for the generation of an intelligent behavior.

This thesis, called the physical symbol systems hypothesis, was attacked by many researchers since then. The many refutations to this thesis allowed the emergence of the new concept of a sub-symbolic representation (also called “numeric” by some authors). As Nilsson emphasized (NILSSON, 1998), the sub-symbolic has a “bottom-up” flavor and, at the lowest levels, the concept of a symbol is not as appropriate as the concept of a

signal. Besides that, he also argues that the human intelligence evolved after billions of years and it is necessary to follow the same steps, i.e., to concentrate first on duplicating the signal-processing abilities in order to duplicate simple animal behaviors and, later, more complex abilities required for handling symbols and more sophisticated concepts.

The sub-symbolic approach has spread with the appearance of neural networks, raising several questions against the completeness of the symbolic approach. This debate new endures for decades. Eliasmith and Bechtel (ELIASMITH; BECHTEL, 2003) tried to summarize some of the critiques:

- **Understanding**: symbolic models are generally sentential, which seems to be reasonable for problems that can be represented linguistically. However, for some basic cognitive tasks (e.g. taste, touch, smell, sound, and sight), the main intent is to respond to patterns detected in the environment. Besides that, how this association between symbols and incoming data is performed? According to the symbolic approach, this is performed through an adapter module that translates the incoming raw data into symbols. This idea is rather problematic. In a first glance, it is not clear how this module works. At the same time, it becomes evident that, regarding human thinking, there should be some intrinsic knowledge below the symbolic level. This issue is also called the symbol grounding problem (HARNAD, 1990).

- **Fault tolerance**: symbolic representations are generally expressed in the form of rules, connecting the conditions to the desired result (linking symbols). However, if some of these rules had been corrupted, it can affect the whole system. On the other hand, sub-symbolic entities are represented as an interconnected net of simple units, so simple corruptions will lead the system to degenerate its accuracy, but it still works.

- **Learning**: for symbolic systems, learning only occurs with the association of predefined symbols, i.e., new symbols can not emerge. On the other hand, sub-symbolic learning involves strengthening or weakening connections which can be used for low-level learning, helping to explain the development of new symbols.

- **Serial/Parallel processing**: symbolic systems are usually implemented as serial processes. Lists of rules are processed in an iterative way, being tested against a list of

propositions, which truth values give or not support for triggering a rule. Iterations on lists are the primary processing structure. In turn, sub-symbolic systems, due to their nature of connected units, have a straightforward bias for parallel processing, where each unit can make its processing more independently.

On the other hand, symbolicists point-out the lack of the following characteristics in the sub-symbolic approach:

- **Productivity**: this is the capacity to produce arbitrarily many expressions. This is easily achieved in symbolic systems using, for example, recursion. Each symbol can be associated to possibly any other symbol, creating a chain of symbols providing an elaborated meaning for the context (e.g., John told it to Sansa who told it to Arya who told it to Robb).
- **Systematicity**: this pertains to a property of natural languages, in which the same idea can be expressed using different articulations and possibly different words (e.g., to say that "John used his sword to kill his enemy" is the same as saying that "A sword was used by John to kill his enemy", or that "John's enemy was killed by his sword"). The meaning of a whole sentence derives from the composition of the meaning of each of its words, combined in a certain way that a complete and same idea can be expressed.

Having this background in the discussion, we can better understand what are the impacts of using one or other approach within a CA. In next sections, the cognitive functions selected in section 5.1 are described in more details (LUCENTINI; GUDWIN, 2015).

6.1 Perception

An important point for any living being concerns how data is acquired from the environment, and further processed, categorized and organized. As human physiology suggests, perception is a process which can be decomposed in several steps, where each step adds an increased layer of abstractness, capturing patterns and relations among different input data. The first steps are responsible for the identification of basic characteristics,

which are gradually being abstracted in the subsequent steps, until a high-level abstraction of a scene can be achieved, after passing through many layers of intermediary abstractions. Each abstraction layer adds a new understanding of the same data, in a more elaborated interpretation. Taking human vision as an example, the primary visual cortex (V1) is responsible for detecting vertical lines, horizontal lines, colors, etc and the lateral occipital complex (LOC) has a general role in object recognition based on the V1 processing.

Different cognitive architectures have different strategies for capturing and modeling information such that perception can take place. In SOAR all information obtained from the environment is stored in the working memory in the form of WMEs (see section 2). In being completely arbitrary, a WME is necessarily a symbol. In SOAR, this symbol is generated externally, i.e., SOAR does not have a specific module responsible for converting the raw data from sensors in WMEs, so all data must be processed outside the architecture, in order to generate WMEs that will be further handled by SOAR (as emphasized in section 2.2). Due to its nature, SOAR is mostly a symbolic architecture concerning perception, does not providing any internal mechanism to accommodate raw data signal processing.

In LIDA, sensory information is captured and sent to the Sensory Memory module, where low-level feature detectors search for patterns and relations in raw data. After that, the result of this first level of processing is sent to PAM, where high-level feature detectors are employed to extract information, propagating its activation through the Slipnet nodes. It is possible to find some similarities between SOAR and LIDA perception processes: LIDA nodes (stored in PAM) perform a role which is very similar to the WME's in SOAR's working memory because they represent information about the current problem state using a symbolic representation. In the same way, the activation of each node in Slipnet remembers the functionality of WMA in SOAR (see section 2.9). Although they are used for different purposes, the idea to have a numeric value associated with a symbolic node is quite similar. On the other hand, LIDA incorporates a mechanism for the processing of incoming raw data. So, LIDA has a more elaborated perception mechanism, where the perception of a symbol is performed progressively through sub-symbolic and symbolic processes.

CLARION has a mixed representation and this is very explicit in the CA.

Each module is split into two parts: the top-level, responsible for the explicit knowledge (symbolic), and the bottom-level, responsible for implicit knowledge (sub-symbolic). The incoming data in CLARION is done via dimension-value pairs that are, in summary, a set of keys and values that represent, respectively, the information type and its corresponding value.

Although CLARION does not have an explicit perception module (as in LIDA), NACS helps in performing this job. On its top-level, chunks are declared associating dimension-value pairs (or other chunks) with a symbolic representation. These chunks also receive an activation value based on the activation of each source of that chunk. This structure of chunks connected with other chunks and/or low-level features is very similar to the Slipnet present in LIDA's PAM. On NACS bottom-level, AMN networks process raw dimension-value pairs in order to generate other dimension-value pairs as outputs (see section 3.2.3) and these outputs can trigger the activation of some chunks in the top-level. Again, as in LIDA, CLARION has a gradual perception mechanism, where symbols are detected progressively through sub-symbolic and symbolic processes.

However, it is important to emphasize one point: for SOAR, CLARION and LIDA symbols should be previously created by the programmer, i.e., none of the architectures allows the creation of a completely new symbol based on other sub-symbolic/symbolic characteristics. Actually, these architectures have some learning processes that associate a symbol to other different symbols, but they do not allow the creation of a completely new symbol at run-time.

6.2 Goals

The ability to handle goals or motivations are what distinguishes deliberative agents from mere reactive agents. Goals are specifications for future states, which are supposed to be achieved by means of agents actions. These specifications can be either of precise states to be achieved or conditions or predicates that these future states must hold. Deliberative agents are capable of creating and executing plans, i.e., orchestrated sequences of actions that, at the end, will try to meet this future specification (possibly bringing a positive reward to the agent). In fact, a lot of studies try to unveil how the human brain selects goals, but this is still an open area for discussion, although we had a

lot of progress in recent years. Some researchers suggest that the pre-frontal cortex (PFC) plays a central role in forming goals and objectives and some other factors (e.g. emotions, conflict monitoring, planning, etc might affect such decision (BAARS; GAGE, 2010)). As Hull emphasized (HULL, 1943), the decision-making process is clearly not only symbolic and not readily accessible cognitively.

The handling of goals can be decomposed into two main sub-steps: the generation of new goals and the generation of actions leading the system to the satisfaction of these goals. Even though SOAR is able to make plans satisfying a given goal (any rule which results in the halt of a SOAR cycle can be seen as a goal-matching rule), SOAR itself does not provide any built-in motivational process, i.e., a process responsible for the generation of new goals. SOAR presupposes that goals are implicit in the rules, and if new goals are to be considered, these might be solved totally outside SOAR. The consideration of new goals in different instants of time might require new rule bases being considered by SOAR for new goals to be pursued. On the other hand, CLARION has an exclusive module, named MS (Motivational Subsystem), in charge of it. As explained in section 3.3, this module is divided into the bottom and top levels. The bottom-level is responsible for handling drives and the top-level has the symbolic representation of a goal which can be related to drives and/or perception inputs. Basically, the drive levels are sent to MCS that updates the goal structure in MS, which then chooses the new goal for the current cognitive cycle.

CLARION follows the drive reduction theory of Hull (HULL, 1943), where motivation is based on drives, i.e. internal impulses to specific actions representing internal needs which might be reduced by means of an asserted repertoire of actions. Examples of drives are hunger, fear, curiosity, honor, etc. Drives create pressures on the motivational system, which reacts choosing a next action meant to reduce these pressures in a proper way. So, for example, if your energy balance is negative (you are finishing your energy reserves) a hunger drive will increase its value, leading you to look for food and eat, and if you eat something after that, you reduce your hunger, in a clearly homeostatic behavior (your need for energy has been satiated). So, although CLARION is not able to create effective plans, composed of a sequence of actions leading to a goal (like SOAR is able to do), it brings an effective contribution to the problem of goal generation, by incorporating the notion that at different instants of time, our needs might change, leading to different

goals which need to be satisfied. Clearly, CLARION motivational system is more than a reactive system, as the actions selected with the aid of drives are not just reactions to the input, but are meant to reach a future state, where the drives are reduced. So, CLARION motivational system can be compared to a planner of a single step, a kind of myopic planning system. Even though it does not take into account all the biological aspects involved in goal selection, it tries to bring to the discussion a cognitive model for that, combining symbolic and sub-symbolic characteristics. Another interesting point is that motivations feed the ACS, i.e., they are relevant in the action selection mechanism. Due to the GKS structure (see 3.4.1), it is possible to set a new goal and also keeping the previous ones (creating a stack of goals), what can be useful for ACS in order to select actions that achieve more than one goal at the same time. In some sense, CLARION and SOAR are both strong in opposite realms. While CLARION has a more sophisticated goal generation mechanism (SOAR has none), SOAR has a powerful planning system in terms of generating a sequence of actions leading to the goal satisfaction (CLARION has strong limitations in that realm). Both of them, though go beyond pure reactive systems, creating different strategies for improving the decision-making processing in order to achieve pre-designed future states.

In LIDA, one of the possible action-selection mechanisms, Maes' behavior net (section 4.9), was originally supposed to handle goals. In her original proposal of behavior networks (MAES, 1989), Maes explicitly says that goals are one possible origin of activation affecting the behaviors within the behavior net, possibly changing the result of the selection process (and, as consequence, the selected action). Besides that, another possible mechanism involving motivated behavior in LIDA is in the top-down attention mechanism. As Baars pointed out (BAARS; GAGE, 2010), there are two kinds of attention: top-down and bottom-up. Bottom-up attention is the simpler one and is usually based on the detection of salient stimulus. For example, if you see a real lion in front of you, you will probably engage immediately in a procedure to deal with the situation (running or fighting the animal, but rarely ignoring its dangerous presence), no matter what were your previous goals. But top-down attention, on the opposite, is a voluntary and goal-oriented kind of attention, where we try to search for stimuli that can help us to meet our goals. An example of top-down attention would be looking for your car's key if you are planning to drive.

McCall recently proposed new mechanisms of motivations for LIDA in order to complement its functionalities (MCCALL, 2014). Based on Damasio's view of feelings (somatic markers, i.e, the perception of a certain body state), McCall proposed the implementation of feelings as PAM nodes, where each feeling is characterized by a valence sign, a positive or negative value, where positive stands for the agent's basic representation of niceness or liking (e.g., a "hungry" node would have negative sign).

Besides feeling nodes, McCall also proposed the representation of *events* in PAM nodes. Events are represented by a group of PAM nodes providing a grounded representation of a whole "situation" (possibly the current situation). An event comprehends whatever happens during a segment of time at a given location, with a beginning and an end (MCCALL *et al.*, 2010). As McCall exemplifies (MCCALL, 2014) in figure 6.1, the event "Charles takes the pen" can be recognized in PAM, using an event node, becoming a part of the current percept.

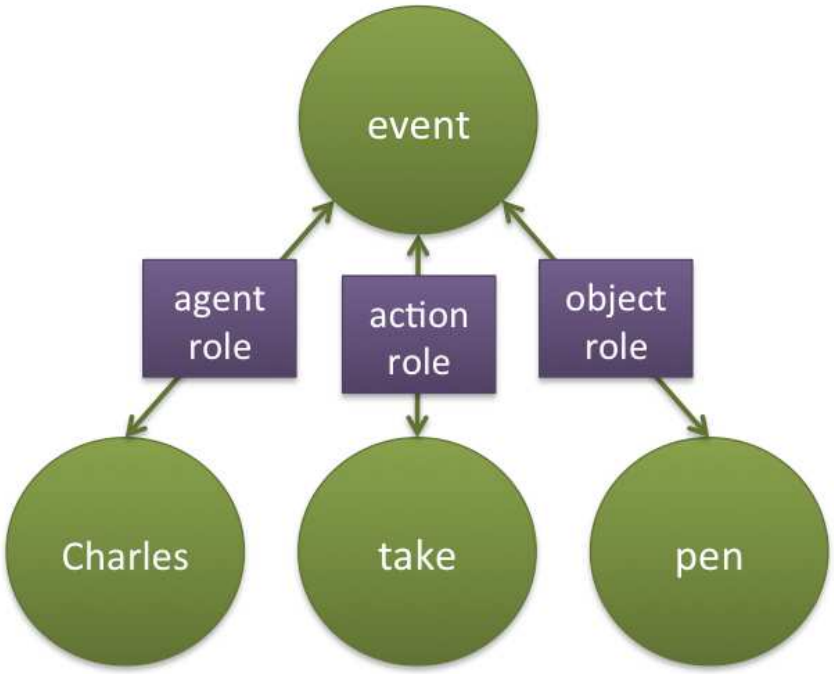


Figure 6.1 – Event "Charles takes the pen" - Extracted from (MCCALL, 2014)

During ordinary perception, feeling nodes might become a part of a preconscious event in the Workspace, either by direct recognition or by close temporal association (MCCALL, 2014). These feeling nodes might be associated with an event. This association is performed by structure building codelets, concerned with particular appraisal dimensions and, once learned, they might be instantly recognized. This association between

feeling nodes and events is described by McCall as an emotion. For example, given a feeling node in PAM representing shame and a currently active event node, the association between them would constitute an instance of the shame emotion (MCCALL, 2014).

