Abstract:
This document constitutes Deliverable D4.2 of the IST Project SAFIRA.

Keywords

Document Id: D-SAFIRA-WP4-D4.2
Commission Ref.: D4.2
Classification: project&commission
Distribution: project&commission
Date: 5 July 2002
### Combined Shell for emotional processing and emotional evolution

<table>
<thead>
<tr>
<th>Deliverable Number:</th>
<th>D-SAFIRA-WP4-D4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Contractual Date of Delivery to CEC</td>
<td>WP4</td>
</tr>
<tr>
<td>Actual Date of Delivery to CEC</td>
<td>P-Public</td>
</tr>
<tr>
<td>Editor</td>
<td>R-Report</td>
</tr>
<tr>
<td>Deliverable Type</td>
<td>WP2, WP3, WP4, WP5, WP6, WP7</td>
</tr>
<tr>
<td>Deliverable Nature</td>
<td>INESC, ADETTI, DFKI, GMD, ICSTM, OFAI, SICS</td>
</tr>
<tr>
<td>Workpackages Contributing</td>
<td>WP2, WP3, WP4, WP5, WP6, WP7</td>
</tr>
<tr>
<td>Partners Contributing</td>
<td>INESC, ADETTI, DFKI, GMD, ICSTM, OFAI, SICS</td>
</tr>
</tbody>
</table>

This document is a Deliverable of the SAFIRA project (“Supporting Affective Interactions for Real-time Applications”), a project performed within the IST programme (project identifier IST-1999-11683).

The SAFIRA project is performed by a consortium consisting of the following partner organizations:

- Instituto de Engenharia de Sistemas e Computadores - INESC (P, Coordinating Partner)
- Associação para o Desenvolvimento das Telecomunicações e Técnicas de Informática - ADETTI (P)
- Deutsches Forschungszenrum fuer Kuenstliche Intelligenz GmbH - DFKI (D)
- GMD – Forschungszentrum Informationstechnik GmbH - GMD (D)
- Imperial College of Science, Technology and Medicine (UK)
- Austrian Research Institute for Artificial Intelligence - OFAI (A)
- Swedish Institute of Computer Science - SICS (SE)
Executive Summary

This Deliverable (D-SAFIRA-WP4-D4.2) is the second deliverable from WP4 (Reasoning and Planning with Emotions) of the IST-sponsored SAFIRA project. It contains a detailed description of the software tools developed by the project for the creation of Emotional Agents.

SAFIRA has defined a general Integration Framework underlying the creation of emotional agents using the software components developed in the project. The produced components can be integrated into a single agent because they were developed following the general framework defined in the project.

Agent designers may also create their own components and integrate them with SAFIRA components, as long as they are aware of the defined general Integration Framework.

SAFIRA Shell for emotional agents is a set of software components developed by the project partners following the defined Integration Framework.

SAFIRA has developed a set of software tools to facilitate the processing of emotions and the evolution of emotions. The emotion processing tools include the Appraisal Compiler and the Emotion Production System Shell. The Appraisal Compiler can be used by component designers to create appraisal components. The Emotional Production System Shell, called Em-PSys, can be used by component designers to create emotional agent control components, using production rules. The Appraisal Component and the Emotional Production System component are built on top of the Salt & Pepper Architecture. The Developmental Model component of the SAFIRA toolkit allows for the representation of long-term changes in agent behaviour and functionality, which may be triggered by affective experiences. The Developmental Model uses a simple, flexible representation of development to support components whose functionality needs to change over time based on an agent’s developmental stage.

The SAFIRA general Integration Framework is defined in chapter 2. The software tools developed for emotional processing are described in chapter 3. Chapter 1 describes the component for emotion evolution. Chapter 5 discusses the developed components and presents conclusions.
< THIS PAGE IS LEFT BLANK INTENTIONALLY >
this page is left blank intentionally
Table of Contents

1 SHELL FOR EMOTIONAL AGENTS ...............................................................................................................11

2 THE SAFIRA INTEGRATION FRAMEWORK ..........................................................................................13

3 SALT & PEPPER COMPONENTS .............................................................................................................16

3.1 SALT & PEPPER ARCHITECTURE .......................................................................................................16
3.1.1 Deliberative vs. Automatic Control .................................................................................................18
3.1.2 Emotion Flow in Salt & Pepper .......................................................................................................20
3.1.3 The Salt & Pepper Memory System ..................................................................................................21
3.1.3.1 Functioning of long-term memory ...............................................................................................21
3.1.3.2 Episodic Memory .........................................................................................................................29
3.1.3.3 Two-level working memory ........................................................................................................30
3.1.4 The Cognitive Engine .....................................................................................................................32
3.1.4.1 Architecture and implementation of the Cognitive Engine .........................................................33
3.1.4.2 Configuration Parameters .........................................................................................................48

3.2 EMOTIONAL PRODUCTION SYSTEM COMPONENT .........................................................................50
3.2.1 Algorithm and Implementation ......................................................................................................55
3.2.2 Built-in Actions ...............................................................................................................................60
3.2.3 Running the Emotional Production System ...................................................................................62
3.2.4 Specification of the long-term memory contents ...........................................................................63

3.3 THE APPRAISAL COMPONENT ..........................................................................................................66
3.3.1 The Appraisal Component Architecture .......................................................................................67
3.3.2 The Component Compiler .............................................................................................................69
3.3.3 Built In Appraisal Modules ..........................................................................................................71

3.4 SCHEMA CLASS BUILDER ...................................................................................................................72
3.4.1 Conversion Model ...........................................................................................................................74
3.4.1.1 From schema to classes ..............................................................................................................74
3.4.1.2 From XML types and type contents to classes ............................................................................75
3.4.1.3 From schema root elements to classes ......................................................................................79
3.4.2 Implementation issues ....................................................................................................................81
3.4.3 Using the generated classes ...........................................................................................................82

3.5 THE INTERACTION MANAGER ........................................................................................................84
3.5.1 Overview of the Interaction Manager ...........................................................................................84
3.5.2 The MessageBox ............................................................................................................................85
3.5.2.1 Handling methods .......................................................................................................................85
3.5.2.2 Messaging methods ....................................................................................................................86
3.5.2.3 Ontology methods .....................................................................................................................86
3.5.2.4 Handling methods .....................................................................................................................87
3.5.3 The Interaction Module ..................................................................................................................88
3.5.4 The Server Socket Listener ..........................................................................................................90
3.5.5 The Interaction Receiver ...............................................................................................................91
3.5.6 Interaction Sender ........................................................................................................................91
3.5.7 The User Interface .........................................................................................................................92
3.5.8 All together now ............................................................................................................................95

4 EMOTION EVOLUTION COMPONENT .................................................................................................103

4.1 BACKGROUND ....................................................................................................................................103
4.2 THE DEVELOPMENTAL GRAPH .......................................................................................................104
4.3 THE DEVELOPMENTAL ALGORITHM ..............................................................................................106
4.3.1 Data Structure ................................................................................................................................106
4.3.2 Algorithm ......................................................................................................................................107
4.4. IMPLEMENTATION .......................................................................................................................... 108
  4.4.1 Developmental Graph ................................................................................................................ 108
  4.4.1.1 Example .................................................................................................................................. 109
  4.4.2 Developmental Model ................................................................................................................. 109
  4.4.2.1 Output .................................................................................................................................... 110
  4.4.2.2 Example .................................................................................................................................. 111

5 EXPERIMENTATION, DISCUSSION AND CONCLUSIONS ............................................................. 114

6 REFERENCES ........................................................................................................................................... 120
1 SHELL FOR EMOTIONAL AGENTS

Although, Artificial Intelligence research in emotions is a very recent endeavour, there is already a remarkable interest in this area, which has produced a considerable variety of agents capable of processing emotions and other affective and personality-related phenomena. Given this increasing interest, the SAFIRA project decided to develop a set of software tools that can help agent designers creating emotional agents in a smooth and integrated way.

Basically, agents can be built from software components communicating with each other using a well-defined framework that specifies the format of the messages to be exchanged. The SAFIRA project defined the general Integration Framework, developed some components that can be used to create affective agents, and produced a set of supporting tools that help installing and integrating the components.

In order to avoid over constraining the agent designer, or imposing a particular view of emotions, SAFIRA has decided to build components that can be used with a variety of theoretical frameworks. Although each component reflects a specific view of affective phenomena, it also grants considerable freedom to the agent designer. Moreover, the general component-based framework allows agent designers to create their agents using only those components that comply with their views on affective computing.

One of the software tools developed by the SAFIRA project is an appraisal compiler. This piece of software allows the agent designer to create components that appraise the agent global state (internal state and external environment) and generates emotions when emotion-eliciting conditions are met. When the agent designer uses the appraisal component, he or she commits its agent to the appraisal view of emotion. Implicitly, he or she also chooses to represent emotion eliciting conditions by a set of IF-THEN rules. However, no commitment is made with respect to the conditions that elicit emotion, and no commitment is made with respect to the particular emotions that are generated under specific circumstances. Not even the representation structure of the generated emotion is imposed on the designer. While the Appraisal Compiler thus does not constrain agent designers to particular theories and models there are two main advantages of its use: (i) it converts a given set of rules into an interoperable component that can be integrated with other components; and (ii) it offers the designer some predefined emotion eliciting modules following some well accepted appraisal theories he or she may decide to use and possibly modify. These are the general design principles behind SAFIRA components.

When the agent designer chooses to use the appraisal compiler, he or she must also use or develop some other component that processes the emotions generated by the appraisal components created with the compiler.

When using the SAFIRA general Integration Framework, agent designers are implicitly committed to the view that agents may be composed by several software components, possibly running on different computers, using a pre-defined protocol and some shared resources to get to know each other and a certain message format to communicate with each other. If these reasonably general constraints are acceptable for the agent designer, he or she will gain the possibility to use our components to build emotional agents and to create or use other components as long as they comply with the Integration Framework.
Each application domain and each agent development team will use different sets of components to build their agents. However, it is possible to say that agents may need sensors, effectors, control components, emotion eliciting components, and components determining the developmental evolution of the agent. Sensors and effectors are generally more tied to specific applications. Control components may use a huge array of mechanisms, from hypothetical reasoning to reactive planning. From the point of view of emotions, agent designers may want to process third person emotions (the emotions of the user), or first person emotions (the emotions of the agent). The sum of all these variables leads to a huge range of possibilities that would be impossible to consider in a single and small project. Thus, we have chosen to create only some components that can be used in a large variety of situations.

SAFIRA has developed a set of software tools that facilitate the processing and evolution of emotions. The emotion processing tools include the Appraisal Compiler and the Emotion Production System Shell. The Appraisal Compiler can be used by component designers to create appraisal components. The Emotional Production System Shell can be used by component designers to create emotional agent control components, using production rules. The Appraisal Component and the Emotional Production System component are built on top of the Salt & Pepper Architecture. The Developmental Model component of the SAFIRA toolkit allows for the representation of long-term changes in agent behaviour and functionality, which may be triggered by affective experiences. The Developmental Model uses a simple, flexible representation of development to support components whose functionality needs to change over time based on an agent’s developmental stage.

The SAFIRA Shell for emotional agents is a set of software components developed by the project partners following the defined Integration Framework.

The SAFIRA general Integration Framework is defined in chapter 2. The software tools developed for emotional processing are described in chapter 3. Chapter 1 describes the component for emotion evolution. Chapter 5 discusses the developed components and presents conclusions.
2 THE SAFIRA INTEGRATION FRAMEWORK

Using the SAFIRA component approach, agents are assemblies of several components, possibly running on different computers. An agent may have sensing components, effector components and internal processing components. All of these components may be integrated through the SAFIRA Central Registry Service (CRS), which provides three integration services: a yellow pages service, a brokering service, and an internal message transport service. It is not mandatory that agent components use the CRS services. It is up to component designers to decide which - if any - CRS services they want to use.

The SAFIRA CRS (Central Registry Service) is described in detail in deliverable D2.2. Here, we merely provide the information necessary for an agent designer to understand how components are integrated.

In the SAFIRA Integration Framework, components (including the CRS) may communicate through message passing using TCP sockets, or through Remote Method Invocation (RMI), in which case a component explicitly calls other component methods.

All the components built in the SAFIRA project use the message based communication. The messages exchanged by the several components that make up an agent are XML documents. Component designers must specify the XML Schema of the messages their components use to communicate with other components.

The main idea underlying the SAFIRA Integration Framework is as follows: a component can only receive commands, information, or emotion signals from other components if it explicitly subscribes to or requests them. For instance, an effector cannot receive a command to perform an action if it has not subscribed to that command (or class of commands). This policy enables the components to decide whether or not they want to receive messages from other components. Therefore, communication is limited to a minimum – only the desired communication takes place.

Human beings do not ask the environment to send them visual information. The only control human beings have on receiving visual information is through opening and closing their eyes and by changing the focus of their visual attention. Once persons are looking at a certain region of their visual field, they cannot control the information they receive. We say that visual information is not deliberately requested, rather, it is voluntarily provided by the environment.

On the other hand, when students ask questions to their teachers, they are deliberately asking to receive certain types of information.

The distinction between deliberately requested information and voluntarily provided information is modelled in the SAFIRA Integration Framework by the difference between subscribing and requesting information. Once a certain class of information is subscribed to, the component will receive it without any further control. On the other hand, if information is received only when the component requests it, the component controls when it wants to receive it.
In the SAFIRA Integration Framework, subscribed to commands, information types and emotion signals are received without the control of the receiver – they are taken to have been volunteered by the providers; requested information is interpreted to be deliberately received. Due to their nature, it is not possible to request a command or an emotion signal. Commands and emotion signals can only be subscribed to.

Whenever a component A subscribes to information from a component B, it creates a thread to handle the information subsequently received. Whenever a component A sends a query to a component B, it creates a thread to handle the answer to be received from B. Using this mechanism, component A will always know whether the received information was sent as a result of a subscription or as a request to perform an informative action (e.g., a query). Consequently, the component can treat the received information differently, according to whether it was explicitly requested or volunteered upon initiative of the sender.

When a component A sends an information-subscription to a component B, it uses (one of the) server sockets of B. This server socket of B creates a private channel for A and B to talk to each other.

B sends the requested information through that channel; therefore, subscriptions and queries do not have to specify a port for reception of the desired information.

Components that want other agents to know the services they provide, the information they produce, or the emotion signals they generate, must register them with the CRS.

When a component wants to send a message to another component, it can send the message itself or it can use the internal messaging service provided by the CRS. Using this service, the communication between components is both easier and more robust. Using this service, the components do not have to ensure the delivery of the messages they want to send to other components. They just have to pass the desired messages to the CRS. It is the responsibility of the CRS to actually deliver the message. For instance, if the destination component is temporarily unavailable, it is the job of the CRS to retry to send the message later. The main drawback of using the internal messaging service is the possibility of having intense message traffic centralised in a single transport medium.

The Yellow Pages service is used by components much in the same way as people use the yellow pages phone directory. It is assumed that publicly available services and information types are registered in the CRS. A component may query the CRS for the identities and the ports of the components that yield certain classes of commands, or provide certain types of information, or generate specific emotion signals. Once the CRS replies, informing the requesting component of the identities of the provider components, it is up to the requesting component to subscribe or request the desired commands, information types and emotion signals from their providers. The requesting component can directly send the subscription or the request to the provider component. Alternatively, it can use the CRS internal messaging service described above.

Using the CRS brokering service, a component does not even need to ask for the identities of the providers. The component just asks the CRS to subscribe or request the desired service from a fitting provider. Using this service, the messages sent by the provider to the requester are also sent to the CRS, which is responsible for handing them to the requester.
The CRS enables a high level of flexibility in agent composition and architecture, depending on which components are available at a given moment. For example, if the agent needs visual information, it may have access to a generic visual sensor or it may use two more specific visual sensors.

The message transport service provided by the CRS ensures more robust internal messaging because, if the recipient component is temporarily not available, the CRS will keep on trying to deliver the message. Therefore, fewer messages will fail to reach their recipients.

In spite of the central services played by the CRS, it cannot be considered a centralised control system. First of all, the CRS does not control anything, it just provides information about available components, and distributes messages among them. Secondly, the CRS is just another component; it is not even a mandatory component.

An important aspect of agent architectures is agent control. Several issues may be analysed in terms of agent control:

(i) Is there a single mechanism that is responsible for giving the orders?

(ii) Is there a unique locus of control?

(iii) What happens if an effector receives two contradictory orders from two different agent control components?

(iv) Is there a meta-control system responsible for deciding which control component takes control over another one?

(v) Is there a centralised storage device where all relevant information is kept and shared among components?

The SAFIRA Integration Framework does not dictate unique answers to the above questions. The SAFIRA framework allows agent designers to choose the answers they want to each of the above questions.

Using the SAFIRA framework, an agent may have a centralised view of control as well as a centralised view of information storage. An agent designer may choose to build an agent with a single agent control component and a single central storage device. A SAFIRA agent may also have several control components and several storage devices. Each control component may have its own storage, and a single storage device may be shared by several components. In case an agent has several control components, it is possible to have a meta-control component that decides which control component imparts the orders, but it is also possible to implement SAFIRA agents without such centralised meta-control mechanisms.
3 SALT & PEPPER COMPONENTS

The basic goal of this section is to describe two SAFIRA components built on top of the Salt & Pepper architecture for autonomous agents. The section begins by describing the Salt & Pepper architecture and its implementation in Java. Then, it proceeds with the description of the two Salt & Pepper components: the appraisal component and the emotional production system component.

Each of the mentioned components includes a module that manages the component’s interactions with other components, according to the SAFIRA Integration Framework (see section 2), using the yellow pages service of the SAFIRA CRS (Central Registry Service). This module is called the Interaction Manager.

The Interaction Manager uses a binding software tool, named Schema Class Builder, which handles the conversion of XML strings representing received messages into Java objects. The Interaction Manager is described in section 3.5. The Schema Class Builder is described in section 0.

3.1 SALT & PEPPER ARCHITECTURE

Salt & Pepper [Botelho and Coelho 2001] is an architecture for autonomous artificial agents in which behaviour results of the interplay of deliberative controlled processes, emotional processes, and reflex actions. In the scope of the SAFIRA project, the most relevant aspect of Salt & Pepper is the interaction between emotion and cognition, in particular the impact of emotion on cognition and behaviour, and the contribution of cognition to emotion.

In Salt & Pepper, emotion is a process that involves appraisal stages, generation of signals used to regulate the agent’s behaviour, and emotion responses. In the first stage, a set of appraisal structures evaluates the global state of the agent (internal state plus external environment). If certain conditions hold, an emotion signal is generated informing the agent of the result of the appraisal stage. A signal can have positive or negative valence. If the agent’s global state is found to be in conflict with the motives of the agent (e.g., instincts, goals, intentions, values, attitudes), a negative emotion signal is generated. If the agent global state is especially favourable according to the agent motives, a positive signal is generated. The emotion experience comprises not only the emotion signal but also the cognitive and behavioural responses that will be produced. Sometimes there will be several signals and several responses. Among other things, an emotion response may be a reflex action, it may increase or decrease the amount of resources assigned to satisfy a given motive, it may create a new motive.

Signals generated during an appraisal stage are sent to the cognitive and behavioural system of the agent, possibly giving rise to an emotion response. In different situations, the same emotion signal may give rise to different emotion responses, depending on the internal state of the agent. Among other things, an emotion response may be a reflex action, it may increase or decrease the amount of resources assigned to satisfy a given motive, it may create a new motive.
(e.g., a new goal), it may set new criteria for selecting plans (or other methods) to achieve current motives. After an emotion signal has been generated and an emotion-response has been performed, the global state of the agent changes and a new appraisal is done possibly generating a new emotion signal. As sketched in Figure 1, this continuous process mirrors the concept of “feedback loop” in control theory.

A simple metaphor for our model is an operating system, where exceptions play the role of emotion signals, the actions performed by the exception handlers play the role of the emotion responses and the appraisal stage is implemented by all conditional statements in the operating system code that may generate exceptions.

![Figure 1 – The emotion process](image)

The appraisal process is governed by a set of emotion eliciting rules that evaluate any part of the agent’s global state, including its external environment and its internal state. For instance, the appraisal process may evaluate whether the result of a planned action achieved the goals it was planned to achieve. In this case, the agent control process provides the desired expectations of the planned actions to the appraisal process, which compares them with the information it receives from the external environment. If the external environment matches the desired expected results, the appraisal process may generate a positive emotion signal. If the external environment does not match the desired expected results, a negative emotion signal may be generated. In this process, the cognitive process (i.e., the agent deliberative control process) influences emotion generation.

It is possible that the emotion eliciting conditions evaluated by the appraisal process are totally independent of the agent’s deliberative control process. For instance, there might be emotion eliciting rules designed to generate emotion signals at the presence of external stimuli of some predefined class. In this case, the cognitive system is not involved.

When an emotion signal is generated, it is sent to the cognitive and behavioural processes of the agent, influencing the way they work. For instance, an emotion signal may increase the accessibility of some data structure stored in memory (e.g., a goal), improving the likelihood of it being used by the agent’s deliberative control process. In this way, emotion can influence an agent’s deliberative control.

Emotion may directly generate reflex actions not mediated by the agent deliberative control process. For instance, an emotion signal may directly activate the agent’s procedural memory, resulting in the execution of some stored procedure.
The described multiple ways by which the external environment, the internal state, emotion and deliberative control processes interact with each other, determining the agent behaviour, are supported by the Salt & Pepper architecture (Figure 2).

**Figure 2 – The Salt & Pepper Architecture**

In the Salt & Pepper architecture, declarative and procedural knowledge is stored in long-term memory. The agent control processes (deliberative or automatic) acquire external information from the agent input buffers and fetch declarative knowledge and procedures from long-term memory in order to determine the agent’s behaviour. Information, knowledge and procedures used by the agent control processes are temporarily stored in working memory. Occasionally, the agent’s control process (deliberative or automatic) may create and store new knowledge structures in long-term memory.

Long-term memory is not a mere repository of declarative and procedural knowledge. Long-term memory is an active device. Its interaction with external stimuli and with the emotion signals generated by appraisal processes determines the relative accessibility of the stored data structures in memory. If a certain data structure, representing declarative or procedural knowledge, becomes more accessible in long-term memory, its likelihood of being used by the agent control process increases. Hence, more accessible stored structures are more likely to influence the agent behaviour than less accessible ones. This property is responsible for some of the most important characteristics of the Salt & Pepper architecture.

When a long-term memory data structure N2 is fetched to working memory while another long-term memory data structure N1 is already present in working memory, a new association from N1 to N2 is formed or the strength of an already existing one is increased. The dynamic formation and strengthening of associations among the long-term memory data structures is responsible for other important properties of the Salt & Pepper architecture.