Further, the feeling nodes activation will spread to the other nodes comprising the event and used to evaluate the final activation of these nodes. As we know, a coalition of nodes with high activation has better chance to win the competition for consciousness. If broadcast, events having a large magnitude of activation better recruit schemes from Procedural Memory, and can instantiate more salient behaviors. This influence of incentive salience on action selection is akin to the influence of "goals" on Maes' behaviors net (MCCALL, 2014). Finally, the feeling nodes activation is updated during perceptual learning (after consciousness broadcast) using reinforcement learning methods. This idea seems to be similar to the drive mechanism present in CLARION.

6.3 Action selection

Action selection is the process of identifying the most suitable action to be taken by an agent, in a given moment, based on sensory data, goals, and possibly other sources of information, like emotions, beliefs, etc. Maybe, this is one of the least understood areas in human physiology, as properly stated by Tyrrell:

> "whereas perception can be studied by presenting known stimuli and measuring neural firing rates, and motor control can be studied by exciting neurons and observing motor responses, it is not so easy to apply either of these techniques to the behavioural parts of the brain. Although areas of the brain governing behavioural responses will respond to perceptual stimuli in certain cases, and produce motor responses, the relationships are more complex because the inputs and outputs only interact with the outside world via other interfacing systems" (TYRRELL, 1993).

There is not a unique way of determining this next action, and consequently, each cognitive architecture proposes a potentially different action selection mechanism (ASM).

In SOAR, there are two ways of determining the next action: the reactive and the deliberative ways. SOAR is a rule-based machine, where from a given initial state, different operators are analyzed and proposed to be applied for the current state. If more

than one operator is proposed, an impasse is generated and further rules are necessary to solve the impasse and choose one unique operator. Using SOAR in a reactive way means that the initial state reflects the current situation, and for that situation, SOAR must find the best operator. In this case, the knowledge in rules reflect the desired behavior, and SOAR simply applies these rules in order to select a unique operator. In this ASM strategy, the rules must detect specific features which are used to choose the best operator. But the real strength of SOAR is not on the reactive ASM. SOAR mechanism allows a different approach, the deliberative approach. In the deliberative approach, instead of having rules that predefines specific operators for different conditions, these rules are constructed from a different perspective. In this case, rules are used to predict what will be the future state after a given operator is applied. Then, SOAR is able to perform a state space exploration, until a desirable state is achieved, applying a sequence of operators that will turn the present situation into the desired situation. The result is a *plan*, leading the current state to the desired state. The cognitive architecture can start executing this plan, step by step, until the desired state is achieved.

CLARION also has two different kinds of action selection. The first one is somewhat equivalent to SOAR's reactive action selection. But CLARION uses a probabilistic way to select the proper action and ACS is the CLARION's module responsible for this job. Instead of using simply a rule-based system, like SOAR, CLARION decomposes the problem into two levels: the top and the bottom. The top-level is composed of rules, following the pattern: condition/action. The bottom-level has a sub-symbolic approach, being composed, usually, by a neural network. Each level will suggest an action in a reactive way, and ACS will select the winner using a stochastic or combination approach. Both top and bottom levels use Boltzmann's distribution in order the select the best action, although they have some slight differences (as described in section 3.1.3). The second action selection mechanism available in CLARION is provided by the Motivational System (MS). In this case, instead of a pure reactive decision, CLARION uses the Hull's concept of needs. A CLARION agent might have many needs, which are characterized by drives and goals, and CLARION promotes a competition among the many needs, given the current situation, such that the many drives can be satisfied and have their values decreased, as pointed out in section 6.2.

In LIDA, action selection is a process involving both the PM (Procedural

Memory) and AS (Action Selection) modules. Initially, PM provides to AS all the schemes that match with the current state and AS will use them as a basis to choose the best action to be taken. This selection is performed either using Maes' Behavior Net, a graph where each node is a behavior, or the *Basic Action Selection* mechanism. The connections among all the behaviors make clear the dependencies among them and, through a spreading activation mechanism, AS chooses the best action.

Three cognitive architectures and three different ways to select an action. First, let's compare how the actions are stored in each architecture and then we can move our attention to the action selection process proper. In SOAR and LIDA, actions are represented by means of rules. Of course that, in each architecture, there are differences due to the way that each rule is represented, but, essentially, they are rules, having a strong symbolic bias. CLARION has also rules located in ACS top-level. However, actions can also be suggested using neural networks located in ACS bottom-level. As we can see, CLARION has a mixed representation of actions, where symbolic and sub-symbolic suggestions are taken into account. These two types of representations bring pros and cons and as CLARION combines these two approaches, it also combines their qualities, minimizing the disadvantages of a single kind of representation.

Now it is time to focus our attention on the mechanism used to choose one single action, given a set of possible actions. As Tyrrell defined (TYRRELL, 1993), an ASM should be able to handle all types of sub-problems that an agent can face in the environment. In SOAR, this job is performed evaluating the operator conditions, their preferences and solving impasses. All operators, preferences, and impasses should be previously identified and addressed using specific rules, which may increase considerably the time and cost for designing a solution (depending on how complex is the experiment). Besides that, it can be very hard in dynamic environments, where the conditions change constantly and it is difficult to evaluate all the possible collisions among operators.

In turn, LIDA should declare all the actions for Maes' behavior net where each one consists in the conditions that will trigger the action, the add-list, the delete-list and the initial activation value for the behaviors. For complex experiments, these definitions may also be very hard to define, but once they are defined, it is not necessary to evaluate all the possible conflicts, because they are solved thanks to the spread of activation in

Maes' behavior net. However, as in SOAR, the agent should be designed to respond to all possible situations, which may be very complex to be performed in dynamic environments.

For CLARION, ACS top-level suggests the rules which meet the current state of the problem and the winner action is selected using a Boltzmann distribution based on the suggestions provided by the top and bottom levels. Differently from the other CA's, CLARION is more versatile to be applied in dynamic environments because it is not required to think about every condition that the agent will face and to create a rule to address each one. Instead of that, it is possible to take advantage of the neural networks present in ACS bottom-level. These networks can be trained to dynamically adjust the weight of each neuron based on the environment response (positive or negative) of a selected action. Furthermore, the rules are still there in ACS top-level and they are very useful because they can be applied for a fine tuning in the action selection, making possible a custom prioritization.

An ASM should also be able to interrupt a sequence of actions related to a certain goal (e.g. "get food"), if a more urgent demand appears (e.g. "avoid predator"). This can be easily achieved using preferences in SOAR. Preferences assert the relative or absolute merits of the candidate operators (LAIRD; CONGDON, 2014), being possible to prioritize actions.

An interruption in LIDA can be achieved, for example, applying a high initial activation for the behaviors that are very critical to the agent. As described in pseudocode 1, the activation is crucial for the selection of the winner behavior, so it is possible to prioritize the behaviors using the activation as a reference. On the other hand, in CLARION, it is possible to achieve this goal storing the critical actions in form of rules (e.g., as fixed rules) and also adjusting the probability of selection of this kind of rule.

Another point of attention is the continuity of action, which reflects the tendency to persist with an action because the "cost" (energy, availability, etc) of changing to another one is higher than keeping in the current strategy. In SOAR, there is no built-in mechanism related to this, because the operators are selected based only on the current working state. For LIDA, the persistence is not being addressed too, because the activation of the winning node is reset to zero (see pseudocode 1) and there is no mechanism to taking into account the switching cost. In turn, CLARION has the rule utility function

that measures the effectiveness of a rule, being a comparison of its benefits (rewards) and its costs (time to execute a rule). It does not totally address all the considerations related to contiguous action, but, at least, there is some evaluation of the cost to switch to another action.

Finally, it is possible to note that SOAR and LIDA use a deterministic decision procedure, i.e., repeating the same inputs, the same winner action will be returned while CLARION prefers to use an approach based on a probabilistic distribution. Both approaches have their pros and cons. The great advantage of using a deterministic decision procedure is the repeatability and consistency of the results, being very good in controlled environments and critical tasks. But, for dynamic environments, the best choice for a given moment will not always be the same for another moment, thus a little bit of randomness can help the agent to explore more opportunities and possibilities. Besides that, for the particular case of CLARION, it is also possible to adjust the constants of the Boltzmann's distribution in order to make it virtually deterministic as already detailed on section 3.1.3.

6.4 Learning

As described by Baars (BAARS; GAGE, 2010), memory can be defined as a perennial representation that is reflected in thought, experience, or behavior and learning is the acquisition and consolidation of such memory. The learning process is very important for any living being and this applies also to artificial agents because learning allows experiences from the past to be available in the present, helping the decision process. This information might be very useful, for example, while trying to avoid making the same mistakes.

Let's understand the several types of memories present in humans. Even though this division is not a commonsense, human memory system can be divided into three different categories: sensory memory, short-term memory and long-term memory (CASTRO; GUDWIN, 2013). The sensory memory is an ultra-short memory (less than one second) responsible for retaining the raw data received from sensors (audition, vision, etc) for the first level of processing. The short-term memory (also called working memory) is a short and size-limited memory (approximately less than ten items and around ten seconds of

retention) that works like a "scratch-pad", where relevant knowledge are brought from different places to be used in several cognitive functions (perception, reasoning, planning, etc).

Finally, long-term memory is responsible for storing different kinds of data for a long period of time (days, months, years, etc). This memory is also divided into two other categories: declarative (or explicit) and non-declarative (or implicit). Declarative memories refer to facts or events explicitly declared and it can also be divided into two another categories: episodic and semantic. Episodic memory is used to store facts particularly contextualized in time and space (CASTRO; GUDWIN, 2013), like the memories of the first time that you went to the beach, your prom party, etc. In turn, semantic memory stores all kinds of data (facts, meanings, concepts) not necessarily being contextualized, like the Pythagorean theorem, or the fact that Brasilia is the capital of Brazil and so on. Finally, non-declarative memories are more related to skills or how to use some objects. The procedural memory, for example, is a representative of non-declarative memory where it is stored "how to do this or that", e.g., how to drive a car, or how to ride a bike.

Now it is time to turn our attention to some types of learning. Initially, it is possible to highlight the learning by association, also known as classical conditioning or Pavlovian conditioning (BAARS; GAGE, 2010), where an arbitrary stimulus (the unconditioned stimulus) is associated with another stimulus (the conditioned stimulus) through multiple repetitions where both stimuli are present together. The classical example of this approach was described by Pavlov, where the sound of a bell was presented just before the instant that a dog receives food. Thus, after a while, every time the dog listened to the bell, even without the real food, it started to salivate, because the dog associated that sound with the presence of food. The other type of learning is known as operant conditioning and it was initially studied by Skinner in the 50's. In this process, the strength of a behavior is modified by the feedback (reinforcement or punishment, positive or negative) provided by the environment, in such a way that the agent will repeat behaviors that provide a positive feedback and avoid those that provide a negative feedback.

Having this background in mind, it is time to turn our attention to how each CA deals with memories and learning. Table 6.1 summarizes if specific memory systems are present in each of the cognitive architectures studied in this work.

Table 6.1 – Availability of memories and learning mechanisms

	SOAR	**CLARION**	**LIDA**
Sensory memory	No	No	Yes
Short-term memory	Yes	No	Yes
Episodic memory	Yes	No	Yes
Semantic memory	Yes	Yes	Yes
Non-declarative memory	Yes	Yes	Yes
Classical conditioning	No	No	Yes
Operant conditioning	Yes	Yes	Yes

SOAR does not provide any kind of sensory memory. In SOAR the raw data is processed outside the architecture, in such a way that inputs are provided already as WME's placed in working memory (refer to section 6.1). On the contrary, LIDA has a classic sensory memory module (see section 4.2), responsible for receiving raw data from the sensors and being the source of information of perception codelets, processing the raw data with low-level detectors. In turn, CLARION does not have an explicit module responsible for this functionality (as in LIDA), i.e., there is no module responsible for concentrating all the architecture inputs in one single place for further processing and, in fact, the dimension-value pairs (used as input in CLARION) can be freely used in a raw format in other modules of the architecture.

The short-term memory is represented in SOAR by the Working Memory and it can be faced as the "theater of operations" of the architecture. As described before, a classical short-term memory should be size limited and it should hold the elements just for a few period of time if they are not being used. SOAR working memory is not size limited, but, it holds the WME's to be used as context of the next cognitive cycles and, due to the working memory activation (see section 2.9), WME's are removed from working memory if they are not being used (measured by the times that they are tested by operators). LIDA has the Workspace module (see section 4.4) that works like the Working Memory in SOAR, i.e., it is responsible to concentrate the information provided by the different modules of the architecture in one single place. The Workspace also has the same characteristics of SOAR Working memory: it is not size limited, but the elements that are not frequently used are removed from there. However, in CLARION, there is no module that works like a short-term memory. Although ACS concentrates all the information provided by the other CLARION modules, it does not work like a memory, i.e., it does

not hold the data received for more than one cognitive cycle.

Episodic and semantic memories are also present in SOAR with the same name, characteristics, and fundamentals. Episodic memory is responsible for storing the full amount of WME's in the current working memory state creating a new episode (being a contextualized memory) and, on the other hand, semantic memory is responsible for storing specific and relevant WME's that do not need to be necessarily contextualized. In LIDA, these two types of memories are also present. Episodes that the agent experienced are stored into Transient Episodic Memory (TEM) with a decay rate of some hours or a day and the episodes that have not yet decayed are stored in Declarative Memory (DM) through an offline consolidation process. In DM, there are episodes with full contextualized data (with "what, where and when") in a placed called Autobiographical Memory. Declarative Memory also contains traces that have lost their "where and when" to interference, while retaining their "what" in the form of facts, rules, being referred as semantic memory (FRANKLIN *et al.*, 2016). On CLARION, episodic memory is not present, i.e., there is no module in the CA responsible for storing full contextualized data. However, NACS works like a semantic memory or declarative memory (SUN, 2003). The explicit knowledge is stored in NACS top-level in form of chunks that can be triggered directly by the top-level mechanisms or even by the reasoning process through the AMN's in the NACS bottom-level.

Moving forward to the non-declarative memories, it is possible to note that, in SOAR, operators are stored in the procedural memory, a type of a non-declarative memory and the chunking functionality (see section 2.10) provides to CA a mechanism for procedural learning. CLARION also has a procedural memory located in ACS top-level where explicit agent behaviors (in the form of rules) are located. The bottom-up learning (see section 3.1.4) also provides to CLARION a procedural learning mechanism. LIDA has a procedural memory that is responsible for storing the agent behaviors in the form of schemes (a structured type of rules) and also a procedural learning mechanism using the consciousness broadcast as a trigger.

With respect to conditioning techniques, there is no module providing a classical conditioning learning in SOAR, in such a way that the links between the WME's do not change due to a learning process, i.e., new correlations and knowledge are not created

at running time. CLARION does not have a classical conditioning as well. The integration between levels in NACS (see section 3.2.3) only provides a mechanism to activate some chunks or dimensioned-value pairs using different approaches, while new links of knowledge are not created. LIDA has a rudimentary mechanism, sometimes referred as *perceptual learning*, to deal with classical conditioning based on the consciousness broadcast. Basically, the consciousness broadcast reaches PAM, where the content of the broadcast (composed by nodes and links) will be analyzed in such a way that the current nodes and links in PAM will be adjusted (added, reinforced, removed) based on this broadcast. As described by Franklin, the conscious broadcast begins and updates the process of learning to recognize and to categorize, both employing perceptual memory (FRANKLIN; JR, 2006).