### 3.1.1 Deliberative vs. Automatic Control

In the Salt & Pepper architecture, agent behaviour may be determined by deliberative (volitional) processes and by automatic non-deliberative processes. Agent behaviour is controlled by the
contents of working memory, which may have been deliberatively or automatically acquired from the external environment or fetched from long-term memory.

The long-term memory monitor is of key importance in the agent control process. The long-term memory monitor observes the accessibility of data structures stored in long-term memory. When the accessibility of a long-term memory data structure becomes high enough, the monitor proposes that data structure to be processed in working memory. If interruptions are currently allowed, the proposed long-term data structure is copied to and processed in working memory, determining the agent behaviour.

Deliberative control is exerted by processes that explicitly select actions to be executed by the agent. These processes may decide to read information from the input buffer, to fetch declarative and procedural knowledge from long-term memory and to create new data structures to be stored in long-term memory. Deliberative control may be implemented by control algorithms such as production systems and planning algorithms. While a deliberative control process is operating, it may happen that a highly accessible long-term memory data structure is automatically copied to and processed in working memory resulting in the interruption of the deliberative control.

It is the responsibility of the control process that actually controls the agent’s behaviour to enable and disable interruptions. This mechanism has been inspired by the inhibition/suppression control mechanism of the subsumption architecture [Brooks 1991].

Another important although less direct role in agent control is played by emotion signals and by external stimuli. Incoming external stimuli and generated emotion signals are always presented to long-term memory. Long-term memory data structures matching the presented stimulus or emotion signal become more accessible. When a node matches an emotion signal, its accessibility increases proportionally to the intensity of the emotion signal. When a node matches an external stimulus, its accessibility increases by a fixed predefined quantity. Since more accessible long-term memory data structures are more likely to be recruited to and processed in working memory, agent behaviour is largely situated in its environment and dependent on generated emotion. It is even possible that presented stimuli or emotion signals directly control the agent’s behaviour. If, due to the presentation of an external stimulus or an emotion signal, a long-term memory data structure containing an action command becomes so accessible that it is recruited to and processed in working memory, the agent’s behaviour will be shaped by this recruited action command. In such cases, the environment or emotion directly control agent behaviour.

When a node interrupts current processing in working memory, its contents must be processed. In order to simplify the explanation, we assume in the following that the interrupting node contains the specification of an action to be executed.

Action specifications may contain arguments that are not instantiated at the time of invocation, therefore making it impossible for the action to be executed. For instance, the action \texttt{step\_north(CurrentPosition)} cannot be executed if the argument \texttt{CurrentPosition} is not instantiated: the system would not know what to do. However, if the agent knows the value of its current position at the time the node was found to match a given stimulus, the action may be executed.
If the interrupting node was activated due to an external stimulus, its contents may be processed only if the external stimulus is still in the input buffer. If the stimulus is not in the input buffer, it is heuristically taken to mean that the context as changed so much that the reason for the node being activated no longer applies.

If the interrupting node was activated due to an emotion signal, its contents may be processed only if the emotion signal was not generated too long ago, according to some criteria that can be specified to the system (notice that emotion signals are not stored anywhere – they are generated, they produce changes in long-term memory, and they are discarded). If the emotion signal has been generated too long ago, we take it to mean that it is no longer relevant for the agent.

Given the described problems (missing information, and long elapsed times), we have decided to store information about the stimulus that matched the contents of matched nodes. Matching information is stored together with each matched clause in the node. Only the information pertaining to the last matching episode of each clause is stored. When a clause matches a certain stimulus, the type of the stimulus is stored together with the matched clause. This type may be either an external stimulus or an emotion signal. If the matching stimulus is an external stimulus, the stimulus itself is stored together with the matched clause, so that it may be checked if it is still in the input buffer before the clause is processed. If the matching stimulus is an emotion signal, the instant of time at which the matching has occurred is stored together with the matched clause.

### 3.1.2 Emotion Flow in Salt & Pepper

In the present framework, emotions are described by a set of parameters:

(i) an emotion signal (S) representing the emotion label (E) and a list of instantiated arguments - the object of the appraisal, the urgency of the repair process (i.e., urgent or not urgent) and the valence which may be negative, positive or neutral (e.g., `attention_shift_warning (object: external_environment, urgency: urgent, valence: negative)`);

(ii) an intensity (I); and

(iii) an emotion response (R) that is performed only if a long-term memory node matching the emotion signal is selected and processed in working memory.

Thus, an emotion is represented by a triplet \(<S,I,P,R>\), in which the signal (S) is itself a four-tuple composed of label, object of appraisal, urgency, and valence: \(<E,O,U,V>\).

In the Salt & Pepper architecture, the appraisal process performs a partial evaluation of the global state of the agent (external and internal). If the eliciting conditions of a particular emotion are met, the appraisal process generates an emotion signal (label, object, urgency and valence) with a given intensity. Long term memory nodes matching the generated emotion signal are activated with an activation level that is a function of the generated emotion intensity (emotion activation function).

---

1 Actually, our current implementation stores an array containing the description of the N last matches, but we only use the last one.
As a result of this activation, it is possible that one of the matching nodes becomes so accessible that the Interrupt Manager proposes it to interrupt the current process in working memory. Highly accessible nodes can only interrupt the current processing if interruptions are enabled, which can be set by any control process. If the node actually interrupts working memory, its contents are processed. In case the interrupting node contains actions, they are executed, forming the emotion response. After an interruption has been processed, the interrupted process is resumed.

The exact response of the agent to a partial evaluation of the global state depends on several factors, such as (i) the pattern of activation of all nodes in long-term memory; (ii) the responses specified in the nodes that represent emotions; and (iii) the available plans that can be used to satisfy possibly created motives.

This context-dependent mechanism suffices to explain Sloman's proposal [Sloman 1987], according to which the same evaluation can generate hate towards someone, but it can also generate disappointment with someone, depending on the motives that are created during the emotion process. According to Sloman, hate will appear only if retributive motives are generated. In our framework, the motives involved in a particular instance of an emotion lead to the execution of the response specified in the node selected to working memory when the signal generated in the appraisal stage activates long-term memory.

When a received emotion signal is a performance evaluation signal, the association between the most strongly associated nodes whose contents are currently being processed in working memory is rewarded (i.e., it is incremented) if the emotion signal has positive valence, and punished (i.e., decremented) if the emotion signal has negative valence. Since the associations between nodes influence their accessibility in long-term memory, this process is yet another automatic mechanism by which emotion conditions the agent’s cognition and behaviour (cf. [Rolls 1999]).

Since input buffers have limited capacity, external information may be lost. One of the roles of the agent's appraisal components is to generate emotion signals that prevent the agent from losing important external information.

### 3.1.3 The Salt & Pepper Memory System

The Salt & Pepper memory system consists of working memory, long-term memory and input/output buffers. Working memory directly controls the agent behaviour. The input buffer and long-term memory are the knowledge and information sources of working memory. Input buffer and working memory contents, and generated emotion signals control the accessibility of long-term memory data structures. Working memory contents determine the formation and strength of associations in long-term memory.

#### 3.1.3.1 Functioning of long-term memory

Long-term memory is an associative network with spreading activation, supported by a directed graph in which the nodes contain long-term data structures and the arcs represent sub-symbolic associations between nodes.
Long-term memory nodes may contain any kind of knowledge, ranging from episodic memories to general knowledge, and including procedures. Long-term memory is not merely a flat static storage device. It is a situated dynamic structured device in which node accessibility reflects the past interactions of the agent with other agents and the environment.

Each long-term memory node has a certain amount of activation. The activation of long-term memory nodes is a measure of their accessibility. More activated nodes are more accessible than less activated ones. An important rule governing the nodes’ accessibility is the exponential decay of activation. According to this rule, the activation of nodes decays exponentially over time. The net result is that accessibility differences between nodes in long-term memory tend to fade. After some time, the accessibility difference between any two nodes can easily be superseded if one of them is stimulated. However, in the absence of stimulation, the most activated node will always remain the most activated, irrespective of the time elapsed since the last stimulation.

Associations between nodes are created or strengthened when nodes are selected to working memory. When a node B is selected to working memory while node A is still present there, the association from A to B is strengthened or created.

Salt & Pepper has the important property of context dependent behaviour. The history of the interactions of the agent determines the accessibility of nodes in long-term memory and the strength of the associations between them. Therefore, it is very likely that, in each context, more adequate cognitive structures are more accessible in long-term memory. This means the behaviour of the agent is likely to be appropriate to each context.

Salt & Pepper long-term memory was originally defined and implemented in Prolog in 1995 [Botelho and Coelho 1995]. Since then, it has been re-implemented and re-defined in several occasions. The SAFIRA project started with the re-implementation in Java of the original Salt & Pepper memory system. Now, near the end of the project, the Salt & Pepper memory system was re-defined and re-implemented in Java. Two alternative designs co-exist in the same implementation, which are different from the original design: Alternative Design I, and Alternative Design II.

The three memory designs diverge one from the others with respect to the mechanisms governing the activation of nodes, the associations between nodes and the way nodes are sought in long-term memory.

**The Original Salt & Pepper Memory System**

Suppose $N(t, A)$ is the number of times that, up to instant $t$, nodes were selected to working memory while node $A$ was already staying there; and $N_B(t, A)$ is the number of times that, at instant $t$, $B$ has been selected to working memory, while $A$ is staying there. Then, the strength of the association from $A$ to $B$, at instant $t$, is defined to be $S(t, A, B) = N_B(t, A)/N(t, A)$. An important aspect of this association formation rule is that an association between two nodes may be formed even when the contents of the two nodes do not overlap. An association from node $A$ to node $B$ represents the likelihood that node $B$ is used after node $A$ has also been used.

When node $A$ receives some amount of activation, that amount of activation spreads to the nodes to which $A$ is associated. Suppose that, at time $t$, node $A$, which is associated to node
B with association strength \( S(t, A, B) \), receives activation \( \Delta(t) \). The activation that spreads through the association from A to B is proportional to the product of \( \Delta(t) \) and \( S(t, A, B) \):

\[
\alpha_B(t+1) = \alpha_B(t) + \rho \sum \Delta_I(t) S(t, I, B) \]

in which \( \alpha_B(t) \) is the activation of node I at instant of time t, I is any node associated to B with strength \( S(t, I, B) \), \( \Delta_I(t) \) is the amount of activation received by node I at instant of time t, and \( \rho \) is a loss factor used so that the spreading of activation stabilizes more quickly. Therefore, the associations’ strengths also determine the amount of activation received by one node to which other nodes are associated.

Nodes receive activation for four reasons: (i) activation spreads from other nodes associated to it; (ii) the content of the node matches some external stimulus placed in the agent input buffer; (iii) the content of the node matches an emotion signal generated by the agent (for more on this see [Botelho and Coelho 2001]); and (iv) the content of the node is being processed in working memory.

When a node receives an amount of activation, the system must propagate that activation to other nodes to which it is associated, and from these to others. This process results in a large amount of computation.

When a node is sought in long-term memory, the most accessible node that matches the requirements is retrieved. In order to achieve this effect, nodes must be sorted by activation order. However, since the activation of nodes is constantly changing, the system must constantly sort nodes. This is an extremely hard computational problem, which has motivated the development of two alternative designs that could preserve the most important properties of the original architecture but could reduce computational complexity.

**Alternative Designs**

The desired properties of the original Salt & Pepper stem from the concepts of association and activation.

The frequency effect arises because more frequently used nodes become more associated in the network than less frequently used nodes. If a node is used often, it is selected to working memory after many other nodes therefore many associations will form from many other nodes to it. Whenever these other nodes receive activation, it will spread to the frequently used node increasing its own activation, which makes it more accessible, therefore easier to find.

The recency effect has to do with activation alone. When a node is used, its activation increases therefore its accessibility improves. When memory is sought again, the node will be found earlier.

Since the desired properties of the original design are derived from the concepts of association and activation, both of the new Salt & Pepper long-term memory designs have counterpart concepts of association and activation.

The two alternative long-term memory designs described in this section use a common data-structure to hold long-term memory. The main differences between them relate to the way activation spreads from one node to the others in the network. In the Alternative Design I, spreading activation is recursively computed whenever nodes receive activation. In the Alternative Design II, activation has two factors: an explicit factor and an implicit factor. The
activation explicit factor spreads through the network but through a lazy evaluation strategy (instead of the eager evaluation strategy of the original model). The implicit activation factor does not spread explicitly: it is an emergent property of the way nodes are searched in long-term memory.

The basic ideas of the new design are as follows:

(i) All long-term memory nodes are stored in a special purpose data structure.

(ii) Explicitly changing one node's activation results in changing its place in the memory data structure.

(iii) Changing the association strength from one node (N1) to another node (N2) results in changing N2's placement in the memory data structure.

(iv) The memory data structure searching algorithm is designed so that the order by which nodes are visited in the data structure ensures that most activated nodes are found first.

(v) The number of nodes to which a particular node is associated is limited.

The long-term memory data structure is a ladder where nodes are placed. The top of the ladder corresponds to the most activated node. Each node in the ladder has an association structure representing the nodes to which it is associated and corresponding association strengths.

A node's association structure is a list of pointers to the nodes to which it is associated. The strength of the association from one node (N1) to another node (N2) relates to the position of N2 in the association structure of N1.

![Figure 3 – Long-term memory data structure](image)

In the long-term memory data structure presented in Figure 3, the most activated node is A (it is placed on top of the memory ladder), and the least activated node is E. B, C and D have intermediate activation values. Node A is associated to B, to E, and to C. The association from A to B is stronger than the association from A to E, which is stronger than the association from A to C. Node B is associated to C, and C is associated to E. D is not associated to other nodes.

---

D-SAFIRA-WP4-D4.2 © IST Project IST-1999-11683 SAFIRA 24
Finally, node E is associated to D with a stronger association than to C. Each node may be associated to at most N other nodes.

When a node receives a certain amount of activation, it climbs the memory ladder, possibly overtaking other nodes in the memory ladder.

When the association from a given node (N1) to another node (N2) increases three things may happen. If N1 is not already associated to N2 and N1's association structure is not full, a pointer to N2 is placed in the last available position of N1's association structure (see also the explanation about association strength, below). If N1 is not already associated to N2 and N1's association structure is full, a pointer to N2 will replace the last element of the association structure. If N1 is already associated to N2, then the pointer to N2 may get closer to the beginning of N1's association structure possibly overtaking other elements.

**Searching long-term memory**

With this alternative algorithm, long-term memory search operations are much more efficient because the search algorithm does not have to sort nodes by activation. It just has to traverse the long-term memory data structure in a given sequential order. The designed traversal order should ensure that more activated nodes are visited earlier than less activated ones. The exact searching algorithm depends on the particular design.

**Long-term memory monitoring**

Long-term memory monitoring is much simplified. All that is needed is to determine the activation of the node placed at the top of the ladder and compare it with the given threshold (actually, a measure of the activation of the nodes whose contents are currently being processed in working memory).

**Activation decay**

What matters, with respect to activation decay, is not the absolute activation of a given node but the activation gap between it and the other nodes in the memory ladder. Activation decay is computed by decreasing the activation gaps between nodes in the memory ladder.

**Association strength**

Association strength is no longer a relative frequency; therefore, no large numbers have to be maintained. The strength of the association from node N1 to node N2 is just a function of the distance from N2 to the beginning of the association structure of N1. Each node may be associated only to a limited number of other nodes.

**Access by name**

Internally, nodes have identifiers (assigned when they are created). Sometimes, it is useful to search nodes by name therefore we have developed a parallel access structure to better access nodes by name. That structure uses a hash table.
Alternative Design I

Most of the properties of Alternative Design I arise from the long-term memory data-structure just described. It differs from Alternative Design II with respect to the way activation spreads from one node to the others, and with respect to the long-term memory searching-algorithm.

Spreading activation

When a node associated to other nodes receives activation, the activation spreads to the nodes to which it is associated. Those, in turn, are associated to other nodes. Therefore, the algorithm must recursively compute the spreading of activation. Whenever a node receives activation it climbs the memory ladder. Possibly, it will overtake other nodes.

The major inefficiency of the original design preserved by Alternative Design I is the computation of the spreading of activation.

Searching long-term memory

The memory ladder contains all memory nodes sorted by activation. The top node is the most activated one. This means the searching algorithm of Alternative Design I just has to look for nodes on the ladder from top to bottom. This is a searching algorithm linear in the number of nodes in long-term memory.

Summary

Alternative Design I significantly improves three of the four sources of inefficiency of the original design: long-term memory search; long-term memory monitoring for interrupt managing; and association strength computation. There is also a minor improvement with respect to spreading activation: each node may be associated only to a maximum number of other nodes, which reduces spreading activation. Besides, there is an architecture parameter that allows to constrain the depth of activation spreading.

The major problem that must be handled in alternative designs is the spreading of activation from one node to the rest of the network through the associations between nodes.

In spite of exhibiting qualitatively similar properties to the original design, this new design is not exactly the same model:

(i) Association strength is no longer a relative frequency.

(ii) Each node can only be associated to a limited number of other nodes.

Alternative Design II

The major drawback of Alternative Design I is the computation of spreading activation. It is still computed in the same way as in the original design. In order to overtake this difficulty, we propose a different approach.

Since each element of an association list of a node refers to the corresponding node in the memory ladder, it is possible to access the nodes to which that node is in turn associated.
When a node receives some activation (i.e., when a node climbs the memory ladder) the whole sub-network rooted in that node should also become more activated. Therefore, if the memory search algorithm takes this into account, we may reduce the computation involved in the process of spreading activation.

Using the original concept of activation, we were not capable of designing a searching algorithm that could traverse the long-term memory data-structure by activation order, because in Alternative Design II, the activation of each node is not always known. Therefore, Alternative Design II presents local properties that differ considerably from the original design. In Alternative Design II, the major factor determining the node's accessibility is other nodes’ association to it, not the node activation.

In Alternative Design II, activation is defined as the accessibility of the node as given by the designed searching algorithm, whereas in the original architecture and in Alternative Design I, accessibility is defined as the activation of the node.

**Spreading activation**

In Alternative Design II, when a node associated to other nodes receives activation, it climbs the memory ladder possibly overcoming other nodes, but the activation does not spread to the rest of the network. Instead, the received activation is made available for each of the associations getting out of the node, proportionally to the association strength. This is not the only effect of spreading activation; the remaining effects emerge of the long-term memory searching-algorithm.

This spreading activation algorithm is much more efficient (fast) than the recursive spreading activation algorithm of the original architecture.

**Searching long-term memory**

Fundamental ideas guiding the design of the searching algorithm:

(i) All other things being equal, a node to which a highly activated node is strongly associated should be more accessible than a node to which only weakly activated nodes are associated. Therefore the searching algorithm visits first the association list of more activated nodes.

(ii) All other things being equal, a node to which a highly activated node is more strongly associated should be more accessible than a node to which a highly activated node is less strongly associated. Therefore, when searching the associations list of a node, the algorithm visits strongly associated nodes before weakly associated nodes.

(iii) All other things being equal, a node to which there are many highly activated nodes associated should be more accessible than a node to which there are few highly activated nodes associated. In order to accommodate this intuition, the algorithm searches only a few nodes in the association list of each node and then, even before the whole list has been visited, the algorithm switches to the next node in the memory ladder and to its association list. This process increases the likeability of finding nodes to which many other nodes are associated.
(iv) When a node is visited by searching a certain path, the activation contribution of that path is explicitly propagated to that node, changing its position along the memory ladder. Available (not yet propagated) activation is kept in the node data structure.

Following the ideas expressed above, we have designed the long-term memory searching algorithm informally presented in Figure 4.

\begin{verbatim}
long term memory search (ltm, pattern) % memory ladder and sought pattern

1. Reset the next-node pointer in the association-list of each node in the long-term memory

/* The search starts with the node at the top of the memory ladder (Figure 3) */

2. \(\eta = \text{ltm}[0]; \ j = 1; \ \text{end\_search} = \text{false}; \ \text{last\_node} = 1;\)

/* If \(\eta\) is the desired node, stop the searching algorithm, reset the next-node pointer in the association-list of each node in the long-term memory, return \(\eta\) */

3. If ( match(\(\eta\), pattern) ) end\_search=true;

4. if( end\_search or end of memory ){
   a) reset the next-node pointer in the association-list of each node in the long-term memory: node.next = node
   b) if( end\_search ) return \(\eta\); else return NULL;
}

/* If \(\eta\) is not the desired node then ... */

/* Difference of activation between node \(\eta\) and the next node in the memory ladder */

5. if( \(j = \text{memory size}\) ) AGap=0; else AGap=\(\alpha(\eta) - \alpha(\text{ltm}[j])\);

/* Search the N next nodes to which \(\eta\) is associated. N is a function of AGap. Update the pointer to the next node in the list of nodes to which \(\eta\) is associated. */

Update the activations of all visited nodes along the searched path */

6. \(n=0;\)

7. while(\(n \leq f(AGap)\) and not end\_associationList(\(\eta\))){
   a) \(\mu=\eta_\.\text{associationList}[\eta\.\text{next}];\)
   b) update activation of \(\mu\)
   c) if( match(\(\mu\), pattern) ){
      end\_search=true; \(\eta=\mu;\)
      break;
   }
   d) \(\eta\.\text{next}=\eta\.\text{next}+1; \ n=n+1;\}

8. if( not end\_search ){  
   a) \(\eta=\text{ltm}[j];\)

      /* Compute new memory indexes, following the pattern 0, 1; 0, 1, 2; 0, 1, 2, 3; ... */

      b) if( last\_node == \text{memory size} ) last\_node=1;
      c) if( \(j > \text{last\_node}\) ){
         last\_node = last\_node + 1; \(j=1;\)
      }
      else \(j=j+1;\)
   }

9. goto 4;
\end{verbatim}

\textbf{Figure 4. Alternative Design II Searching Algorithm}
Using this algorithm, long-term memory nodes and their association lists are searched as in the following pattern: node\textsubscript{0}, node\textsubscript{1}; node\textsubscript{0}, node\textsubscript{1}, node\textsubscript{2}; node\textsubscript{0}, node\textsubscript{1}, node\textsubscript{2}, node\textsubscript{3}; etc. node\textsubscript{0} represents the top-level node in the memory ladder and the nodes to which it is associated, node\textsubscript{1} represents the second node in the memory ladder and the nodes to which it is associated, and so on.

**Summary**

Alternative Design II significantly improves the four sources of inefficiency of the original design: recursive spreading activation computation; long-term memory search; long-term memory monitoring for interrupt managing; and association strength computation. However, the Alternative Design II is more different from the original Salt & Pepper than Alternative Design I.

(i) Association strength is no longer a relative frequency.

(ii) Each node can only be associated to a limited number of nodes.

(iii) Activation is defined in terms of node accessibility as given by the searching algorithm.

(iv) The explicit component of activation spreads through memory in a lazy evaluation manner, instead of an eager evaluation approach as in the original model.