Finally, SOAR has the reinforcement learning module which is one of the most recent modules in the architecture, directly inspired in the behaviorist psychology, where the main idea is to provide a positive or negative feedback for each executed operator, in such a way that operators receiving more positive feedback's will tend to be selected more times. CLARION also uses the feedback provided by the environment in the neural networks located at ACS bottom-level, having, therefore, the operant conditioning present. LIDA also has a similar functionality in the architecture, but not so explicit as in SOAR or CLARION. One of the destinations of the consciousness broadcast is the Procedural Memory. Basically, as described by Franklin, if a behavior is selected and executed, and the result of that behavior subsequently comes to consciousness, selective learning is triggered and the base-level activation of the scheme that generated the behavior is reinforced (FRANKLIN *et al.*, 2016). Increasing the base-level activation will increase the chances of a behavior to be selected for execution (see pseudocode 1), so these procedural skills are shaped by reinforcement learning, operating by way of conscious processes over more than one cognitive cycle (FRANKLIN; JR, 2006).

6.5 Consciousness

Consciousness, as a human cognitive function, just recently entered in the realm of topics suitable to be investigated in a scientific way. Nevertheless, in the intelligent systems community, an approach being called "machine consciousness" started to appear,

proposing what to be synthesized if consciousness was to be found in an engineered artifact (ALEKSANDER, 1995).

The subject is in itself very controversial and some authors understand that the term "consciousness" is, in fact, a mongrel concept, relating not just to one phenomenon, but denoting several meanings. Block classified four different concepts assigned to the word consciousness (ATKINSON *et al.*, 2000; BLOCK, 1995):

- **Access consciousness (A-consciousness)**: it refers to our ability to access information in our mind through speech, reasoning and the control of behavior, i.e., there is a representation in that person's brain for the possible rational controls of speech and action given the current situation.
- **Phenomenal consciousness (P-consciousness)**: it refers to the qualitative nature of experience, also referred as *qualia*. Every time we use our five senses (hearing, sight, touch, smell, and taste), we are having P-consciousness and we can go further, including sensations, feelings, and perceptions. Basically, it is related to the question "what it is like". Sometimes, P-consciousness is referred as the hard problem of consciousness, because it is not clear how sensations acquire different meanings or subjective aspects in each individual (CHALMERS, 1995).
- **Monitoring consciousness (M-consciousness)**: thoughts about our awareness, our sensations, monitoring, internal scanning or, in other words, a p-consciousness of the self.
- **Self-consciousness (S-consciousness)**: possessing the concept of the self and the ability to use this concept. For instance: recognizing itself in front of mirrors.

Actually, several theories emerged to explain how consciousness works in human beings based on the concepts created by Block or even in other theories and some of these approaches were applied in computer programs. Given the cognitive architectures that are in the scope of this work, only LIDA uses an established model for consciousness, the Global Workspace Theory by Baars (widely described in section 4).

In this approach, consciousness is implemented computationally by means of a broadcast of contents from a global workspace, which receives input from the senses and

from memory (FRANKLIN *et al.*, 2005). Making a parallel with Block's concepts for consciousness, Baars' theory focus the attention in A and P consciousness as a computational substrate of consciousness (ATKINSON *et al.*, 2000).

In summary, Global Workspace Theory says that the brain is composed of a set of specialized networks (each one responsible for a specific task) and the content of the global workspace (a virtual memory region where sensory and memory inputs compete for a position based on the result of the attention sensors) is broadcast (also called consciousness broadcast) to these specialized regions in order to trigger different functions given the content of the broadcast. According to Baars' approach, the consciousness broadcast enables most types of learning (FRANKLIN *et al.*, 2005). For instance, when we compare the semantic memory learning in LIDA and in SOAR, the role of consciousness becomes more clear: in SOAR, the semantic memory is also present, but, in order for an agent to store a long-term identifier into semantic memory, it must invoke an explicit command (LAIRD; CONGDON, 2014). In turn, LIDA uses the content of consciousness to automatically trigger this type of learning for the most relevant information already filtered by consciousness. Thus, one of the main roles of consciousness in Baars' theory is to "automatically" trigger the updating of perceptual memory, transient episodic memory, and procedural memory, using as input the most relevant information provided by the consciousness broadcast (FRANKLIN *et al.*, 2005).

The other CA's do not have an explicit consciousness mechanism computationally implemented. However, taking as basis the several types of consciousness described by Block, it is possible to correlate some types of consciousness with some features present in each CA. For instance, the main role of Meta-Cognitive Subsystem (MCS) in CLARION is to monitor and to control all the other modules in the architecture (see section 3.4), whose behavior is very similar to what is described as M-consciousness. Besides that, in SOAR and CLARION the interaction among all the modules is responsible for receiving raw data from the environment, to process it and to select what is the best action to be taken, what address some of the characteristics described as being A and P consciousness.

6.6 Summary

In this chapter, we compared each cognitive architecture model discussing how the following cognitive functions are addressed in each of the architectures: perception, goals, action selection, memories/learning, and consciousness. The analysis showed the pros and cons of each design based on the expected biological behavior for those cognitive functions. In next chapter, we analyze how each architecture can be used to solve the same experimental problem, providing an implementation analysis, considering a pragmatic evaluation of either architecture.

7 A Practical Comparison

To complement our study on similarities and differences among cognitive architectures, started with the theoretical analysis developed in the last chapter, we performed also a practical analysis, which is described in this chapter. The practical analysis consists of proposing a prototypical problem in the field of cognitive architectures and trying to employ SOAR, CLARION, and LIDA in order to analyze the different aspects of each implementation. Our aim here is to clarify what are the pros and cons of each approach and acquire a better understanding of the pragmatic issues while trying to use either of these architectures. The problem is the construction of an artificial mind to control a virtual character in a computer game-like environment. Our experiment consists of one single agent located in a virtual environment surrounded by walls, where it should collect the maximum number of jewels that randomly appear during the simulation, avoiding possible collisions. Although the experiment is the same, due to the different interface strategies in each of the architectures, we had to employ different technical solutions for binding each cognitive architecture to the virtual world. Nevertheless, the same principles were adopted in each of the cases (client-server architecture and three layers application on the client side), as described in section 5.2.

In the next sections, we describe how we implemented the solutions using each of the three cognitive architectures and, at the end of this chapter, we provide an analysis for the three implementations, comparing them in several aspects.

7.1 Implementation in SOAR

As previously described, in SOAR there are two major elements of concern: states and operators. The initial state is the current representation of a problem in terms of a set of WME's describing the current state of affairs in the problem demanding a solution. Operators are modifiers of the current state in the form of rules, generating a new state, after its application to the current state. The SOAR kernel works by trying to select a suitable operator to be applied to the current state, applying this operator and making changes to the current state, turning it into a new state, and repeating this

same procedure until a goal state is detected and the search halts. There are though two different approaches while using SOAR to solve a problem. The first one is this canonical deliberative approach, which is in the origins of SOAR. In this approach, operators are applied, changing the current state, and SOAR performs a search on state space until a goal state is found, and SOAR can halt, returning a solution. But there is an alternative approach while using SOAR: the reactive approach. In this approach, rules are conceived in order to detect the most suitable operator to a given condition, and after deciding which operator to apply, the application of this operator defines a command at the state's output link, and halts. In this approach, SOAR does not keep generating new states, performing a search until finding a desirable state. In a single step, it decides what to do and sends the command to the output link, which will trigger an actuator. In our experiments, due to the intrinsic characteristics of the problem, we decided to employ this reactive approach.

The connection between the SOAR kernel and the external environment is realized through two links associated to the current state: the input link and the output link. In each cognitive cycle, before calling the SOAR kernel, a piece of software must collect the sensory information from the environment, format this information in terms of WME's, and feed the input link. After the SOAR kernel is called and halted, another piece of software should pick the information generated by SOAR at the output link and send to the environment. In our simulations, this role is performed by the bridge layer (on the client side, implemented in JAVA). So, the first step in order to use SOAR as a decision-making algorithm is to design how the WME's will be structured in Working Memory and, after that, feed the architecture with the proper data. SOAR provides a set of APIs making possible this conversion of raw sensory information into WME's as exemplified in the code snippets in Box 7.1.

The bridge tier is responsible for creating a tree of WME's in SOAR Working memory based on the raw data provided by the virtual environment (using the Sensors/Actuators layer). The topology of this information written in the Working Memory is illustrated in figure 7.1.

In our experiment, the bridge provides SOAR with the creature's current position (in the Cartesian plane) and a list with all the perceived entities in the agent's

Box 7.1: Input data in SOAR

```
// Importing SOAR API's
import sml.Agent;
import sml.Identifier;
import sml.Kernel;

public class SoarBridge
{
  Kernel kernel = null;
  Agent agent = null;
  Identifier inputLink = null;
  Identifier creature = null;

  //  Instantiating a new agent
  public SoarBridge(String agentName, String pathToTheRules)
  {
     kernel = Kernel.CreateKernelInNewThread();
     agent = kernel.CreateAgent(agentName);
     agent.LoadProductions(pathToTheRules);
     inputLink = agent.GetInputLink();
     creature = agent.CreateIdWME(inputLink, "CREATURE");
  }

  public void SetCreaturePosition(double positionX, double positionY)
  {
    // Setting creature Position
    Identifier creaturePosition = agent.CreateIdWME(creature, "POSITION");
    agent.CreateFloatWME(creaturePosition, "X", positionX);
    agent.CreateFloatWME(creaturePosition, "Y", positionY);
  }
}
```

field of vision, together with several properties of each entity, like unique name (useful for some operations), type (jewel or wall), distance to the agent (it is possible to perform this math using SOAR operators, but it is simpler to do it from outside of the scope of the architecture), and location (X1, Y1, X2, Y2 - for jewels, X1 is equal to X2 and Y1 is equal to Y2, but for walls these two points represent the diagonal points of a rectangle). So, for instance, if the agent is seeing three entities, there will be three branches coming out from the "VISUAL" node with the same attribute "ENTITY" and each of them with its own attributes according to the pattern described in figure 7.1. Besides that, there will be also a "MEMORY" WME used for storing the position of the last viewed items in order to make the search for jewels more efficient, however the entities below the "MEMORY" node are filled with the result of SOAR operators and not by the bridge layer. It is important to emphasize that this "MEMORY" WME does not use the built-in mechanisms for semantic memory present in SOAR. This approach was used because it is simpler if the elements can be easily and deliberately stored or removed.

The following operators were created to accomplish our intents with the ex-

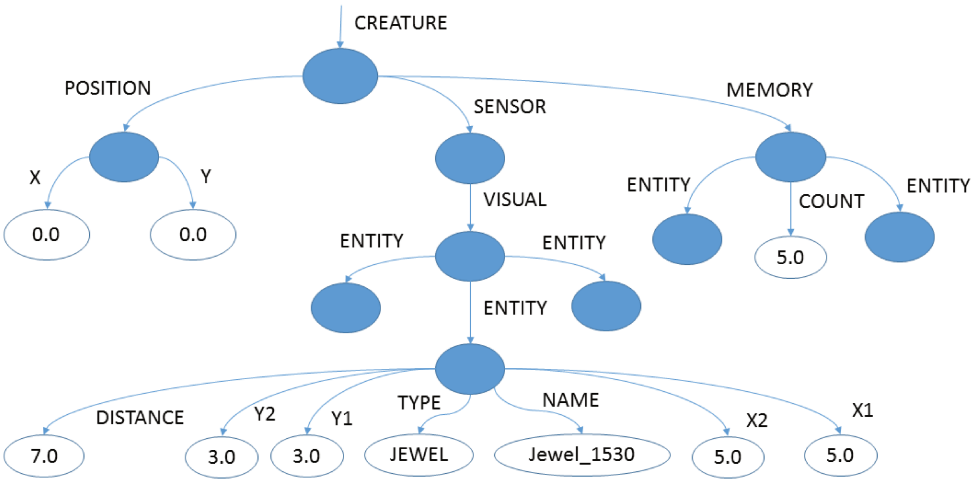


Figure 7.1 – Experiment representation in SOAR

periment and their pre-conditions and results are described below:

1. *Wander*
   - Pre-conditions: An agent must exist.
   - Result: Move the agent forward with constant velocity.
2. *Memorize*
   - Pre-conditions: A jewel must exist in the visual field, must not be in the "MEMORY" WME (checked using the "NAME" attribute) and there must be an available position to store this entity ("COUNT" should be lower than 7).
   - Result: Add the jewel to the "MEMORY" branch and increment the "COUNT" attribute.
3. *Move to Jewel*
   - Pre-conditions: There must be at least one jewel at the "MEMORY" branch
   - Result: Send the command to move the agent to the position for the closest jewel in the "MEMORY" branch, but if there is no jewel at the visual field in the same position as the memorized one, remove it from memory, decrementing the "COUNT" attribute.
4. *Get Jewel*
   - Pre-conditions: There must be at least one jewel in the visual field that is at a picking distance.
   - Result: Send the command to get that jewel and remove it from memory,

decrementing the counter if the jewel is in memory.

5. *Avoid Wall*
   - Pre-conditions: There must be at least one wall in the visual field close to the agent.
   - Result: Rotate the agent clockwise.

In Box 7.2, we show the rules used for implementing the *Wander* operator. As already mentioned in section 2, each operator is composed for, at least, two rules: the first one *proposes* the operator, i.e., describes the conditions for the operator to run. The second *applies* the operator if the pre-conditions are met and that operator was chosen by SOAR in its inner selection algorithm.

Box 7.2: *Wander* operator

```
# Propose*wander:
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)}

# Apply*wander:
# If the wander operator is selected, then generate an output command to it
sp {apply*wander
   (state <s> ^operator <o>
              ^io <io>)
   (<io> ^output-link <ol>)
   (<o> ^name wander)
-->
   (<ol> ^MOVE <command>)
   (<command> ^Vel 1)
   (<command> ^VelR 1)
   (<command> ^VelL 1)}
```

However, as we already mentioned, an operator is not necessarily applied, just because its pre-conditions are met. There might be more than one operators proposed at the same time, by different rules, all of them having conditions to be applied, and SOAR must select one among them. In order to perform this decision, SOAR processes further rules, which set preferences between different operators. If using these rules, SOAR is not able to select a unique operator, it generates an impasse, possibly halting the search for an

operator, without a chosen one[1]. In order to solve impasses, the developer should provide a complete set of preferences among operators, such that in any case, SOAR is able to find the operator with the highest preference. Table 7.1 summarizes the preferences among all operators in our experiment.

Table 7.1 – Preference among operators in SOAR

	Wander	**Memorize**	**Move Jewel**	**Get Jewel**	**Avoid Wall**
Wander	=	<	<	<	<
Memorize	>	=	>	<	<
Move Jewel	>	<	*	<	<
Get Jewel	>	>	>	*	<
Avoid Wall	>	>	>	>	=

The symbol < means that the operator in the line has always lower preference than the operator in the column. The symbol > means that the operator in the line has always higher preference than the operator in the column. The symbol = means that the preference is not important, i.e., SOAR can choose one of them randomly. The symbol $*$ means that we should solve the preference using impasses and, for this experiment, we evaluate all the impasses using the same approach: the operator referring an entity which is closest to the agent will have the preference. So, for instance: if there are two jewels in the MEMORY, rules will propose two operators of the kind *Move Jewel*, each of them pointing to a different jewel. As a consequence, an impasse will occur, a sub-state will be generated, and while processing this new sub-state, additional rules will set the preference to the *Move Jewel* operator pointing to the jewel which is closest to the agent.

It is important to remember that there are different approaches to set preferences. It is possible to define a global preference, i.e., one operator with the highest or lowest preference amongst all the other operators (of course that if we have two operators of this same type proposed at the same time, this will not solve the impasse). It is possible to define a static preference between two operators, i.e., operator A will always have more preference than operator B. Finally, it is possible to define preferences by allowing the impasse to happen and providing additional rules just to solve the impasse. Box 7.3 provides some examples of how to deal with preferences among operators using different

[1] In fact, before halting the search, SOAR tries to solve the impasse creating a sub-state and trying to apply further preference rules. Nevertheless, if it is not able to solve the impasse, the same procedure is recursively applied, up to a maximum number of times, and if it reaches this limit, then it gives up and halts

approaches.

Box 7.3: Preference among operators

```
# Wander Preferences - Global preference
sp {wander*preferences
(state <s> ^operator <o> +)
(<o> ^name wander)
-->
(<s> ^operator <o> <)}

# Move Jewel vs Get Jewel - Static Preference
sp {moveJewel*getJewel*preferences
(state <s> ^operator <o> +
                     <o2> +)
(<o> ^name getJewel)
(<o2> ^name moveJewel)
-->
(<s> ^operator <o> > <o2>)}

# Move Jewel vs Move Jewel Preferences - Using Impasse
sp {moveJewel*moveJewel*less*distance
(state <s> ^attribute operator
           ^impasse tie
           ^item <o> {<> <o> <o2>}
           ^superstate <ss>)
(<ss> ^io.input-link <il>)
(<il> ^CREATURE <creature>)
(<o> ^name moveJewel)
(<o2> ^name moveJewel)
(<o2> ^parameter.distance <distance2>)
(<o>  ^parameter.distance <distance> <= <distance2>)
-->
(<ss> ^operator <o> > <o2>)}
```

Using our reactive approach, once a winner operator is selected by SOAR, the SOAR kernel halts and a command is provided at the output link to be sent to the creature's actuators. Now, at the bridge side, this command must be read, processed and applied to the virtual world. From the definition of the *Wander* operator in Box 7.2, the command MOVE is placed at the *output-link* node and, as described in box 7.4, this command together with its parameters are read in the bridge side and then sent to the virtual world.

Box 7.4: Output processing in SOAR

```
public void getReceivedCommands()
{
    if (agent != null)
    {
        int numberCommands = agent.GetNumberCommands();

        for (int i = 0 ; i < numberCommands ; i++)
        {
            // Check what is the command selected by SOAR
            Identifier pCommand = agent.GetCommand(i);
            String name  = pCommand.GetCommandName();

            if (name.equalsIgnoreCase("MOVE"))
            {
                // Get the parameters associated with the command
                Float rightVelocity = tryParseFloat(pCommand.GetParameterValue("VelR"));
                Float leftVelocity = tryParseFloat(pCommand.GetParameterValue("VelL"));
                Float linearVelocity = tryParseFloat(pCommand.GetParameterValue("Vel"));

                SendMoveCommandToWorldServer(rightVelocity, leftVelocity, linearVelocity)
            }
        }
    }
}
```

7.2 Implementation in CLARION

Even though the experimental conditions are the same as in the SOAR case, a totally different implementation was required in CLARION. This difference comes basically from the fact that SOAR is implemented as a framework, i.e., to include SOAR in your program, you just create an instance of its kernel and starts it. CLARION, otherwise, is implemented in the form of a toolkit. This means that you need to instantiate the parts of CLARION which will be used and connect them together in order to compose the real architecture used in a given application. In this sense, the architecture described in chapter 3 (figure 3.1) is a kind of reference architecture, which guides the construction of the real architecture making use of the classes available in the CLARION toolkit. Besides that, CLARION gives the option to the designer using (more or less) the resources of the top-down (rules and symbolic representations) and the bottom-up (neural networks). The designer might need to choose one among the other, or work with mixed styles while building its real cognitive architecture instance. Figure 7.2 shows how the CLARION reference architecture was customized in the current experiment.

The issue of knowledge representation becomes very important in the case of

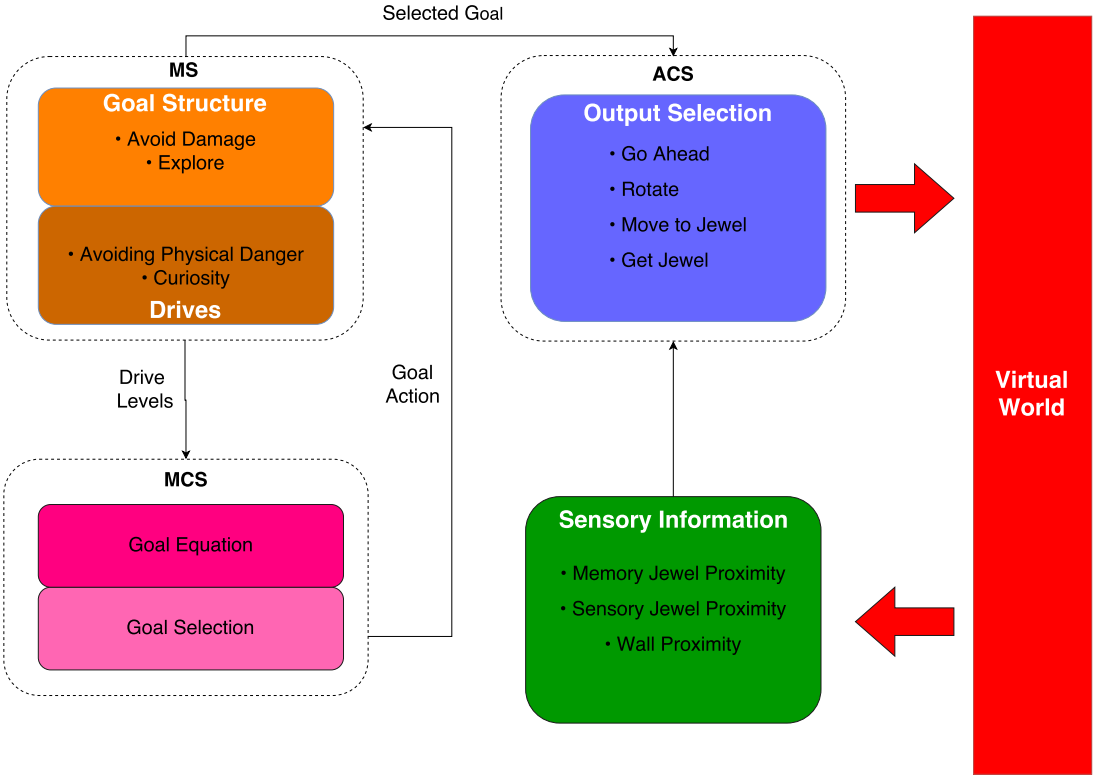


Figure 7.2 – Customization of CLARION for the Current Experiment

CLARION. Instead of working just with a tree of WMEs describing the current situation, as in SOAR, CLARION uses dimension-value pairs as input to the cognitive architecture. This requires a tuple composed by a string (describing the input) and an activation value (generally a numeric value) for each input. CLARION does not have a specific module for perception (see section 6.1), so all the processing to convert the raw data coming from the virtual world into dimension-value pairs is performed outside of the scope of the architecture, in the bridge layer. For our experiment, we defined three input dimension-value pairs to represent the current state situation:

1. **Wall Proximity**: the activation of this dimension-value pair is inversely proportional to the distance between the agent and the closest wall, i.e., the shorter the distance to the wall, the higher is the activation.

2. **Sensory Jewel Proximity**: the activation of this dimension-value pair is inversely proportional to the distance between the agent and the closest jewel in the visual field, i.e., the shorter the distance to the jewel, the higher is the activation.

3. **Memory Jewel Proximity**: Similar to the idea used in SOAR, we created in

CLARION's agent a short-term and size limited memory, in the form of a limited capacity list of jewels. The jewels in the visual field are stored in this memory if there is an available space and if that jewel was not memorized yet. So, this input will have the same behavior as the "Sensory Jewel Proximity", but now applied to the jewels stored in this memory, the closer the jewel, the higher the activation.

Box 7.5 code shows how the bridge layer (implemented in C#) receives data from the virtual world, process it and generates a new *Wall proximity* dimension-value pair.

We defined two goals in our MS Goal Structure: "Avoid Damage" and "Ex-

Box 7.5: Input data in CLARION

```
public class ClarionAgent
{
  Agent CurrentAgent = World.NewAgent("ClarionAgent");
  DimensionValuePair InputWall = World.NewDimensionValuePair("Sensorial", "Wall");

  private void RunAgent()
  {
    while (true)
    {
        //Perceive the sensory information
        SensoryInformation si = World.NewSensoryInformation(CurrentAgent);

        // Get New Sensorial Information from World Server
        IList<Thing> inputs = GetNewSensorialInformationEvent();

        // Get the creature information
        Creature creature = inputs.Where(item =>
                            (item.CategoryId == Thing.CATEGORY_CREATURE)).FirstOrDefault()
                            as Creature;

        // Get all the walls ordered by the distance to the creature
        Thing wall = (from item in inputs
                      where item.CategoryId == Thing.CATEGORY_BRICK
                      orderby Utils.GetMinimalGeometricDistanceToPoint(item, creature)
                              ascending
                      select item).FirstOrDefault();

        double inputWallActivation = 0.0;

        // Activation is proportional to the distance between agent and the wall
        if (wall != null)
        {
           double returnedValue = Utils.GetMinimalGeometricDistanceToPoint(wall, creature);
           inputWallActivation = GetActivationByDistance(returnedValue);
        }

        // Transfer perception information to the cognitive architecture
        si.Add(InputWall, inputWallActivation);
        CurrentAgent.Perceive(si);
    }
  }
}
```

plore". These two goals are related to the following drives in the MS bottom-level: "Avoiding Physical Danger" and "Curiosity". These drives are general use pre-implemented drives available in CLARION. The behaviors for each of these drives is defined using *delegates*, a type-safe pointer to a method signature available in C#. So, the "Avoiding Physical Danger" is calculated based on the "Wall Proximity" input according to the code in box 7.6:

Box 7.6: Equation for avoiding physical damage drive

```
private void AvoidPhysicalDamageDriveEquationImpl(ActivationCollection input,
ActivationCollection output)
{
  double wallActivation = input[InputWall.WorldID];
  double driveActivation = 0.0;

  if (wallActivation >= PROXIMITY_NEAR_ACTIVATION)
  {
        driveActivation = 1.0;
  }

  output[Drive.MetaInfoReservations.DRIVE_STRENGTH,
  typeof(AvoidingPhysicalDangerDrive).Name] = driveActivation;
}
```

The "Curiosity" drive equation is simpler than the "Avoiding Physical Danger" drive. It just returns a constant activation, symbolizing an agent with a persistent degree of curiosity. It is important to highlight that with the usage of delegates it is possible to create any custom implementation for a drive.

The process to attach these drives to goals is described in more details in the code presented in box 7.7. In this solution, these two goals are auto-exclusive, i.e, when one is activated the other one is not. This is accomplished using the "SET_RESET" attribute (when a new goal is selected, the previous one is removed from goal structure), using the coefficient value for the relevance between the goal and drive in such a way that "Avoid Damage" goal will have more priority than "Explore" (see equation 3.6 on section 3.4.1 for more details about goal setting).

Box 7.7: Goal and drive association in CLARION

```
GoalSelectionModule gsm =
AgentInitializer.InitializeMetaCognitiveModule(CurrentAgent, GoalSelectionModule.Factory);

GoalSelectionEquation gse =
AgentInitializer.InitializeMetaCognitiveDecisionNetwork(gsm, GoalSelectionEquation.Factory);

gse.Input.Add(avoidPhysicalDamageDrive.GetDriveStrength());
gse.Input.Add(curiosityDrive.GetDriveStrength());

// Initialize Basic Update Action Goals
GoalStructureUpdateActionChunk updateActionAvoidDamage =
World.NewGoalStructureUpdateActionChunk();

GoalStructureUpdateActionChunk updateActionExplore =
World.NewGoalStructureUpdateActionChunk();

updateActionAvoidDamage.Add(
GoalStructure.RecognizedActions.SET_RESET,AvoidDamageGoal);

updateActionExplore.Add(
GoalStructure.RecognizedActions.SET_RESET, ExploreGoal);

gse.Output.Add(updateActionAvoidDamage);
gse.Output.Add(updateActionExplore);

gsm.SetRelevance(updateActionAvoidDamage, avoidPhysicalDamageDrive, 1.0);
gsm.SetRelevance(updateActionExplore, curiosityDrive, 0.2);
```

With goals and inputs, it is time to feed ACS. We created just one entry in ACS bottom-level represented by a generic equation (again using delegates). No rules were created on ACS top-level, so the suggestion provided by ACS bottom-level is the chosen one to be executed by the agent. It is important to highlight one aspect: besides this approach using delegates, CLARION also allows ACS bottom-level to be implemented as a Q-Learning neural network which is fed with a feedback signal provided by the environment.

The code in box 7.8 highlights how ACS can be configured to receive the appropriate inputs and how it suggests an action based on these inputs.

The ACS levels are responsible to get all the information (inputs, goals, and memories) and suggest one single external action chunk as a possible output for the cognitive cycle. In this experiment, we only considered outputs to the environment (not to NACS or back to MS) and there are four possible outcomes:

1. **Move to jewel**: The agent is commanded to move towards a specific jewel.
2. **Get jewel**: The agent is commanded to catch a specific jewel when it is close enough.

Box 7.8: ACS configuration

```
public class ClarionAgent
{
  private ExternalActionChunk rotateClockWise = World.NewExternalActionChunk("Rotate");

  private void SetupAcs()
  {
    GenericEquation eq = AgentInitializer.InitializeImplicitDecisionNetwork(
    CurrentAgent, GenericEquation.Factory, (Equation)AcsBottomLevelDecisionEq);

    eq.Input.Add(AvoidDamageGoal,"goals");
    // (...)
    eq.Output.Add(rotateClockWise)
    // (...)
    CurrentAgent.Commit(eq);
  }

  private void AcsBottomLevelDecisionEq(ActivationCollection in, ActivationCollection out)
  {
    double avoidDamageGoalActivation = in[AvoidDamageGoal];
    double rotateClockWiseActivation = 0.0;

    // (...)
    if (avoidDamageGoalActivation >= 1.0)
    {
        rotateClockWiseActivation = 1.0;
    }
    // (...)

    out[OutputRotateClockWise] = rotateClockWiseActivation;
  }
}
```

3. **Rotate**: The agent is commanded to rotate clockwise in order to avoid collisions.