(v) The implicit / emergent component of activation is not additive, in the sense that the several contributions for a node’s activation do not sum up.

Quantitatively, the Alternative Design II preserves the same fundamental properties of the original design: the frequency and the recency effect. These properties ensure that the Alternative Design still exhibits context-dependent (situated) behaviour.

### 3.1.3.2 Episodic Memory

Episodic memory consists of the set of episodic nodes in long-term memory. Episodic nodes represent specific episodes of the agent interaction.

Some working memory processes create new information structures or modify previously existing ones. For instance:

- Facts tracking intermediate agent positions along its trajectory from the initial to the goal position;
- Successive explicit representations of partial plans;
- Increasingly more complete maps of the external environment;
- Partial results of complex computations.

In most cases (e.g., intermediate agent positions, partial computations) such information structures can perfectly well be discarded after being used. On some occasions, they could be stored, possibly together with contextual information (e.g., partial plans). There are even cases
of context independent useful information resulting from working memory computations (e.g., acquired maps of the external environment). One and the same kind of information should be memorised in some cases and discarded under some other circumstances.

The general problem is (i) to decide when produced information should be memorised; (ii) to decide when produced information should replace previous versions; and (iii) how to design a systematic mechanism capable of discarding useless temporary information.

Episodic information is information about specific states of the agent’s interaction. It pertains only to a certain state of the interaction. Examples are the position of the agent in a given instant of time, or the number of parked vehicles in a given instant of time.

In this new version of the Salt & Pepper, relevant episodic information is stored in episodic nodes in long-term memory. All new information contained in working memory when a strong emotion signal is generated is stored in an episodic node together with the generated emotion signal, the goal being tried at that time, and the instant of time in which the signal was generated.

There are two alternatives with respect to the format of episodic nodes.

A. Each episodic node is composed by a set of episodic clauses. An episodic clause is represented by an episode/4 symbolic structure: episode(time-stamp, goal, emotion signal, clause).

B. Each episodic node has only a single episodic clause, which is represented by an episode/4 symbolic structure: episode(time-stamp, goal, emotion signal, clause-set).

Since for pattern matching purposes alternative A is more tractable and more flexible than alternative B, we have decided to pursue alternative A. For the time being, episodic nodes are being used by a learning algorithm that creates conditioned action chunks. This algorithm is not yet built into the Salt & Pepper software.

3.1.3.3 Two-level working memory

Besides being a storage system, working memory is the process that controls the component. In the case of the Em-PSys, working memory includes the production rule inference process and the interruption management process.

The set of rule nodes present in working memory at any given point of time forms the context-dependent knowledge base upon which the control algorithm operates. This context dependency results from the fact that selection of rule nodes from long-term memory to working memory is based on individual node accessibility which depends on node activation and the agents' motivational state.

Since the current implementation of the Salt & Pepper does not use the motivation strength to determine the number of nodes that are fetched to working memory, all rule nodes are unconditionally retrieved to working memory and made accessible to the control algorithm (not only the most accessible ones). As a consequence, these retrievals from long-term memory to
working-memory cannot be used as condition for updating of node activation and strengthening of node associations, as normally performed in the Salt & Pepper paradigm. In order to put remedy to this issue and achieve context dependent cognition with the currently employed production rule inference process, it was decided that only nodes containing currently satisfied rules should be activated and their associations increased.

To this end, two alternative solutions were considered:

A. Organising working memory in two levels: evaluation memory and use memory. Only the nodes containing information in the use memory are activated and their associations increased. Using this approach, the first step of the described algorithm fetches rules to the evaluation memory; the second step selects satisfied rules to the use memory.

B. Collapsing the first two steps of the described algorithm into a single step in which only satisfied rules are fetched to working memory. According to this approach, the fetching operation is an intelligent process capable of determining whether or not the left-hand side of a rule is satisfied.

Since the second alternative assumes the existence of a complex cognitive process that uses working memory facts in order to select nodes from long-term memory, the first alternative was chosen.

To summarise, the production rule system implemented on top of Salt&Pepper which controls working memory currently works as follows:

1. All rule nodes are fetched to evaluation memory.

2. Satisfied rules are selected to use memory, forming the conflict set. The activation and association strengths of the respective nodes are updated.

3. The most activated of the rules in use memory is selected, and its right-hand side is executed.

4. Go to step 1.

Note that the two-level working memory solution lends itself also to other types of control mechanisms. For instance, if a planning algorithm were used in place of the production rule system, all action description nodes would be retrieved to evaluation memory, and goal/sub-goal satisfaction would be used as criterion for selection to use memory as well as updating of node activations and association strengths.

**Garbage collection**

Information to be discarded may be cleaned by a garbage collection process. The simplest garbage collection process deletes temporary information based on temporary information retention time. From an implementation point of view, the agent’s mind has an attribute defining the temporary information retention time. All temporary clauses are stored in special nodes whose
node identifier takes the form \( \text{wm}(\text{time-stamp}) \), which designates the instant of time at which the information was generated in memory. From time to time, the system checks the time-stamps of stored working memory temporary clauses and deletes those that have been stored longer than the temporary information retention time.

The garbage collection algorithm applies only to temporary information memory. Persistent information copied from long-term memory is not discarded.

Examples

```prolog
node_clause(
    \text{wm}(\text{time}(2001, 10, 14, 17, 34, 18, 89)), \% \text{Node identifier}
    \text{position}(\text{dirty_harry}, 23, 125)
).
node_clause(
    \text{wm}(\text{time}(2001, 10, 14, 17, 34, 20, 2))%,
    \text{position}(\text{dirty_harry}, 23, 126)
).
node_clause(
    \text{wm}(\text{time}(2001, 10, 14, 17, 34, 21, 77))%,
    \text{position}(\text{dirty_harry}, 23, 127)
).
```

Other kinds of \textit{node clauses}, that is, \textit{node clauses} whose node identifier is not of the form \textit{wm(\textit{Time})}, are not subjected to garbage collection.

### 3.1.4 The Cognitive Engine

In order to facilitate the creation of cognitive components, we have developed a layer on top of the Salt & Pepper Memory system that implements the cognitive part of the Salt & Pepper architecture. The remaining other parts of the architecture include appraisal components (see section 0), effectors and sensors. They are implemented as independent components that communicate with each other and with the cognitive component(s). Since sensors and effectors are domain dependent components, they are not described in this deliverable. The emotional production system component (Em-PSys, section 3.1.3.2) has been implemented on top of the Cognitive Engine.

The Cognitive Engine is a general-purpose skeleton of Salt & Pepper cognitive components, which contains architecture parameters, long-term memory, working memory, input and output buffers, and implements the basic Salt & Pepper control mechanism described in section 3.1.1. The Cognitive Engine also integrates an Interaction Manager module (see section 3.5) that controls its interaction with other modules.

Agent designers building Salt & Pepper cognitive components may use the Cognitive Engine as their starting point. Software components based on the Cognitive Engine, such as the Em-PSys, must specify the values of the several architectural parameters; they must provide a reasoning mechanism to manage working memory; and they must fill the long-term memory of the component with the knowledge structures actually used in the agent’s reasoning process.
The component must also specify the kind of information, the commands and the emotion signals it needs to receive from other components. Finally, it must also specify the classes of generated outputs that can be used by other components.

3.1.4.1 Architecture and implementation of the Cognitive Engine

The Cognitive Engine includes several modules organised around different tasks: the agent-designer interface; component interaction; long-term memory tasks; and working memory and component control tasks.

The agent designer interface

The agent designer must provide some inputs for components built on top of the Cognitive Engine, including the specification of configuration parameters, the contents of long-term memory, the component-specific hardwired code, and the information required to set up the interaction of the component with other components.

At present, configuration parameters must be set by editing the Java file containing the definition of the Cognitive Engine class.

Long-term memory content is specified through any number of text files with .ltm extension placed in a source directory whose pathname must be specified to the component in the component configuration file, which is a Java code file located in the folder containing the component source. The content of long-term memory consists of a set of long-term memory nodes written in Prolog.

In case agent designers want to include hardwired code in the component, they may do so, using any number of text files with extension .pl to be placed in the same source directory as the .ltm files. Hardwired code must be written in Prolog.

When the Cognitive Engine is created (new CognitiveEngine…), its constructor loads all .ltm and .pl files contained in the specified source directory.

Currently, the data required for setting up the interaction of the component with other components (see the description of the Component Interaction tasks in this section) is hardwired in the Java code of the CognitiveEngine constructor. This will be changed in the future in order to increase the generality of the components built on top of the Cognitive Engine.

Component Interaction

The modules responsible for component interaction are the Interaction Handler, the MessageBox, the input buffer, the output buffers, and the memory-change semaphore.

```
private Semaphore memoryChangeSemaphore;
private MessageBox messageBox;
private InteractionModule interactionHandler;
private RoundRobin input_buffer;
private Vector command_output_buffer;
private Vector information_output_buffer;
```

Code-Block 1 – Interaction Handlers of the Cognitive Engine
memoryChangeSemaphore is a semaphore used by the Cognitive Engine to signal memory changes. This semaphore is very useful for control purposes since there may be processes that depend on memory changes. For instance, the Emotional Production System algorithm (section 3.1.3.2) suspends itself when it cannot find any satisfied rules. Memory changes are relevant events with respect to rule satisfaction, therefore it is necessary to resume the production rule algorithm whenever long-term memory changes occur.

The Message Box is a data structure containing the definitions of all the XML schemas of the messages that can be received or sent by the component, and the information and emotion signal subscriptions to be made by the component.

The Interaction Handler, which is an instance of the Interaction Module class, handles the interaction of any component built on top of the Cognitive Engine with other components. The definitions and settings required by the Interaction Handler are passed to it through the Message Box. The class Interaction Manager is described in detail in section 3.5.

```
this.messageBox.AddSchema(SAFIRADefinition.SAFIRAPath +
"SAFIRA/WP4/Adetti/AgentMind/Schemas/InformationSubscriptionSchema.xsd ");

this.messageBox.SubscribeSchema("EmotionSignal",
this, IMDefinition.NON_PERSISTENT);

this.interactionModule = new InteractionModule(cf.getName(),
cf.getAddress(), cf.getPort(), this.messageBox,
IMDefinition.ASYNCHRONIZED, this.gui);
```

**Code-Block 2 – Interaction related code fragments**

Code-Block 2 shows the definition of a message schema from a XSD file; the definition of an emotion signal subscription to be made by the component; and the initialisation of the Interaction Handler. The method `AddSchema` takes the full pathname of the file containing the Schema definition.

The method `SubscribeSchema` takes the name of the subscribed schema ("EmotionSignal"), the path of the class of the component relative to the class path, the name of the method used to handle the subscribed messages ("handleEmotionSignal"), the reference to the component, and a parameter specifying whether the subscription is persistent.

The constructor of the Interaction Handler receives the name, the address and the port of the component; the message box; the type of the interaction (asynchronous/synchronous); and the graphical user interface.

Components can receive two kinds of messages: subscription/request messages, and subscribed/requested messages. Components built on top of the Cognitive Engine have two methods to handle information and command subscriptions, and two methods to handle received (subscribed) emotion signals and information.

The method `handleCommandSubscription` is passed to the Interaction Handler to send generated commands to their subscribers. When a command is generated by the
component, the method `handleCommandSubscription` gets the command from the command output buffer and sends it to the components that have subscribed it.

The method `handleInformationSubscription` is passed to the Interaction Handler to send generated information to the components that have subscribed it. When information of a certain class is generated by the component, the method `handleInformationSubscription` gets the information from the information output buffer and sends it to the components that have subscribed it.

The method `handleEmotionSignal` is passed to the Interaction Handler to handle received emotion signals. When the Interaction Handler receives an emotion signal it calls the method `handleEmotionSignal` to handle the received signal.

The method `handleInput` is passed to the Interaction Handler to handle received input information. When the Interaction Handler receives some information of any subscribed class, it calls the method `handleInput` to handle the received information.

```java
static public Object handleCommandSubscription(CommandSubscription subscription,
                                              Object actions, KnowledgeBaseListener listener){
    int i;
    if( actions == null ) return null;
    int len = ((Vector)actions).size();
    if( len == 0 ) return null;
    Vector subscribed_actions = new Vector();
    for(i = 0; i < len; i++){
        if( command_subscription_match(((Vector)actions).elementAt(i),
                                         subscription) ){
            subscribed_actions.addElement(((Vector)actions).elementAt(i));
        }
    }
    if( subscribed_actions.size() > 0 )
        return (Object)subscribed_actions;
    else
        return null;
}
```

**Code-Block 3 – Handle subscribed commands**

The method `handleCommandSubscription` loops over all commands in the command output buffer and compares them with the specified subscription. Output command buffer elements matching the specified subscription are added to the command vector to be returned. If there are no commands in the command vector, null is returned.
static public Object handleInformationSubscription(InformationSubscription subscription, Object informs, KnowledgeBaseListener listener){
    int i;
    if( informs == null ) return null;
    int len = ((Vector)informs).size();
    if( len == 0 ) return null;
    Vector subscribed_informs = new Vector();
    for(i = 0; i < len; i++){
        if( information_subscription_match(((Vector)informs).elementAt(i), subscription) )
            subscribed_informs.addElement(((Vector)informs).elementAt(i));
    }
    if( subscribed_informs.size() > 0 )
        return (Object)subscribed_informs;
    else return null;
}

**Code-Block 4- Handling information of subscribed classes**

The method `handleInformationSubscription` loops over all predicate tuples placed in the information output buffer. Tuples matching the information subscription are placed in the information vector to be returned. It returns the information vector or null if there are no tuples in the information vector.
static public Object handleEmotionSignal(EmotionSignal esig, Object kb, KnowledgeBaseListener listener){
    CognitiveEngine ce = (CognitiveEngine) kb;
    Double spreadingDepth = (double) ce.spreadingDepth;
    SPMemoryModule ltm = ce.ltm();
    PRuleControlSystem pRuleControl = ce.pRuleControl();
    StringBuffer emotion_args = null;
    String pattern = null;
    MatchingInformation matching_info = null;
    Node[] association = new Node[2];
    Node from = association[0], to = association[1];

    // Create the pattern to present to LTM: pattern_action(<signal_name>( args ), _)
    // Create the string “(arg1, arg2, ...)”containing the arguments of the emotion signal
    emotion_args = new StringBuffer(JavaPrologInterface.JavaToProlog(esig.signalArguments.s_arg));

    // Present EmotionSignal to LTM
    pattern = "pattern_action(" + esig.signalName + emotion_args + ", ", _).;"
    matching_info = new MatchingInformation(SPDefinition.EMOTION_SIGNAL, (Object)(new Date()));
    ltm.ShowStimulus(pattern, esig.intensity.doubleValue(),
                     (double)spreadingDepth, 1, (Object) matching_info);

    // If performance evaluation signal, update node associations
    if( performanceEvaluationSignal(esig.emotionCategory) ){
        // Get most associated nodes being processed in Working Memory
        if(pRuleControl.mostAssociatedNodes(association)==SPDefinition.ASSOCIATION_FOUND){
            if( esig.valence.booleanValue() ==
                AppDefinition.NEGATIVE_VALENCE.booleanValue())
                ltm.DissociateNodes(from, to, associationPunishment);
            else
                if(esig.valence.booleanValue()==AppDefinition.POSITIVE_VALENCE.booleanValue())
                    ltm.AssociateNodes(from, to, associationReward);
        }
        return null;
    }

    // Code-Block 5 – Handling received emotion signals

    The method handleEmotionSignal handles received emotion signals. It presents the signal to long-term memory. Since emotion responses are represented by pattern-action clauses in long-term memory, the emotion signal E is wrapped in the pattern “pattern-action(E, _)” before it is shown to long-term memory. The method showStimulus of the Salt & Pepper memory (SPMemoryModule) is used to present the stimulus to long-term memory.

    If the signal is a performance-evaluation signal, the method adjusts the association strength between the most associated nodes currently being processed in Working Memory. This is only a heuristic approach assuming that the strongest association between nodes being processed in working memory is the responsible for the identified poor/exceptionally good
performance. A general approach could be to trigger a rationalization process aimed at identifying the responsible association. If the performance evaluation has negative valence, the association strength is decreased. If the valence is positive, the association strength is increased.

The method currently does not signal the memory change semaphore, although it could be argued that it should, since memory accessibility changes and this very motivation could suffice to find something in memory.

```java
static public Object handleInput(Object obj, Object kb){
    CognitiveEngine ce = (CognitiveEngine) kb;
    SPMemory ltm = ce.ltm();
    RoundRobin input_buffer = ce.input_buffer();
    String stimulus;
    Vector patterns = new Vector();
    int i;

    // Create the stimuli to be presented to long term memory
    stimulus = JavaPrologInterface.JavaToProlog(obj);
    patterns.addElement((Object)stimulus);
    patterns.addElement((Object)"pattern_action(" + stimulus + ", ")");

    // Present object to ltm
    for(i = 0; i < patterns.size(); i++)
        ltm.ShowStimulus((String)patterns.elementAt(i), inputActivation, spreadingDepth, 1, obj);
    input_buffer.newElement(obj);
    return null;
}
```

**Code-Block 6 – Handling all subscribed inputs but emotion signals**

The method `handleInput` handles all subscribed inputs but emotion signals. Inputs are just presented to long-term memory. They are not stored in long-term memory or in working memory. It is the responsibility of the agent control process to explicitly read received inputs from the input buffer. As above, this method also currently does not signal the memory change semaphore.

Since a received stimulus may match nodes containing clauses matching it as well as nodes containing pattern-action clauses in which the pattern matches it, this method creates two patterns to present to long-term memory: the stimulus itself and the pattern `pattern-action(S, _)`, in which S is the received stimulus. These two patterns are presented to long-term memory. Actually, the received stimulus may also match the condition of production rules, but it is not possible to check that possibility using the currently supported pattern specifications.

Since any component that may be developed on top of the Cognitive Engine may have to access the contents of the input buffer and may have to place information or commands in the output buffers, it is necessary to provide general methods to access these two kinds of buffers.

The method `nextInputBufferElement` returns a string containing the next element in the component input buffer. If there are no more elements in the input buffer, `nextInputBufferElement` returns null. The Prolog predicate that calls `nextInputBufferElement` must convert the returned String into a Prolog term.
public String nextInputBufferElement(){
    Object element = this.input_buffer().getElement();
    if( element == null ) return null;
    return JavaPrologInterface.JavaToProlog(element);
}

**Code-Block 7 – Reading the next element in the input buffer**

The static method `JavaToProlog` of the class `JavaPrologInterface` converts a Java object to a string that may be converted into a Prolog term. The returned string complies with the Prolog syntax. The functor of the term is the same as the name of the class of the element found in the input buffer. The attributes of the object are converted into the arguments of the predicate.

The class `JavaPrologInterface` contains a set of general-purpose methods that perform conversions between Prolog and Java data types.

The methods `sendActionToOutputBuffer` and `sendExpectationsToOutputBuffer` are used to place commands and information in the output buffers. The predicates `prs_send_action_to_the_effector/3` and `prs_send_expectations_to_the_output_buffer/3` of the Emotional Production System use these methods.

```java
public void sendActionToOutputBuffer(String class_name, SPTerm arguments){
    placeInfoInOutputBuffer(this.command_output_buffer, class_name, arguments);
}

public void sendExpectationsToOutputBuffer(String class_name, SPTerm arguments){
    placeInfoInOutputBuffer(this.information_output_buffer, class_name, arguments);
}
```

**Code-Block 8 – Output buffers access methods**

Both methods in Code-Block 8 are mere interfaces to the general access method `placeInfoInOutputBuffer`, which receives a reference to the output buffer.
public void placeInfoInOutputBuffer(Vector buffer, String class_name, SPTerm arguments){
    SPTerm[] spterms_list = null;
    try{ spterms_list = arguments.toTermArray(); }
    catch(Exception e){ e.printStackTrace();}
    int i = 0;
    int len = spterms_list.length;
    Vector attributes = new Vector(len);
    Object prolog_term;
    try{
        for(; i < len; i++)
            switch(spterms_list[i].type()){
                case SPTerm.SP_TYPE_INTEGER:
                    attributes.add(i, new Integer(((int)spterms_list[i].getInteger())));break;
                case SPTerm.SP_TYPE_FLOAT:
                    attributes.add(i, new Double(spterms_list[i].getDouble()));break;
                case SPTerm.SP_TYPE_ATOM:
                    attributes.add(i, spterms_list[i].getString());break;
                default:
                    attributes.add(i, spterms_list[i].getObject());break;
            }
    }catch(Exception e) { e.printStackTrace();}
    prolog_term =
            class_name, attributes);
    if( prolog_term != null ){
        buffer.addElement(prolog_term);
        this.sendallSignal(buffer);
    }
}

**Code-Block 9 – General access to output buffers**

The method `placeInfoInOutputBuffer` creates a vector containing the Java representations of arguments of the Prolog term that is to be placed in the specified output buffer. It can successfully handle Prolog arguments of types Integer, Float and Atom. Other types are dealt with as general Java objects, but they are not processed.

After the conversion of the term arguments, `placeInfoInOutputBuffer` calls the method `PrologToJava` of the class `JavaPrologInterface` to create a Java object from the functor and the arguments of the Prolog term to be placed in the output buffer.

Finally, `placeInfoInOutputBuffer` places the created object in the specified output buffer and signals the Interaction Manager that a new term was produced. It waits until the Interaction Manager delivers the term to the appropriate component, and then cleans the output buffer and proceeds. These last two steps are performed by the method `sendallSignal`. 
Long-term memory tasks

There are two modules responsible for long-term memory operation in the Cognitive Engine: ltm, which is an instance of the Salt & Pepper long-term memory (SPMemoryModule), and the Activation Decrementer.²

```java
private SPMemoryModule ltm;
private ExponentialDecay activation_decrementer;
```

**Code-Block 10 – Components related to long-term memory**

Code-Block 10 shows the code lines containing the declarations of long-term memory (ltm) and the Activation Decrementer modules of the Cognitive Engine.

In the current implementation, the Activation Decrementer is a thread of the Cognitive Engine that periodically visits all long-term nodes and decrements their activation, following a negative exponential law.

The Cognitive Engine uses several methods of the Salt & Pepper long-term memory (SPMemoryModule), of their nodes, and of the Semaphore class.

**Salt & Pepper Long-Term Memory Methods**

- **LoadFile(Filename)**: Loads a .ltm file containing the specification of long-term memory nodes. It is used in the constructor of the Cognitive Engine to load the files located in the source directory.

- **Memory()**: Returns a data structure containing the set of nodes in long-term memory. It is used by the Activation Decrementer to visit all nodes in long-term memory.

- **GetNextUnloadedNode()**: Returns the next node that has not yet been loaded in long-term memory.

- **GetNumberOfNodes()**: Returns the number of nodes in long-term memory.

- **ShowStimulus(pattern, inputActivation, spreadingDepth, returnType, matchingInfo)**: Presents a pattern to long-term memory. All nodes that match the presented pattern receive an amount of activation equal to the value of `inputActivation`. The activation received by the nodes spreads to other nodes through the node’s associations. The activation is allowed to spread the number of levels set by the `spreadingDepth` parameter.

  The parameter `returnType` specifies the information returned by `showStimulus`. `returnType == 1` causes `showStimulus` to return all the nodes with that match the pattern; with `returnType == 2`, `showStimulus` returns the most activated node matching the given pattern.

  If the presented pattern represents an emotion signal, `matchingInfo` is the instant of time the emotion signal was generated. If the presented pattern is an external stimulus or a searched clause, `matchingInfo` is the pattern itself.

² Actually, these two modules should both be part of a unique Salt & Pepper long-term memory module.
ShowStimulus is used by the Cognitive Engine when it receives external stimuli and emotion signals.

Fetch(Pattern, ReturnType, Motivation): Fetches nodes matching Pattern from long-term memory. ReturnType specifies the type of return (see showStimulus, above). The parameter Motivation conditions the maximum number nodes to be visited by the search algorithm.

Fetch(Pattern, ReturnType, Motivation, PatternType, AdditionalData): A newer version of Fetch/3. AdditionalData represents additional matching information. If the pattern is an external stimulus, the additional information is the stimulus itself. If the pattern is an emotion signal, the additional information is the time at which the signal was generated.

SPFetch(pattern, returnType, Motivation): This method is an interface with the method Fetch(). It converts Java data types into Prolog types. The predicate prs_fetch/6 of the Emotional Production System uses this method to fetch rules and facts from long-term memory.

ewEmptyNode(NodeId): Creates a new empty node in long-term memory with the specified identifier. Returns the reference to the node just created. This method is used by the Emotional Production System predicate sp_create_empty_node/3.

InsertClauseIntoNode(Node, Clause): Inserts a clause into the specified node. Node is a reference to the node in which the clause is to be inserted. Clause is a string containing the text of the clause to be inserted. This method is used by the Emotional Production System predicate sp_insert_clause_into_node/3.

DissociateNodes(from, to, associationPunishment): decrements the association defined by the nodes from and to by a factor set by associationPunishment. DissociateNodes is used by the Cognitive Engine to punish the association that has been found responsible for poor performance.

AssociateNodes(from, to, associationReward): increments the association defined by the nodes from and to by a factor set by associationReward. AssociateNodes is used by the Cognitive Engine to reward the association that has been found responsible for exceptionally good performance.

ActivateNode(node, inputActivation, spreadingDepth): Increases the activation of the specified node by the value specified by inputActivation. The activation conveyed to the specified node is allowed to spread for a number of levels set by the value spreadingDepth. ActivateNode is used by the Cognitive Engine to activate the nodes contained in working memory. This method is used when it is not convenient to use the equivalent Node method. The predicate prs_activate_node/4 of the Emotional Production System uses this method.

Memory Semaphore Methods

WaitOnSem(): The thread that executes this method suspends itself on the semaphore to which the method is applied. This method is used for instance by the predicate prs_waiting_on/1 of the Emotional Production System, when it does not find any satisfied production rule. The system suspends itself until there is any change in the information relevant for rule satisfaction.

ReleaseWaitOnSem(): This method has the opposite effect of WaitOnSem(). It is also used by the predicate prs_releasing/1 of the Emotional Production System.
Node methods

Associate(node, associationIncrement): used to increase the association strength from the node to which it is applied to the node passed as the first parameter. The association increment is set by the parameter associationIncrement.

GetPredicates(): Returns a vector with the clauses contained in the node to which the method is applied. It is used by the Cognitive Engine to process the node contents.

GetId(): Returns the node identifier of the node to which the method is applied. The predicate prs_node_id/2 uses this method in the Emotional Production System

Activation(): Returns the activation of the node to which it is applied. The predicate prs_node_activation/2 uses this method in the Emotional Production System

GetNodeClauses(): Returns the vector of the clauses contained in the node to which it is applied.

SPGetNodeClauses(PO): Interface to GetNodeClauses(). Converts the returned Java data types into Prolog data types. PO is the object representing SICStus Prolog. This method is used by the Emotional Production System predicate node_clauses/3

Clause Methods

lastInstantiationData(): Returns an Object containing information about the last time the clause contained in the node was instantiated due to the presentation of a pattern. If the pattern is an emotion signal, the instantiation data is the time of the instantiation. If the pattern is an external stimulus, the instantiation data is the Java Object representing the stimulus itself.

The Method lastInstantiationData is used by the Cognitive Engine to determine whether the contents of a node interrupting the current process in working memory are processed.

functor(): returns the main predicate of the clause to which the method is applied. It is used by the Cognitive Engine when processing the contents of nodes that interrupt the working memory current processing.

Working memory and component control tasks

Working memory is not just a passive repository of temporary information. Working memory controls the workings of the Cognitive Engine. Component designers wishing to create components on top of the Cognitive Engine must provide a possibly deliberative control mechanism for the component. In periods of regular operation, the provided mechanism will control the component’s working memory, and hence the entire component. However, under special circumstances, the Interrupt Manager may interrupt the regular working of the deliberative control mechanism.

The Cognitive Engine main program invokes its constructor, which creates the component’s long-term memory, and starts the Activation Decrementer, the Interaction Handler, the component deliberative control algorithm (if any), and the Interrupt Manager- each in a separate thread. Since the Emotion Production System mechanism currently is the only available
deliberative control algorithm, the Cognitive Engine launches it explicitly. In the future, the Cognitive Engine will be modified so that different, and possibly multiple, control mechanisms may be used.

The Interrupt Manager can only interrupt working memory if interrupts are enabled. There are two methods used to turn the interrupt enable flag on and off: enable_interrupts() and disable_interrupts().

When the interrupt manager decides that a node should interrupt current processing in working memory, it calls the Interrupt Handler with that node. The deliberative control process is suspended (only in our Linux version)3, the node is copied to and processed in working memory (process_node_content), and finally the deliberative control process is resumed.

```java
public void interruptHandler(Node node) {
    int i;
    int len = this.pRuleControl.SelectedNodes.nElements();
    Vector nodes = this.pRuleControl.SelectedNodes.storedElements();

    if(this.interrupt_enabled()){
        // Associate all nodes already in WM to the new node
        for(i=0; i < len; i++)
            ((Node)nodes.elementAt(i)).Associate(node, this.associationIncrement);

        // Add the new node to WM
        this.pRuleControl.SelectedNodes.newElement((Object)node);

        // Activate the new selected node
        this.ltm.ActivateNode(node, this.inputActivation, this.spreadingDepth);

        // Process the contents of the node, interrupting WM processing
        this.processNodeContent(node);
    }
}
```

**Code-Block 11 – Cognitive Engine Interrupt Handler**

The method processNodeContent is called to process the contents of the node interrupting the current process in working memory. The contents of the node can only be processed if the matching information is still valid as evaluated by the method stillValidMatch of the MatchingInformation class.

External actions contained in the node are placed in the command output buffer; internal actions are executed; external actions appearing in the action part of pattern-action clauses are placed in the command output buffer; internal actions appearing in the action part of pattern-action clauses are executed; all other contents are not subject to any kind of processing.

---

3 In the Linux version this is achieved via a system call using the JNI libraries. In the Windows Operating System, this cannot be done; here, the thread is suspended only when the Java Virtual Machine decides to switch to the next thread.
Each time an external action is placed in the command output buffer, the Cognitive Engine calls the method `sendallSignal`, which signals the Interaction Handler that a new command can be delivered to the components that have subscribed it.

```java
public void processNodeContent(Node node) {
    String action_string = null;
    Object action_designator = null;
    Vector clauses = null;
    int i;
    Predicate cl = null, action = null;
    MatchingInformation m_info = null;
    Date current_time = new Date();

    clauses = node.getPredicates();
    for(i = 0; i != clauses.size(); i++) {
        cl = (Predicate) clauses.elementAt(i);  // Node clause, including instantiation data
        m_info = ((MatchingInformation)cl.lastInstantiationData());
        if(m_info != null && m_info.stillValidMatch(this.input_buffer, current_time,
            this.emotion_retention_time())) {
            if(cl.functor().equalsIgnoreCase("external_action") ) {
                // String with an instantiated action designator
                action_string = ((Predicate)cl.getInstantiated(null)).args().getElementAt(0).toString();
                // Converts the action designator string into a java object
                action_designator = createActionDesignator("SAFIRA.WP4.Adetti.Effector",
                    action_string);
                this.command_output_buffer.add(action_designator);
                this.sendallSignal(this.command_output_buffer);
            } else if(cl.functor().equalsIgnoreCase("internal_action") ) {
                // String with an instantiated action designator
                action_string = ((Predicate)cl.getInstantiated(null)).args().getElementAt(0).toString();
                // Converts the action designator string into a java object
                action_designator = createActionDesignator("SAFIRA.WP4.Adetti.AgentMind",
                    action_string);
                // Converts the Java representation of an action designator into a string
                action_string = action.designator().getElementAt(0).toString();
                // Converts the action designator string into an action designator object
                if(action.functor().equalsIgnoreCase("external_action") ) {
                    // Instatiated action object: external_action(Action); or internal_action(Action)
                    action = (Predicate)cl.getInstantiated(null).getElementAt(1);
                    // Converts the Java representation of an action designator into a string
                    action_string = action.designator().getElementAt(0).toString();
                }
            }
        }
    }
}
```
Currently, the internal actions contained in interrupting nodes must be implemented as Java classes having the method \textit{execute} that actually carries out the action operation. This is unfortunate, because component designers must program both in Java and in Prolog to create and instantiate a component on top of the Cognitive Engine. One of the future steps in the development of the Cognitive Engine will be to allow all these actions to be programmed in Prolog.

The only type of interrupting node content that can be processed is \textit{action}. No other type is processed. In future developments, we will address the problem of determining what to do with interrupting node contents that are not actions.

\textit{Working memory methods}

The Cognitive Engine uses the \textit{pRuleControl} system (the Java object implementing the Java part of the Em-PSys). Unfortunately, this ties the Cognitive Engine—which is supposed to be a generic layer integrating the Salt & Pepper cognitive architecture-to a specific deliberative control mechanism. Instead, the Cognitive Engine should have been defined with a generic working memory control system. The Cognitive Engine explicitly uses the method \textit{SPSelectedNodes}, the attribute \textit{SelectedNodes} and the method \textit{mostAssociatedNodes} of the \textit{pRuleControl} object. The components built on top of the Cognitive Engine may also use some of these methods.

\textit{mostAssociatedNodes(association)}: Instantiates the parameter \textit{association} with the \textit{origin} and \textit{destination} nodes that are the most strongly associated out of all nodes whose contents are being processed in working memory at the time the method is called. The parameter \textit{association} is an array of two nodes. The Cognitive Engine uses the method \textit{mostAssociatedNodes} as a heuristic to identify the association that is responsible for poor/exceptionally good performance of the component. In case of poor performance, the association strength is decreased (punishment). In case of exceptionally good performance, the association strength is increased (reward).

\textit{SPSelectedNodes}(): This method returns the vector of the nodes currently being processed in working memory. The predicate \textit{prs_working_memory_nodes/3} uses this method in the Emotional Production System.
**SelectedNodes**: Vector of the nodes whose contents are currently being processed in working memory. This attribute is often used in the Cognitive Engine. For instance, when it is necessary to increase the association of all nodes whose contents are being processed in working memory to a new node just copied to working memory.

**AddNode()**: This method adds the contents of a node to working memory. The predicate prs_add_node_to_wm/2 of the Emotional Production System uses this method.

**Time methods**

In order to manage temporary information in working memory, it was necessary to create the class TimeObject. This class has two methods that are used by the Emotional Production System to determine when to remove temporary information in working memory.

**currentTime()**: Returns the current system time. This method is used by the predicate prs_time_object/2 of the Emotional Production System.

**timeDiff(\(T_1, T_2\))**: Returns the difference between the two dates, \(T_1\) and \(T_2\). This method is used in the Emotional Production System by the predicate wm_time_difference/4.

**Parameters passed to the deliberative control component**

The Cognitive Engine passes several parameters to the program actually implementing the component, in the current case, the Emotion Production System (Em-PSys). Most of these parameters are required just because the actual component is implemented in Prolog, whereas the Cognitive Engine is implemented in Java. Since the Prolog code needs to apply Java methods to their objects, it is necessary to pass the references to the relevant objects to the Prolog code.

Other parameters that need to be passed to the program that implements the actual component are configuration parameters whose meaning is described in section 3.1.4.2.

The necessary information is passed to the Component code by the Cognitive Engine through the creation of special-purpose facts in the Prolog program.

**prs_wm(\(WM\))**: \(WM\) is the reference to the Java object that represents working memory.

**prs_ltm(\(LTM\))**: \(LTM\) is the reference to the Java object that represents the component long-term memory.

**prs_cognitive_engine(\(CE\))**: \(CE\) is the reference to the Java object that represents the Cognitive Engine.

**prs_ltm_change_semaphore(\(SEM\))**: \(SEM\) is the reference to the Java object that maintains the memory change semaphore.

**prs_prolog_object(\(PO\))**: \(PO\) is the reference of the Java object representing the Prolog object itself. This is necessary because the component code will have to use methods of the Prolog object, such as data conversion methods.
prs_default_motivation(Motivation): Motivation is the default system motivation. Obviously, since the motivation should change as the system works, this fact should not exist. Instead, there should be a predicate to consult the current motivation when it is needed.

prs_wm_activation_increment(ActivIncrement): ActivIncrement is the value to be added to the activation of nodes whose content is being processed in working memory.

prs_spreading_depth(SpreadingDepth): SpreadingDepth is the number of levels that activation is allowed to spread.

prs_association_increment(AssocIncrement): AssocIncrement is the value to be added to the association from node A to node B, when the contents of B are copied to working memory at a time at which the contents of A are already in working memory.

wm_retention_time(RT): RT is the time (in milliseconds) that temporary information is allowed to persist in working memory.

Several of these parameters, namely motivation, should not be passed to the component code as static facts because they may change dynamically. Future versions of the Cognitive Engine will address this issue.

This way to pass information to the actual component built on top of the Cognitive Engine assumes the component to be implemented in Prolog, which is another issue that will be addressed in future versions.

### 3.1.4.2 Configuration Parameters

The Cognitive Engine may be shaped by a set of configuration parameters, which are described below. Any of these parameters can be changed by the agent designer at compilation time, or dynamically through the agent control process.

**interruptingThreshold**

The Memory Monitor detects when there is a node in long-term memory whose activation exceeds the activation of working memory by a certain threshold. The interruptingThreshold parameter specifies the value of that threshold. Currently it is set to 0, but component designers are free to modify this setting.

**interruptionMonitoringPeriod**

With a given frequency, the Memory Monitor checks whether there is any long-term memory node activated enough to interrupt current processing taking place in the component’s working memory. The interruptionMonitoringPeriod parameter specifies the monitoring frequency of the Memory Monitor. It is currently set to 100 milliseconds, but can be changed.

**spreadingDepth**

When a pattern is presented to long-term memory, the activation of the nodes that match the pattern is increased by a certain amount. This change must then be spread to other nodes to which they are associated (depending on the association strengths), and from those to others,
and so on. The spreadingDepth parameter controls the maximum number of levels the activation will be allowed to spread. Currently, spreadingDepth is set to 5, but this can be modified.

**associationPunishment and associationReward**

When an association between two nodes is found to be responsible for poor performance or for exceptionally good performance, the strength of the association should be decremented or incremented, according to the following general equation:

\[ A(t+1) = A(t) \times (1 + \delta) \]

In which \( \delta \) is the punishment or the rewarding factor.

The associationReward and associationPunishment parameters are used to set the value of \( \delta \) in the above equation. Currently, they are both set to 0.1 but they can be changed.

**inputActivation**

When an external stimulus is presented to long-term memory, the activation of the nodes that match it is increased by a certain amount. The inputActivation parameter sets the amount of activation passed on to nodes matching the external stimulus. Currently, inputActivation is set to 10, but this can be modified.

**associationIncrement**

When node N1 is copied to working memory while N2 is already being processed in working memory, the strength of the association from N1 to N2 increases by a certain amount, according to the following general formula:

\[ A(t+1) = A(t) \times (1 + \delta) \]

The parameter associationIncrement is the value of \( \delta \) in the formula. Currently, associationIncrement is set to 0.1, but this can be modified.

**decay_period**

Periodically, the process responsible for the activation-decay updates the activations of all nodes. The decay_period parameter sets the period of time that mediates two consecutive updates. By default, it is set to 10 milliseconds but it can be changed.

**default_motivation**

Whenever the long-term memory is sought for nodes matching a given pattern, the searching algorithm visits a number of nodes that depend on the agent motivation. The default_motivation parameter specifies the initial number of nodes that will be visited by the searching algorithm. default_motivation is currently 1000 but it may be set to a different value. Additionally, any component built from the Cognitive Engine may provide a dynamic way to update the value of default_motivation in run-time.
emotion_retention_time

When an emotion signal results in a node being recruited to working memory due to the received activation, the contents of the node will be processed only if the culprit emotion signal was not received too long ago. The emotion_retention_time parameter specifies the maximum time elapsed since the generation of the emotion signal in order for the contents of the interrupting node to be processed. Currently, emotion_retention_time is set to 10 seconds but this value can be modified.

wm_retention_time

When temporary information is created in working memory, it is tagged with the time of its creation. Periodically, a garbage collection process erases temporary information that has been created too long ago. The wm_retention_time parameter specifies the maximum time that temporary information is allowed to live in working memory. Currently, wm_retention_time is set to 2 minutes but it can be modified.

interrupt_enabled

When the activation of a node becomes high enough to interrupt working memory current processing, it can only actually interrupt the current processing if interruptions are enabled. The interrupt_enabled parameter is used to control interruptions. Both the agent reasoning process and emotion responses may change the value of the interrupt_enabled parameter. Initially, interruptions are enabled (interrupt_enabled = true) but the component designer may change this default setting.

3.2. EMOTIONAL PRODUCTION SYSTEM COMPONENT

Em-PSys (Emotional Production System) is an agent control component built on top of the Salt & Pepper Cognitive Engine (section 3.1.4). Its behaviour therefore is conditioned by emotion and by the differing accessibilities of long-term memory nodes. Satisfied rules contained in more accessible nodes are selected over satisfied rules contained in less accessible nodes. Emotion responses may interrupt the regular functioning of the Em-PSys component.

The production rule based agent control component receives external information from the agent's sensing components and receives emotion signals from the agent's appraisal components. It sends information to the agent's appraisal components, and sends commands (i.e., actions to be executed) to the agent's effectors.

The agent designer must create a set of long-term memory nodes containing production rules, declarative Prolog clauses, facts, pattern-action clauses, action descriptions, and possibly action definitions. These knowledge structures have to be specified in the Prolog programming language. It is not mandatory to place all the component knowledge structures in

---

4 Currently, the actions appearing in the action side of pattern-action clauses must be defined in Java. This is an unfortunate consequence of using SICStus Prolog. We are currently finishing the development of Nazgul Prolog that will allow to use Prolog to define these actions as well. tuProlog is also being considered as an alternative (http://tuprolog.sourceforge.net/)
long-term memory nodes. If desired, some knowledge structures may be “hardwired” into the agent control component. In particular, knowledge structures using Prolog “cuts” cannot be placed in long-term memory nodes; they must be “hardwired”. Furthermore, agent designers wanting to increase efficiency may also want to hardwire procedural code.

Declarative Prolog clauses represent relations between objects and properties of objects. Declarative Prolog clauses are used by the component inference engine in a backward chaining fashion. Declarative Prolog clauses are used only when required to evaluate the condition of a production rule.

Rules represent *IF condition THEN action* knowledge structures. The component inference engine evaluates the rule conditions, in a forward chaining fashion. If needed, the condition of a rule may be defined through a set of Prolog clauses. When this is the case, the component inference engine evaluates the condition in a backward chaining fashion.

Prolog facts represent persistent factual knowledge or episodic knowledge automatically recorded by the component.

Pattern-action clauses are used mainly to represent the conditions under which a reflex action, such as an emotion response, may be executed. Pattern-action clauses consist of a pair of a pattern and an action. The component inference engine checks whether the pattern of the pattern-action clause matches received external stimuli or emotion signals. Contrarily to the evaluation of rule conditions, no inference is performed to check whether the patterns of pattern-action clauses match specified patterns.

Action descriptions are special purpose clauses used to represent the effects and typical execution times of the actions used in component production rules. These descriptions are used by the component to inform some appraisal component of the expected results of actions selected for execution.

Action definitions are used to define internal actions. External actions are executed by effectors, which are independent components. Internal actions are to be performed by the component itself. These must be defined by Prolog procedures. Prolog procedures defining internal actions may be contained in long-term memory nodes or hardwired in the component code.

The main loop of Em-PSys fetches production rules from the agent long-term memory, selects those with satisfied conditions, and picks the one contained in the most accessible node (the node with highest long-term memory activation). The system then executes (or sends for execution) the actions of the rule chosen. Internal actions are directly executed; external actions are sent to the corresponding effector component to be executed.

Em-PSys does not automatically read input information from the input buffer. If designers want the agent to acquire input information, they must provide rules with read actions.

The regular functioning of the production system may be suspended whenever interruptions are enabled and long-term memory nodes become activated enough. In such cases, the most activated of the nodes exceeding the interrupt threshold is selected to working memory and is processed. Long-term memory nodes may interrupt working memory for two reasons:
The node matches an external stimulus and becomes more activated than current working memory activation

The node matches an emotion signal becoming more activated than current working memory activation

The actions of pattern_action clauses of interrupting nodes are executed whenever

(i) the cause for the node becoming more activated is an emotion signal that has occurred not too long ago; or

(ii) the cause for the node becoming more activated is an external stimulus that is still present in the input buffer.

After the interruption being processed, the production rule inference process resumes, taking control of the component until the next interruption. Due to this interruption mechanism, the production system and emotion control the agent behaviour in unpredictable ways.

Agent designers must also create nodes with emotion-responses. The best way to create emotion responses is through the creation of nodes with pattern-action clauses in which the pattern specifies the emotion signal and the action is the emotion response. The emotion response may be external actions, internal actions, or both. Any action can be used as an emotion response. It is natural to specify reflex external actions; or to use actions that create goals; or complex internal procedures such as rationalization processes, triggered to find explanations for the experienced emotion.

**Automatic influence of cognition on emotion**

Emotion influences cognition and behaviour by increasing the accessibility of information in long-term memory, by changing the strength of associations between nodes in long-term memory, and possibly by triggering emotion responses. Cognition and behaviour also influences emotion.