4. **Go ahead**: The agent is commanded to go ahead when it does not find any jewel in its visual field.

Finally, the bridge layer processes the selected output and a new command is sent to World Server, finalizing the cognitive cycle. At this point, if the experiment uses a Q-Learning network, it is time to collect the feedback after applying the action in order to provide to CLARION this data. This is illustrated in box 7.9.

Although we could use NACS to identify a jewel, the current structure of NACS makes handling multiple jewels a difficult task, because there is no way to add multiple instances of the same type of element in NACS top-level, being necessary to process this information outside of the architecture. Explaining in more details: in SOAR, a jewel in working memory has all the parameters related to that object (position, color, name, etc). So, handling multiple objects in SOAR is more straightforward, because once all the

Box 7.9: Output processing in CLARION

```
public class ClarionAgent
{
  private ExternalActionChunk rotateClockWise = World.NewExternalActionChunk("Rotate");

  private void RunAgent()
  {
    //(...)
    ExternalActionChunk chosen =  CurrentAgent.GetChosenExternalAction(si);
    if (chosen == rotateClockWise)
    {
    // Send appropriated command to World Server
    // (...)
    }
  }
}
```

information is provided to the architecture, SOAR can create multiple WME's, each one representing one specific jewel. In CLARION, NACS is very good to detect patterns, so probably we would easily identify that there is a jewel in the visual field of the agent, but NACS top-level does not have the same flexibility of having multiple instances of the same type of object (as in SOAR Working Memory). So, CLARION is good while selecting an action with the intent to move the agent towards a jewel. However, the precise direction of that jewel must be calculated outside of the architecture.

7.3 Implementation in LIDA

Yet, the experiment of implementing a LIDA solution to our problem became also quite different, compared to CLARION and SOAR. First, because in SOAR and CLARION we have a better control of the cognitive cycle. Generally, the client application is kept in an infinite loop where certain APIs are called successively, each one responsible for different tasks perceiving data, making decisions, etc. However, LIDA keeps this cognitive loop inside its framework, so instead of binding LIDA to your code, you need to provide some code which will be bound to LIDA. LIDA has the responsibility of calling your code during its inner cognitive cycle in order to have the expected results. Second, because it is also necessary to properly configure a set of XML files that are used as input for the architecture. These files are responsible to instantiate the agent modules, behaviors, properties and for enabling or disabling some debugging options. In this context, it is important to highlight three files:

- **factoryData.xml**: It defines the common data structures, strategies, and tasks that will be used in the architecture. In short, in this file, we can define what are the classes responsible for implementing basic entities on LIDA: nodes, links, decay strategy, etc. These classes should follow a specific interface in order to be called by LIDA's core.

- **agent.xml**: It is on this file where you define the LIDA's modules to be used in your agent, i.e., this is the place where you build your agent, specifying all the classes responsible for each LIDA's module (see chapter 4 for a description of all available LIDA's modules).

- **lidaConfig.properties**: This file provides general environment configurations: file path to the other configuration files, debug options, etc.

Let's highlight some important characteristics of the LIDA architecture. As we already mentioned, LIDA is based on *codelets*, i.e., small pieces of code, each one running independently, and focused on a specific task. So, to create an agent in LIDA is basically to develop the codelets which will be, together, responsible for the agent's behaviors.

The process of binding LIDA to an application requires several steps. First, we need to define a class to implement the `Environment` module in LIDA. This class should extend a base class (`EnvironmentImpl`) from the LIDA Framework. The `Environment` module is responsible for the agent's domain-specific code and should override three basic methods: the `init` method, used for some sort of initialization, the `getState` method, responsible for getting data from sensors, and the `processAction`, responsible for applying the selected action into the environment. In our case, we created a class called `AgentEnvironment` implementing the `Environment` module. Besides the class implementation, it is also necessary to edit the `agent.xml` file in order to declare this new class to be responsible for the implementation of the `Environment` module. Box 7.10 shows the class declaration in `agent.xml` file while Box 7.11 provides the class implementation in Java.

Beginning its cognitive cycle, LIDA uses the `Environment` module to collect the raw data from sensors, providing an input to the Sensory Memory. This is done by means of the `getState` method, which in our case basically connects to the virtual world

Box 7.10: AgentEnvironment class definition in agent.xml

```
<module name="Environment">
<class>agent.environment.AgentEnvironment</class>
<param name="height" type="int"> 10 </param>
<param name="width" type="int">10 </param>
<taskspawner>defaultTS</taskspawner>
</module>
```

Box 7.11: AgentEnvironment class declaration

```
public class AgentEnvironment extends EnvironmentImpl
{
   @Override
    public void init()
    {
        // Override for initial configuration
    }

    @Override
    public void processAction(Object o)
    {
        // Process output
    }

    @Override
    public Object getState(Map<String, ?> map)
    {
        // Process input

        Object rawDataFromSensors = GetDataFromWorldServer();

        return rawDataFromSensors;
    }
}
```

and get the current state, translating it into a Java `Object`. This `Object` is internally sent to the `SensoryMemory` module, where this raw data is further cataloged and structured by sensory codelets.

Similar to what was done before, it is necessary to create a custom Java class implementing the `SensoryMemory` module, extending the `SensoryMemoryImpl` base class in LIDA, which should be responsible for implementing the behaviors of this module. After that, it is necessary to declare the class in the `agent.xml` file.

Box 7.12 shows how this class is declared in `agent.xml` and Box 7.13 shows how this class is implemented. In our experiment, the agent only has one sensor (visual), so all environment data is related to the visual sensor. However, for agents with many sensors, the sensory memory will have a key role in cataloging and low-level processing the incoming data. The method `runSensors` gets the raw data from environment and store

Box 7.12: SensoryMemory module configuration in agent.xml

```
<module name="SensoryMemory">
<class>agent.environment.SensoryMemory</class>
<associatedmodule>Environment</associatedmodule>
<taskspawner>defaultTS</taskspawner>
<initialTasks>
<task name="backgroundTask">
<tasktype>SensoryMemoryBackgroundTask</tasktype>
<ticksperrun>5</ticksperrun>
</task>
</initialTasks>
</module>
```

Box 7.13: Sensory Memory in LIDA

```
public class SensoryMemory extends SensoryMemoryImpl
{
    private Map<String,Object> sensorParam = new HashMap<String, Object>();
    private VisualSensorReading visualSensorReading = null;

    public SensoryMemory()
    {
        sensorParam.put(AgentConstants.SENSOR_VISUAL, null);
    }

    @Override
    public void runSensors()
    {
       // Get Readings from visual Sensor
       Object readings = environment.getState(sensorParam);
       visualSensorReading = (VisualSensorReading) readings;
    }

    @Override
    public Object getSensoryContent(String sensorType, Map<String, Object> map)
    {
        Object contentData = null;

        if (sensorType.equalsIgnoreCase(AgentConstants.SENSOR_VISUAL)
        {
            /// Get visual sensor content in visualSensorReading
        }

        return contentData;
    }
}
```

it in an appropriate place for future processing, and the method `getSensoryContent` is called by the codelets, returning that specific sensor content with a low-level processing (if necessary).

The next step in the cognitive cycle is perception, which in LIDA is performed by PAM (Perceptual Associative Memory). As detailed in section 4.3, PAM can be viewed as a network of linked nodes, where each node is related to a concept (quite similar to

NACS top-level in CLARION) and, thanks to the spreading activation mechanism, the perceptual meaning is generated. The LIDA framework already has an implementation for PAM, but we need to configure the PAM nodes, according to the application and declare the module in `agents.xml`. We defined the following nodes in our PAM:

- **Wall in front**: There are one or more walls close enough to the agent to have a collision.
- **Wall clear**: There are no walls close enough to have a collision.
- **Jewels presence**: There are jewels in the visual field or in the agent memory (not necessarily close to the agent).
- **Jewel clear**: There are no jewels in the visual field or in the agent memory.
- **Jewel in front**: There is at least one jewel close enough to the agent, in order to be picked.

Box 7.14 shows how PAM module is declared in `agents.xml`. As we can see, the `node` parameters define all the nodes in our PAM (no links were defined in this experiment). If the node activation is higher or equal than `pam.perceptThreshold` property, this node will be placed in the Workspace. PAM nodes have their activation level calculated by a specific codelet attached to each node. In a general case, if we have links between nodes, the activation is spread through these links, so it is not necessary to have a codelet for all the nodes, just some of them. In our case, it is necessary because we don't have such links. The definition of these codelets is made in three places:

- First, it is necessary to create a JAVA class implementing the codelet, which should extend the `BasicDetectionAlgorithm` class. This class should be responsible for reading the data from the sensory memory (via the overloaded `detect` method), detecting the relevant features for that node and returning the proper activation level.
- Next, in `factoryData.xml` we need to define a task (the codelet) and associate it to this JAVA class, locating it in the right package.

Box 7.14: PAM definition in agents.xml

```
<module name="PerceptualAssociativeMemory">
<class>edu.memphis.ccrg.lida.pam.PerceptualAssociativeMemoryImpl</class>
<param name="pam.Upscale" type="double">.7 </param>
<param name="pam.Downscale" type="double">.6 </param>
<param name="pam.perceptThreshold" type="double">.7 </param>
<param name="nodes">wallFront, wallClear, jewelPresence, jewelClear, jewelFront</param>
<taskspawner>defaultTS</taskspawner>
<initialTasks>
<task name="WallClearDetector">
<tasktype>NodeActivationInversion</tasktype>
<ticksperrun>15</ticksperrun>
<param name="node" type="string">wallClear</param>
<param name="nodeToInvert" type="string">wallFront</param>
</task>
<task name="WallProximityDetector">
<tasktype>ObjectProximityDetector</tasktype>
<ticksperrun>15</ticksperrun>
<param name="node" type="string">wallFront</param>
<param name="object" type="string">WALL</param>
</task>
<task name="JewelClearDetector">
<tasktype>NodeActivationInversion</tasktype>
<ticksperrun>15</ticksperrun>
<param name="node" type="string">jewelClear</param>
<param name="nodeToInvert" type="string">jewelPresence</param>
</task>
<task name="JewelPresenceDetector">
<tasktype>ObjectPresenceDetector</tasktype>
<ticksperrun>13</ticksperrun>
<param name="node" type="string">jewelPresence</param>
<param name="object" type="string">JEWEL</param>
</task>
<task name="JewelProximityDetector">
<tasktype>ObjectProximityDetector</tasktype>
<ticksperrun>15</ticksperrun>
<param name="node" type="string">jewelFront</param>
<param name="object" type="string">JEWEL</param>
</task>
</initialTasks>
<initializerclass>edu.memphis.ccrg.lida.pam.BasicPamInitializer</initializerclass>
</module>
```

- Finally, in `agent.xml`, we need to correlate each PAM node to a task defined in `factoryData.xml` (it is possible to reuse the same type of task to different nodes).

These steps are illustrated in boxes 7.15, 7.16, and 7.17.

Attention codelets are responsible for monitoring specific nodes in Workspace and, if their activation level is greater than a certain threshold, one or more nodes and links (forming a coalition) are sent to Global Workspace, where they will compete for consciousness. Each attention codelet should be defined in `agent.xml` file as described in box 7.18. Two kinds of Attention codelets can be used, based either on the `BasicAttentionCodelet` or on the `DefaultAttentionCodelet` tasks, provided by the LIDA framework. All our

Box 7.15: Perceptual codelet class

```
public class ObjectProximityDetector extends BasicDetectionAlgorithm
{
  private Map<String, Object> smParams = new HashMap<String, Object>();

  @Override
  public void init()
  {
    super.init();
    String objectToDetect = (String)getParam("object", "");
    smParams.put(objectToDetect, "");
  }

  @Override
  public double detect()
  {
    double activationValue = 0.0;

    // Get Data from sensory memory
    List<Thing> allObjectsOfSomeType = (List<Thing>) sensoryMemory.getSensoryContent(
    AgentConstants.SENSOR_VISUAL,smParams);

    // Process data (...)

    return activationValue;
  }
}
```

Box 7.16: Perceptual codelet definition in factoryData.xml

```
<tasks>
  <task name="ObjectProximityDetector">
  <class>agent.featuredetectors.ObjectProximityDetector</class>
  <ticksperrun>10</ticksperrun>
  <associatedmodule>SensoryMemory</associatedmodule>
  <associatedmodule>PerceptualAssociativeMemory</associatedmodule>
  </task>
</tasks>
```

Box 7.17: Perceptual codelet definition in agent.xml

```
<task name="WallProximityDetector">
  <tasktype>ObjectProximityDetector</tasktype>
  <ticksperrun>11</ticksperrun>
  <param name="node" type="string">wallFront</param>

  <!-- Arguments that will be processed by the codelet -->
  <param name="object" type="string">WALL</param>
</task>
```

attention codelets are based on the `BasicAttentionCodelet` task. Using this task, LIDA only checks if a set of desired nodes are currently in the Workspace. If they are, the coalition is moved to the Global Workspace where it will compete for consciousness. LIDA also provides an alternative behavior, using a `DefaultAttentionCodelet`, where the promo-

Box 7.18: Attention codelet definition in agent.xml

```
<module name="AttentionModule">
<class>edu.memphis.ccrg.lida.attentioncodelets.AttentionCodeletModule</class>
<associatedmodule>Workspace</associatedmodule>
<associatedmodule>GlobalWorkspace</associatedmodule>
<taskspawner>defaultTS</taskspawner>
<initialTasks>
  <task name="JewelPresenceCollisionCodelet">
  <tasktype>BasicAttentionCodelet</tasktype>
  <ticksperrun>10</ticksperrun>
  <param name="nodes" type="string">jewelPresence, wallClear</param>
  <param name="refractoryPeriod" type="int">5</param>
  <param name="initialActivation" type="double">0.5</param>
  </task>
</initialTasks>
```

tion of a node to the Global Workspace requires the activation of each node in the coalition to be equal or higher than a certain threshold (defined by the `attentionThreshold` property).

For Workspace and Global Workspace modules, we did not create a custom class, i.e., for these modules we used implementations provided by LIDA. However, it is still necessary to define these modules in `agent.xml`. Box 7.19 shows the configuration for Workspace and box 7.20 for Global Workspace. Particularly for Global Workspace, it is important to highlight two aspects:

1. The activation of each coalition is the average activation of each node and link multiplied by the attention codelet's base-level activation (set in the `initialActivation` property in box 7.18).

2. The winner coalition will be chosen following the criteria defined in section 4.7. Recapping:

    a) At least one coalition has an activation level greater than a certain threshold (set by the `globalWorkspace.individualActivationThreshold` property in box 7.20).

    b) The sum of the activation level of the coalitions is greater than another threshold (set by the `globalWorkspace.aggregateActivationThreshold` property in box 7.20).

    c) No new coalitions arrived in the global workspace in a certain period of time (set by the `globalWorkspace.delayNoNewCoalition` property in box 7.20).