The computational framework of Jonathan Gratch, in which an intelligent planner influences emotion [Gratch 1999, 2000], draws upon the OCC cognitive theory of emotion [Ortony, Clore and Collins 1988]. Although Gratch's proposal has been presented for planners, the basic ideas may be adapted for production rule agent control. If an action fails to produce the expected results, a negative emotion signal will be generated. Conversely, if an action outperforms the agent's expectations, a positive emotion signal will be generated. If external events, objects, or agents impair the achievement of agent's goals, negative emotion signals will be generated. Conversely, if the achievement of agent's goals is facilitated, a positive emotion signal will be generated.

In the SAFIRA architecture, these and other kinds of influences of cognition and behaviour on emotion are handled by appraisal components. Each appraisal component subscribes to relevant information from the production rule based control component (e.g., actions to be executed together with their expected outcomes) in order to produce emotion signals that, in turn, are sent back to the agent control components, closing the feed-back loop.
In order for the agent control component to condition the generation of emotion, the agent designer must provide action-descriptions for actions used in the right hand side of production rules. The Em-PSys system automatically computes the desired outcome(s) of the selected action and sends them to the appraisal components that have subscribed to them.

In order for the system to compute the desired action expectations, production rules must specify the goal that is being satisfied by the rule. In addition, each action must be described together with the effects it is expected to cause:

\[\text{IF (Goal, Condition) THEN Action}\]

\[\langle \text{Action, Pos(Action), Duration(Action)} \rangle\] in which \(\text{Pos(Action)}\) represents the expected effects of executing the action, and \(\text{Duration(Action)}\) is the typical duration of its execution (in milliseconds).

When a rule \(\text{IF (G, C) THEN A}\) is selected to be executed, the desired expectations of executing action \(A\) are the common parts of \(G\) and \(\text{Pos(A)}\). I.e., if we assume that the action effects and rule goals are available in clause format, an action expectation is the intersection of the clauses representing the effect of the action with those representing the rule goal.

Assume action \(A\) is chosen to be executed by the rule \(\text{IF (G, C) THEN A}\). The expectations of executing action \(A\) are the maximal set of clauses \(G_\sigma \cap \text{Pos(A)}_\sigma\), in which \(\sigma\) is a substitution and \(E_\sigma\) is the result of applying the \(\sigma\) to an expression \(E\):

\[\text{Expectations(A, G)}=\text{Max}\{G_\sigma \cap \text{Pos(A)}_\sigma: \sigma\ \text{belongs to the set of all substitutions}\}\]

in which \(\text{Max(S)}\) is the set of \(S\) with the highest cardinality.

The expectations of an action are assumed to arise within the action’s typical duration plus a given temporal delta, which must be specified to the system.

If the right hand side of a rule is a sequence of actions \(A=\{a_1, a_2, a_3, \ldots\}\), then the expectation of executing action \(a_i\) is the common parts of the rule goal and the expected effects of \(a_i\).

The agent programmer is not required to write the rules in this format. In particular, the goal part of the rule may be omitted altogether, or it may be too far away from the expected effects the rule’s action. If any of these conditions arise, it is possible that the desired expectations of a given action selected by a given rule will be the empty set. Under such particular circumstances, the system will not have any expectations regarding its actions and, hence, no performance evaluation emotion signal may be generated.

This mechanism for the computation of desired expected results of actions is not exclusive of rule based control systems. This mechanism could be used in any agent control system. However, applicability of the Salt & Pepper architecture is not confined to agent control systems. It can also be used as the basis for other internal agent processes. Therefore, the mechanism for the computation of desired expectations of actions should not be defined in the Salt & Pepper architecture, but rather within a general control system. Unfortunately, it currently has been implemented in the Emotional Production System (Em-PSys) agent control component.
Examples of proper rule design

Consider the actions step_north, step_south, step_west, and step_east whose post-conditions are as follows (for simplicity, we omit the typical durations of actions).

\[
\text{pos(step_north}(X, Y), \text{[position}(X, Y1)]) :- \text{Y1 is } Y + 1.
\]

\[
\text{pos(step_south}(X, Y), \text{[position}(X, Y1)]) :- \text{Y1 is } Y - 1.
\]

\[
\text{pos(step_east}(X, Y), \text{[position}(X1, Y)]) :- \text{X1 is } X + 1.
\]

\[
\text{pos(step_west}(X, Y), \text{[position}(X1, Y)]) :- \text{X1 is } X - 1.
\]

In the following, we provide an example rule that is not written so that the agent can compute the expected desired outcome of its actions. Then, we provide an example of a well-specified rule. Both of them are syntactically correct, but only the second one enables the computation of the desired expectations for the action.

Wrong example

IF (goal(position(Gx, Gy)), position(X, Y) and X > Gx) THEN step_west(X, Y)

The above rule, although perfectly usable for the purpose of moving the agent around until it reaches the goal position, is not well specified with respect to allowing the agent to compute the desired outcomes of the action step_west.

Let us suppose that the goal position is (Gx=1, Gy=4) and the current position is (X=10, Y=3). In this case, there is no intersection between the rule goal (position(1, 4)) and the action’s effect (position(9, 3)).

Good example

IF (goal(position(Gx, Gy)), position(X, Y) and X > Gx) THEN (X1 is X – 1, create_goal(position(X1, Y)))

IF (goal(position(Gx, Gy)), position(X, Y) and immediately_west_of(Gx, X)) THEN step_west(X, Y)

immediately_west_of(X, Y) :- X is Y - 1.

The above rules are well specified. The first one is used only to create an appropriate sub-goal. If the agent’s position is (X=10, Y=3) and the goal position is (Gx=1, Gy=4) the first rule creates a new intermediate goal position (Gx'=9, Gy'=3)), so that the second rule may be used. Now the goal of the second rule is position(9, 3) and the effect of step_west(10, 3) is exactly position(9, 3), therefore the set of desired expectations of the action step_west is not empty.

Temporary information in working memory

Temporary information that is (incidentally or purposefully) left in working memory without being removed explicitly is automatically removed by Salt & Pepper when its age exceeds a given
threshold. For the moment, there is only a single global retention time: all temporary information is retained for the same period of time. In the future, we will introduce diverse retention times.

When Em-PSys yields actions to be executed by some agent effectors, the actions are automatically sent to the effectors that have subscribed them. Whether or not these commands are really executed is up to the effectors themselves. Sending the command to the appropriate effectors is not the responsibility of the designed rules. This is the responsibility of the Interaction Manager of the component (see section 3.5).

3.2.1 Algorithm and Implementation

The basic algorithm of the Emotional Production System is a loop that repeats the following sequence of operations:

(i) Clean temporary clauses whose age exceeds temporary information retention time

(ii) Load long-term memory nodes that have not yet been loaded (nodes may have been dynamically created in the meantime)

(iii) Fetch nodes containing rules to the evaluation memory of working memory; in the following, we refer to these nodes as Rule Nodes

(iv) Identify the conflict set (the set of rules with satisfied condition) and copy it to the use memory; associate all nodes whose contents have previously been copied to use memory to the new nodes; associate each new node to all other new nodes; and activate all new nodes whose content has been added to use memory

(v) Select the rule of the conflict set that is contained in the most activated node of Rule Nodes; call it, the Selected Rule

(vi) Execute the internal actions of the Selected Rule and send its external actions for execution

(vii) Compute the desired expectations of the actions sent for execution, and send them to the appraisal components that have subscribed them

This algorithm is implemented in a Prolog program (see Code-Block 13 through Code-Block 20), which is called from the Cognitive Engine Java program (see section 3.1.4).
psys:-
% LTM, CE, PO, and WM are (pointers to) representations of Java Objects
%  
% prs_prolog_object(PO),
% prs_cognitive_engine(CE),
% prs_ltm(LTM),
% prs_wm(WM),
% prs_default_motivation(Motivation),
% wm_retention_time(RT),
repeat,
  once({
    prs_clean_temporary_clauses(RT),
    load_ltm, % Loads all nodes that were not loaded yet
    prs_ltm_rule_nodes(LTM, PO, Motivation, RuleNodes)
  }),
  prs_process_rules(WM, PO, RuleNodes),
!.

Code-Block 13 – Main loop of the Em-PSys

The facts prs_cognitive_engine(CE), prs_ltm(LTM) — long-term memory —, prs_prolog_object(PO), and prs_wm(WM) — working memory —, hold references to the Java objects that are needed in the Prolog program just to allow calling Java methods from the Prolog program because those methods must be applied to the corresponding objects. prs_default_motivation(Motivation) holds the current system motivation. This parameter is used when nodes are searched for in long-term memory.  

wm_retention_time(RT) holds the retention time of temporary information created in working memory.

prs_process_rules(_, _, []):- !.
prs_process_rules(WM, PO, RuleNodes):-
  once({
    prs_working_memory_nodes(WM, PO, WMNodes), % Nodes already in working memory
    wm_temporary_clause_ids(TempNodes),
    % WMNodes – List of Node Java Objects already in working memory
    % Nodes – List of Node Java Objects newly copied to working memory
    % AllNodes – union of Nodes and temporary nodes that have not yet expired
    prs_nodeset_union(WMNodes, RuleNodes, Nodes),
    append(TempNodes,Nodes, AllNodes),
    prs_conflict_set(AllNodes, ConflictSet)
  }),
  % ConflictSet – List of node_clause(N, Rule) in which N is a node identifier (prolog atom)
  prs_process_conflict_set(Nodes, ConflictSet).

Code-Block 14 – Computation of the conflict set

If there are no rule nodes, the process halts (first clause). Otherwise, the set of satisfied rules (the conflict set) is computed and processed. WM and PO are references to the Working Memory and the Prolog Java objects. They are needed in order to call their methods from the Prolog program.
The detailed explanation of the predicates append/3, prs_working_memory_nodes/3, wm_temporary_clause_ids/1, and prs_nodeset_union/3 is cumbersome. For the present purposes it suffices to state that they are used to compute the set of all nodes containing the knowledge structures in working memory (after retrieval of all current rule nodes).

% Currently, this stops as soon as it reaches an empty conflict set.
% In the future, it will stop only if no modifications are made in LTM.
%prs_process_conflict_set(_, []):- !,
   prs_wm(WM),
   prs_waiting_on(WM),
   fail.
prs_process_conflict_set(NewNodes, ConflictSet):-
   select_nodes_to_use_memory(NewNodes), % All nodes of satisfied rules are added to WM
   prs_prolog_object(PO),
   prs_wm(WM),
   prs_working_memory_nodes(WM, PO, Nodes), % All selected nodes
   wm_temporary_clause_ids(TempClauseNodes),
   append(Nodes, TempClauseNodes, AllNodes),
   prs_select_most_activated_rule(Nodes, ConflictSet, node_clause(N,rule(Condition, Action))),
   prs_process_action(AllNodes, Action),
   prs_condition_goals(Condition, Goals),
   prs_action_expectations(AllNodes, Goals, Action, Expectations),
   prs_process_action_expectations(Expectations), !,
   fail.

Code-Block 15 – Processing of the rules contained in the conflict set

In case of an empty conflict set, the production rule system suspends itself and waits for new information in the system input buffer (prs_waiting_on/1). When new information enters the input buffer, the process resumes.

All nodes containing satisfied rules are copied to use memory. Consequently, the nodes containing knowledge structures already in use memory become more strongly associated to the nodes containing the rules in the conflict set; each node containing rules in the conflict set becomes associated to all other nodes containing rules belonging to the conflict set; and all new nodes added to use memory receive a predefined amount of activation.

After nodes containing rules that belong to the conflict set are copied to use memory, the satisfied rule contained in the most accessible node (that contains rules) is selected; its action is processed, and the desired action expectations are computed and processed.

As before, a detailed discussion of the predicates prs_prolog_object/1, prs_wm/1, prs_working_memory_nodes/3, wm_temporary_clause_ids/1, and append/3, whose function has been briefly explained above, is omitted on purpose.
An important part of the main algorithm is the computation of the conflict set (Code-Block 16). This computation involves evaluating the conditions of production rules, which are implemented within long-term memory nodes. In order to evaluate general Prolog goals, taking into account only the clauses contained in a set of nodes, it was necessary to find an adequate representation of the node contents in Prolog. It was further necessary to write an inference mechanism that uses that representation.

Node contents, which may be specified by the component user (i.e., the agent programmer), are represented in Prolog through the special fact `node_clause/2`. `node_clause(NodeId, Clause)` means that `Clause` belongs to the node `NodeId`. The created inference mechanism, explained below, is a backward chaining algorithm that handles `node_clause/2` clauses.

/* Computes the set of satisfied rules */
prs_conflict_set(Nodes, ConflictSet):-
    findall(
        node_clause(N, rule(Condition, Action)),
        (node_clause(N, rule(Condition, Action)), prs_member(N, Nodes),
        solve(Nodes, Condition)),
        ConflictSet).

Code-Block 16 – Conflict set computation algorithm

The predicate `prs_conflict_set/2` uses `findall` to collect into `ConflictSet` the set of node clauses containing rules with satisfied conditions. The conditions are evaluated with respect to the clauses contained in the set of nodes currently in working memory, passed in the parameter `Nodes`. The predicate `solve/2` (Code-Block 17) is the meta-interpreter responsible for the evaluation of rule conditions.

In the meta-interpreter `solve/2`, `true` holds for any set of nodes. `(A, B)` holds for the set of nodes `Nodes`, if both `A` holds for `Nodes` and `B` holds for `Nodes`. `(A;B)` holds for `Nodes` if either `A` or `B` hold for `Nodes`. Not `P` (i.e., `\+P`) holds for `Nodes` if `P` does not hold for `Nodes`. Otherwise, `P` is an atomic goal. The predicate `solve_atomic_prop/2` handles atomic goals.

solve(Nodes, true).
solve(Nodes, (A,B)):- !,
solve(Nodes, A), solve(Nodes, B).
solve(Nodes, (A;B)):- !,
solve(Nodes, A); solve(Nodes, B).
solve(Nodes, \+P):- !,
\+ solve(Nodes, P).
solve(Nodes, P):- solve_atomic_prop(Nodes, P).

Code-Block 17 – Node Meta-Interpreter
solve_atomic_prop/2 tries to find solutions an atomic goal in the set of nodes currently in working memory (basic_solve_atom/2). If it does not find at least one solution for the atomic goal in the nodes currently in working memory, solve_atomic_prop/2 tries to find solutions in other nodes it must fetch from long-term memory, conditioned by the available amount of motivation (more_solve_atom/2).

\[
\text{solve_atomic_prop}(\text{Nodes}, P) :- \\
\text{findall}(P, \text{basic_solve_atom}(\text{Nodes}, P), [S | \text{Solutions}]), !, \\
\text{member}(P, [S | \text{Solutions}]).
\]

\[
\text{solve_atomic_prop}(\text{Nodes}, P) :- \\
\text{\}+ \text{hardwired_predicate}(P), \\
\text{\}more_solve_atom}(\text{Nodes}, P).
\]

**Code-Block 18 – Evaluation of atomic goals**

The predicate basic_solve_atom/2 evaluates an atomic goal with respect to a fixed, predefined set of nodes.

\[
\text{basic_solve_atom}(\text{Nodes}, P) :- \\
\text{\}hardwired_predicate}(P), !, \\
\text{\}call}(P).
\]

\[
\text{basic_solve_atom}(\text{Nodes}, P) :- \\
\text{\}+ \text{defined_predicate}(P), \\
\text{\}\text{node_clause}(\text{Node}, (P:-Q)), \\
\text{\}\text{prs_member_node}(\text{Node}, \text{Nodes}), \\
\text{solve}(\text{Nodes}, Q).
\]

\[
\text{basic_solve_atom}(\text{Nodes}, P) :- \\
\text{\}+ \text{defined_predicate}(P), \\
\text{\}\text{node_clause}(\text{Node}, P), \\
\text{\}\text{prs_member_node}(\text{Node}, \text{Nodes}).
\]

**Code-Block 19 – Evaluation of an atomic goal w.r.t. a set of nodes**

If P is hardwired, the Prolog interpreter is called to evaluate P. P is hardwired if it is a Prolog built-in or if it is defined by the agent designer outside of long-term memory (that is, in a .pl file). If there is a clause \[P:- Body\] in the set of clauses contained in Nodes, and Body can be proven using those clauses, then P is true. P is also true if it belongs to the clauses contained in Nodes.

\[
\text{more_solve_atom}(\text{OldNodes}, P) :- \\
\text{\text{search_matching_nodes}(P, NewNodes)}, \\
\text{\text{temporary_out_of_all_nodes}(OldNodes, TemporaryNodes)}, \\
\text{\text{add_nodes_to_working_memory}(NewNodes, WMNodes)}, \\
\text{\text{append}(Temporary, WMNodes, Nodes)}, !, \\
\text{\text{basic_solve_atom}(Nodes, P)}.
\]

**Code-Block 20 – Fetching additional long-term nodes to solve an atomic goal**

The predicate search_matching_nodes/2 fetches nodes matching the atomic goal being evaluated from long-term memory, using the agent’s current motivation. The predicate add_nodes_to_working_memory/2 basically calls select_nodes_to_working_memory/1 to add the newly fetched nodes to use memory as explained earlier in the discussion of processing.
of the conflict set (Code-Block 15). Nodes whose contents were already in working memory become more strongly associated to the newly selected nodes; each new node becomes more strongly associated to all other new nodes; and all newly fetched nodes become more activated.

The roles of `temporary_out_of_all_nodes/2` and `append/3` are irrelevant for the current discussion.

The main idea behind `more_solve_atom/2` is to fetch nodes from long-term memory, add them to use-memory and then call `basic_solve_atom/2` to evaluate the atomic goal again.

By virtue of being implemented on top of the Cognitive Engine (section 3.1.4), the normal functioning of the Emotional Production System described at the beginning of this section may be interrupted by a new node that has become sufficiently activated. When this happens, the contents of the interrupting node are copied to and processed in working memory. Long-term memory nodes may become highly accessible mainly because of emotion signals, but also because of new stimuli and even because they receive activation due to activation increment in other nodes.

### 3.2.2 Built-in Actions

Em-PSys offers actions to create temporary Working Memory facts and goals, actions to remove temporary Working Memory facts and goals, actions to read the input buffer, actions to create Long-Term Memory nodes, and actions to enable/disable working memory interruptions.

- **wm_assert_temporary_clause(+Clause)**
  
  Creates a temporary clause in Working Memory.

- **wm_create_goal(+Goal)**
  
  Uses `wm_assert_temporary_clause` to create a temporary goal in Working Memory. The created goal will have the format `goal(Goal)`.  

- **wm_remove_temporary_clause(+Clause)**
  
  Removes a temporary clause from Working Memory.

- **wm_remove_goal(+Goal)**
  
  Removes a temporary goal from Working Memory.

- **wm_read_input_buffer(CE, Object)**
  
  CE - Reference of the Cognitive Engine Java Object

  Object - Prolog term read from the input buffer

  Reads the next element in the input buffer. When the end of the buffer is reached, the predicate returns the empty list but does not fail.
ltm_create_node(+NodeId, +Clauses, -Node)

Creates a new node in long-term memory, with the given identifier and the given set of clauses.

NodeId – Atom representing the node identifier
Clauses – List of clauses to be inserted into the new node
Node – Reference to the Java Object representing the created node

ltm_create_node(+Clause, -Node)

Generates a new node identifier and creates a new node in long-term memory, with that identifier containing the given clauses.

Clauses – List of clauses to be placed in the new created node
Node – Reference to the Java Object representing the created node

ltm_add_clause_set_to_node(+LTM, +Node, +ClauseSet)

Inserts a set of clauses in an existing long-term memory node.

LTM – Reference to the Java Object representing long-term memory
Node – Reference to the Java Object representing the created node
ClauseSet – List of Prolog clauses.

ltm_new_node_id(+Prefix, -NodeId)

Create a new long-term memory node identifier

Prefix of the node identifier (e.g., node).
NodeId – New node identifier

prs_disable_interrupts (+CE)

CE – Reference to a Java Object representing the agent cognitive engine (Class CognitiveEngine)

Turns the flag interrupts_enabled off.

Interrupts are enabled by default.

prs_enable_interrupts (+CE)

CE – Reference to a Java Object representing the agent cognitive engine (Class CognitiveEngine)
Turns the flag interrupts_enabled on.

Interrupts are enabled by default.

3.2.3 Running the Emotional Production System

The production rule based agent control component is run as any other Java program. The component package is called AgentMind, and the main class is the CognitiveEngine class.

To run the component, the following arguments must be given:

The component name (for instance, "cog");

The component hostname (for instance, "localhost");

The component server socket port (for instance, 5001);

The pathname of the directory containing the files with the production rules (for instance "/Ustr/Amos/Cog/PSys").

The component should be executed at the Linux prompt or at the DOS prompt:

% java AgentMind.CognitiveEngine cog localhost 5001 /usr/Amos/Cog/PSys

or

C:> java AgentMind.CognitiveEngine cog localhost 5001 C:/usr/Amos/Cog/PSys

Note: the directory separator character should be '/' even for Windows users

Agent Designer Specification

The agent designer must provide the following elements:

- One or several files containing the component production rules. Each rule RHS is a sequence of internal or external actions. External actions are sent to the agent's effector components. Internal actions must be executed by the component itself, therefore they must be defined in Prolog. If a rule must execute a java method instead of a Prolog defined procedure, the designer must provide the interface between Prolog and Java plus the corresponding Java method.

- One or several files containing Prolog code to be hardwired in the component. These files must have .pl extension.

- One or more files containing node specifications with emotion responses. Each emotion response is executed by the Java Virtual Machine therefore there must be a Java Method of the InternalActionExecuter class (see next point). In the next version of the Em-PSys, emotion responses will also be written in Prolog.
• One file containing the Java code of the component internal actions. Long-term memory nodes containing internal actions to be executed by the Java Virtual Machine (such as emotion responses) must be defined as methods of the 'InternalActionExecutor' Java class. This should be made in the file 'InternalActionExecutor.java' provided together with the component. This class already contains the method execute, which may not be deleted. After all required internal action methods being defined, the class 'InternalActionExecutor.java' must be compiled.

• Descriptions of available actions, through pos/3 clauses.

• Component designers must also specify the information categories, and the emotion signals that must be subscribed by the component.

### 3.2.4 Specification of the long-term memory contents

The contents of the component long-term memory must be specified in one or more text files placed in a single directory whose pathname is passed as a parameter to the component. All the long-term memory specification files must have .ltm extension. This section describes the way long-term memory node contents are written.

Each file specifies a set of long-term memory nodes containing rules, facts, action definitions and action descriptions. Each node may contain one or more rules, facts, action definitions, and action descriptions.

Rules, facts, action definitions and action descriptions are written in standard Prolog.

#### Facts

Standard Prolog facts, for example:

```
position(cog, 25, 120).
```

#### Action definitions

A set of one or more Prolog clauses defining a procedure, for example

```
clean_memory:-
    memorized_stuff(MemIndex),
    clean_memory(MemIndex).

clean_memory([P|MemIndex])
    retractall(P),
    clean_memory(MemIndex).

clean_memory([]).
```

#### Rules

A rule is represented by the special Prolog fact rule/2, such that rule(Condition, Action) represents the production rule *IF Condition THEN Action*. 
Condition =
    Atomic_Condition
    | Negated_Condition
    | "(" Composed_Condition ")"

Composed_Condition =
    Atomic_Condition
    | Negated_Condition
    | Composed_Condition "," Composed_Condition
    | Composed_Condition ";" Composed_Condition.

Atomic_Condition = Prolog Term.
Negated_Condition = "\+" Condition.

In simple words, a condition is a Prolog atom, a negated condition, or a complex condition, enclosed in parentheses, containing conjunctions and disjunctions.

Action =
    Atomic_Action
    | "(" Action_Sequence ")".
Action_Sequence =
    Atomic_Action
    | Atomic_Action "," Action_Sequence.
Atomic_Action =
    "internal_action" "(" Prolog Term ")"
    | "external_action" "(" Prolog Term ")".

In simple words, an action is an atomic action or a sequence of actions enclosed in parentheses, separated by commas.

    Each action in a rule may be either an internal action defined in Prolog (including Prolog hooks for Java methods), or an external action to be executed by some effector component.

Examples

"If the villain and the hero are in the same position, then kill the villain"
rule(
    (position(hero, X, Y), position(villain, X, Y)),
    external_action(kill_villain)
).

"If the hero sees a visual object of type trap, identified by Id, and located at X, Y then create the fact trap(Id, X, Y)"
rule(visual_object(trap, TrapID, X, Y),
    internal_action(createTrap(Id, X, Y))
).

Action Descriptions

Optionally, the component designer may want to have information regarding the expected results of each action, together with its expected duration. This information is very useful if you want to
have an agent with an appraisal component that may detect situations in which the execution of a given action fails to produce the desired results.

pos(step_north(X, Y), [position(X, Y1)], 100):-  
  Y1 is Y + 1.

pos(step_south(X, Y), [position(X, Y1)], 100):-  
  Y1 is Y - 1.

pos(step_east(X, Y), [position(X1, Y)], 100):-  
  X1 is X + 1.

pos(step_west(X, Y), [position(X1, Y)], 100):-  
  X1 is X - 1.

The first pos/3 clause says that, after the agent setsps north of its current position, its new position will be one step further to the north; the expected duration of stepping north is 100 miliseconds. The other three pos/3 clauses specify the expected effects of step_south, step_east and step_west actions.

Nodes

Each node starts with the token “node” followed by a full stop. The “node” term is followed by the node contents (which is formed by a set of Prolog clauses) followed by the token “end_node” followed by a full stop. Both the node and the end_node terms cannot be put together with other stuff in the same line. Prolog comments may be inserted anywhere in the file.

Syntax

node = “node” “.” node_content “end_node” “.”.

node_content =

    clause “.”

    | clause “.” node_content.

Examples

node.
action(jump_backwards(X, Y)).
end_node.
node.
rule(
    (agent_position(AX, AY), villain_position(VX, VY), VX > AX),
    (external_action(step_north(AX, AY)),
    increase_motivation(5))
).
rule(
    (agent_position(AX, AY), villain_position(VX, VY), VX < AX),
    (external_action(step_south(AX, AY)),
    increase_motivation(5))
).
end_node.

3.3. THE APPRAISAL COMPONENT

The Appraisal Compiler is a software tool to assist in building appraisal components that may be integrated with other components to make up an agent. The Appraisal Compiler automatically generates SAFIRA Integration Framework compliant Appraisal Components that communicate with other components using the inter-component communication language and protocols defined within the SAFIRA Agent Framework. Internally, the generated appraisal component consists of several modules including possibly several appraisal modules. All appraisal modules of an appraisal component must be specified and filled by the agent designer.

The SAFIRA appraisal approach is based on the Salt & Pepper theory [Botelho and Coelho 2001]. The main hypothesis behind Salt & Pepper is that emotion plays an adaptive role in humans and that we can implement artificial agents having processes (Artificial Emotions) that can play the same adaptive role emotion plays in humans. Following Simon’s approach to emotional control [Simon 1967], the adaptive emotional response is triggered by interruption mechanisms based on Emotion Signals. Emotion Signals are generated by appraisal components that evaluate incoming stimuli as well as internal state information. If the appraisal component decides that the situation is sufficiently relevant, it sends an emotion signal to some of the other agent components. The effect of the emotion signal does not concern the appraisal component: each agent component that receives emotion signals has to decide what to do with them by itself. For example, as a consequence of a signal, a planning algorithm may decide to change the priority of goals not yet achieved. It may also decide that the signal is not sufficiently relevant to cause its behaviour to change.

In its first version, the Appraisal Compiler offered an automatic mechanism to build general appraisal components. Now, it was extended to offer general built-in Appraisal Modules that may be used in any application. The appraisal modules provided detect domain-independent emotion eliciting conditions, generate the appropriate emotion signals, and send them to the components that have subscribed to them. The domain-independent emotion eliciting conditions detected by these appraisal modules closely follow Jonathan Gratch’s proposal [Gratch 1999, 2000] which is based on the widely applied OCC appraisal theory [Ortony, Clore and Collins 1988].
3.3.1 The Appraisal Component Architecture

The general appraisal component architecture is presented in Figure 5. Each appraisal module is made up of a prioritised set of appraisal rules controlled by a forward chaining inference mechanism. The appraisal rules are written in text files (one for each component) and are processed by Jess Java Classes. A dedicated module loads the text file rules into the Jess inference engine. Each module produces only one kind of emotion signal.

The appraisal rules are specified using Jess rules syntax:

(condition) => (action)

The component uses sockets to communicate with other components. Information received from provider components (as a consequence of information subscription) is stored in an internal mailbox (that works like a Round Robin). The emotion signals are stored in another mailbox and then delivered to components that subscribed them.

Modules can subscribe emotion signals from other modules of the same component.

Figure 5 General Appraisal Component Architecture

The communication among modules and components is managed by the Interaction Manager, as illustrated in Figure 6. The Interaction Manager parses XML messages into java objects, and vice versa.

In order to do accomplish its tasks, the Interaction Manager must dispose of information - as illustrated in Figure 7 - including:

a) The XML schemas of subscription messages (to subscribe to provider components);

b) The XML schemas of the messages subscribed to;
c) The XML schema of emotion signal subscription (standardised for all SAFIRA components);

d) The XML schema of emotion signal message (standardised for all SAFIRA components);

e) The component name and address and the component server socket port (not illustrated in the figure).

![Figure 6 The role of the Interaction Manager](image)

![Figure 7 Information required for the Interaction Manager](image)

The communication between module(s) and the Interaction Manager (based on semaphores), as well as the communication with the Jess engine, are illustrated in Figure 8.
3.3.2 The Component Compiler

The Appraisal Compiler input data is provided using the form presented in Figure 9.

The data shown in the example form Figure 9 means the following:

1) Module 1 produces the emotion signal ESA of CatA category. The capacity of the input and output mailboxes of Module 1 (maximum messages) is 20 messages, although they could have different sizes. Module 1 subscribes from other components information InfA (which is not depicted in the figure since it shows only the data pertaining to Module 2).

2) Module 2 produces the emotion signal ESB of CatB category. The capacity of the input and output mailboxes of Module 2 (maximum messages) is 25 messages. Module 2 subscribes from other components information InfB and also subscribes the emotion signals of Module 1.

The Generate Button triggers the production of the corresponding Java source code. All generated code can be modified. However, for simple components, there is no need to change the code (except for a small detail). The code does not need to be changed if the data sent by the providers does not need to be pre-processed before it is stored in the Jess database (modules automatically include a simple parser in order to take care of Jess syntax).
Besides completing the form, users must supply the XML schema files and fill the Jess database with the appropriate production rules.

Some conventions were adopted (in the following, \textit{em} in the filenames stands for the unique emotion signal identifier [+ category?!]):

\begin{itemize}
\item[a)] The names of the XML Schema files are the same as the corresponding information category names (in the given example: \textit{InfA.xsd} and \textit{InfB.xsd});
\item[b)] The XML schema file of an emotion signal subscription is \textit{req_em.xsd};
\item[c)] The XML schema file of an emotion signal message is \textit{em.xsd};
\item[d)] The name of the main class file is the name of the component (AppComponent.java in the example);
\item[e)] Each module has one associated file with corresponding classes; the filename is the same as the module name (the first letter of the module name should be an uppercase letter).
\end{itemize}

\textbf{Emotion Signals}

All SAFIRA compliant emotion signals must have the following arguments: Signal Name, Emotion Category (used to classify the signals), Intensity (a float value that represents the signal
urgency), Valence (positive or negative) and Parameters List (an optional parameter to hold any additional domain dependent parameters).

The emotional signal intensity in the built-in appraisal modules (section 3.3.3) exhibits the following properties related to the action performance evaluation:

a) The intensity of a negative signal increases with the number of failed evaluated actions;

b) The intensity of a positive signal decreases when the evaluated action succeeds more often;

c) When the action status changes (from success to failure or vice versa) the intensity of the emotion signal (negative valence for changes from success to failure or positive valence otherwise) is proportional to the difference between the number of succeeded and failed actions.

The designer must specify the intensity of user specified emotion signals.

Emotion eliciting rules cannot fire too frequently; otherwise, the agent wouldn’t be capable of processing all generated emotion signals. The suitable frequency depends on the domain application. Therefore the Appraisal Compiler generates a control mechanism to deal with the delay between emotion signals generation: the designer only needs to specify the minimum time each rule has too wait until it may be used again.

3.3.3 Built In Appraisal Modules

The Appraisal Compiler offers two Appraisal Modules that may be used in any application without requiring any further specification. These Appraisal Modules are used in combination with the Em-PSys SAFIRA component. They may also be modified at will by the designer to suit her or his needs. The Appraisal Compiler does not force the agent designer to use any of the Appraisal Components provided. Agent designers may use the Appraisal Compiler to create other Appraisal Components.

The offered Appraisal Components detect domain-independent emotion eliciting conditions that may be used by default. Designers may specify other domain-specific emotion eliciting conditions. When an emotion eliciting condition is detected, the Appraisal Component generates the appropriate emotion signal and sends it to the components that have subscribed to it. If desired, the generation of emotion signals relative to domain-independent emotion eliciting conditions may be overridden by the generation of emotion signals relative to specified domain-specific emotion-eliciting conditions.

The two independent appraisal modules are the Expectations Evaluator Module and the Unpredicted Actions Module.

The Expectations Evaluator Module is responsible for warning the agent that some actions did not produce the expected results. The intensity of the emotional signals should depend on the importance of the goal that should have been fulfilled by the action that failed to achieve it, and the effort the agent has already spent to achieve the goal (the effort variable was also pointed out by Ortony et al., whereas the goal importance is indicated by Gratch). A heuristic
measure of the importance of the goal in the Salt & Pepper architecture can be its current activation level (in the agent long-term memory). If the importance is not received from the appraisal module the intensity is computed as described in section 3.3.2. The Appraisal Component subscribes to information from the Em-PSys Component in order to receive information about the actions, the instant of time when the action should have been executed and the expected duration of action execution.

The Unpredicted Actions Module is responsible for advising the agent that some unpredicted actions (not scheduled by the Em-PSys Component) where executed. The intensity of this emotional signal will be a pre-determined constant value. The Appraisal Component must subscribe to information from the Effector Components in order to know which actions were requested to be executed and which component has requested their execution. The action requester is necessary to determine whether or not the action was requested by the Em-PSys Component.

3.4. SCHEMA CLASS BUILDER

In the SAFIRA project, inter-component communication is made through XML messages. The Schema Class Builder tool provides a set of functionalities that allow the conversion between XML and Java. The purpose of this tool is to automatically extract Java objects from XML strings, and vice-versa.

The Schema Class Builder is based on the principle that an XML Schema definition is a signature for a class of XML objects. Therefore, it can be converted into a Java class that is a signature for Java objects. The Schema Class Builder is a code conversion/generation mechanism that takes an XML schema definition and outputs a Java class. The output class is the Java representation of the XML schema. According to the XML schema definition, the Java class is filled with the corresponding attributes (Code-Block 21). A more detailed description of the conversion model is given in Section 3.4.1.
Given the following XML schema:

```xml
<schema>
  <complexType name="Address">
    <all>
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="number" type="long"/>
      <element name="zip-code" type="long"/>
      <element name="city" type="string"/>
    </all>
  </complexType>
</schema>
```

The generated class would be:

```java
public class Address{
  public String name;
  public String street;
  public Long number;
  public Long zip_code;
  public String city;
  ...
}
```

**Code-Block 21**

In each created class a set of methods is provided that allow the conversion between XML and Java. There are two sets of methods, from XML to Java and from Java to XML (Code-Block 22).

The first set is composed by a method named `tagToObject`. This method is a class method that takes a Java representation of an XML tag and converts it into a new object of the declaring class. The Java representation of the tag is the output of an XML parser. If the XML tag passed as an argument is a tag that cannot be mapped to this class, an exception is thrown.

The second set is composed by two methods, both named `toTag`. Both methods are instance methods. One, that takes no arguments, converts this object into a Java representation of a XML tag. The other method takes a string that is used to name the tag, and converts the attributes of this object to elements of the tag.

Each method described above, performs validations according to the XML schema. XML schema has some elements that are implicitly associated with restrictions (`all, sequence` and `choice` groups), and others that explicitly restrict the values of the XML elements (`restriction` elements). When a Java class is generated from an XML schema, these validations are converted into conditions and placed in the conversion methods.

Given the following XML schema:

```xml
<schema>
  <complexType name="Address">
    <all>
      <element name="name" type="string"/>
    </all>
  </complexType>
</schema>
```
The usage of the generated methods would be:

```java
Address address = new Address();
address.name="John Smith";
address.street="Rainbow Street";
address.number=new Long(777);
address.zip_code=new Long(7856);
String address_tag = address.toTag().toString();
System.out.println("1. 
" + address_tag);
Tag t = xmlparser.parse(address_tag);
Address converted = Address.tagToObject(t);
System.out.println("2. 
John Smith lives on Rainbow Street, number 777 with zip-code 7856 
" + converted.street + ", number " + converted.number + 
" with zip-code " + converted.zip_code);
```

The output of this Java program would be:

1. `<address>
   <name>John Smith</name>
   <street>Rainbow Street</street>
   <number>777</number>
   <zip-code>7856</zip-code>
</address>`

2. `John Smith lives on Rainbow Street, number 777 with zip-code 7856`

### 3.4.1 Conversion Model

As described at the beginning of section 0, the Schema Class Builder converts an XML schema definition into a Java class that is placed in a Java file.

The XML schema definition is a description of classes of XML tags. Therefore it has a set of reserved words that operate as identifiers of the properties that a representative class description should have. Schema Class Builder does not implement all of the XML schema definition features. The implemented features are: `complexType`, `simpleType`, `element`, `all`, `sequence`, `choice`, `complexContent`, `simpleContent`, `extension`, `restriction`, `abstract`, `enumeration`, and `attribute`.

#### 3.4.1.1 From schema to classes

The global principle of conversion is as follows (Code-Block 23):

- XML types, including `complexType` and `simpleType`, are converted into Java classes.
• XML root elements, including the schema’s child element, are converted into Java classes.

• XML annotations are discarded.

• XML application information, including appInfo, is passed in the header of the Java class file.

The XML Schema:

```xml
<schema>
  <annotation>
    <appInfo>
      package hello.bye;
      import java.util.*;
    </appInfo>
  </annotation>
  <complexType name="Bye">
    ...
  </complexType>
  <element name="hello" type="Bye" />
</schema>
```

would be converted into:

Bye.java:
```java
package hello.bye;
import java.util.*;
public class Bye{
  ...
}
```

hello.java:
```java
package hello.bye;
import java.util.*;
public class hello{
  ...
}
```

---

### 3.4.1.2 From XML types and type contents to classes

XML types are converted into Java classes. The principle of conversion is as follows (Code-Block 24):

- XML type elements are converted into Java attributes.
- XML types defined as abstract are converted into Java classes with no attributes.
- XML groups, including all, sequence and choice, are defined as conditions inside the conversion methods.
- XML content, including complexContent and simpleContent is converted into:
a Java class extending another Java class, if the content is defined as an
extension.

a Java class, if the content is defined as restriction and the restriction is defined
as a condition in the conversion methods.

Given the following XML schema:

```xml
<schema>
  <complexType name="Zip" abstract="true" />
  <complexType name="Address">
    <sequence>
      <element name="name" type="string" />
      <element name="street" type="string" />
      <element name="number" type="long" />
      <element name="zip-code" type="Zip" />
    </sequence>
  </complexType>
  <complexType name="ZipEurope">
    <simpleContent>
      <extension base="Zip">
        <choice>
          <element name="zip-pt" type="string" />
          <element name="zip-uk" type="long" />
          <element name="zip-fr" type="float" />
        </choice>
      </extension>
    </simpleContent>
  </complexType>
</schema>
```

The output of compiling this schema with Schema Class Builder would be three generated files:

**Zip.java:**

```java
public class Zip{
    public Zip(){
    }
    public Zip(Zip z){
        this();
    }
    public Tag toTag() throws XmlConvertException{
        return new Tag(new Token("Zip"));
    }
    public Tag toTag(String s) throws XmlConvertException{
        return new Tag(new Token(s));
    }
    public static Object tagToObject(Tag t) throws JavaConvertException{
        return new Zip();
    }
}
```
Address.java:

```java
public class Address{
    public String name;
    public String street;
    public Long number;
    public Zip zip_code;
    public Address(){
    }
    public Address(Address a){
        this.name = a.name;
        this.street = a.street;
        this.number = a.number;
        this.zip_code = a.zip_code;
    }
    public Address(String name, String street, Long number, Zip zip_code){
        this.name = name;
        this.street = street;
        this.number = number;
        this.zip_code = zip_code;
    }
    public Tag toTag() throws XmlConvertException{
        Tag ret = new Tag(new Token("Address"));
        if((this.name == null) || (this.street == null) ||
        (this.number == null)) || (this.zip_code == null))
            throw new XmlConvertException("This tag, " +
            this.getClass().getName() + ", is a sequence group. There are
            some values that are not allowed!");
        else{
            /*convert values*/
        }
        return ret;
    }
    public Tag toTag(String s) throws XmlConvertException{
        Tag ret = new Tag(new Token(s));
        /*same as in the previous method*/
    }
    public static Object tagToObject(Tag t) throws JavaConvertException{
        Address ret = new Address();
        /*convert elements*/
        if((ret.name == null) || (ret.street == null) || (ret.number == null)
            || (ret.zip_code == null)) throw new JavaConvertException();
        return ret;
    }
}
```

ZipEurope.java:

```java
public class ZipEurope extends Zip{
    public String zip_pt;
    public Long zip_uk;
    public Float zip_fr;
    public ZipEurope(){
    }
    public ZipEurope(ZipEurope z){
        this.zip_pt = z.zip_pt;
        this.zip_uk = z.zip_uk;
        this.zip_fr = z.zip_fr;
    }
}
public ZipEurope(String zip_pt, Long zip_uk, Float zip_fr){
    this.zip_pt = zip_pt;
    this.zip_uk = zip_uk;
    this.zip_fr = zip_fr;
}

public ZipEurope(Zip z, String zip_pt, Long zip_uk, Float zip_fr){
    super(z);
    this.zip_pt = zip_pt;
    this.zip_uk = zip_uk;
    this.zip_fr = zip_fr;
}

public ZipEurope(Zip z){
    super(z);
}

public Tag toTag() throws XmlConvertException{
    Tag ret = super.toTag("ZipEurope");
    if(((this.zip_pt != null) && ((this.zip_uk != null) ||
        (this.zip_fr != null))) || ((this.zip_uk != null) &&
        (this.zip_pt != null) || (this.zip_fr != null)) ||
        ((this.zip_fr != null) && ((this.zip_pt != null) ||
        (this.zip_uk != null))))
        throw new XmlConvertException("This tag, " +
        this.getClass().getName() + ", is a choice group. There
        are some values that are not allowed!");
    else{
        /*convert elements*/
    }
    return ret;
}

public Tag toTag(String s) throws XmlConvertException{
    Tag ret = super.toTag(s);
    /*same as in previous method*/
}

public static Object tagToObject(Tag t) throws JavaConvertException{
    ZipEurope ret = new ZipEurope((Zip)Zip.tagToObject(t));
    Tag tag = t;
    /*convert elements*/
    if(((ret.zip_pt != null) && ((ret.zip_uk != null) ||
        (ret.zip_fr != null))) || ((ret.zip_uk != null) &&
        (ret.zip_pt != null) || (ret.zip_fr != null)) ||
        ((ret.zip_fr != null) && ((ret.zip_pt != null) ||
        (ret.zip_uk != null))))
        throw new JavaConvertException();
    return ret;
}
}

Code-Block 24

In this example, three classes are generated by the Schema Class Builder: Zip; Address; and
ZipEurope. The Zip class was declared as abstract. Therefore, it has no attributes and all its
methods are merely initialisation methods.

The Address class is a representation of an ordinary XML schema class. Its XML
schema elements were converted into attributes and the sequence group was converted into a
Java condition. This condition is placed in the conversion methods and validates all attributes as
not null (the sequence group does not allow for the omission of any element).

The ZipEurope class is an extending class of Zip. The main difference from the
Address class is that the super class of ZipEurope, Zip, is always initialised in the conversion
methods. *Choice* groups were converted into Java conditions placed in the conversion method, which validates only one attribute as not null (the *choice* group allows the appearance of only one of the defined *elements*).

### 3.4.1.3 From schema root elements to classes

Schema root elements, i.e., the element tags defined as children of the schema tag, are converted into Java classes. These classes have only one attribute, and all their conversion methods simply invoke the conversion methods of the element type (Code-Block 25).
Taking the XML schema used in Code-Block 24 as an example, let us add a schema root element:

```xml
<schema>
  <complexType name="Zip" abstract="true" />
  <complexType name="Address">
    <sequence>
      <element name="name" type="string" />
      <element name="street" type="string" />
      <element name="number" type="long" />
      <element name="zip-code" type="Zip" />
    </sequence>
  </complexType>
  <complexType name="ZipEurope">
    <simpleContent>
      <extension base="Zip">
        <choice>
          <element name="zip-pt" type="string" />
          <element name="zip-uk" type="long" />
          <element name="zip-fr" type="float" />
        </choice>
      </extension>
    </simpleContent>
  </complexType>
  <element name="MyAddress" type="Address" />
</schema>
```

The output of compiling this schema with the Schema Class Builder would be four generated files, three of them identical to those described in Code-Block 24, and the fourth as follows:

**MyAddress.java:**

```java
public class MyAddress{
  public Address MyAddress;
  public MyAddress(){
    this.MyAddress = new Address();
  }
  public MyAddress(Address a){
    this.MyAddress = a;
  }
  public Tag toTag() throws XmlConvertException{
    return this.MyAddress.toTag("MyAddress");
  }
  public Tag toTag(String s) throws XmlConvertException{
    return this.MyAddress.toTag(s);
  }
  public static Object tagToObject(Tag t) throws JavaConvertException{
    return new MyAddress((Address)Address.tagToObject(t));
  }
}
```

**Code-Block 25**
3.4.2 Implementation issues

The Schema Class Builder was implemented using the JavaCup and JLex parser generation tools. JavaCup and JLex are the Java counterparts of the well-known parser generators lex and yacc. These parser generators are LR parsers (Left Right parsers) that are based on the principle of precedence of the defined symbols. JLex creates a lexical analyser that validates the correctness of the symbol character sequences. In other words, it analyses the input text and checks whether all of the words contained are part of the grammar symbol set. JavaCup generates a syntax analyser that validates the sequence of the grammar symbols, i.e., whether the sequence of symbols contained in the input text is valid in the grammar rule set.

The JavaCup parser generator accepts an input file containing a set of grammar rules. These rules are specified as follows:

\[
\text{symbol ::=}
\begin{align*}
\text{rule1} & \{ : \text{action1} : \} \\
\text{rule2} & \{ : \text{action2} : \} \\
& \quad \quad \cdots \\
\text{ruleN} & \{ : \text{actionN} : \}
\end{align*}
\]

The action attached to each rule determines what the generated parser will do whenever rule is satisfied by the current input string. The action is a set of Java instructions and is copied to the generated parser. In the Schema Class Builder, each of the actions attached to each grammar rule creates a string according to the XML schema rule (Code-Block 26).