Box 7.19: Worskapce module in agent.xml

```
<module name="Workspace">
<class>edu.memphis.ccrg.lida.workspace.WorkspaceImpl</class>
<submodules>
<module name="PerceptualBuffer">
<class>edu.memphis.ccrg.lida.workspace.workspacebuffers.WorkspaceBufferImpl</class>
<taskspawner>defaultTS</taskspawner>
</module>
<module name="CurrentSituationalModel">
<class>edu.memphis.ccrg.lida.workspace.workspacebuffers.WorkspaceBufferImpl</class>
<taskspawner>defaultTS</taskspawner>
</module>
</submodules>
<taskspawner>defaultTS</taskspawner>
<initialTasks>
<task name="UpdateCsmBackgroundTask">
<tasktype>UpdateCsmBackgroundTask</tasktype>
<ticksperrun>5</ticksperrun>
</task>
</initialTasks>
</module>
```

d) A consciousness broadcast did not happen in a certain period of time (set by the `globalWorkspace.delayNoBroadcast` property in box 7.20).

Box 7.20: Global Worskapce module in agent.xml

```
<module name="GlobalWorkspace">
<class>edu.memphis.ccrg.lida.globalworkspace.GlobalWorkspaceImpl</class>
<param name="globalWorkspace.coalitionRemovalThreshold" type="double">0.0</param>
<param name="globalWorkspace.coalitionDecayStrategy">coalitionDecay</param>
<param name="globalWorkspace.refractoryPeriod" type="int">40 </param>
<param name="globalWorkspace.delayNoBroadcast" type="int">5 </param>
<param name="globalWorkspace.delayNoNewCoalition" type="int">5 </param>
<param name="globalWorkspace.aggregateActivationThreshold" type="double">2.0</param>
<param name="globalWorkspace.individualActivationThreshold" type="double">0.9</param>
<taskspawner>defaultTS</taskspawner>
<initializerclass>
edu.memphis.ccrg.lida.globalworkspace.GlobalWorkspaceInitalizer
</initializerclass>
</module>
```

Starting now the behavior generation tasks of LIDA's cognitive cycle, we need to provide the information necessary for the construction of the Scheme Net, used in LIDA for decision-making. The procedural memory stores all the possible agent behaviors in the form of schemes. These schemes are defined in the `agent.xml` file (see box 7.21), in the form of a formatted string, containing all the data required to define the scheme. This string is formatted as a sequence of *properties*, separated by pipes (|). In current LIDA implementation, the scheme contains the following properties:

Box 7.21: Procedural memory definition in agent.xml

```
<module name="ProceduralMemory">
<class>edu.memphis.ccrg.lida.proceduralmemory.ProceduralMemoryImpl</class>
<param name="proceduralMemory.ticksPerStep" type="int"> 14 </param>
<param name="scheme.1">if wallFront, rotate|(wallFront)()|action.rotateClockwise|()()|0.01
</param>
<taskspawner>defaultTS</taskspawner>
<initializerclass>
edu.memphis.ccrg.lida.proceduralmemory.BasicProceduralMemoryInitializer
</initializerclass>
</module>
```

Box 7.22: Action selection module in agent.xml

```
<module name="ActionSelection">
<class>edu.memphis.ccrg.lida.actionselection.BasicActionSelection</class>
<param name="actionSelection.ticksPerStep" type="int"> 10</param>
<taskspawner>defaultTS</taskspawner>
</module>
```

1. Scheme label - this is usually a human-readable explanation for the scheme, not used for processing
2. Context nodes (inside a parenthesis, separated by commas) and context links (inside the parenthesis, separated by commas).
3. Action name - the name of an action in the `SensoryMotorMemory` module (as described in section 4.10).
4. Result nodes (inside a parenthesis, separated by commas) and result links (inside the parenthesis, separated by commas).
5. Scheme base-level activation.

The schemes matching the current environment state are enabled and sent to the action selection module, being further on called *behaviors*. There, they start competing against each other, until just one behavior is selected for execution, following either Maes' behavior net algorithm or the basic action selection algorithm (see section 4.9). The action selection module has a pre-implementation provided in the LIDA Framework, so we just need to declare its use in the `agent.xml` file (as described in box 7.22). In our experiment, we configured the basic action selection mechanism to be used in this module.

Behaviors should also be linked to the algorithms implementing their actions. So, it is necessary to declare, at the sensory motor memory module, where all the behaviors (via action name) are correlated to the desired output (an alias given for that output) as illustrated in box 7.23.

Box 7.23: Sensory motor memory in agent.xml

```
<module name="SensoryMotorMemory">
<class>edu.memphis.ccrg.lida.sensorymotormemory.BasicSensoryMotorMemory</class>
<associatedmodule>Environment</associatedmodule>our
<param name="smm.mapping.1">action.rotateClockwise,algorithm.rotateClockWise</param>
<param name="smm.mapping.2">action.goAhead,algorithm.goAhead</param>
<param name="smm.mapping.3">action.goJewel,algorithm.goJewel</param>
<param name="smm.mapping.4">action.getJewel,algorithm.getJewel</param>
<taskspawner>defaultTS</taskspawner>
<initializerclass>edu.memphis.ccrg.lida.sensorymotormemory.BasicSensoryMotorMemoryInitializer
</initializerclass>
</module>
```

Finishing the cognitive cycle, we turn back to the `Environment` module (see boxes 7.10 and 7.11), where each action should be translated into an external command sent to the environment. Box 7.24 details the method `processAction` in box 7.11, responsible for translating LIDA actions into commands to the Virtual World.

Box 7.24: Output processing in LIDA

```
public class AgentEnvironment extends EnvironmentImpl
{
    @Override
    public void processAction(Object o)
    {
      if("algorithm.rotateClockWise".equalsIgnoreCase(action))
      {
         logger.log(Level.INFO, "Action rotateClockWise", TaskManager.getCurrentTick());
         // Send command to World Server
         CommandUtility.sendSetTurn(creature.getIndexID(), 0, 0, 2);
      }
    }
}
```

7.4 Analysis

After presenting the implementation details for the three CAs we used in this work, it is now time to present the simulation results and evolve an analysis of them. This analysis requires a careful examination of particular features of each CA, which

might be unique to each architecture, turning this analysis into a difficult task, due to the many differences among them. Some implementations can bring benefits only for specific conditions in specific scenarios. Thus, we should only conclude one CA to be better than other after considering all the variables in a well-controlled experiment.

In order to overcome these difficulties, we chose to focus this analysis on the *non-functional requirements* of each implementation. Differently from functional requirements, where a set of specific behaviors or functionalities are expected from a software, non-functional requirements are sometimes referred as the "quality attributes" of the software (CHUNG; LEITE, 2009), i.e., they define the criteria to be used for judging different aspects related to the software operation. Using this approach, it is possible to compare the implementations using the same "quality standard" that should be present in every implementation without considering the merit of the different theoretical foundations for each CA.

Usually, these non-functional requirements are called "ilities", because most of them refer to attributes with words ending with "ility" or "ity". These quality attributes can be divided into two big categories: *execution* and *evolution* qualities. As Mari and Eila define:

> "Execution qualities are observable at run-time. That is, they express themselves in the behavior of the system. Evolution qualities cannot be discerned at run-time, meaning that the solutions for evolution qualities lay in static structures of the software system. Therefore, they should be considered in the phases of the product's life cycle, i.e. in development and maintenance of a software system." (MARI; EILA, 2003).

For this comparison, we chose the following attributes:

- *Execution*
  - i **Performance**: how fast the software is doing the work it should do, i.e., the responsiveness of the system.
  - ii **Scalability**: the ability to handle (or to be prepared for) a growth in the amount of work.
- *Evolution*
  - i **Open source**: if the source code is available with a license which provides unrestricted use rights for any purpose.

ii **Maintainability and Modifiability**: how easy a software can be maintained or modified, i.e., how easy defects can be isolated and fixed, how easy new features can be added, how easily other people can support the product.

iii **Supportability**: refers to the technical support while installing or configuring the software.

Before going further, it is important to highlight some important differences among the implementations of each architecture. As we already mentioned, CLARION is provided in the form of a toolkit, while SOAR and LIDA are provided in the form of software frameworks. Thus, CLARION requires the agent to be coded using a set of functions available in the CLARION library (written in C#). Basically, in order to use CLARION, a designer must create a program for the agent to get information from the environment, call the CA functions using the interfaces defined by the library and act in the environment based on the responses given at the end of the CLARION cognitive cycle. So, the designer has the burden of connecting the many parts required for the CA to operate, in the source code. Now, instead, SOAR and LIDA provide a complete framework (in the Java language), which just needs to be instantiated and started. It is true, though, that both frameworks must be complemented with custom code. In the case of SOAR, this custom code should be written in a SOAR specific language, containing SOAR rules. In turn, LIDA requires additional custom code written in Java, together with configuration files in XML. These details have been considered during our analysis.

7.4.1 Execution attributes

Performance is an important attribute for most systems, because, generally, we want our tasks to be performed as faster as possible. Besides that, there might be time constraints for real-time systems, requiring a special attention in some situations. Scalability is another attribute that is very desirable in most systems, because it is very common the requirement to expand our system, increasing its workload or adding new features. So, for both scenarios, it is desirable for the system to handle growth in a smooth way.

Thus, in order to evaluate how each architecture deals with an increase in its workload, we run our experimental task considering a different (increasingly) number of

jewels (J) in the environment. A total of ten simulations was performed for each condition (i.e. for each J) and each simulation took about ninety seconds, having a jewels re-spawn time of seventeen seconds. Every cognitive cycle time was measured during the simulation, and the worst and average times were recorded. At the end of each condition, we evaluated the following statistics (based on the data of the ten simulations):

- Mean of the cognitive cycle average time in milliseconds (μ_C)
- Standard deviation value of the cognitive cycle average time (σ_C)
- Mean of the cognitive cycle worst time in milliseconds (μ_W)
- Standard deviation value of the cognitive cycle worst time (σ_W)

For all simulations, the agent's memory size (of jewels locations) is seven, a reference to the "magical number seven" described by Miller (MILLER, 1956). The recorded time considered only the time in the cognitive cycle (not including the time to read data and to send commands to World Server). Of course, depending on the computer running the experiment and the experiment itself, the numbers can be different. However, the main concern here is to compare differences while increasing the number of jewels and the corresponding response for each cognitive architecture. All the simulations were performed on the same computer.

The results for SOAR are summarized in table 7.2 and in figure 7.3. The results show that increasing the number of jewels in the experiment implies in an increment in the time spent in the cognitive cycle (for worst and average times). Actually, this is not a surprise. The increase of a jewel implies in the proposition of a new operator (e.g. to move to this new jewel, one operator for each new jewel), requiring SOAR to decide from a bigger

Table 7.2 – Performance and scalability results for SOAR

	$\boldsymbol{\mu_C}$ (ms)	$\boldsymbol{\sigma_C}$ (ms)	$\boldsymbol{\mu_W}$ (ms)	$\boldsymbol{\sigma_W}$ (ms)
$\boldsymbol{J = 1}$	4.664	0.234	27.958	5.190
$\boldsymbol{J = 3}$	5.797	0.464	33.507	6.760
$\boldsymbol{J = 5}$	6.103	0.509	33.226	8.530
$\boldsymbol{J = 10}$	7.280	0.262	32.780	10.276
$\boldsymbol{J = 20}$	10.481	0.639	46.723	9.487
$\boldsymbol{J = 50}$	17.942	1.100	88.477	13.023

set of proposed operators, following some priority. Recalling table 7.1, we can see that the preference among these operators is solved evaluating impasses. Basically, operators are compared two-by-two in order to find a winner (in our case, the winner operator refers to the closest jewel). So, equation 7.1 describes the number of comparisons C required to evaluate each impasse for this kind of operator, given the number of jewels J (where $J \geq 2$).

$$C = \binom{J}{2} = \frac{J!}{2!(J-2)!} = \frac{J(J-1)(J-2)!}{2!(J-2)!} = \frac{J^2 - J}{2} \tag{7.1}$$

Thus, a simple task of finding the closest jewel in a set of unsorted jewels would require an $\mathcal{O}(n)$ algorithm in any other procedural language, however for SOAR it takes $\mathcal{O}(n^2)$.

It is important to emphasize that other aspects can also influence in SOAR's performance (e.g. the chunking mechanism, as pointed on section 2.10). But, as we could see by the results, solving more impasses results in the degradation of performance. Moreover, using successive impasses to find the preference for a specific operator, the number of combinations can increase considerably because the solution of one impasse might require a further impasse and so on (this scenario is also referred as *nested impasses*).

Equation 7.1 can be checked using SOAR Debugger as shown in box 7.25. Looking carefully at this box, it is possible to confirm how many comparisons are being

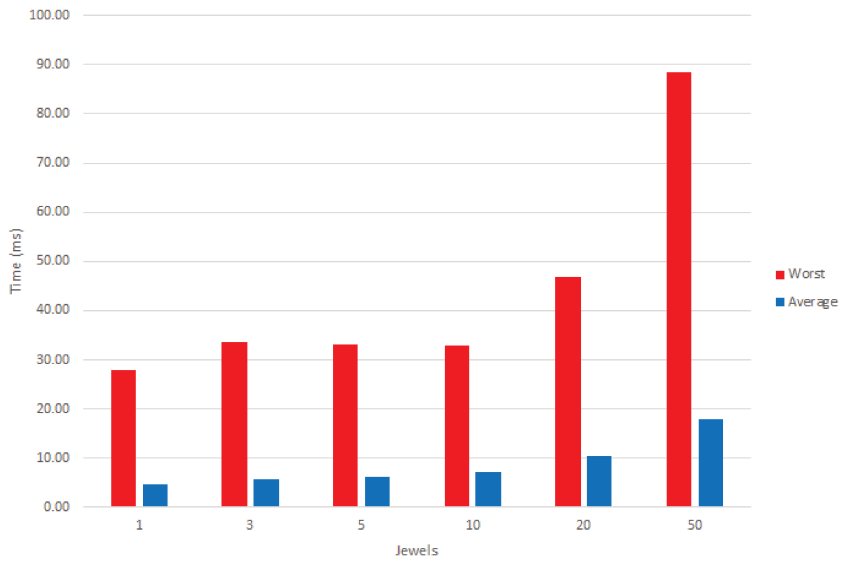


Figure 7.3 – Worst and average time for SOAR

evaluated for the command "Move to Jewel" only counting how many times the production `moveJewel*moveJewel*less*distance` appears. As a note, in this box each jewel is represented by a unique identifier (`O673`, `O674`, `O675`, `O676`, and `O677`), each comparison is performed between two jewels for $J = 5$ (thus $C = 10$).