```
xmlelement_definition ::= 
    element name:n type:t slash
    {
        String s = new String();
        if(XMLSchemaUtilities.basicType(t)){
            t = XMLSchemaUtilities.getBasicType(t);
        }
        s = "public " + t + " " +
            XMLSchemaUtilities.getValidString(n) + ";\n";
        RESULT = new XMLSchemaElement(n, s, t);
    } | 
    \dots
};
```

**Code-Block 26**

The grammar rules for XML schema definitions are recursive, i.e., XML schema symbol A, may have symbol B as child and symbol B may have symbol A as child. Due to this peculiarity of the XML schema grammar, the implementation of three JavaCup based parsers was necessary: one that generates the Java class header, attribute definition and constructor methods; one that generates the XML to Java conversion methods; and one that generates the Java to XML conversion methods. The strings generated by these parsers are concatenated and written to a file (Code-Block 27).
3.4.3 Using the generated classes

To provide a user-friendly interface with the generated classes, an interface class was implemented. The XmlInterface class has a set of methods that simplify the conversion of XML and Java.

After the creation of an XmlInterface object, the user must provide the information about which Java classes will be used to translate the XML or Java objects. This mechanism is necessary so that a relation can be made between an XML tag and a given generated class.

Once the initialisation has been performed, the user may use the methods `getObject` and `getTag`. The `getObject` method receives a string and tries to convert it into a corresponding
Java object. The `getTag` method receives a Java object and tries to convert it into an XML string.

The usage of this is class is described in the following example:

```java
import java.lang.*;
import java.util.*;
import org.adetti.xml.xmltypes.*;
import org.adetti.exceptions.*;
import org.adetti.xml.*;

public class Main{
    public static void main(String args[]){
        XmlInterface xi = new XmlInterface();
        xi.assertSchema("Message"); // the name of the class that translates all tags named message
        String s = "<message> <sender>John</sender> <receiver>Peter</receiver> <content>hello!!!</content> </message>";
        Message msg_java = (Message) xi.getObject(s);
        String msg_xml = xi.getTag(msg_java);
        System.out.println(msg_java.sender + " says " + msg_java.content + " to " + msg_java.receiver);
        System.out.println(msg_xml);
    }
}
```

the result of running this program is:

```
John says hello!!! to Peter
<Message><sender>John</sender><receiver>Peter</receiver><content>hello!!!</content></Message>
```

Code-Block 28
3.5. THE INTERACTION MANAGER

Since SAFIRA has adopted a component-based approach to agent development, it is useful to have a general-purpose module that can handle most of the interaction work for the component developer. The Interaction Manager is this very module.

3.5.1 Overview of the Interaction Manager

The Interaction Manager is composed of a set of modules, where each module manages a different part of the interaction process (Figure 10). The Interaction Module is its main module. It manages the creation of the other modules and is responsible for registering and querying the CRS (section 3.5.3) about service information. The Server Socket Listener is the module that accepts connections from other components (section 3.5.4). The Interaction Receiver is the module responsible for receiving and processing messages from other components (section 3.5.5). The Interaction Sender is the module responsible for answering and sending messages to other components (section 3.5.4).

Each of these modules runs in separated threads, sharing data. This data is stored in a class called Message Box (section 3.5.2). This class includes the data that is updated and required by all of the modules, including received messages; messages scheduled to be sent; the component identification; and the list of available services.

The Interaction Manager uses and is partially based on the Schema Class Builder tool (section 3.4). It uses the classes created by the Schema Class Builder to transform the received XML messages into Java objects. These Java objects are passed to handlers that process each type of XML tag or Java object. The handler methods are defined by the user (Figure 11).
3.5.2 The MessageBox

The MessageBox is a class that contains a set of data and functionalities that allow the modules making up the Interaction Manager to share information, synchronise, and correctly receive and reply to XML messages. It also gives the user/programmer that uses the Interaction Manager some control over the interaction flow inside the system.

Different types of functionalities compose this class: service methods; messaging methods; ontology methods; and handling methods.

Basically, the MessageBox class provides initialisation methods that will provide the Interaction Manager with the information necessary to work properly.

3.5.2.1 Handling methods

The service methods allow the user to register services with the Central Registry Service. This registration makes the component services visible to all components and enables the querying or subscription to these services (Code-Block 29).

```java
MessageBox mb = new MessageBox();
mb.registerService("request-time");
```

Code-Block 29
### 3.5.2.2 Messaging methods

The messaging methods allow the user to schedule messages to be sent to other components. This scheduling is not direct, i.e., the user does not choose to which component to send the message. Instead, the Interaction Manager verifies if there is some information about a provider for the corresponding service. If it exists, the message is sent. Otherwise, the Interaction Manager queries the CRS for a provider, and the message is sent when the reply is received (Code-Block 30).

```java
MessageBox mb = new MessageBox();
mb.ScheduleMessage("<request-time>today</request-time>" );
```

**Code-Block 30**

### 3.5.2.3 Ontology methods

The SAFIRA components have a restricted and known ontology. Component developers know which services will be available, and which services their components will provide. The ontology methods allow the developer to pass the list of XML tags that will be exchanged to the Interaction Manager (Code-Block 31).

```java
Given the schema:
</schema>  
<annotation>  
  <appInfo>package mycomponents.timecomponent;</appInfo>  
</annotation>  
<element name="Request-Time" type="string" />  
</schema>

Using the Schema Class Builder, the output is the Java file Request_Time.java:

```java
package mycomponents.timecomponent;
public class Request_Time{
  ...
}
```

to the following provides the Interaction Manager with the ability to process and handle an XML tag of type request-time:

```java
MessageBox mb = new MessageBox();
mb.AddSchema("mycomponents.timecomponent.Request_Time");
```

**Code-Block 31**

Since the Interaction Manager uses the Schema Class Builder, the format of the XML tag information is a string with the full name of the generated class that will be used to convert XML
to Java and vice-versa. The purpose of this functionality is just to associate XML tags to Java classes, in order to be able to convert an incoming XML string into a Java object.

### 3.5.2.4 Handling methods

The handling methods are used to store information about the handlers that will process a given incoming XML message.

The information passed into the Interaction Manager is composed of the name of the XML tag to process; the name of the handler that will process it; the Java class where the handler is stored; and a flag, which describes whether the tag is to be processed forever or only once - i.e., whether the tag represents a subscription for information or just a query (Code-Block 32). These handlers take on the form of Java methods and are defined by the component developer following a strict format (Code-Block 33). The methods must be static, must not return valid information (anything returned by the method will be discarded), and must accept the following four arguments:

- the received message, converted to Java object
- an object, which can be any object that the developer may need to use inside the method
- the identification of the sender of the message
- an InteractionData object, a class that provides the same functionalities as the MessageBox class.

```java
MessageBox mb = new MessageBox();
Vector knowledgebase = new Vector();
mb.SubscribeSchema("request-time", "handleRequestTime",
    "mycomponents.timecomponent.Handler", knowledgebase,
    NON_PERSISTENT);
```

**Code-Block 32**
Given the following Java code:

```java
MessageBox mb = new MessageBox();
Calendar knowledgebase = Calendar.getInstance();
mb.SubscribeSchema("request-time", "handleRequestTime",
    "mycomponents.timecomponent.Handler", knowledgebase,
    NON_PERSISTENT);

the handler for the "request-time" XML tag, "handleRequestTime",
would have to be placed in the Handlers Java class, defined as:
package mycomponents.timecomponent;

public class Handlers{
    public static void handleRequestTime(Request_Time req, Object kb,
        ComponentId sender,
        InteractionData idata){
        Calendar cal = (Calendar) kb;
        Date date = cal.getDate();
        idata.scheduleMessage(sender, date);
    }
}
```

Code-Block 33

The arguments received by the methods are initialised and filled in by the Interaction Manager when some message arrives.

3.5.3 The Interaction Module

The Interaction Module is the central nervous system of the Interaction Manager. Basically, it creates and assigns the threads that process the different parts of the interaction.

The Interaction Module is initialised as a normal Java object, inside any class. The constructor method for this class takes the following arguments: the component name; the MessageBox; and a boolean flag representing the type of interface of the Interaction Manager (true = graphical, false = text) (Section 3.5.7). The Interaction Module can be started by invoking the `Start` method (Code-Block 34).

```java
MessageBox mb = new MessageBox();
InteractionModule im = new InteractionModule("MyComponent", mb, true);
im.Start();
```

Code-Block 34

When started, the Interaction Module connects to the CRS in order to register the component services and request information about needed services. If there is any service to register (Code-Block 29), the Interaction Module sends a message to the CRS, providing the necessary information to make the service available (Figure 12).
If there is any scheduled message (Code-Block 30), the Interaction Module checks whether it has any information about providers for the desired services. If it does, the message is sent. Otherwise, a query message is sent to the CRS. When the reply from the CRS arrives, the Interaction Manager creates new instances of the Interaction Sender and the Interaction Receiver to manage the connection with the new provider (Figure 13).

The creation of two distinct threads for each connection is indispensable for a proper interaction, even though it may seem unnecessary and processor consuming at first sight. The main reason to take this approach is that a single thread per socket (or a single thread for all sockets) would break the interaction’s flow by suspending the processing in the socket reading. Inevitably, that would delay or even suspend the sending of any scheduled messages. Another option would be not to suspend the processing in the socket reading, but that would certainly be even more processor consuming than the present solution.

![Figure 12](image-url)
3.5.4 The Server Socket Listener

The Server Socket Listener is a simple thread that accepts connections from other components. When some connection request arrives, the Server Socket Listener creates new instances of the Interaction Sender and Interaction Receiver, to manage the new connection (Figure 14).

![Figure 13]

![Figure 14]
3.5.5 The Interaction Receiver

The Interaction Receiver is the module responsible for receiving messages from other components. When a message is received, the Interaction Receiver consults the MessageBox to retrieve the handler information addressed to the XML tag received (Figure 15).

If the XML tag is marked as **NON_PERSISTENT**, the handler is immediately executed and the message discarded. If the XML tag is marked as **PERSISTENT**, the message is stored in a list of persistent messages and the handler is executed in each step of the Interaction Sender cycle (Section 3.5.6). The handler is executed with the received message and the sender identification as arguments (Section 3.5.2.4).

The Interaction Receiver receives its messages on a TCP/IP socket. Therefore, it blocks in the socket reading in order to avoid wasting CPU time.

![Figure 15](image)

3.5.6 Interaction Sender

The Interaction Sender module is responsible for replying and sending messages to other components. Due to its processor intensive nature, special care was taken in the development of this module.

When a thread is running in an infinite loop and it has no blocking mechanism, it consumes large amounts of processor time. The Interaction Receiver and the Server Socket
Listener both have automatic socket related mechanisms that block the processing. The Interaction Sender uses a UNIX-like semaphore based mechanism to block itself. Basically, at the end of each loop step, the Interaction Sender blocks itself until a message is scheduled or the developer explicitly unblocks it.

The developer may unblock the Interaction Sender by invoking the MessageBox methods that manage the semaphore system. The developer can use the `dataChanged` and `dataChangedAndListen` methods. The first one unblocks all the Interaction Senders in order for them to try to reply to received messages. The second one has the same effect, with the added ability of blocking the invoking thread until all the Interaction Senders are blocked again. This latter mechanism is extremely useful to synchronise removal operations.

When unblocked, the Interaction Sender processes the list of persistent messages, executing the handlers associated to each one. Afterwards, it sends all scheduled messages to the component attached to its socket (Figure 16).

![Figure 16](image)

### 3.5.7 The User Interface

The Interaction Manager provides an interface that allows the user to monitor the component’s activity. The window is composed of the following elements (Figure 17):

1. a log box
2. a received messages box
3. a sent messages box
Figure 17

The log box allows the user to view a continuous log of the Interaction Manager’s activity (Figure 18).

Figure 18

The received messages box lists the received messages. By clicking on an item of this list (Figure 19), the user may view the full message in a separate window (Figure 20).
The sent messages box allows the user to see the sent messages in a list (Figure 21). By double clicking on an item of this list, the respective message is fully shown in a separate window (as in Figure 20 for received messages).
3.5.8 All together now

In this section, an example is given on how to use the Interaction Manager.

Let us take the following component as an example:

```java
public class AppMonga {
    public AppMonga() {
        Vector kbout = new Vector();
        Vector kbin = new Vector();
        this.mb = new MessageBox();
        this.im = new InteractionModule("AppMonga", this.mb, IMDefinition.ASSYNCHRONIZED, this.gui);
        this.mb.registerService("SubscribeEmotionSignal");
        this.mb.registerService("SubscribeEmotionSignal");
        this.mb.ScheduleMessage(new RequestWorldObject("all", "all");
    }

    public static void Assert(WorldObjectList wol, ComponentId sender, Object kb, InteractionData id) {
        Vector kbin = (Vector) kb;
        System.out.println("Received a WorldObjectList");
        kbin.add(wol);
    }

    public static void Evaluate(SubscribeEmotionSignal sub, ComponentId sender, Object kb, InteractionData id) {
        Vector kbout = (Vector) kb;
        System.out.println("Received a SubscribeEmotionSignal");
        id.scheduleMessage(kbout.remove(0));
    }
}
```

**Code-Block 35**
The component described in Code-Block 35 provides two services – \texttt{SubscribeEmotionSignal}\ and \texttt{QueryExpectedResults}\ – and schedules one message, for the service \texttt{"RequestWorldObject"}. It also subscribes the handling of two types of message: \texttt{WorldObjectList}\ and \texttt{SubscribeEmotionSignal}.

When the user clicks the start button, the Interaction Module contacts the CRS, in order to register the two services provided by \texttt{AppMonga}, and to query for the service that the \texttt{AppMonga} requires (Figure 22 through Figure 24).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure22.png}
\caption{Figure 22}
\end{figure}
Figure 23

Sender: AppMonga@natassja.adetti-linhat.org
Receiver: DummyCRS@natassja.adetti-linhat.org
Message ID: 0
Time Stamp: 12/05/2002 00:36:44
Content:

RegisterService:
component:
  component_id:
    component_name: AppMonga
    component_address: natassja.adetti-linhat.org
    component_port: 431049
  service_name: RequestWorldObject

Figure 24

Sender: AppMonga@natassja.adetti-linhat.org
Receiver: DummyCRS@natassja.adetti-linhat.org
Message ID: 3
Time Stamp: 12/05/2002 00:41:52
Content:

QueryService:
  service_name: RequestWorldObject
When a component registers with the service required by AppMonga, the CRS replies with the component identification. Then AppMonga sends the message to the service provider (Figure 25 through Figure 27).

Figure 25
**Figure 26**

Sender: DummyCRS@natassja.adetti-linha4.org
Receiver: AppMongo@natassja.adetti-linha4.org
Message ID: 0
Time Stamp: 12/05/2002 00:38:28

Content:

ServicePerformed
  sensor_name: RequestVandObject
  provider:
    ComponentInfo:
      component_name: VisionSensor
      component_address: natassja.adetti-linha4.org
      component_port: 42054

---

**Figure 27**

Sender: AppMongo@natassja.adetti-linha4.org
Receiver: VisionSensor@natassja.adetti-linha4.org
Message ID: 4
Time Stamp: 12/05/2002 00:38:28

Content:

RequestVandObject:
  object_type: all
  object_id: all
4 EMOTION EVOLUTION COMPONENT

4.1. BACKGROUND

The goal of the emotion evolution component of the SAFIRA toolkit is to develop a lightweight, reusable support for emotional development. By emotional development, we mean lasting changes in an agent’s behaviour, abilities, or structure that are related to emotion and occur over a longer time period, like that which occurs in the development of children’s personality. This contrasts with ordinary moment-to-moment fluctuations in emotional state, which also lead to changes in agent behaviour, but occur on a shorter time scale and are not permanent. Affective agents can use emotional development software as a support for an emotion-based relationship with the user that develops over time, becoming richer and/or adapting to the specifics of the agent’s relationship and history with that particular user.

Development is not well-understood, technically. Our goal for the SAFIRA toolkit is to generate a first tool for development, which can support emotion-based change over time in a demonstrator and for users of the SAFIRA toolkit. The tool should be simple, easy-to-use, and not constraining (general and flexible).

In Deliverable 4.1, we described the background research done for this component, which used as an example domain the depth-psychological analysis of emotion-linked changes in the development of children’s drawing. Research in this area describes how children express emotions in their drawings through graphical forms (such as circles, suns, and head-foot shapes) and attributes such as colour, pressure, speed, and placement on page, and how these forms and their meanings change over time. This background research revealed the following aspects of emotion-linked development:

1. Development is deeply interlinked with all aspects of an agent’s structure. Developmental changes can occur in many aspects of an agent’s cognition and functioning, and can also be triggered by many of its aspects, including the agent’s biological development as well as its intellectual development, historical experiences and relationships with other people. Therefore:
   a. A general model of development cannot directly model the specific triggers and effects of development, since it would be impossible to enumerate them fully, at least at the current state of the art and perhaps at all.
   b. Development must potentially be inter-linkable with many different components of an agent’s functioning.

2. Development is often described as movement through a set of different states. Agents are said to move from one kind of behaviour or functionality to another. This may not be an attribute of development itself, but may instead be an artefact of how we most easily explain change; it may simply be easier to explain it as a change from one discrete state to another, rather than as a continuous and undifferentiable process of gradual change. Despite the fact that we cannot say with certainty that a state-space model is the most accurate description of development, movement through state space may form the basis
of an easily-understood software model of development which can support a variety of state-space-based theories of development.

These aspects form the basis of the developed algorithm and software for emotional development. Specifically, the developmental model is designed to be a minimum-commitment model that can provide support to many different components of an agent's mind. The model is based on a data structure, the developmental graph, which represents the possible developmental states of the agent and of the triggers for change between developmental states. Using this graph as a basis, we develop a developmental algorithm, which keeps track of the current set of developmental states, as well as the triggers that may cause those states to change.

4.2. THE DEVELOPMENTAL GRAPH

Given that development is frequently spoken of as a movement across developmental states, a natural representation for development is as a graph structure. The states of the graph represent behaviors or functionalities that are available to the agent at a particular point in development. The edges of the graph represent moments of developmental change, or ways in which the agent can develop from one state to another. An example of what such a graph might look like for the early stages of children’s drawings is shown in Figure 28.
We now formalize this intuition. The states of the graph represent properties (behaviours or functionalities), which the agent has. Since agents may have multiple functionalities, agents may simultaneously occupy multiple developmental states. The initial states of the graph represent the functionalities with which the agent begins. Each edge of the graph is labelled with a condition that, if the initial state is active, will trigger the final state to be active as well. If a state in the graph has more than one incoming edge, all edges must be triggered before the state becomes active (i.e., incoming edges form an "and"-combination).

The informal notion of developing from one state to another can have two different meanings: it may mean that the second state is active in addition to the first state, or it may mean that the first state ends and the second state begins. In our formal representation, we use the first meaning; the second kind of development is represented by labelling states with a post condition, which, when triggered, causes the state to no longer be active.

For example, in the developmental graph shown in Figure 29, the agent begins with two functionalities, represented by State 1 and State 2. From State 1, the agent can add State 3.
(which might represent a new, more complex form) when condition 1 occurs. When conditions 2 and 3 occur, the agent will add State 4. When both State 3 and State 4 are occupied, and conditions 4 and 5 occur, the agent will add State 5. Since this graph contains no post conditions, the states will all be active, once triggered.

![Figure 29: An example developmental graph](image)

### 4.3. The Developmental Algorithm

The goal of the Emotional Development software model, as outlined previously, is to provide support within an agent’s mind for keeping track of the developmental state of the agent. The software runs as a separate thread, monitoring conditions that may cause the developmental state to change and answering queries about the current emotional state.

The developmental algorithm is based on the developmental model described in the previous section. Accordingly, we limit our definition of emotional development to the following: emotional development is the process of moving through a graph of developmental states, either as a result of the passage of time or in consequence of conditions which occur outside of the developmental model. Multiple states may be simultaneously occupied. Associated with each state is a property that holds in this state. These properties are purely abstract, which means that the development component does not ‘know’ anything about these properties; for it, they are only distinct labels. The association of a label with actions in or outside the mind of the agent is done outside the scope of the component.

Neither is the developmental model aware of the ‘meaning’ of the conditions that can cause changes to developmental state. Instead, it keeps track of the names of the conditions that currently must be monitored. Actually evaluating those conditions lies again outside the scope of this component.

#### 4.3.1 Data Structure

The Developmental Model consists of the following components:

- A developmental graph, defined by the following pieces:
A set of developmental states, where a single state \( N(p) \) is described in terms of

- A property \( p \), as described previously
- A predicate \( \text{activated()} \), which tests if the state has been activated. A state becomes activated when, for all developmental changes \( DC(P,N) \) (see below) leading to that state, at some point it has been the case that \( P \) has been active and the condition associated with that developmental change has simultaneously evaluated to true.
- A predicate \( \text{active()} \), which tests if the state is currently active. A state is active if it has been activated, and its post condition has not been fulfilled.
- A predicate \( \text{postcondition()} \), which is evaluated outside of the developmental model component, and which determines when the property describes by the developmental state no longer holds.

One state \( N_i \) is distinguished as the initial state. It is always activated.

- A set of developmental changes \( DC(S_i,S_f) \), which link developmental states into the graph. Each change has associated with it
  - An initial state \( S_i \)
  - A final state \( S_f \)
  - A predicate \( \text{condition()} \), which is evaluated outside the developmental model, and intuitively determines when it is time to change from the initial to the final state.

- A set of run-time generated data structures as follows:
  - ActiveStates, the set of developmental states which are currently active
  - PotentialDevelopmentalChanges, the set of developmental changes which lead from a state in ActiveStates to a state that has not been activated
  - ActivelyTestedConditions, the set of conditions of developmental changes in PotentialDevelopmentalChanges
  - ActivelyTestedPostconditions, the set of post conditions of states in ActiveStates

### 4.3.2 Algorithm

Generally speaking, the developmental model must keep track of the conditions which currently must be monitored by the other components. Whenever such a condition is triggered, the system
updates the set of current developmental states. Whenever a request comes for the current set of developmental states, the system returns it.

At initialisation, the system reads in the developmental graph provided by an external source, and initialises the ActiveStates set with the initial state $N_i$. The other run-time data structures are initialised as described above. Then, for each condition in ActivelyTestedConditions and ActivelyTestedPostconditions, the system sends out a request for notification when that condition becomes true.

The system then waits for incoming messages. These messages are processed as follows:

- When the system is notified that a condition in ActivelyTestedConditions for $DC(N_1,N_2)$ has become true, the number of preconditions for state $N_2$ is reduced by one. If the number of preconditions for state $N_2$ is no more than zero, it is added to ActiveStates, and the other run-time data structures are updated accordingly.

- When the system is notified that a condition in ActivelyTestedPostconditions for state $N$ has become true, state $N$ is removed from ActiveStates, and the other run-time data structures are updated accordingly. This potentially includes withdrawing previously stated requests for condition confirmation.

- When the system is requested to return the current set of DevelopmentalStates, it does so.

### 4.4. IMPLEMENTATION

The Developmental Model consists of the two pieces described in the algorithm above: (1) a Developmental Graph, or static graph which represents the developmental states; and (2) the Developmental Model, which keeps track of the currently active set of states, and the conditions which are related to them. These are represented in two separate classes.

To use the developmental model, a user would first create a DevelopmentalGraph corresponding to development in his or her target application. Then, he or she would create a DevelopmentalModel that uses this graph.

#### 4.4.1 Developmental Graph

The DevelopmentalGraph class provides functionality to set up a developmental graph as needed for the target application.

```java
public DevelopmentalGraph()
```

Creates a new developmental graph, which can then be initialised with a particular graph’s states and links.

```java
public void addState(String name)
```
Add a state to the graph. The parameter is the name of the state (equivalent to the property the state represents). This state is considered not to be an initial state.

```java
public void addInitialState(String name)
```

Add an initial state to the graph. The parameter is the name of the state.

```java
public void addChange(String initialName, String finalName, String transitionCondition)
```

Add a developmental change to the developmental graph. A change goes from an initial state specified by initialName to a final state specified by finalName. These states must be defined prior to calling addChange, i.e. if the states specified have not yet been defined, addChange will fail to create a developmental change and will print a warning message. The transitionCondition is a string representing the conditions under which the transition should take place. It will be output by the system when it is time for that condition to be monitored, i.e. should be understandable to the program which is in charge of monitoring conditions for the Developmental Model.

```java
public void print()
```

Print the developmental graph.

### 4.4.1.1 Example

This example shows how the DevelopmentalGraph class can be used in practice. The next section will show how to use the graph in order to have a running developmental model.

```java
private static DevelopmentalGraph makeGraph() {
    DevelopmentalGraph graph = new DevelopmentalGraph();
    /* Add states to graph */
    graph.addInitialState("Circle Scribbles");
    graph.addState("Circle Scribbles with Mark");
    graph.addState("Spiral");
    graph.addState("Circle or Oval");
    /* Add links to graph */
    graph.addChange("Circle Scribbles", "Circle Scribbles with Mark", "Control more than 2");
    graph.addChange("Circle Scribbles with Mark", "Spiral", "Control more than 3, Physicality less than 4");
    graph.addChange("Spiral", "Circle or Oval", "Sense of Self more than 4");
    return(graph);
}
```

### 4.4.2 Developmental Model

This class implements the developmental model for the Safira toolkit, based on the notion of a developmental graph laid out in the previous section.
4.4.2.1 Output

The Developmental Model maintains autonomous communication with the outside world through the means of a set of First-In, First-Out queues, where it places information it would like to make known. Calling programs may empty these queues if appropriate for their internal processing; they are not used as internal data structures, but only for communication with the calling program.

```java
public FIFOQueue requests;
```

This queue of strings holds requests the model makes to the outside world, i.e. current conditions in the developmental graph. These conditions are extracted from the transitionConditions that are part of the developmental graph. It is the responsibility of the calling program to monitor the conditions in this queue and to let the Developmental Model know when any of them become true.

```java
public FIFOQueue endedRequests;
```

This queue holds conditions for which requests were made but that are now unnecessary. They may be removed from the calling program’s list of conditions to be monitored.

```java
public FIFOQueue activatedStates;
```

This queue lists states that have been activated.

```java
public FIFOQueue deactivatedStates;
```

This queue lists states that have been deactivated.

4.4.2.1.1 Processing

```java
public DevelopmentalModel(DevelopmentalGraph graph)
```

Creates a new DevelopmentalModel. The parameter is the graph (as defined above) that the model uses.

```java
public boolean succeedCondition(String conditionName)
```

This method takes care of changes to the current state of developmental model that are a result of a condition succeeding in the world. It checks all currently monitored preconditions and post conditions, and as a result of their succeeding, activates and deactivates states and updates the data structures holding them as necessary. The parameter is the string describing the succeeded condition (as previously placed on the “requests” queue). The method returns true if the condition was found, and false otherwise. In this case it also prints a warning message.

```java
public HashSet activeStates()
```

Returns a set of the names of currently active states. The names are strings (as passed in during the creation of the DevelopmentalGraph).

```java
public void printActiveStates()
```


Prints the active states in the model.

```java
public void printActiveStatesDetailed()
```

Prints a detailed report on the active states in the model.

### 4.4.2.2 Example

This simple example shows how to use the developmental model: how to initialise it, how to keep track of the conditions it emits, and how to incorporate the activation of states into an outside program. In the example, the Developmental Model is initialised with a graph. Then, the requested conditions are triggered randomly, until the last state in the graph is reached.

First, here are the variables used in the class:

```java
/** The developmental model used */
private static DevelopmentalModel dModel;
/** The list of conditions which I am keeping track of for */
/** the developmental model */
private static HashSet queriedConditions;
/** Has the whole graph been traversed? */
private static boolean donep;
```

Now, we are ready to define our main function. It can be called directly.

```java
/** The main function. */
public static void main (String args[]) {
```

First, we initialise the variables in the class. To create the developmental model, we pass in a graph that is made elsewhere (for more on making graphs, see the previous section).

```java
// initialize data structures
queriedConditions = new HashSet();
dModel = new DevelopmentalModel(makeGraph());
donep = false;
```

Now, we enter the main loop. At each tick, the program will probabilistically trigger one of the conditions for which the Developmental Model. This results in newly activated and deactivated states, which we print to standard out.

```java
while(!donep) {
    // update developmental state and get new queries
```

First, we check the conditions that the developmental model is monitoring, and probabilistically decide whether to tell the model they are true. This will result inside the developmental model with new states being activated or deactivated.
// tell DM about any true conditions
for (Iterator i = queriedConditions.iterator();
i.hasNext(); ) {
    String conditionName = (String) i.next();
    // randomly trigger them
    if (Math.random() > 0.8) {
        System.out.println("Triggering condition " +
                            conditionName);
        dModel.succeedCondition(conditionName);
        i.remove();
    }
}

Next, we check for any new conditions that may have become timely because of newly
activated states. We add these to our own list of conditions to monitor.

    // get new conditions from DM
    while (!dModel.requests.empty()) {
        String newCondition = (String)
            dModel.requests.pop();
        System.out.println("Testing new condition " +
                            newCondition);
        // add stuff to my list of things to keep track of
        queriedConditions.add(newCondition);
    }

Then, we check for any conditions that we have been monitoring, which the developmental
model no longer finds pertinent. We can safely remove these from our list.

    // get conditions which are no longer pertinent
    while (!dModel.endedRequests.empty()) {
        String oldCondition = (String)
            dModel.endedRequests.pop();
        System.out.println("Deleting condition request " +
                            oldCondition);
        queriedConditions.remove(oldCondition);
    }

Now we print the states which have just been activated or deactivated. If one of the states is
the final state, we can finish our loop and our travels through the developmental model.

    // print the activated and deactivated states
    while (!dModel.activatedStates.empty()) {
        String ds = (String) dModel.activatedStates.pop();
        System.out.println("State " + ds +
                            " is activated");
        // check if the final state has been triggered
        if (ds.equals("All States Reached")) {
            donep = true;
        }
    }
while(!dModel.deactivatedStates.empty()) {
    String ds = (String)
        dModel.deactivatedStates.pop();
    System.out.println("State " + ds +
        " is deactivated");
}
}
5 EXPERIMENTATION, DISCUSSION AND CONCLUSIONS

The general SAFIRA component approach to software engineering has proved to be a useful development paradigm to build complex agents. Both component developers and agent designers may profit of using the SAFIRA integration paradigm. Component developers are not tied to particular technological choices. Agent designers may choose the components they want to use in their agents and may even decide to use other software as well. Most importantly, not all the developed components fit together in a single framework. While the appraisal component can be used with the developmental component or with the Emotional Production System (Em-PSys), the developmental component hardly fits with the Em-PSys component. This means that agent designers may even choose the theoretical framework. Another advantage is the Central Registry Service (CRS). Using this service, no component has to know about the others. They just need to register their services and to know what they want to get from others.

In the context of reasoning, planning, and learning with emotions, an interactive experience called “Monga World” was implemented to test the affective components developed. It could thereby be shown that the components lend themselves to the implementation of whole control architectures, where the connection between stimulus events and selection of behavioural reactions is mediated by an emotional system offering both, fast and heuristic, as well as more elaborated and slower stimulus processing. Preliminary results indicate that agents built with these components exhibit behaviour that can be deemed emotional.

From the software engineering point of view, Salt & Pepper can be seen as a programming framework in which agent developers are encouraged to separate an agent’s deliberative reasoning and control from the agent’s evaluative process. The programming of the rules that control the agent behaviour is largely independent of the programming of the emotion eliciting conditions and corresponding emotion-responses. The agent control rules do not mention generated emotion signals. The responses to emotion signals must be programmed in pattern-action clauses. The agent development team can and should split into separate groups, each of which is responsible for one of these aspects.

Since the conflict resolution strategy used to select one rule from a set of satisfied rules is unpredictable, the agent programmers cannot implicitly use their knowledge of the used strategy to design rules that only work when that strategy is used. Consequently, rules do not use implicit control knowledge. All necessary control must be explicitly programmed into the rules, which makes them clearer.

The use of Salt & Pepper offers other possibilities: the interplay between deliberative control and automatic control mechanisms. Agent control is ultimately decided by the relative accessibility of knowledge structures in long-term memory, which is strongly dependent on the agent’s external environment and mainly on emotion. As a result of this property, deliberative control may be superseded by automatic non-deliberative control. The main short-term consequence is that agent behaviour becomes unpredictable and more difficult to control. At the time of this writing, this property is difficult to deal with from the point of view of agent programming.
The Cognitive Engine wraps the cognitive aspects of the Salt & Pepper architecture in a component skeleton that is very convenient for the component designer. It includes an interface through which the agent designer can easily specify the agent’s knowledge structures, interaction requirements, and parameter settings. The Cognitive Engine automates the larger part of component interaction tasks. Most importantly, it implements the Salt & Pepper basic control mechanisms responsible for the integration of deliberative and automatic agent control, and it provides an interface to the Salt & Pepper long term memory system. The task of creating a concrete component on top of the Cognitive Engine is thus greatly simplified. The Emotional Production System component was developed on top of the Cognitive Engine.

Although the Cognitive Engine facilitates the creation of Salt & Pepper deliberative components, its design needs some improvement. In particular, several of its features were tied to the creation of the Emotional Production System instead of being independent of any specific future use. On the other hand, several tasks currently carried out by the Emotional Production System, such as activating and associating nodes selected to “use memory” (see section Error! Reference source not found.), should have been built into the Cognitive Engine instead. Some other tasks explicitly performed by the Cognitive Engine should have been built into the Salt & Pepper long-term memory. The most obvious example is the Activation Decrementer.

Finally, configuration parameter settings and component interaction specification should be removed from the code of the Cognitive Engine and stored in specification files instead, in order to improve the software’s generality.

The Emotional Production System Component (Em-PSys), developed on top of the Cognitive Engine, is used to specify two kinds of agent control mechanisms. The agent designer must specify the set of production rules that represent the agent’s deliberative control, and a set of emotion responses to be performed by the agent in response to generated emotion signals. While the specification of the production rules and related predicates and internal actions can easily be done through Prolog files, the specification of the emotional responses involves both Prolog and Java programming. This was a result of having used SICStus Prolog to implement Em-PSys. The development of the first beta version of Nazgûl Prolog, a Prolog fully implemented in Java to be used instead in the future, has just been completed. With Nazgûl Prolog, it will be possible to implement the agent’s emotional responses in Em-PSys entirely in Prolog.

In the Salt & Pepper framework, cognition influences emotion, which in turn influences cognition and behaviour. In order to use Em-PSys to create an agent in which cognition influences emotion, the agent designer has to provide descriptions for the available actions, and has to design agent control rules in such a way that the action of the rule can contribute to achieve part or all the goals explicitly stated in the condition part of the rule (see beginning of section 3.1.3.2). Em-PSys automatically computes desired-expectations tied to specific actions from the intersection of rule goals with action effects. This greatly simplifies the burden on the designer of agents with emotions.

The set of built-in programming facilities in Em-PSys is quite reasonable. Since it is implemented in Prolog, all Prolog built-in predicates can be used by the agent designer. Besides, Em-PSys includes a relatively large collection of actions that make sense in the context of the Salt & Pepper. These include long-term memory manipulation actions, working memory manipulation actions, input buffer access actions, output buffer access actions, and control
actions. The usefulness of the set of built-in actions can only be further evaluated by using Em-PSys to create real agents.

The Appraisal Compiler automatically generates appraisal components that can be used by any component that knows how to subscribe to it (i.e., uses the XML schema message subscription). Appraisal components are very flexible because they allow agent designers to supply domain dependent appraisal rules. This feature is complemented with two built-in appraisal modules: the Expectations Evaluator Module and the Unpredicted Action Module. Until now, the appraisal components could not yet be tested with sophisticated appraisal rules (like, for example, the full proposals by Ortony). We only used simple rules in the test bed we developed (explained later in this section).

In the proposed appraisal architecture, cognitive appraisals can coexist with more basic ones. The designer can decide what kind of knowledge should enrich each module. Since each module runs on a separate thread, basic modules can quickly trigger emotion responses independently of some more demanding cognitive eliciting conditions evaluations.

The Appraisal Compiler is easy to use because designers need only supply simple domain dependent information. They have to fill the Jess knowledge base with domain dependent rules. In the future, it would be useful to add a graphical toolkit to help the writing of the rules (syntax checking and XML schema validation).

In the present version, the appraisal compiler only generates independent components. One natural extension is to generate appraisal modules that can be easily integrated into existing components.

We have created a game, the Monga World, which is being used as a very demanding test bed for evaluating agents built out of Salt & Pepper components, namely the Em-PSys and the Appraisal Component, together with sensors and effectors.

Monga World is a dynamic, unpredictable, and uncertain environment in which the agent faces several challenges, including traps, dangerous creatures such as Grosserontes, and poisoned food. The only goal of the agent is to get out of this vicious world through the exit door. Besides having to find its way out, the agent will possibly have to eat and take medicines in order to increase its energy and health.

The analysis of and preliminary tests in this environment allowed us to reach some early conclusions. Firstly, emotion-eliciting conditions should not be all programmed in appraisal components. It was found very useful to program emotion eliciting conditions into the agent’s effectors. From a theoretical point of view, this is a relevant finding, since it seems to support the idea that there is certainly more to emotion than just cognitive appraisal.

Secondly, we found it very useful to delegate an important play in the agent control to the agent effectors, instead of centralizing every detail in the agent production system. For instance, the effectors will be capable of two non-primitive actions: go to and go around. Got to action is used to move the agent to a certain location, expressed in relative coordinates, that can be seen by the agent. Go around action is used to move the agent to a certain location, expressed in relative coordinates, that is behind an obstacle. The agent effector is capable of executing these two actions without the help of the agent deliberative control system. These
ideas were inspired in the control architecture proposed by James Firby and co-workers [Firby et al. 1995].

Thirdly, we also found it useful to separate the evaluative aspect of the agent behaviour (emotion eliciting conditions) from the agent deliberative control (production rules) because the Monga World environment is a very demanding problem. This separation enabled us to focus our attention in a specific aspect of the problem at a time. Unfortunately, there is a negative side in this coin. It has been difficult to write the rules and emotion responses that control the agent. In fact, due to the control mechanism of the Cognitive Engine, the agent behaviour becomes totally unpredictable. Therefore, it is difficult to write the appropriate rules and emotion-responses since they will interact in unpredictable ways. However, we are firmly convinced that this problem arises because it is the first time we have to program an agent using an emotional control device.

Rule based programming is already difficult for programmers not used to the paradigm, emotional rule based programming has proven to be even more difficult. However, the problem should be overcome as more experience is gained.

In the end, the implemented component for development ended up being a kind of active memory or data structure which keeps track of developmental states and changes that are occurring elsewhere in the system. The developmental model is lightweight; it does not engage in much deep computation itself, off-loading this labour to the components that calculate its predicates. On the one hand, this is a simplistic model of development. On the other, it is simple, flexible, and general. What would it take to develop a deeper model of development? Because of the intertwined nature of development with other aspects of cognition and behaviour, we suspect that a deeper model would need to be task-specific, focusing on supporting, for example, the development of spatial cognition, or on the immediate needs of the application for which development is being used.

Strangely for a project that is focused on affective interaction, the developmental model, in the end, does not focus on emotions. It supports emotion-based development with a more general model, which can handle affective properties and triggers along with other kinds. Originally, in the design of the development component we had a more specialised perspective. In particular, we thought of predicates for developmental change as being based on particular emotional levels, such as “anger > 3”. However, this model of the predicate did not seem general enough, at least lacking a notion of the passage of time. More generally, emotion is only one aspect of development. In the end, it turned out to be inappropriate to have a model of development that was based primarily or largely on emotion, since emotion is interweaved with other factors even in what we might call mostly emotion-based development. An additional factor was discussion with project partners, who believed the predicate model was too limiting for the way they saw development as being possibly useful in their work in affective modelling. In the end, the general model seemed more useful than an emotion-specific one.

It is rewarding to know that the supporting tools built to facilitate using and building affective components can be used within and outside the scope of the project, even in applications totally unrelated to affective computing.

The Schema Class Builder currently is the software tool closest to commercial quality and general usefulness. A web page with a servlet was created, that enables the general use of
the Schema Class Builder (http://www.adetti-linha4.org/SchemaClassBuilder/). Through that web page, users may submit their XML Schema files and receive the corresponding Java classes. Then, they can use the received classes in a program to parse and validate XML messages, and convert them into instances of the generated classes.

In the project, we use the Schema Class Builder to create the Java classes that are used by the components to process the XML messages exchanged during component interaction.

The Schema Class Builder has had other uses. For instance, it was used to create the Java class that represents the XML message envelope used in inter-agent communication in the AgentCities project. There, agents use the generated class to parse the received messages and convert them into internal Java objects.

The Interaction Manager is a software tool that can be used in any Java software component that needs to communicate with other components. Its use is not as general as the use of the Schema Class Builder, but it is not limited to emotion related components. In order to use the Interaction Manager, the component designer just has to specify the files containing the XML Schemas of the exchanged messages; the XML Schemas of the information that is subscribed to by the component from other components, and the handlers used in the component to process the specified messages.

Finally, although Nazgûl Prolog has not been described in this deliverable, it has been developed for the SAFIRA project, in order to overcome some restrictive features of SICStus Prolog. The development of Nazgûl Prolog will have great impact in other projects in which the SAFIRA Partners are involved and beyond. It is often useful to integrate Java and Prolog to create agents. There are many useful Java software tools for building agents, mainly at the infrastructure level (agent platforms, parsers, message transport, procedural control, and user interfaces). Another important advantage of the Java technology is operating system independence. On the other hand, Prolog technology offers several advantages to create flexible reasoning and control mechanisms, mainly because of its symbolic processing capabilities (especially, pattern-matching), its inference engine, and its backtracking capabilities.

There are several Prolog implementations providing an interface with Java. Unfortunately, many of them lack commercial quality, mainly because they are not supported and because they do not have enough documentation (e.g., JavaLog and GNU Prolog for Java). There are also exceptions, such as SICStus Prolog. But these exceptions are usually not free and are not independent of the operating system. Being produced in-house, Nazgûl Prolog should prove easier to use than the other alternatives.

Nazgûl Prolog is also being used in some of our classrooms, it will be used in some of our AgentCities agents, and Em-PSys is currently being updated to use Nazgûl.

The SAFIRA project has defined a general Integration Framework and has built the Central Registry Service, which can be used to integrate software components to create agents. SAFIRA has also developed three demonstrators that may be used to build affective agents, plus some general affective software tools that may be used to create other affective components. Although these tools provide interesting features for agent developers, they do not have commercial quality yet. Finally, it would be desirable to create a greater variety of components. These needs could perfectly be handled by another project with three main objectives: (i) to fix
identified design problems and convert the created prototypes into documented commercial quality products; (ii) to develop additional affective components; and (iii) to actually use the developed components to build real agents for different applications.

As a side effect, SAFIRA has also developed useful general tools that can be used outside the project, in totally unrelated application domains. Among these, we emphasise the Schema Class Builder and Nazgûl Prolog.
6 REFERENCES


Firby, R.J., Kahn, R.E., Prokopowicz, P.N. and Swain, M.J. 1995. “An architecture for vision and action”, *IJCAI’95*, p72-79


Gratch, J. 2000. Émile: Marshalling Passions in Training and Education