Box 7.25: Number of comparisons for "Move Jewel" command in SOAR

```
--- Firing Productions (IE) For State At Depth 2 ---
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O677 > O676)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O673 > O676)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O677 > O675)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O676 > O675)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O673 > O675)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O677 > O674)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O676 > O674)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O675 > O674)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O673 > O674)
Firing moveJewel*moveJewel*less*distance
-->
(S1 ^operator O677 > O673)
```

CLARION results, presented in table 7.3 and figure 7.4, show a different scenario. First, it is possible to verify that the increase in the number of jewels does not impact significantly the performance of the agent. CLARION does not use an impasse

Table 7.3 – Performance and scalability results for CLARION

	μ_C (ms)	σ_C (ms)	μ_W (ms)	σ_W (ms)
$J = 1$	137.191	5.397	227.414	39.875
$J = 3$	133.738	2.796	211.200	26.271
$J = 5$	134.792	2.095	232.215	55.626
$J = 10$	134.134	2.234	213.002	20.916
$J = 20$	131.876	1.127	211.401	19.154
$J = 50$	136.376	3.721	232.666	65.270

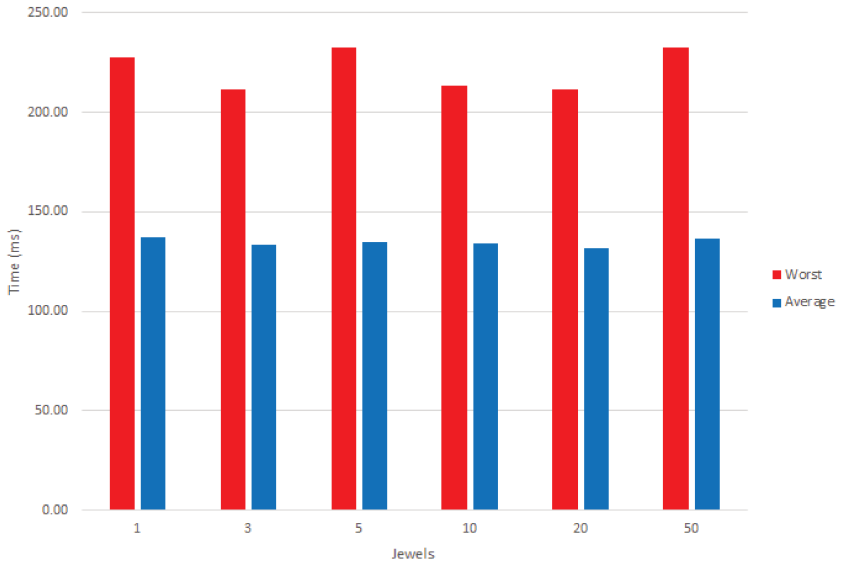


Figure 7.4 – Worst and average time for CLARION

approach as in SOAR, so the time spent to identify the closest jewel is proportional to the number of jewels in agent's memory or in the visual field. However, on the other hand, the average time in CLARION is significantly higher when compared to SOAR, possibly due to the chosen programming language (C#) or optimizations in the source code.

Analyzing LIDA results, table 7.4 and figure 7.5, it becomes evident that the number of jewels does not have a significant influence on the architecture performance for the same reason as explained for CLARION, i.e., the number of comparisons to be performed in order to identify the closest jewel is proportional to the number of jewels. The results show that an increase in the number of jewels does not necessarily imply in a big change in the architecture performance.

However, what draws more our attention is the magnitude of time measurements in all cases. The worst time almost reached 1.5 seconds, i.e., the cognitive architecture spent almost 1.5 seconds to decide its next action. Actually, during simulation, we

Table 7.4 – Performance and scalability results for LIDA

	μ_C (ms)	σ_C (ms)	μ_W (ms)	σ_W (ms)
$J = 1$	379.021	44.410	1263.323	146.424
$J = 3$	368.633	30.980	1261.328	135.154
$J = 5$	362.536	60.058	1274.776	177.761
$J = 10$	344.924	43.229	1285.525	159.579
$J = 20$	396.619	35.738	1345.021	175.139
$J = 50$	406.571	42.529	1351.791	104.707

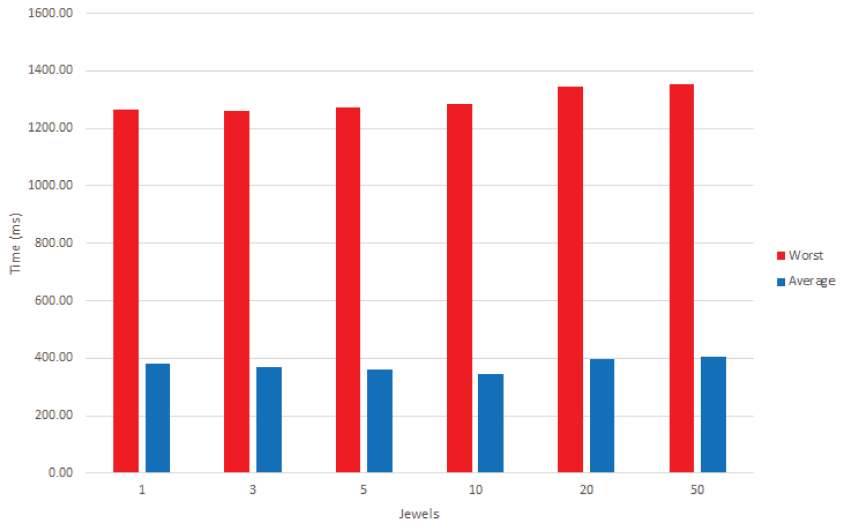


Figure 7.5 – Worst and average time for LIDA

identified many cases resulting in agents colliding with objects, because the architecture was not responsive enough to decide a better action, avoiding the collision. Even considering only average time, it is around three times higher than CLARION. Since LIDA uses a framework based application, we have less control over the cognitive cycle, so we have less flexibility to overcome possible issues.

In order to have a better overview of the results, figure 7.6 and figure 7.7 show respectively the average and worst time comparison for all architectures.

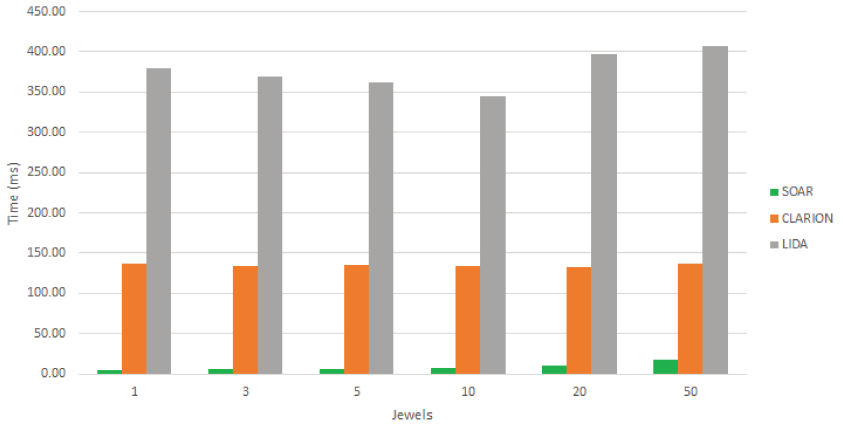


Figure 7.6 – Average time comparison

7.4.2 Evolution attributes

SOAR and LIDA are open source projects, making their source code available to anyone interested in developing with them and studying the source code. On the

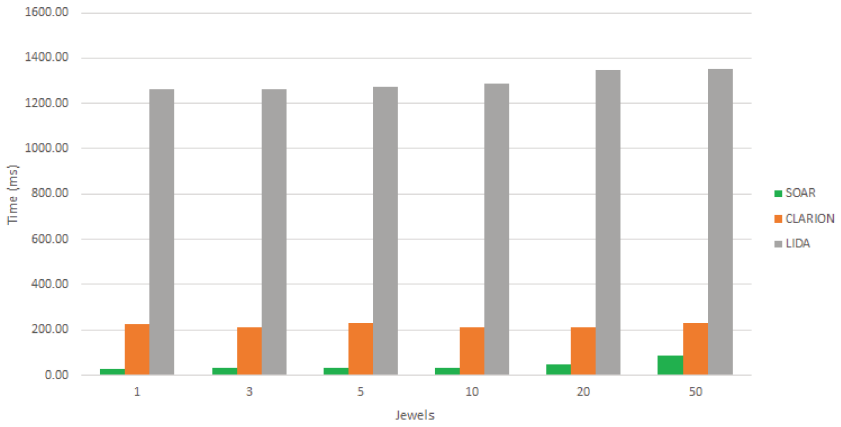


Figure 7.7 – Worst time comparison

contrary, for CLARION only the binaries are available. We don't have access to its source code. Since we are dealing with a complex system, with a lot of possibly undocumented features, this closed source policy increases the time to detect issues longer and avoids the introduction of new features to the architecture.

With respect to the developmental support, SOAR has a better documentation than the other architectures, with a lot of tutorials, examples, and explanations about each module in the architecture. LIDA and CLARION try to address the basic issues for newcomers, with a few tutorials. However, the documentation lacks more detailed information, regarding more advanced features. Besides that, SOAR has a more active community, where questions can be quickly answered (being an older architecture, SOAR has a different level of maturity).

All architectures provide some sort of support to the integration with legacy systems developed in different languages. For all the architectures, it is necessary to import a specific library in order to have access to the features provided by each CA. CLARION provides native support to C# and LIDA to Java. On the other hand, SOAR provides support for integrating with several languages: C++, Java, Tcl, Python and C# applications.

However, SOAR has a caveat. As described in section 7.1, the operators and preferences in SOAR are described using its own language to encode rules and facts. Obviously, there are pros and cons to using this approach. An evident benefit is the easiness of making changes in the agent behaviors. Usually, it will imply the updating of a rule file and its deployment. The next time the agent runs, it will read this new configuration and

automatically the new behaviors will be applied. For the other architectures, probably, it would be necessary to re-compile the entire agent code and deploy the solution into the target. Of course, this is not a big deal if you are in a controlled environment, but, in some production scenarios, it will be necessary to create another program, responsible for securely updating the agent, implying in a more complex solution. It is important to highlight other two points regarding this aspect: if new inputs (in a WME format) or new outputs are necessary, an update in the agent's code might also be necessary. The other point is that LIDA has something similar, due to the way the architecture was implemented. That is, it is possible to change some of the agent behaviors just making changes to XML files (see section 7.3). However, SOAR's rule files provide more flexibility than LIDA's XML files.

The con refers to two factors: first, SOAR's rule language is not a powerful language. When we are programming using Java or C#, we have a powerful framework providing methods and classes to increase the productivity. SOAR's rule language allows just a few mathematical functions. So, depending on what we want to do, it is better to calculate things outside the architecture and provide them to SOAR in a WME format. The second factor: there are not so many people proficient in SOAR's rule language, so encoding behaviors in SOAR's language might probably require a learning curve for newcomers. Regarding this last point, it is important to make a parallel with LIDA. LIDA's XML files can turn the development of a new agent into a very complex task. There are so many undocumented properties and features defined in the XML configuration files that, in order to have a clue on how doing the right thing, the user most probably will need to look into the source code. This will result also in a shortcoming for a newcomer.

7.5 Summary

In this chapter, we described how each cognitive architecture can be used to solve the same experimental problem. We analyzed the simulation results based on a set of non-functional requirements (execution and evolution categories). Finally, next chapter provides a final conclusion for the whole work.

8 Conclusions

Cognitive architectures rely on different cognitive models for explaining the appearance of cognitive functions in living beings (especially in humans). Considered as possible solutions for the construction of intelligent systems, their usage is typically indicated for complex scenarios, as e.g. dynamic environments, where a large number of possible actions might be taken, in a given moment. Although they could be applied for simpler problems, the complexity of the solution maybe does not justify that effort. Thus, for the cases that they are indicated, cognitive architectures can bring a lot of benefits, replicating biological features in artificial systems in order to create more resilient agents, capable to deal with several types of situations without the need of human intervention.

SOAR, CLARION, and LIDA were the three cognitive architectures described during this work and their detailed comparison is our main contribution. At the beginning, we described each cognitive architecture, detailing in one single place how each component works in such a way that a new researcher can easily understand how each cognitive function was addressed. In this context, the reader was introduced to several concepts: reinforcement learning, symbolic and sub-symbolic dichotomy, sparse distributed memory, Mae's behavior net, etc.

Two different approaches were used to compare these architectures: a theoretical inquiry, where the architecture models were compared according to some cognitive functions and an empirical approach, where the same experiment was applied in all the architectures, in order to evaluate different aspects regarding the implementations and the applicability of each solution in real problems.

During the theoretical inquiry, we could compare the different approaches used to deal with some cognitive functions. First, it was possible to observe that SOAR is a predominantly symbolic architecture, despite having some sub-symbolic modules. On the other hand, LIDA and CLARION have a mixed approach that tries to combine the benefits of both paradigms. Regarding goals, SOAR does not have any built-in motivational process (responsible for the generation of new goals), in turn, CLARION has a more elaborated approach using the Hull's drive reduction theory and concerning LIDA, just recently

McCall proposed a new mechanism of motivations based on Damasio's view of feelings whose the main idea is similar to CLARION's approach.

Besides that, CLARION also has a unique probabilistic decision procedure mechanism for action selection, which can bring more randomness, but it also affects the repeatability of some experiments when compared to the more deterministic approach used in LIDA and SOAR. When we covered the types of memories and the learning processes, LIDA was the only architecture which addressed all the types of memories and learning processes described in this work. Finally, LIDA is also the only architecture to use an established model for consciousness, the Global Workspace Theory, which performs an important role in the learning process.

Moving further in this work, the empirical comparison probably is the most important contribution. It brought the theoretical debate to the real world. Over this study, a new researcher could understand step-by-step how to use each cognitive architecture in real experiments. Moreover, based on the comparison using non-functional requirements, we can see, at a first glance, if a CA is appropriate for a given situation. Although SOAR has demonstrated the best performance (based on the average cognitive cycle time), the rising number of impasses directly contributed to a performance degradation. Besides that, CLARION's approach of delivering the cognitive architecture as a toolkit seemed to be more interesting because we have more control over the cognitive cycle and we also have more flexibility. The common issue found in all architectures is the lack of formalism, i.e., a software development guided by a process and applied during the specification, development, and verification of the system. This approach would increase the reliability, robustness, and maintainability of the architectures.

With the expertise gained in learning how to operate these three architectures, their pros and cons, we can envision, as a future work, the proposal of a new cognitive architecture, gathering the good aspects of each CA and avoiding the common pitfalls encountered during our first contact with them.

Structurally, we might use LIDA as the foundation for this new architecture, because it uses codelets as basic elements, a solution we found original and elegant. The codelet approach is particularly interesting because it brings modularity and decoupling to the overall architectures, allowing different kinds of solutions (e.g. rule-based systems,

fuzzy systems, evolutionary systems, neural networks of different kinds, like deep learning convolutional networks or HTM: hierarchical temporal memories) to be easily integrated. LIDA's learning mechanisms are also a source of inspiration, with its vast amount of memory types and learning processes. Consciousness is another area where LIDA takes advantage, because it is the only architecture implementing a scientific model for consciousness, something which proved to be very useful in several scenarios, especially in the learning process.

A mixed representation for perception (symbolic and sub-symbolic) brings a lot of benefits. So, in this area, it is possible to use the mechanism applied either in LIDA or in CLARION, where sub-symbolic elements can trigger symbolic entities. However, SOAR can also contribute to the representation process. The big advantage of SOAR in this area is the fact that we can virtually add anything to the working memory. Taking as an example a case from our experiment, if an agent is observing five jewels at a moment, there will be five representations of jewels with all the parameters related to them in SOAR working memory. On the other hand, CLARION and LIDA only recognize that there is a jewel in the visual field, but the number of jewels and their properties should be handled outside the scope of the architecture. Ideally, besides only detecting the patterns distinguishing an object (using a sub-symbolic process to detect symbolic elements), a cognitive architecture should also address this situation where there are multiple instances of the same type of object, but with different properties.

Goals and the action selection process are areas where CLARION can bring interesting contributions. A specific module responsible for goals and motivations is only present in CLARION and it proved to be very useful when dealing with long-term targets in order to generate orchestrated actions to accomplish the objective. Regarding action selection, a mechanism based on Maes' behavior net (used in LIDA) or even using preferences evaluation (used in SOAR) relies on the fact that all the situations should be mapped by design. This can be a very challenging task while dealing with dynamic environments. At this point, CLARION's approach of using a sub-symbolic representation (generally based on neural networks) and combining it with a symbolic representation (generally based on rules) can bring more flexibility to the action selection process.

Finally, an important consideration regarding this new architecture is choos-

ing to implement it as a framework or a toolkit. In our opinion, it will be adequate to implement it as a toolkit (something close to how CLARION is implemented). The big disadvantage of having a framework, which makes everything and control all the tasks (as in LIDA and SOAR), is the lack of control and flexibility to adapt some points for different scenarios. Besides that, as already discussed in the present work, maintainability, modifiability, and supportability are very important aspects of any development. For these items, SOAR proved to be a good option, having a deep documentation and a large community. CLARION and LIDA need to improve in this area.

Of course, the most important criteria covered in this work regarding implementation was the performance and scalability attributes. As we could see, CLARION showed a good performance and also a good scalability. SOAR showed an excellent performance but is less scalable. LIDA had the worst performance, although its cycle time was not influenced by an increase in demand (showing some scalability). These attributes must be taken into account depending on the requirements involved in the experiment.

As we could see, each cognitive architecture has its pros and cons, so the intent of this work is to show all the particularities of each CA in order for a new researcher to identify which one is better applicable to a given situation. Besides that, the other outcome is to propose a debate on possible improvements on already existing cognitive architectures in order to improve them. Finally, as a future work, we propose a new set of experiments in order to explore more features of each architecture and also to model and implement a new cognitive architecture based on this comparative study, trying to collect all the positive aspects of SOAR, CLARION, and LIDA.

Bibliography

ALEKSANDER, I. Artificial neuroconsciousness an update. In: SPRINGER. *International Workshop on Artificial Neural Networks*. [S.l.], 1995. p. 566–583. Cited on page 103.

ATKINSON, A. P.; THOMAS, M. S.; CLEEREMANS, A. Consciousness: mapping the theoretical landscape. *Trends in Cognitive Sciences*, Elsevier, v. 4, n. 10, p. 372–382, 2000. Cited 3 times on pages 82, 103, and 104.

BAARS, B. J.; GAGE, N. M. *Cognition, brain, and consciousness: Introduction to cognitive neuroscience*. [S.l.]: Academic Press, 2010. Cited 17 times on pages 59, 60, 62, 64, 66, 67, 68, 72, 73, 74, 75, 78, 81, 91, 92, 98, and 99.

BADDELEY, A. The episodic buffer: a new component of working memory? *Trends in cognitive sciences*, Elsevier, v. 4, n. 11, p. 417–423, 2000. Cited on page 67.

BADDELEY, A. Working memory: theories, models, and controversies. *Annual review of psychology*, Annual Reviews, v. 63, p. 1–29, 2012. Cited on page 30.

BLOCK, N. On a confusion about a function of consciousness. *Behavioral and Brain Sciences*, v. 18, p. 227–247, 5 1995. ISSN 1469-1825. Cited on page 103.

CASTRO, E. C.; GUDWIN, R. R. A scene-based episodic memory system for a simulated autonomous creature. *International Journal of Synthetic Emotions (IJSE)*, IGI Global, v. 4, n. 1, p. 32–64, 2013. Cited 3 times on pages 83, 98, and 99.

CHALMERS, D. J. Facing up to the problem of consciousness. *Journal of consciousness studies*, Imprint Academic, v. 2, n. 3, p. 200–219, 1995. Cited on page 103.

CHUNG, L.; LEITE, J. C. S. do P. On non-functional requirements in software engineering. In: *Conceptual modeling: Foundations and applications*. [S.l.]: Springer, 2009. p. 363–379. Cited on page 130.

CONWAY, M. A. Sensory–perceptual episodic memory and its context: Autobiographical memory. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, The Royal Society, v. 356, n. 1413, p. 1375–1384, 2001. Cited on page 67.

COX, M. T. Metacognition in computation: A selected research review. *Artificial intelligence*, Elsevier, v. 169, n. 2, p. 104–141, 2005. Cited on page 54.

CRAIG, R. T.; MULLER, H. L. *Theorizing communication: Readings across traditions*. [S.l.]: Sage, 2007. Cited on page 86.

DENNING, P. J. Sparse distributed memory. 1989. Cited 2 times on pages 67 and 70.

DRESCHER, G. L. *Made-up minds: a constructivist approach to artificial intelligence*. [S.l.]: MIT press, 1991. Cited 2 times on pages 59 and 75.

ELIASMITH, C.; BECHTEL, W. Symbolic versus subsymbolic. *Encyclopedia of cognitive science*, Wiley Online Library, 2003. Cited on page 87.

FRANKLIN, S.; BAARS, B. J.; RAMAMURTHY, U.; VENTURA, M. The role of consciousness in memory. *Brains, Minds and Media*, v. 1, n. 1, p. 38, 2005. Cited on page 104.

FRANKLIN, S.; JR, F. P. The lida architecture: Adding new modes of learning to an intelligent, autonomous, software agent. *pat*, v. 703, p. 764–1004, 2006. Cited on page 102.

FRANKLIN, S.; MADL, T.; D'MELLO, S.; SNAIDER, J. Lida: A systems-level architecture for cognition, emotion, and learning. *Autonomous Mental Development, IEEE Transactions on*, IEEE, v. 6, n. 1, p. 19–41, 2014. Cited 2 times on pages 60 and 71.

FRANKLIN, S.; MADL, T.; STRAIN, S.; FAGHIHI, U.; DONG, D.; KUGELE, S.; SNAIDER, J.; AGRAWAL, P.; CHEN, S. A lida cognitive model tutorial. *Biologically Inspired Cognitive Architectures*, Elsevier, v. 16, p. 105–130, 2016. Cited 10 times on pages 58, 59, 60, 66, 71, 73, 75, 78, 101, and 102.

FRANKLIN, S.; RAMAMURTHY, U.; D'MELLO, S. K.; MCCAULEY, L.; NEGATU, A.; SILVA, R.; DATLA, V. *et al.* Lida: A computational model of global workspace theory and developmental learning. In: AAAI PRESS ARLINGTON, VA. *AAAI Fall Symposium on AI and Consciousness: Theoretical Foundations and Current Approaches*. [S.l.], 2007. p. 61–66. Cited on page 59.

GAMEZ, D. Progress in machine consciousness. *Consciousness and cognition*, Elsevier, v. 17, n. 3, p. 887–910, 2008. Cited 2 times on pages 58 and 81.

GOERTZEL, B.; PENNACHIN, C.; GEISWEILLER, N. *Engineering General Intelligence, Part 1: A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*. [S.l.]: Springer, 2014. v. 5. Cited on page 80.

HARNAD, S. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, Elsevier, v. 42, n. 1, p. 335–346, 1990. Cited on page 87.

HOFSTADTER, D. R.; MITCHELL, M. *et al.* The copycat project: A model of mental fluidity and analogy-making. *Advances in connectionist and neural computation theory*, v. 2, n. 31-112, p. 29–30, 1994. Cited 3 times on pages 59, 63, and 64.

HUANG, B.-Q.; CAO, G.-Y.; GUO, M. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In: IEEE. *2005 International Conference on Machine Learning and Cybernetics*. [S.l.], 2005. v. 1, p. 85–89. Cited on page 42.

HULL, C. L. Principles of behavior: an introduction to behavior theory. Appleton-Century, 1943. Cited 2 times on pages 50 and 91.

KANERVA, P. *Sparse distributed memory*. [S.l.]: MIT press, 1988. Cited 2 times on pages 59 and 68.

KENNEDY, W. G.; JONG, K. A. D. Characteristics of long-term learning in soar and its application to the utility problem. In: *ICML*. [S.l.: s.n.], 2003. p. 337–344. Cited on page 34.

KOCH, C. *The Quest for Consciousness: A Neurobiological Approach*. Roberts and Company, 2004. ISBN 9780974707709. Disponível em: <https://books.google.com.br/books?id=7L9qAAAAMAAJ>. Cited on page 71.

KOTSERUBA, I.; GONZALEZ, O. J. A.; TSOTSOS, J. K. A review of 40 years of cognitive architecture research: Focus on perception, attention, learning and applications. *arXiv preprint arXiv:1610.08602*, 2016. Cited on page 80.

LAIRD, J. E. Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, IOS Press, v. 171, p. 224, 2008. Cited 2 times on pages 17 and 18.

LAIRD, J. E.; CONGDON, C. B. *The Soar User's Manual Version 9.4.0*. [S.l.], 2014. Cited 9 times on pages 18, 19, 27, 28, 31, 34, 35, 97, and 104.

LUCENTINI, D. F.; GUDWIN, R. R. A comparison among cognitive architectures: A theoretical analysis. *Procedia Computer Science*, Elsevier, v. 71, p. 56–61, 2015. Cited on page 88.

MADL, T.; BAARS, B. J.; FRANKLIN, S. The timing of the cognitive cycle. *PloS one*, Public Library of Science, v. 6, n. 4, p. e14803, 2011. Cited 2 times on pages 60 and 61.

MAES, P. How to do the right thing. *Connection Science*, Taylor & Francis, v. 1, n. 3, p. 291–323, 1989. Cited 3 times on pages 59, 75, and 92.

MARI, M.; EILA, N. The impact of maintainability on component-based software systems. In: IEEE. *Euromicro Conference, 2003. Proceedings. 29th*. [S.l.], 2003. p. 25–32. Cited on page 130.

MASLOW, A. H. A theory of human motivation. *Psychological review*, American Psychological Association, v. 50, n. 4, p. 370, 1943. Cited on page 51.

MCCALL, R. J. *Fundamental motivation and perception for a systems-level cognitive architecture*. [S.l.]: The University of Memphis, 2014. Cited 3 times on pages 75, 93, and 94.

MCCALL, R. J.; FRANKLIN, S.; FRIEDLANDER, D.; D'MELLO, S. Grounded event-based and modal representations for objects, relations, beliefs, etc.. In: *FLAIRS Conference*. [S.l.: s.n.], 2010. Cited on page 93.

MILLER, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, American Psychological Association, v. 63, n. 2, p. 81, 1956. Cited 2 times on pages 26 and 132.

NASON, S.; LAIRD, J. E. Soar-rl: Integrating reinforcement learning with soar. *Cognitive Systems Research*, Elsevier, v. 6, n. 1, p. 51–59, 2005. Cited on page 35.

NEWELL, A.; SIMON, H. A. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, ACM, v. 19, n. 3, p. 113–126, 1976. Cited on page 86.

NILSSON, N. J. *Artificial intelligence: a new synthesis*. [S.l.]: Morgan Kaufmann, 1998. Cited on page 86.

NUXOLL, A.; LAIRD, J. E.; JAMES, M. Comprehensive working memory activation in soar. In: *International conference on cognitive modeling*. [S.l.: s.n.], 2004. p. 226–230. Cited 2 times on pages 32 and 33.

RAMAMAURTHY, U.; MELLO, S. K.; FRANKLIN, S. Modified sparse distributed memory as transient episodic memory for cognitive software agents. In: IEEE. *Systems, Man and Cybernetics, 2004 IEEE International Conference on*. [S.l.], 2004. v. 6, p. 5858–5863. Cited on page 70.

RAMAMURTHY, U.; D'MELLO, S. K.; FRANKLIN, S. Realizing forgetting in a modified sparse distributed memory system. In: *Proceedings of the 28th Annual Conference of the Cognitive Science Society*. [S.l.: s.n.], 2006. p. 1992–1997. Cited on page 70.

RAMAMURTHY, U.; FRANKLIN, S. Memory systems for cognitive agents. In: *Proceedings of human memory for artificial agents symposium at the artificial intelligence and simulation of behavior convention (AISB'11)*. [S.l.: s.n.], 2011. p. 35–40. Cited on page 68.

SAMSONOVICH, A. V. Toward a unified catalog of implemented cognitive architectures. *BICA*, v. 221, p. 195–244, 2010. Cited on page 80.

SEARLE, J. Chinese room argument, the. *Encyclopedia of Cognitive Science*, Wiley Online Library, 2001. Cited on page 17.

SNAIDER, J.; MCCALL, R.; FRANKLIN, S. The lida framework as a general tool for agi. In: *Artificial general intelligence*. [S.l.]: Springer, 2011. p. 133–142. Cited 2 times on pages 63 and 65.

SNAIDER, J.; MCCALL, R.; STRAIN, S.; FRANKLIN, S. The lida tutorial. *Cognitive Computing Research Group, University of Memphis*, 2012. Cited on page 71.

SOCIETY, B. *Comparative Table of Cognitive Architectures*. 2012. <http://bicasociety.org/cogarch/architectures.htm>. Cited on page 15.

SUN, R. A tutorial on clarion 5.0. *Unpublished manuscript*, 2003. Cited 11 times on pages 42, 43, 44, 46, 47, 48, 50, 52, 54, 56, and 101.

SUN, R. Desiderata for cognitive architectures. *Philosophical Psychology*, Taylor & Francis, v. 17, n. 3, p. 341–373, 2004. Cited 5 times on pages 15, 39, 50, 81, and 86.

SUN, R. The clarion cognitive architecture: Extending cognitive modeling to social simulation. *Cognition and multi-agent interaction*, Cambridge, UK: Cambridge Univ. Press, p. 79–99, 2006. Cited on page 40.

SUTTON, R. S.; BARTO, A. G. *Reinforcement learning: An introduction*. [S.l.]: MIT press, 1998. Cited on page 37.

TYRRELL, T. *Computational mechanisms for action selection*. Tese (Doutorado) — University of Edinburgh Edinburgh, Scotland, 1993. Cited 2 times on pages 94 and 96.

WATKINS, C. J.; DAYAN, P. Q-learning. *Machine learning*, Springer, v. 8, n. 3-4, p. 279–292, 1992. Cited on page 35.

WHITESON, S.; TAYLOR, M. E.; STONE, P. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, SAGE Publications, v. 15, n. 1, p. 33–50, 2007. Cited on page 45.